Software Specification

G-API

Documentation
Version 1.4 (r7601)

GOEPEL electronic GmbH
Goeschwitzer Str. 58/60
D-07745 Jena
+49-3641-6896-597
ats_support@goepel.com
http://www.goepel.com

G-API: Documentation

Version 1.4 (r7601)
Published 01/23/2020

Table of Contents

List of Figures

List of Tables

Bus Independent Functions

In order to communicate with a device via **G-API** it is necessary first that the application opens a port by the command G_Common_OpenInterface which assigns a port-related handle to this application. This port handle is used for all further function calls. If a second application establishes a connection to the API, another handle is assigned to it. One application can open several ports. For finishing the application, call the command G_Common_CloseInterface which closes the port and releases the used resources.

These **G-API** commands provide the basic functionality for the hardware communication. If the return value of type **G_Error_t** is not equal to `G_NO_ERROR`, an error has occurred. The error description can be retrieved by command G_GetLastErrorDescription .

For a list with all error codes, see file *g_error.h* in the *bin* directory of your installation.

1 Definitions

Common **G-API** definitions

G_Common_Signal_t

G_Common_Signal_t — Signal definition

Description

A Common definition for different kinds of signals.

Definition

```
typedef struct {
 u8_t  Size;
 G_Common_Signal_ByteOrder_t  ByteOrder;
 u16_t  Offset;
} G_Common_Signal_t;
```

Parameters

`Size`
Signal size in bits

`ByteOrder`
Byte order of message

The following values are possible:

`G_COMMON__SIGNAL__BYTE_ORDER__LITTLE_ENDIAN`
Byte order is **little endian** or **Intel** format

`G_COMMON__SIGNAL__BYTE_ORDER__BIG_ENDIAN`
Byte order is **big endian** or **Motorola** format

Signals with **little endian** byte order start in byte **N** and end in byte **N + 1** . Signals with **big endian byte order** start in byte **N** but end in byte **N - 1** .

`Offset`
Signal offset in bits

2 Common

These commands provide control over common properties.

G_Common_OpenInterface

G_Common_OpenInterface — Open a port

Description

This command opens a port for the communication with the hardware.

The function **G_Common_OpenInterface** opens a communication port to the interface using its logical name. For a description how to assign a logical name to an interface, refer to the **G-API** Quickstart Guide.

For opening a port by using **G_InterfaceInfo_t** parameters, see G_OpenInterface .

Definition

```
G_Error_t
G_Common_OpenInterface(
 const char * const interfaceName,
 G_PortHandle_t * portHandle
);
```

Parameters

`interfaceName`
Pointer to the variable `interfaceName` which indicates the logical name of the hardware to be addressed. This logical name can be entered in the **GOEPEL electronic Hardware Explorer** .

`portHandle`
Pointer to the variable of type **G_PortHandle_t**. The port handle to be indicated for all further function calls is saved in this variable.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Initialization of a **basicCAN** USB module

```
G_PortHandle_t portHandle;
G_Error_t rc;

rc =
 G_Common_OpenInterface(
   "CAN1",
   &portHandle
 );
```

The indicated name `CAN1` is the identifier for the hardware to be addressed. It has been set with the **GOEPEL electronic Hardware Explorer** and is passed on as a string. This is done using the surrounding " " signs. As for the variable `portHandle` only the address is passed on and this is made by means of the prefixed "&". This is necessary because in the called function the port handle is written in the variable.

The application making the first connection to a controller by this command has subsequently the exclusive right to reset this controller.

G_OpenInterface

G_OpenInterface — Open a port

Description

This command opens a port that enables the communication with the hardware.

The function **G_OpenInterface** opens a communication port to the hardware under specification of the corresponding parameters in structure `interfaceInfo`. For calling the hardware per logical name, please use the function G_Common_OpenInterface .

Definition

```
G_Error_t
G_OpenInterface(
 const G_InterfaceInfo * const interfaceInfo,
 G_PortHandle_t * const portHandle
);
```

Parameters

`interfaceInfo`
: Pointer to a structure of type **G_InterfaceInfo_t**. The content of the structure defines the hardware for communication.

`portHandle`
: Pointer to a variable of type **G_PortHandle_t**. On function return, this variable contains the port handle that is used in all of the following function calls.

Please consider that - depending on the hardware type - it is necessary to initialize the structure `interfaceInfo` differently.

The following parameters are required for the communication with USB, PCI and PXI devices :

- `interfaceInfo.Type` - Hardware type, defined by **G_HostInterfaceType_t**
- `interfaceInfo.Parameters.ControllerNumber` - Number of the controller to be called by this function
- `interfaceInfo.Parameters.InterfaceNumber` - Number of the interface to be called by this function
- `interfaceInfo.Parameters.Flags` - Interface info flags
- `interfaceInfo.Parameters.u.CardNumber` - Number of the card to be called by this function.

  When using several devices of the same type, this number corresponds to the device serial number. The `CardNumber` of the device with the lowest serial number is **1** , the device with next higher serial number has a `CardNumber` value of **2** , etc.

The following parameters are required for the communication with Ethernet devices:

- `interfaceInfo.Type` - Host interface type `G_HOST_INTERFACE_TYPE__ETHERNET`
- `interfaceInfo.Parameters.ControllerNumber` - Number of the controller to be called
- `interfaceInfo.Parameters.InterfaceNumber` - Number of the interface to be called
- `interfaceInfo.Parameters.Flags` - Interface info flags
- `interfaceInfo.Parameters.u.Ethernet.PortNumber` - Port number of the device (default: **5134**)
- `interfaceInfo.Parameters.u.Ethernet.reserved1` - reserved byte - has to be initialized with **0**
- `interfaceInfo.Parameters.u.Ethernet.reserved2` - reserved byte - has to be initialized with **0**

- `interfaceInfo.Parameters.u.IpAddress` - IP address of the device to be called

  When flag `G_INTERFACE_INFO__FLAG__IP_ADDRESS_IN_BYTES` is set, the ip address has to be declared as a 4-byte-value.

```
// declare the ip address "192.168.1.62" as a 4-byte-value
interfaceInfo.Parameters.u.IpAddress[0] = 192;
interfaceInfo.Parameters.u.IpAddress[1]=  168;
interfaceInfo.Parameters.u.IpAddress[2] = 1;
interfaceInfo.Parameters.u.IpAddress[3] = 62;
```

  Otherwise, the ip address has to be declared as a string.

```
// declare the ip address "192.168.1.62" as a string
(char *) interfaceInfo.Parameters.u.IpAddress = "192.168.1.62";
```

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Initialization of a **basicCAN** USB module

```
G_PortHandle_t portHandle;
G_InterfaceInfo_t interfaceInfo;
G_Error_t rc;
interfaceInfo.Type = G_HOST_INTERFACE_TYPE__USB_BASICCAN;
interfaceInfo.Parameters.ControllerNumber = 1;
interfaceInfo.Parameters.InterfaceNumber = 1;
interfaceInfo.Parameters.Flags = G_INTERFACE_INFO__FLAG__NONE;
interfaceInfo.Parameters.u.CardNumber = 1;

rc = G_OpenInterface(&interfaceInfo, &portHandle);
```

The application making the first connection to a controller by this command has subsequently the exclusive right to reset this controller.

G_Common_CloseInterface

G_Common_CloseInterface — Close a port

Description

This command releases the port requested by G_Common_OpenInterface and the used resources.

Definition

```
G_Error_t
G_Common_CloseInterface(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
rc = G_Common_CloseInterface(portHandle);
```

The port belonging to `portHandle` is closed and the used resources are released.

G_CloseInterface

G_CloseInterface — Close a port

Description

This command releases the port requested by G_OpenInterface and the used resources.

Definition

```
G_Error_t
G_CloseInterface(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
rc = G_CloseInterface(portHandle);
```

The port belonging to `portHandle` is closed and the used resources are released.

G_AsyncCommunication_ErrorCallback

G_AsyncCommunication_ErrorCallback — Callback function for the asynchronous error handler

Description

This function is called as soon as an error has occurred while an asynchronous operation was pending. It must be entered by the user and indicated when activating the asynchronous communication with G_AsyncCommunication_Enable.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_AsyncCommunication_ErrorCallback(
 const G_PortHandle_t  portHandle,
 const G_Error_t  error,
 const u8_t * const errorInfo,
 const u32_t  errorInfoLength
);
```

Parameters

`portHandle`
: Handle to the communication port

`error`
: Error code of type **G_Error_t**

`errorInfo`
: Pointer to the information about the error occurred

`errorInfoLength`
: Length of information saved under `errorInfo` in byte

G_AsyncCommunication_Enable

G_AsyncCommunication_Enable — Activate the asynchronous communication for the port

Description

This command activates the asynchronous communication for the selected port.

The command acknowledgment is deactivated for the commands and the user is notified in case of errors occurred by means of the callback function indicated under `errorCallback`. If an error occurs already before sending the command, it is shown as before by the return value `rc`.

The action of functions while asynchronous operation of the API can be additionally influenced by the functions G_AsyncCommunication_FunctionsWithoutResponse_Sync and G_AsyncCommunication_FunctionsWithoutResponse_Async .

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_API_DLL G_Error_t
G_AsyncCommunication_Enable(
 const G_PortHandle_t  portHandle,
 const G_AsyncCommunication_ErrorCallback_t  errorCallback
);
```

Parameters

`portHandle`
Handle to the communication port

`errorCallback`
Function pointer of the error callback function (see G_AsyncCommunication_ErrorCallback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
rc =
G_AsyncCommunication_Enable(
 portHandle,
 ErrorCallbackFunction
);
```

G_AsyncCommunication_Disable

G_AsyncCommunication_Disable — Deactivate the asynchronous communication for the port

Description

This command deactivates the asynchronous communication for the selected port.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_API_DLL G_Error_t
G_AsyncCommunication_Disable(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
rc = G_AsyncCommunication_Disable(portHandle);
```

G_AsyncCommunication_FunctionsWithoutResponse_Sync

G_AsyncCommunication_FunctionsWithoutResponse_Sync — Activate the command acknowledgment while asynchronous communication

Description

This command activates the command acknowledgment for commands without response while asynchronous communication.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_API_DLL G_Error_t
G_AsyncCommunication_FunctionsWithoutResponse_Sync(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
rc =
G_AsyncCommunication_FunctionsWithoutResponse_Sync(
 portHandle
);
```

G_AsyncCommunication_FunctionsWithoutResponse_Async

G_AsyncCommunication_FunctionsWithoutResponse_Async — Deactivate the command acknowledgment while asynchronous communication

Description

This command deactivates the command acknowledgment for commands without response while asynchronous communication.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_API_DLL G_Error_t
G_AsyncCommunication_FunctionsWithoutResponse_Async(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
rc =
G_AsyncCommunication_FunctionsWithoutResponse_Async(
 portHandle
);
```

G_AsyncCommunication_FunctionsWithoutResponse_IsSync

G_AsyncCommunication_FunctionsWithoutResponse_IsSync — Query of the command acknowledgment status while asynchronous communication

Description

This command queries the status of the command acknowledgment for commands without response while asynchronous communication.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_API_DLL u8_t
G_AsyncCommunication_FunctionsWithoutResponse_IsSync(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

Return Value

Status of the command acknowledgment:
0: Command acknowledgment deactivated
1: Command acknowledgment activated

Example

```
u8_t  sync;

sync =
G_AsyncCommunication_FunctionsWithoutResponse_IsSync(
 portHandle
);
```

G_Common_InitInterface

G_Common_InitInterface — Initialize the interface

Description

This commmand initializes the indicated interface with standard values.

Definition

```
G_Error_t
G_Common_InitInterface(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
rc = G_Common_InitInterface(portHandle);
```

The hardware interface indicated by `portHandle` is initialized with standard values.

G_Common_GetFirmwareVersion

G_Common_GetFirmwareVersion — Read out the firmware version

Description

This command queries the firmware version number of the selected controller.

Definition

```
G_Error_t
G_Common_GetFirmwareVersion(
 const G_PortHandle_t portHandle,
 char * const response,
 u32_t * const responseLength
);
```

Parameters

`portHandle`
Handle to the communication port

`response`
Pointer to the response data buffer

`responseLength`
Pointer to the variable `reponseLength`

Content when invoked: Size of response data buffer

Returned content: Size of response data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Query of the firmware version by **G_Common_GetFirmwareVersion**

```
char response[4096];
u32_t responseLength;

…

responseLength = 4096;

rc =
G_Common_GetFirmwareVersion(
 portHandle,
 response,
 &responseLength
);

if (rc == G_NO_ERROR) {
printf("%s", response);
}
```

In this example a response buffer of 4096 bytes is declared with `response`. Its size is also input into the variable `responseLength`. The function parameters of **G_Common_GetFirmwareVersion** are the port handle `response` as pointer to the response buffer and the address of the variable `responseLength`, which on function call contains the size of the response buffer and on function return the size of the response data. The command is executed completely only after receiving a response or an error. If the return code `rc` with the value `G_NO_ERROR` indicates an error-free execution, the firmware version value will be printed with **printf**, that is saved as zero-terminated string in the response buffer `response`.

G_Common_GetFirmwareVersion_Async

G_Common_GetFirmwareVersion_Async — Query of the firmware version (by asynchronous query)

Description

This command initiates the asynchronous firmware query of the selected controller.

In contrast to G_Common_GetFirmwareVersion , this command returns immediately (that means without waiting for the response). As soon as the corresponding response is available, the callback function being defined by G_Common_GetFirmwareVersion_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Common_GetFirmwareVersion_Async(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_GetFirmwareVersion_Async_Callback

G_Common_GetFirmwareVersion_Async_Callback — Callback function for asynchronous query of the firmware version

Description

This function will be called as soon as a response to the asynchronous query of the firmeware version is received. It has to be entered by the user and set with G_Common_GetFirmwareVersion_Async_AddCallback.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Common_GetFirmwareVersion_Async_Callback(
 const G_PortHandle_t  portHandle,
 const char * const version,
 const u32_t  versionLength
);
```

Parameters

`portHandle`
: Handle to the communication port

`version`
: Pointer to string with firmware version

`versionLength`
: Length of the string with firmware version in bytes

G_Common_GetFirmwareVersion_Async_AddCallback

G_Common_GetFirmwareVersion_Async_AddCallback — Set the callback function for asynchronous query of the firmware version

Description

This command defines a callback function for the asynchronous query of the firmware version with G_Common_GetFirmwareVersion_Async.

Definition

```
G_Error_t
G_Common_GetFirmwareVersion_Async_AddCallback(
 const G_PortHandle_t portHandle,
 const G_Common_GetFirmwareVersion_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`callback`
Function pointer of the callback function (see G_Common_GetFirmwareVersion_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_GetFirmwareVersion_Async_RemoveCallback

G_Common_GetFirmwareVersion_Async_RemoveCallback — Remove the callback function for G_Common_GetFirmwareVersion_Async

Description

This command removes the callback function for asynchronous query of the firmware.

Definition

```
G_Error_t
G_Common_GetFirmwareVersion_Async_RemoveCallback(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_GetFirmwareErrorDescription

G_Common_GetFirmwareErrorDescription — Get the description for the firmeware error code

Description

This command queries the description of a firmware error code.

Definition

```
G_Error_t
G_Common_GetFirmwareErrorDescription(
 const G_PortHandle_t portHandle,
 const G_Error_t errorCode,
 char * const response,
 u32_t * const responseLength
);
```

Parameters

`portHandle`
Handle to the communication port

`errorCode`
Variable of type **G_Error_t** indicating the firmware error code whose description has to be queried

`response`
Pointer to the response data buffer

`responseLength`
Pointer to the variable `reponseLength`

Content when invoked: Size of response data buffer

Returned content: Size of response data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
char response[4096];
u32_t responseLength = 4096;
G_Error_t errorCode = 0x40;

rc =
G_Common_GetFirmwareErrorDescription(
 portHandle,
 errorCode,
 response,
 &responseLength
);

if (rc == G_NO_ERROR) {
printf("Description: %s", response);
}
```

In this example a response buffer of 4096 bytes is declared with `response`. Its size is also input into the variable `responseLength`. The description of the firmware error code `0x40` is subsequently queried by **G_Common_GetFirmwareErrorDescription** . If the return code `rc` with the value `G_NO_ERROR` indicates an error-free execution, the description of the firmware error code will be printed with **printf** that is saved as zero-terminated string in the response buffer `response`.

G_Common_SoftwareReset

G_Common_SoftwareReset — Reset a controller

Description

This command resets the controller belonging to the port handle.

Definition

```
G_Error_t
G_Common_SoftwareReset(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
rc = G_Common_SoftwareReset(portHandle);
```

The controller belonging to `portHandle` is reset.

G_Common_GetControllerInfo

G_Common_GetControllerInfo — Query information about controller

Description

This command is used to query information about the selected controller.

Definition

```
G_Error_t
G_Common_GetControllerInfo(
 const G_PortHandle_t portHandle,
 G_Common_GetControllerInfo_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`rsp`
Pointer to response structure

The structure contains the following elements:

`HostInterfaceType`
Type of host interface, defined by **G_HostInterfaceType_t**

`ControllerFamily`
Family of controller

The following values are possible:

`G_COMMON__CONTROLLER_FAMILY__UNKNOWN`
Unknown controller family

`G_COMMON__CONTROLLER_FAMILY__C16X`
Infineon C16x

`G_COMMON__CONTROLLER_FAMILY__TRICORE`
Infineon TriCore

`G_COMMON__CONTROLLER_FAMILY__PPC`
Power PC

`ControllerType`
Type of controller

The following values are possible:

`G_COMMON__CONTROLLER_TYPE__UNKNOWN`
Unknown controller type

`G_COMMON__CONTROLLER_TYPE__C164CI`
Infineon C164CI

`G_COMMON__CONTROLLER_TYPE__TC1765`
Infineon TC1765

G_COMMON__CONTROLLER_TYPE__TC1775B
Infineon TC1775B

G_COMMON__CONTROLLER_TYPE__TC1796B
Infineon TC1796B

G_COMMON__CONTROLLER_TYPE__PPC460EX
AMCC PPC460EX

reserved1
reserved parameter (must be initialized with **0**)

reserved2
reserved parameter (must be initialized with **0**)

Flags
Response flags

The following values are possible and can be combined:

G_COMMON__GET_CONTROLLER_INFO__RSP_FLAG__NONE
No response flag set

G_COMMON__GET_CONTROLLER_INFO__RSP_FLAG__IS_BOOT_LOADER
The controller is in **Boot Loader Mode**.

ResetStatusRegister
Hardware-specific reset information

HardwareVersion
Hardware version

SoftwareVersion
Software version

NumberOfInterfaces
Number of interfaces of the controller

Interfaces
Array with interface information

Each array-element contains the following elements:

InterfaceType
Interface Type

The following values are possible:

G_INTERFACE_TYPE__UNKNOWN
Unknown interface type

G_INTERFACE_TYPE__BOOT_LOADER
Boot Loader interface

G_INTERFACE_TYPE__SEQUENCES
Sequences interface

G_INTERFACE_TYPE__SEQUENCES
Interface for controlling command sequences

G_INTERFACE_TYPE__IO
Input / Output interface

`G_INTERFACE_TYPE__CAN`
CAN interface

`G_INTERFACE_TYPE__LIN`
LIN interface

`G_INTERFACE_TYPE__KLINE`
K-Line interface

`G_INTERFACE_TYPE__MOST`
MOST interface

`G_INTERFACE_TYPE__LVDS`
LVDS interface

`G_INTERFACE_TYPE__FLEXRAY`
FlexRay interface

`InterfaceNumber`
Interface number

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_GetControllerInfo_Async

G_Common_GetControllerInfo_Async — Query controller information (by asynchronous query)

Description

This command initiates the asynchronous query of controller information.

In contrast to G_Common_GetControllerInfo, this command returns immediately (that means without waiting for the response). As soon as the corresponding response is available, the callback function being defined by G_Common_GetControllerInfo_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Common_GetControllerInfo_Async(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_GetControllerInfo_Async_Callback

G_Common_GetControllerInfo_Async_Callback — Callback function for the asynchronous query of controller information

Description

This function will be called as soon as a response to the asynchronous query of the controller information is received. It has to be defined by the user and set with G_Common_GetControllerInfo_Async_AddCallback.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Common_GetControllerInfo_Async_Callback(
 const G_PortHandle_t  portHandle,
 const G_Common_GetControllerInfo_Rsp_t * const rsp
);
```

Parameters

`portHandle`
: Handle to the communication port

`rsp`
: Pointer to response structure (see GetControllerInfo Response Structure)

G_Common_GetControllerInfo_Async_AddCallback

G_Common_GetControllerInfo_Async_AddCallback — Set the callback function for asynchronous query of controller information

Description

This command is used to define a callback function for the asynchronous query of controller information with G_Common_GetControllerInfo_Async.

Definition

```
G_Error_t
G_Common_GetControllerInfo_Async_AddCallback(
 const G_PortHandle_t portHandle,
 const G_Common_GetControllerInfo_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`callback`
Function pointer to callback function (see G_Common_GetControllerInfo_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_GetControllerInfo_Async_RemoveCallback

G_Common_GetControllerInfo_Async_RemoveCallback — Remove callback function for G_Common_GetControllerInfo_Async

Description

This command removes the callback function for asynchronous query of controller information.

Definition

```
G_Error_t
G_Common_GetControllerInfo_Async_RemoveCallback(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_GetInterfaceInfo

G_Common_GetInterfaceInfo — Query information about interface

Description

This command is used to query information about the selected interface.

Definition

```
G_Error_t
G_Common_GetInterfaceInfo(
 const G_PortHandle_t portHandle,
 G_Common_GetInterfaceInfo_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`rsp`
Pointer to response structure

The structure contains the following elements:

`InterfaceType`
Type of interface (see Interface Type)

`Flags`
Response flags

The following values are possible and can be combined:

`G_COMMON__GET_INTERFACE_INFO__RSP_FLAG__NONE`
No flag is set

`G_COMMON__GET_INTERFACE_INFO__RSP_FLAG__INSTANCE_ID_PRESENT`
The instance ID of the interface is returned in `InstanceId`

`InstanceId`
Instance ID of the interface (only valid if flag `G_COMMON__GET_INTERFACE_INFO__RSP_FLAG__INSTANCE_ID_PRESENT` is set)

`reserved`
reserved parameter (must be initialized with `0`)

`InterfaceVersion`
Version of the interface

`NumberOfFeatures`
Number of features of the interface

`Features`
Array with interface features

The following values are possible:

`G_COMMON__FEATURE__UNKNOWN`
Unknown feature

G_COMMON__FEATURE__CAN__TP__TP_1_6
CAN transport protocol **TP 1.6**

G_COMMON__FEATURE__CAN__TP__TP_2_0
CAN transport protocol **TP 2.0**

G_COMMON__FEATURE__CAN__TP__ISOTP
CAN transport protocol **ISO TP** (ISO 15765-2)

G_COMMON__FEATURE__CAN__TP__GMLAN
CAN transport protocol **GMLAN**

G_COMMON__FEATURE__CAN__TP__J1939
CAN transport protocol **J1939** (SAE J1939-21)

G_COMMON__FEATURE__DIAG__CAN__KWP2000__TP_1_6
Diagnostic protocol **Keyword 2000** (ISO 14230) on **TP 1.6** for CAN

G_COMMON__FEATURE__DIAG__CAN__KWP2000__TP_2_0
Diagnostic protocol **Keyword 2000** (ISO 14230) on **TP 2.0** for CAN

G_COMMON__FEATURE__DIAG__CAN__KWP2000__ISOTP
Diagnostic protocol **Keyword 2000** (ISO 14230) on **ISO TP** (ISO 15765-2) for CAN

G_COMMON__FEATURE__DIAG__CAN__GMLAN
Diagnostic protocol **GMLAN** (ISO 15765-2) for CAN

G_COMMON__FEATURE__DIAG__CAN__UDS__ISOTP
Diagnostic protocol **UDS** (ISO 14229-1) on **ISO TP** (ISO 15765-2) for CAN

G_COMMON__FEATURE__DIAG__CAN__J1939
Diagnostic protocol **J1939** (SAE J1939-73) for CAN

G_COMMON__FEATURE__DIAG__LIN__RAW
Raw Diagnostics for LIN

G_COMMON__FEATURE__DIAG__LIN__LIN_2_0
Diagnostic protocol **LIN 2.0** for LIN

G_COMMON__FEATURE__DIAG__KLINE__KWP2000
Diagnostic protocol **Keyword 2000** (ISO 14230) for K-Line

G_COMMON__FEATURE__DIAG__KLINE__KWP1281
Diagnostic protocol **Keyword 1281** for K-Line

G_COMMON__FEATURE__DIAG__KLINE__FORD
Ford-specific Diagnostics for K-Line

G_COMMON__FEATURE__DDP__CAN__TP_1_6__ECU
Display Data Protocol on **TP 1.6** for CAN from ECU side

G_COMMON__FEATURE__DDP__CAN__TP_1_6__DISPLAY
Display Data Protocol on **TP 1.6** for CAN from display side

G_COMMON__FEATURE__DDP__CAN__TP_2_0__ECU
Display Data Protocol on **TP 2.0** for CAN from ECU side

G_COMMON__FEATURE__DDP__CAN__TP_2_0__DISPLAY
Display Data Protocol on **TP 2.0** for CAN from display side

G_COMMON__FEATURE__CUSTOM__LIN_ADVANCED_LIBRARY
Advanced Library of LIN commands

`G_COMMON__FEATURE__CUSTOM__CCP`
Can Calibration Protocol (CCP)

`G_COMMON__FEATURE__CUSTOM__XCP`
Extended Calibration Protocol (XCP)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_GetInterfaceInfo_Async

G_Common_GetInterfaceInfo_Async — Query interface information (by asynchronous query)

Description

This command initiates the asynchronous query of interface information.

In contrast to G_Common_GetInterfaceInfo, this command returns immediately (that means without waiting for the response). As soon as the corresponding response is available, the callback function being defined by G_Common_GetInterfaceInfo_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Common_GetInterfaceInfo_Async(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_GetInterfaceInfo_Async_Callback

G_Common_GetInterfaceInfo_Async_Callback — Callback function for asynchronous query of interface information

Description

This function will be called as soon as a response to the asynchronous query of the interface information is received. It has to be defined by the user and set with G_Common_GetInterfaceInfo_Async_AddCallback.

Definition

```
void
G_Common_GetInterfaceInfo_Async_Callback(
 const G_PortHandle_t  portHandle,
 const G_Common_GetInterfaceInfo_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`rsp`
Pointer to response structure (see GetInterfaceInfo Response Structure)

G_Common_GetInterfaceInfo_Async_AddCallback

G_Common_GetInterfaceInfo_Async_AddCallback — Set the callback function for asynchronous query of interface information

Description

This command is used to define a callback function for the asynchronous query of interface information with G_Common_GetInterfaceInfo_Async.

Definition

```
G_Error_t
G_Common_GetInterfaceInfo_Async_AddCallback(
 const G_PortHandle_t portHandle,
 const G_Common_GetInterfaceInfo_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`callback`
Function pointer to callback function (see G_Common_GetInterfaceInfo_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_GetInterfaceInfo_Async_RemoveCallback

G_Common_GetInterfaceInfo_Async_RemoveCallback — Remove callback function for G_Common_GetInterfaceInfo_Async

Description

This command removes the callback function for asynchronous query of interface information.

Definition

```
G_Error_t
G_Common_GetInterfaceInfo_Async_RemoveCallback(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_GetTransceiverInfo

G_Common_GetTransceiverInfo — Query information about transceiver

Description

This command is used to query information about the transceivers of your device.

The query of transceiver information is only possible on specific devices.

Definition

```
G_Error_t
G_Common_GetTransceiverInfo(
 const G_PortHandle_t portHandle,
 G_Common_GetTransceiverInfo_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`rsp`
Pointer to response structure

The structure contains the following elements:

`NumberOfTransceiverInfos`
Number of returned transceiver information

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`InterfaceVersion`
Version of the interface

`TransceiverInfos`
Array with transceiver information

Each entry consists of the following elements:

`HardwarePosition`
Slot number of transceiver

`TransceiverFlags`
Transceiver flags

The following values are possible and can be combined:

`G_COMMON__TRANSCEIVER_FLAG__NONE`
No flag is set

G_COMMON__TRANSCEIVER_FLAG__ISOLATED
The transceiver is isolated

TransceiverType
Transceiver type

The following values are possible:

G_COMMON__TRANSCEIVER_TYPE__UNKNOWN
Unknown transceiver type

G_COMMON__TRANSCEIVER_TYPE__CAN__TJA1041
Highspeed CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__CAN__TJA1041A
Highspeed CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__CAN__82C252
Lowspeed CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__CAN__AU5790
Single Wire CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__CAN__TJA1054
Lowspeed CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__CAN__TJA1054A
CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__CAN__TJA1050
CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__CAN__82C250
CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__CAN__82C251
CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__CAN__TLE6250
CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__CAN__B10011S
CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__LIN__TJA1020
LIN transceiver

G_COMMON__TRANSCEIVER_TYPE__LIN__TLE6258
LIN transceiver

G_COMMON__TRANSCEIVER_TYPE__LIN__TLE6259
LIN transceiver

G_COMMON__TRANSCEIVER_TYPE__LIN__TJA1020_LOAD
LIN transceiver

G_COMMON__TRANSCEIVER_TYPE__KLINE__L9637
K-Line transceiver

G_COMMON__TRANSCEIVER_TYPE__KLINE__RX_TX
K-Line transceiver

`G_COMMON__TRANSCEIVER_TYPE__FLEXRAY__TJA1080`
FlexRay transceiver

`G_COMMON__TRANSCEIVER_TYPE__FLEXRAY__AS8221`
FlexRay transceiver

`G_COMMON__TRANSCEIVER_TYPE__ADIO__TYPE_1`
Analog / digital - input / output board

`G_COMMON__TRANSCEIVER_TYPE__ADIO__TYPE_2`
Analog / digital - input / output board

`G_COMMON__TRANSCEIVER_TYPE__ADIO__TYPE_3`
Analog / digital - input / output board

`G_COMMON__TRANSCEIVER_TYPE__ETHERNET__1000BASE_T`
Ethernet transceiver 1000Base-T

`G_COMMON__TRANSCEIVER_TYPE__ETHERNET__BRR_TJA1100`
Ethernet transceiver BRR TJA1100

`G_COMMON__TRANSCEIVER_TYPE__ETHERNET__AETH_88Q2112`
Ethernet transceiver AETH 88Q2112

`G_COMMON__TRANSCEIVER_TYPE__ETHERNET__AETH_88Q2112_A2`
Ethernet transceiver AETH 88Q2112 A2

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_GetTransceiverInfo_Async

G_Common_GetTransceiverInfo_Async — Query transceiver information (by asynchronous query)

Description

This command initiates the asynchronous query of transceiver information.

In contrast to G_Common_GetTransceiverInfo, this command returns immediately (that means without waiting for the response). As soon as the corresponding response is available, the callback function being defined by G_Common_GetTransceiverInfo_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Common_GetTransceiverInfo_Async(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_GetTransceiverInfo_Async_Callback

G_Common_GetTransceiverInfo_Async_Callback — Callback function for asynchronous query of transceiver information

Description

This function will be called as soon as a response to the asynchronous query of the transceiver information is received. It has to be defined by the user and set with G_Common_GetTransceiverInfo_Async_AddCallback.

Definition

```
void
G_Common_GetTransceiverInfo_Async_Callback(
 const G_PortHandle_t  portHandle,
 const G_Common_GetTransceiverInfo_Rsp_t * const rsp
);
```

Parameters

`portHandle`
: Handle to the communication port

`rsp`
: Pointer to response structure (see GetTransceiverInfo Response Structure)

G_Common_GetTransceiverInfo_Async_AddCallback

G_Common_GetTransceiverInfo_Async_AddCallback — Set the callback function for asynchronous query of transceiver information

Description

This command is used to define a callback function for the asynchronous query of transceiver information with G_Common_GetTransceiverInfo_Async.

Definition

```
G_Error_t
G_Common_GetTransceiverInfo_Async_AddCallback(
 const G_PortHandle_t portHandle,
 const G_Common_GetTransceiverInfo_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`callback`
Function pointer to callback function (see G_Common_GetTransceiverInfo_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_GetTransceiverInfo_Async_RemoveCallback

G_Common_GetTransceiverInfo_Async_RemoveCallback — Remove callback function for G_Common_GetTransceiverInfo_Async

Description

This command removes the callback function for asynchronous query of transceiver information.

Definition

```
G_Error_t
G_Common_GetTransceiverInfo_Async_RemoveCallback(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_GetTransceiverInfoForCurrentInterface

G_Common_GetTransceiverInfoForCurrentInterface — Query transceiver information for current interface

Description

Use this command to query the transceiver information for the current interface.

In contrast to G_Common_GetTransceiverInfo, only the transceiver information for the current interface (defined by `portHandle`) is returned.

Definition

```
G_Error_t
G_Common_GetTransceiverInfoForCurrentInterface(
 const G_PortHandle_t portHandle,
 const G_Common_GetTransceiverInfoForCurrentInterface_CmdFlags_t  cmdFlags,
 G_Common_GetTransceiverInfoForCurrentInterface_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_COMMON__GET_TRANSCEIVER_INFO_FOR_CURRENT_IFACE__CMD_FLAG__NONE`
No flag is set

`rsp`
Returns response parameters

The structure consists of the following elements:

`HardwarePosition`
Hardware position of the transceiver (starting with **0**)

`HardwareSubPosition`
Hardware sub position of the transceiver (starting with **0**)

If the transceiverboard has more than 1 transceiver slot, the actual position of the queried transceiver is returned as the hardware sub position.

`Type`
Transceiver type

The following values are possible:

`G_COMMON__TRANSCEIVER_TYPE__UNKNOWN`
Unknown transceiver type

`G_COMMON__TRANSCEIVER_TYPE__CAN__TJA1041`
Highspeed CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__CAN__TJA1041A
Highspeed CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__CAN__82C252
Lowspeed CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__CAN__AU5790
Single Wire CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__CAN__TJA1054
Lowspeed CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__CAN__TJA1054A
CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__CAN__TJA1050
CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__CAN__82C250
CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__CAN__82C251
CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__CAN__TLE6250
CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__CAN__B10011S
CAN transceiver

G_COMMON__TRANSCEIVER_TYPE__LIN__TJA1020
LIN transceiver

G_COMMON__TRANSCEIVER_TYPE__LIN__TLE6258
LIN transceiver

G_COMMON__TRANSCEIVER_TYPE__LIN__TLE6259
LIN transceiver

G_COMMON__TRANSCEIVER_TYPE__LIN__TJA1020_LOAD
LIN transceiver

G_COMMON__TRANSCEIVER_TYPE__KLINE__L9637
K-Line transceiver

G_COMMON__TRANSCEIVER_TYPE__KLINE__RX_TX
K-Line transceiver

G_COMMON__TRANSCEIVER_TYPE__FLEXRAY__TJA1080
FlexRay transceiver

G_COMMON__TRANSCEIVER_TYPE__FLEXRAY__AS8221
FlexRay transceiver

G_COMMON__TRANSCEIVER_TYPE__ADIO__TYPE_1
Analog / digital - input / output board

G_COMMON__TRANSCEIVER_TYPE__ADIO__TYPE_2
Analog / digital - input / output board

G_COMMON__TRANSCEIVER_TYPE__ADIO__TYPE_3
Analog / digital - input / output board

G_COMMON__TRANSCEIVER_TYPE__ETHERNET__1000BASE_T
Ethernet transceiver 1000Base-T

G_COMMON__TRANSCEIVER_TYPE__ETHERNET__BRR_TJA1100
Ethernet transceiver BRR TJA1100

G_COMMON__TRANSCEIVER_TYPE__ETHERNET__AETH_88Q2112
Ethernet transceiver AETH 88Q2112

G_COMMON__TRANSCEIVER_TYPE__ETHERNET__AETH_88Q2112_A2
Ethernet transceiver AETH 88Q2112 A2

`Flags`
Transceiver flags

The following values are possible and can be combined:

G_COMMON__TRANSCEIVER_FLAG__NONE
No flag is set

G_COMMON__TRANSCEIVER_FLAG__ISOLATED
The transceiver is isolated

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_GetInterfaceNumber

G_Common_GetInterfaceNumber — Query interface number of specific interface

Description

Use this command to query the interface number of a specific interface.

Definition

```
G_Error_t
G_Common_GetInterfaceNumber(
 const G_PortHandle_t portHandle,
 const G_Common_GetInterfaceNumber_Parameters_t * const parameters,
 G_Common_GetInterfaceNumber_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_COMMON__GET_INTERFACE_NUMBER__CMD_FLAG__NONE`
No flag is set

`InterfaceType`
Interface Type

The following values are possible:

`G_INTERFACE_TYPE__UNKNOWN`
Unknown interface type

`G_INTERFACE_TYPE__BOOT_LOADER`
Boot Loader interface

`G_INTERFACE_TYPE__SEQUENCES`
Sequences interface

`G_INTERFACE_TYPE__SEQUENCES`
Interface for controlling command sequences

`G_INTERFACE_TYPE__IO`
Input / Output interface

`G_INTERFACE_TYPE__CAN`
CAN interface

`G_INTERFACE_TYPE__LIN`
LIN interface

`G_INTERFACE_TYPE__KLINE`
K-Line interface

`G_INTERFACE_TYPE__MOST`
MOST interface

`G_INTERFACE_TYPE__LVDS`
LVDS interface

`G_INTERFACE_TYPE__FLEXRAY`
FlexRay interface

`InstanceId`
Instance ID of the interface (starting with **0**)

If multiple interfaces of the same type are present, they can be distinguished by the instance ID which is an increasing number depending on the transceiver slot.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`InterfaceNumber`
The interface number of the specific interface

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_GetFeatureCode

G_Common_GetFeatureCode — Query feature codes

Description

Use this command to query feature codes.

Definition

```
G_Error_t
G_Common_GetFeatureCode(
 const G_PortHandle_t portHandle,
 const G_Common_GetFeatureCode_CmdFlags_t  cmdFlags,
 const G_Common_FeatureCode_Mode_t  mode,
 u32_t  featureCode[4]
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible:

`G_COMMON__GET_FEATURE_CODE__CMD_FLAG__NONE`
No flag is set

`mode`
Determines what feature code is to be queried

The following values are possible:

`G_COMMON__FEATURE_CODE__FIRMWARE`
Firmware feature code

This feature code represents the available firmware features for the device.

`G_COMMON__FEATURE_CODE__APPLICATION`
Application feature code

This feature code represents the available application features for the device.

`featureCode`
Returns the feature code as 4 32-Bit values.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Command

G_Command — Command with acknowledge

Description

Send a command to the device and wait for the corresponding acknowledge.

This command is rudimental. If applicable, please use a higher level **G-API** function instead!

Definition

```
G_Error_t
G_Command(
 const G_PortHandle_t portHandle,
 const u8_t commandCode,
 const u8_t * const parameters,
 const u32_t parameterLength
);
```

Parameters

`portHandle`
Handle to the communication port

`commandCode`
Marks the command which is sent to the hardware

`parameters`
Pointer to the parameter list of the command

`parameterLength`
Parameter length in byte

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Define a message on the CAN bus with **G_Command**

```
struct {
u32_t  Id;
u16_t  CycleTime;
u8_t   Mode;
u8_t   PrepareMode;
u8_t   MsgCount;
u8_t   Dlc;
u8_t   Data[8];
u8_t   reserved1;
u8_t   reserved2;
} parameters;

parameters.Id = 0x123;
parameters.CycleTime = 100;  // 100 milliseconds
parameters.Mode = 1;
parameters.PrepareMode = 1;
parameters.Dlc = 2;          // 2 bytes data
parameters.Data[0] = 0x11;
parameters.Data[1] = 0x22;
parameters.reserved1 = 0;
parameters.reserved2 = 0;

…

rc =
G_Command(
 portHandle,
 0x22
 &parameters,
 sizeof(parameters)
);
```

In this example the variable `parameters` is declared first and subsequently completed with the required values. After then the command is sent to the device by the **G_Command** . The transferred parameters are the port handle, the command code (`0x22` for defining a CAN message), the address of the parameter list and the length of the parameter list in bytes. The command **G_Command** is executed completely only after getting the command acknowledgment or an error message. If the parameter list of a command contains reserved bytes (`reserved`), they must be always initialized with `0`!

G_CommandWithResponse

G_CommandWithResponse — Command with response

Description

This command sends a command to the device and waits for a response.

This command is rudimental. If applicable, please use a higher level **G-API** function instead!

Definition

```
G_Error_t
G_CommandWithResponse(
 const G_PortHandle_t portHandle,
 const u8_t commandCode,
 const u8_t * const parameters,
 const u32_t parameterLength,
 u8_t * const response,
 u32_t * const responseLength
);
```

Parameters

`portHandle`
Handle to the communication port

`commandCode`
Marks the command that is sent to the hardware

`parameters`
Pointer to the parameter list of the command

`parameterLength`
Parameter length in bytes

`response`
Pointer to the response data buffer

`responseLength`
Pointer to the variable `reponseLength`

Content when invoked: Size of response data buffer

Returned content: Size of response data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Read out the firmware version with **G_CommandWithResponse**

```
u8_t response[4096];
u32_t responseLength;

…

responseLength = 4096;

rc =
G_CommandWithResponse(
 portHandle,
 0xF0,
 ZERO,
 0,
 response,
 &responseLength
);

if (rc == G_NO_ERROR) {
printf("%s", response);
}
```

In this example a response buffer of 4096 bytes is declared with `response`. Its size is also input into the variable `responseLength`. The function parameters of **G_CommandWithResponse** are the port handle, the command code (`0xF0` for reading out the firmware version), `ZERO` as pointer to the parameter list and `0` for the length of the parameter list because the command does not need parameters, `response` as pointer to the response buffer und the address of the variable `responseLength` containing the size of the response buffer on function call and the size of the response data on function return. The command is executed completely only after receiving a response or an error. If the return code `rc` with the value `G_NO_ERROR` indicates an error-free execution, the firmware version value is printed with **printf** that is saved as zero-terminated string in the response buffer `response`.

G_CommandWithResponseEx

G_CommandWithResponseEx — Command with response (extended)

Description

Send a command to the device. Any further action depends on `mode`.

This command is rudimental. If applicable, please use a higher level **G-API** function instead!

Definition

```
G_API_DLL G_Error_t
G_CommandWithResponseEx(
 const G_PortHandle_t  portHandle,
 const G_CommandWithResponseMode_t mode,
 const u8_t  commandCode,
 const u8_t  controlDataLength,
 const u8_t  * const parameters,
 const u32_t  parameterLength,
 u8_t  * const response,
 u32_t  * const responseLength
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
Variable of type **G_CommandWithResponseMode_t**

The following return values are possible:

`G_COMMAND_WITH_RESPONSE_MODE__SYNC`
Always wait for the response

`G_COMMAND_WITH_RESPONSE_MODE__ASYNC`
Do not wait for response (only available if the asynchronous communication was activated)

`commandCode`
Command code specifying the command

`controlDataLength`
Number of the data bytes of the parameter list to be checked for on receiving the response

`parameters`
Pointer to the parameter list of the command

`parameterLength`
Parameter length in byte

`response`
Pointer to the response data buffer

`responseLength`
Pointer to the variable `reponseLength`

Content when invoked: Size of response data buffer

Returned content: Size of response data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_GetLastErrorCode

G_GetLastErrorCode — Get the error code

Description

This command calls the error code of the last error that has occurred.

Definition

```
G_Error_t
G_GetLastErrorCode(
 void
);
```

Parameters

None

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Send the last error that has occurred with **G_GetLastErrorCode**

```
G_Error_t rc;

…

rc = G_GetLastErrorCode();

printf("Last Error: %x", rc);
```

In this example the error code of the last error that has occurred, is saved with **G_GetLastErrorCode** in the variable `rc` and subsequently printed in hexadecimal code with **printf** .

G_GetLastErrorDescription

G_GetLastErrorDescription — Get the error description

Description

This command queries the description of the last error that has occurred.

Definition

```
G_DLL const char *
G_GetLastErrorDescription(
 void
);
```

Parameters

None

Return Value

Error description as zero-terminated string

Example

```
   printf(
    "Last Error Description: %s",
    G_GetLastErrorDescription()
   );
```

In this example the error description of the last error that has occurred is queried by **G_GetLastErrorDescription** and printed by **printf** .

G_GetErrorDescription

G_GetErrorDescription — Get error description

Description

This command queries the description of an error code.

Definition

```
const char *
G_GetErrorDescription(
 const G_Error_t errorCode
);
```

Parameters

`errorCode`
Error code whose description has to be obtained

Return Value

Error description as zero-terminated string

Example

```
    printf(
     "Error Description: %s",
     G_GetErrorDescription(0x02000008)
    );
```

In this example the error description of the error code `0x02000008` is queried by **G_GetErrorDescription** and printed by **printf** .

G_GetErrorDescription2

G_GetErrorDescription2 — Get error description including firmware errors

Description

This command queries the description of an error code. In contrast to G_GetErrorDescription it is possible to query the detailed description of firmware error codes.

Definition

```
G_Error_t
G_GetErrorDescription2(
 const G_PortHandle_t  portHandle,
 const G_Error_t  errorCode,
 u32_t * const length,
 char * const errorDescription
);
```

Parameters

`portHandle`
Handle to the communication port

`errorCode`
Error code whose description has to be obtained

`length`
in : Size of buffer `errorDescription` in bytes

out : Size of error description in bytes

`errorDescription`
Returns error description

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
u32_t  length = 256;
char desc[256];
G_Error_t  rc;

rc =
 G_GetErrorDescription2(
  PortHandle1,
  (G_Error_t) 0x03,
  &length,
  desc
 );
```

G_GetLastErrorByPortHandle

G_GetLastErrorByPortHandle — Get error description for port handle

Description

This command queries the description of an error code for a specific port handle.

Definition

```
G_Error_t
G_GetLastErrorByPortHandle(
 const G_PortHandle_t  portHandle,
 G_Error_t * const errorCode,
 u32_t * const length,
 char * const errorDescription
);
```

Parameters

`portHandle`
Handle to the communication port

`errorCode`
Returns error code

`length`
in : size of buffer for error description in byte

out : size of error description in byte

`errorDescription`
Buffer for error description

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_GetApiVersion

G_GetApiVersion — Returns **G-API** version

Description

This command returns the current **G-API** version.

Definition

```
G_Error_t
G_GetApiVersion(
 u32_t * const major,
 u32_t * const minor,
 u32_t * const revision
);
```

Parameters

`major`
Returns major version number

`minor`
Returns minor version number

`revision`
Returns revision number

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_UpdateToDedicatedFirmware

G_UpdateToDedicatedFirmware — Update the device to the dedicated firmware version

Description

Use this command to ensure that the device is using the dedicated firmware version.

Each firmware device supported by the **G-API** was tested with a specific firmware version. To ensure that the device is using this dedicated firmware, the specific **GOEPEL electronic firmware package** that belongs to this version of the **G-API** needs to be installed.

This function will check the firmware version of the device and flash the dedicated firmware if required.

Definition

```
G_Error_t
G_UpdateToDedicatedFirmware(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

3 EA

Event and Actions

These commands provide access to the Events and Actions functionality of the device.

In contrast to Section 5, "Events", these events are triggered by the firmware of the device.

The following steps are required to configure Events and Actions:

1. Allocate an Event
2. Allocate an Action
3. Initialize Event
4. Initialize Action
5. Assign Action to Event
6. Enable event

G_Common_EA_ResetAll

G_Common_EA_ResetAll — Reset all Events and Action resources

Description

Use this command to reset all resources used by Events and Actions.

Definition

```
G_Error_t
G_Common_EA_ResetAll(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_EA_AllocateEvent

G_Common_EA_AllocateEvent — Allocate Event

Description

Use this command to allocate an event.

The returned event handle is used for all subsequent operations regarding this event.

Definition

```
G_Error_t
G_Common_EA_AllocateEvent(
 const G_PortHandle_t  portHandle,
 G_Common_EA_EventHandle_t * const eventHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

`eventHandle`
: Returns handle to allocated event

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_EA_AllocateAction

G_Common_EA_AllocateAction — Allocate Action

Description

Use this command to allocate an action.

The returned action handle is used for all subsequent operations regarding this action.

Definition

```
G_Error_t
G_Common_EA_AllocateAction(
 const G_PortHandle_t  portHandle,
 G_Common_EA_ActionHandle_t * const actionHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

`actionHandle`
    Returns handle to allocated action

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_EA_InitEvent

G_Common_EA_InitEvent — Initialize event

Description

Use this command to initialize an event.

The event needs to be allocated with G_Common_EA_AllocateEvent before it can be initialized.

Definition

```
G_Error_t
G_Common_EA_InitEvent(
 const G_PortHandle_t  portHandle,
 const G_Common_EA_InitEvent_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`EventHandle`
Event handle

`EventType`
Event type

The following values are possible:

`G_COMMON__EA__EVENT_TYPE__TP__RX_INDICATION`
Transport protocol data was received

Supported interface types: **FlexRay**

`G_COMMON__EA__EVENT_TYPE__COM__RX_INDICATION`
COM layer data was received

Supported interface types: **Net2Run**

`G_COMMON__EA__EVENT_TYPE__COM__TX_CONFIRMATION`
COM layer data was transmitted

Supported interface types: **Net2Run**

`G_COMMON__EA__EVENT_TYPE__COM__TRANSMIT`
The COM layer triggered data for transmission

Supported interface types: **Net2Run**

`G_COMMON__EA__EVENT_TYPE__COM__RX_PDU_SIGNAL__RX_IND`
A COM layer Rx PDU signal has been received

Supported interface types: **Net2Run**

`G_COMMON__EA__EVENT_TYPE__COM__TX_PDU_SIGNAL__TX_CON`
A COM layer Tx PDU signal has been sent

Supported interface types: **Net2Run**

`G_COMMON__EA__EVENT_TYPE__NM__BUS_SLEEP_MODE`
The network management entered bus sleep mode

Supported interface types: **Net2Run**

`G_COMMON__EA__EVENT_TYPE__NM__NETWORK_MODE`
The network management entered network mode

Supported interface types: **Net2Run**

`G_COMMON__EA__EVENT_TYPE__TRANSCEIVER__WAKE_UP`
The transceiver woke up (e.g. due to a network management event)

Supported interface types: **FlexRay**

`G_COMMON__EA__EVENT_TYPE__TRIGGER__SOFTWARE_IN`
An event occurred at the configured software input

Supported interface types:

- FlexRay
- Net2Run
- Io
- MOST
- Sequence
- CAN
- LIN

`G_COMMON__EA__EVENT_TYPE__SOFTWARE__USER_EVENT`
A user event was triggered

Supported interface types:

- FlexRay
- Net2Run
- Io
- MOST
- Sequence
- CAN
- LIN

`G_COMMON__EA__EVENT_TYPE__FLEXRAY__CYCLE_START`
FlexRay cycle start

Supported interface types: **FlexRay**

G_COMMON__EA__EVENT_TYPE__FLEXRAY__PDU__RX_IND
FlexRay PDU Rx indication

Supported interface types: **FlexRay**

G_COMMON__EA__EVENT_TYPE__FLEXRAY__PDU__TX_CON
FlexRay PDU Tx confirmation

Supported interface types: **FlexRay**

G_COMMON__EA__EVENT_TYPE__FLEXRAY__SYNCHRONIZED
FlexRay is synchronized

Supported interface types: **FlexRay**

G_COMMON__EA__EVENT_TYPE__FLEXRAY__LINK_LOSS
FlexRay lost link

Supported interface types: **FlexRay**

G_COMMON__EA__EVENT_TYPE__BUS__RECEIVED_WAKE_UP
A wake up pattern was received

Supported interface types:

- FlexRay
- LIN

G_COMMON__EA__EVENT_TYPE__ETHERNET__LINKED
Ethernet connection is linked

Supported interface types: **Ethernet**

G_COMMON__EA__EVENT_TYPE__ETHERNET__LINK_LOSS
Ethernet connection link is lost

Supported interface types: **Ethernet**

G_COMMON__EA__EVENT_TYPE__ETHERNET__RX_FIFO__NEW_ENTRY
Ethernet Rx FiFo received new entry

Supported interface types: **Ethernet**

reserved1
reserved parameter (must be initialized with **0**)

reserved2
reserved parameter (must be initialized with **0**)

u
Union with event parameters, depending on the value of EventType

The union contains the following structures:

Tp_RxIndication
For event type G_COMMON__EA__EVENT_TYPE__TP__RX_INDICATION

The structure contains the following elements:

Channel
The transport protocol channel

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Com_RxIndication`
For event type `G_COMMON__EA__EVENT_TYPE__COM__RX_INDICATION`

The structure contains the following elements:

`Flags`
Flags

The following values are possible and can be combined:

`G_COMMON__EA__INIT_EVENT__SUB_CMD__COM__RX_INDICATION__CMD_FLAG__NONE`
No flag is set

`RxPduId`
ID of RX Pdu

`CompareOffset`
Byte offset for compare operation

`CompareLength`
Number of COM RX PDU data bytes to compare with `CompareMaskAndData` starting on byte position `CompareOffset`

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`CompareMaskAndData`
Mask bytes (0..(`CompareLength`-1)) directly followed by data bytes (0..(`CompareLength`-1))

e.g.:

```
CompareLength = 3;
CompareMaskAndData[0] = compareMask[0];
CompareMaskAndData[1] = compareMask[1];
CompareMaskAndData[2] = compareMask[2];
CompareMaskAndData[3] = compareData[0];
CompareMaskAndData[4] = compareData[1];
CompareMaskAndData[5] = compareData[2];
```

When the following pseudo-code returns with true, the event is processed:

```
for (i = 0; i < CompareLength; i++) {
```

```
 if ((comRxData[CompareOffset + i] & compareMask[i])
 != (compareData[i] & compareMask[i])) {
  return false;
 }
}

return true;
```

`Com_TxConfirmation`
For event type `G_COMMON__EA__EVENT_TYPE__COM__TX_CONFIRMATION`

The structure contains the following elements:

`Flags`
Flags

The following values are possible and can be combined:

`G_COMMON__EA__INIT_EVENT__SUB_CMD__COM__TX_CONFIRMATION__CMD_FLAG__NONE`
No flag is set

`TxPduId`
ID of TX Pdu

`CompareOffset`
Byte offset for compare operation

`CompareLength`
Number of COM TX PDU data bytes to compare with `CompareMaskAndData` starting on byte position `CompareOffset`

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`CompareMaskAndData`
Mask bytes (0..(`CompareLength`-1)) directly followed by data bytes (0..(`CompareLength`-1))

e.g.:

```
 CompareLength = 3;
 CompareMaskAndData[0] = compareMask[0];
 CompareMaskAndData[1] = compareMask[1];
 CompareMaskAndData[2] = compareMask[2];
 CompareMaskAndData[3] = compareData[0];
 CompareMaskAndData[4] = compareData[1];
 CompareMaskAndData[5] = compareData[2];
```

When the following pseudo-code returns with true, the event is processed:

```
for (i = 0; i < CompareLength; i++) {
```

```
 if ((comRxData[CompareOffset + i] & compareMask[i])
 != (compareData[i] & compareMask[i])) {
  return false;
 }
}

return true;
```

`Com_Transmit`
For event type `G_COMMON__EA__EVENT_TYPE__COM__TRANSMIT`

The structure contains the following elements:

`Flags`
Flags

The following values are possible and can be combined:

`G_COMMON__EA__INIT_EVENT__SUB_CMD__COM__TRANSMIT__CMD_FLAG__NONE`
No flag is set

`TxPduId`
ID of TX Pdu

`CompareOffset`
Byte offset for compare operation

`CompareLength`
Number of COM TX PDU data bytes to compare with `CompareMaskAndData` starting on byte position `CompareOffset`

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`CompareMaskAndData`
Mask bytes (0..(`CompareLength`-1)) directly followed by data bytes (0..(`CompareLength`-1))

e.g.:

```
 CompareLength = 3;
 CompareMaskAndData[0] = compareMask[0];
 CompareMaskAndData[1] = compareMask[1];
 CompareMaskAndData[2] = compareMask[2];
 CompareMaskAndData[3] = compareData[0];
 CompareMaskAndData[4] = compareData[1];
 CompareMaskAndData[5] = compareData[2];
```

When the following pseudo-code returns with true, the event is processed:

```
for (i = 0; i < CompareLength; i++) {
```

```
 if ((comRxData[CompareOffset + i] & compareMask[i])
 != (compareData[i] & compareMask[i])) {
  return false;
 }
}

return true;
```

`Com_RxPduSignal_RxInd`
For event type `G_COMMON__EA__EVENT_TYPE__COM__RX_PDU_SIGNAL__RX_IND`

RX Pdu signal has been received

The structure contains the following elements:

`Flags`
Flags

The following values are possible:

`G_COMMON__EA__INIT_EVENT__SUB_CMD__COM__RX_PDU_SIGNAL__RX_IND_ _CMD_FLAG__NONE`
No flag is set

`RxPduId`
ID of RX Pdu

`RxPduSignalId`
ID of the RX Pdu signal

`ConditionType`
Condition type

The following values are possible:

`G_COMMON__EA__CONDITION_TYPE__ALWAYS`
Condition: always true

`G_COMMON__EA__CONDITION_TYPE__MASKED_NEW_EQUALS_X`
Condition: (newValue & `ConditionMask`) == `ConditionX`

`G_COMMON__EA__CONDITION_TYPE__MASKED_NEW_DIFFERS_X`
Condition: (newValue & `ConditionMask`) != `ConditionX`

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`ConditionMask`
Condition mask

`ConditionX`
ConditionX

`Com_TxPduSignal_TxCon`
For event type `G_COMMON__EA__EVENT_TYPE__COM__TX_PDU_SIGNAL__TX_CON`

TX Pdu signal has been sent

The structure contains the following elements:

`Flags`
Flags

The following values are possible:

`G_COMMON__EA__INIT_EVENT__SUB_CMD__COM__TX_PDU_SIGNAL__TX_CON_ _CMD_FLAG__NONE`
No flag is set

`TxPduId`
ID of TX Pdu

`TxPduSignalId`
ID of the TX Pdu signal

`ConditionType`
Condition type

The following values are possible:

`G_COMMON__EA__CONDITION_TYPE__ALWAYS`
Condition: always true

`G_COMMON__EA__CONDITION_TYPE__MASKED_NEW_EQUALS_X`
Condition: (newValue & `ConditionMask`) == `ConditionX`

`G_COMMON__EA__CONDITION_TYPE__MASKED_NEW_DIFFERS_X`
Condition: (newValue & `ConditionMask`) != `ConditionX`

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`ConditionMask`
Condition mask

`ConditionX`
ConditionX

`Nm_BusSleepMode`
For event type `G_COMMON__EA__EVENT_TYPE__NM__BUS_SLEEP_MODE`

The structure contains the following elements:

`Flags`
Flags

The following values are possible and can be combined:

`G_COMMON__EA__INIT_EVENT__SUB_CMD__NM__BUS_SLEEP_MODE__CMD_ FLAG__NONE`
No flag is set

`BusType`
Bus type

The following values are possible:

`G_COMMON__EA__BUS_TYPE__CAN`
CAN bus

`G_COMMON__EA__BUS_TYPE__FLEXRAY`
FlexRay bus

`ControllerId`
Controller ID starting with **0** (e.g. **0** = **CAN1**, **1** = **CAN2**)

`EcuId`
ECU ID

`reserved`
reserved parameter (must be initialized with **0**)

`Nm_NetworkMode`
For event type `G_COMMON__EA__EVENT_TYPE__NM__NETWORK_MODE`

The structure contains the following elements:

`Flags`
Flags

The following values are possible and can be combined:

`G_COMMON__EA__INIT_EVENT__SUB_CMD__NM__NETWORK_MODE__CMD_FLAG_`
`_NONE`
No flag is set

`BusType`
Bus type

The following values are possible:

`G_COMMON__EA__BUS_TYPE__CAN`
CAN bus

`G_COMMON__EA__BUS_TYPE__FLEXRAY`
FlexRay bus

`ControllerId`
Controller ID starting with **0** (e.g. **0** = **CAN1**, **1** = **CAN2**)

`EcuId`
ECU ID

`reserved`
reserved parameter (must be initialized with **0**)

`Transceiver_WakeUp`
For event type `G_COMMON__EA__EVENT_TYPE__TRANSCEIVER__WAKE_UP`

The structure contains the following elements:

`Flags`
Flags

The following values are possible and can be combined:

G_COMMON__EA__INIT_EVENT__SUB_CMD__TRANSCEIVER__WAKE_UP__CMD_FLAG__NONE
No flag is set

Channel
Transceiver channel

CAN , LIN : 0

FlexRay : 0 = channel A, **1** = channel B

reserved1
reserved parameter (must be initialized with **0**)

reserved2
reserved parameter (must be initialized with **0**)

reserved3
reserved parameter (must be initialized with **0**)

Trigger_SoftwareIn
For event type G_COMMON__EA__EVENT_TYPE__TRIGGER__SOFTWARE_IN

The structure contains the following elements:

Flags
Command Flags

The following values are possible and can be combined:

G_COMMON__EA__INIT_EVENT__SUB_CMD__TRIGGER__SOFTWARE_IN__CMD_FLAG__NONE
No flag is set

G_COMMON__EA__INIT_EVENT__SUB_CMD__TRIGGER__SOFTWARE_IN__CMD_FLAG__RISING_EDGE
The event is triggered on a rising edge at the software in channel

G_COMMON__EA__INIT_EVENT__SUB_CMD__TRIGGER__SOFTWARE_IN__CMD_FLAG__FALLING_EDGE
The event is triggered on a falling edge at the software in channel

Channel
Software in channel (starting with 1)

reserved1
reserved parameter (must be initialized with **0**)

reserved2
reserved parameter (must be initialized with **0**)

reserved3
reserved parameter (must be initialized with **0**)

Software_UserEvent
For event type G_COMMON__EA__EVENT_TYPE__SOFTWARE__USER_EVENT

The structure contains the following elements:

Flags
Flags

The following values are possible and can be combined:

`G_COMMON__EA__INIT_EVENT__SUB_CMD__SOFTWARE__USER_EVENT__CMD_FLAG__NONE`
No flag is set

`FlexRay_CycleStart`
For event type `G_COMMON__EA__EVENT_TYPE__FLEXRAY__CYCLE_START`

The structure contains the following elements:

`Flags`
Flags

The following values are possible and can be combined:

`G_COMMON__EA__INIT_EVENT__SUB_CMD__FLEXRAY__CYCLE_START__CMD_FLAG__NONE`
No flag is set

`BaseCycle`
FlexRay base cycle

`CycleRepetition`
Cycle repetition, **0** = no cycle check

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`FlexRay_Pdu_RxInd`
For event type `G_COMMON__EA__EVENT_TYPE__FLEXRAY__PDU__RX_IND`

FlexRay Pdu RX indication

The structure contains the following elements:

`Flags`
Flags

The following values are possible and can be combined:

`G_COMMON__EA__INIT_EVENT__SUB_CMD__FLEXRAY__PDU__RX_IND__CMD_FLAG__NONE`
No flag is set

`NumberOfPduIds`
Number of Pdu IDs within `PduIds`

`reserved1`
reserved parameter (must be initialized with **0**)

`CompareLength`
Number of data bytes to compare with `CompareData`

`CompareOffset`
The compare starts at this PDU data offset

`PduLengthMin`
Minimum Pdu length

`PduLengthMax`
Maximum pdu length

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`PduIds`
Array with Pdu IDs

`CompareMask`
Array with compare mask bytes

`CompareData`
Array with compare data bytes

For example, with `CompareLength=2`, `CompareOffset=1`:

```
data[0] is not compared (because CompareOffset is 1)
data[1] is compared with CompareData[0] by CompareMask[0]
data[2] is compared with CompareData[1] by CompareMask[1]
data[3] is not compared (because data offset is out of compare
 range)
data[4] is not compared (because data offset is out of compare
 range)
data[5] is not compared (because data offset is out of compare
 range)
data[6] is not compared (because data offset is out of compare
 range)
data[7] is not compared (because data offset is out of compare
 range)
```

`FlexRay_Pdu_TxCon`
For event type `G_COMMON__EA__EVENT_TYPE__FLEXRAY__PDU__TX_CON`

FlexRay Pdu TX confirmation

The structure contains the following elements:

`Flags`
Flags

The following values are possible and can be combined:

`G_COMMON__EA__INIT_EVENT__SUB_CMD__FLEXRAY__PDU__TX_CON__CMD_FLAG__NONE`
No flag is set

`NumberOfPduIds`
Number of Pdu IDs within `PduIds`

`reserved1`
reserved parameter (must be initialized with **0**)

`CompareLength`
Number of data bytes to compare with `CompareData`

`CompareOffset`
The compare starts at this PDU data offset

`PduLengthMin`
Minimum Pdu length

`PduLengthMax`
Maximum pdu length

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`PduIds`
Array with Pdu IDs

`CompareMask`
Array with compare mask bytes

`CompareData`
Array with compare data bytes

For example, with `CompareLength=2`, `CompareOffset=1`:

```
data[0] is not compared (because CompareOffset is 1)
data[1] is compared with CompareData[0] by CompareMask[0]
data[2] is compared with CompareData[1] by CompareMask[1]
data[3] is not compared (because data offset is out of compare
 range)
data[4] is not compared (because data offset is out of compare
 range)
data[5] is not compared (because data offset is out of compare
 range)
data[6] is not compared (because data offset is out of compare
 range)
data[7] is not compared (because data offset is out of compare
 range)
```

`FlexRay_Synchronized`
For event type `G_COMMON__EA__EVENT_TYPE__FLEXRAY__SYNCHRONIZED`

FlexRay is synchronized

The structure contains the following elements:

`Flags`
Flags

The following values are possible and can be combined:

`G_COMMON__EA__INIT_EVENT__SUB_CMD__FLEXRAY__SYNCHRONIZED__CMD_FLAG__NONE`
No flag is set

`FlexRay_SyncLoss`
For event type `G_COMMON__EA__EVENT_TYPE__FLEXRAY__SYNC_LOSS`

FlexRay lost synchronization

The structure contains the following elements:

`Flags`
Flags

The following values are possible and can be combined:

`G_COMMON__EA__INIT_EVENT__SUB_CMD__FLEXRAY__SYNC_LOSS__CMD_FLAG__NONE`
No flag is set

`Bus_ReceivedWakeUp`
For event type `G_COMMON__EA__EVENT_TYPE__BUS__RECEIVED_WAKE_UP`

The structure contains the following elements:

`Flags`
Flags

The following values are possible and can be combined:

`G_COMMON__EA__INIT_EVENT__SUB_CMD__BUS__RECEIVED_WAKE_UP__CMD_FLAG__NONE`
No flag is set

`Channel`
Channel

LIN : **0**

FlexRay : **0** = channel A, **1** = channel B, **2** = channel A or B

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Ethernet_Linked`
For event type `G_COMMON__EA__EVENT_TYPE__ETHERNET__LINKED`

Ethernet connection is linked

The structure contains the following elements:

`Flags`
Flags

The following values are possible and can be combined:

`G_COMMON__EA__INIT_EVENT__SUB_CMD__ETHERNET__LINKED__CMD_FLAG__NONE`
No flag is set

`Ethernet_LinkLoss`
For event type `G_COMMON__EA__EVENT_TYPE__ETHERNET__LINK_LOSS`

Ethernet connection link is lost

The structure contains the following elements:

`Flags`
Flags

The following values are possible and can be combined:

`G_COMMON__EA__INIT_EVENT__SUB_CMD__ETHERNET__LINK_LOSS__CMD_FLAG__NONE`
No flag is set

`Ethernet_RxFifo_NewEntry`
For event type `G_COMMON__EA__EVENT_TYPE__ETHERNET__RX_FIFO__NEW_ENTRY`

A new RX FiFo entry has been received

The structure contains the following elements:

`Flags`
Flags

The following values are possible and can be combined:

`G_COMMON__EA__INIT_EVENT__SUB_CMD__ETHERNET__RX_FIFO__NEW_E__CMD_FLAG__NONE`
No flag is set

`RxFifoId`
ID of the RX FiFo

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_EA_InitAction

G_Common_EA_InitAction — Initialize Action

Description

Use this command to initialize an action.

The action needs to be allocated with G_Common_EA_AllocateAction before it can be initialized.

Definition

```
G_Error_t
G_Common_EA_InitAction(
 const G_PortHandle_t  portHandle,
 const G_Common_EA_InitAction_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`ActionHandle`
Action handle

`ActionType`
Action type

The following elements are possible:

`G_COMMON__EA__ACTION_TYPE__SEQUENCES__REPLAY__START_PLAYING`
Start a sequence

Supported interface types:

- FlexRay
- Net2Run
- Io
- MOST
- Sequence
- CAN
- LIN

`G_COMMON__EA__ACTION_TYPE__TRIGGER__SOFTWARE_OUT__WRITE`
Manipulate software output

Supported interface types:

- FlexRay
- Net2Run
- Io
- MOST
- Sequence
- CAN
- LIN

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`u`
Union with event parameters, depending on the value of `ActionType`

The union contains the following structures:

`Sequences_Replay_StartPlaying`
For action type `G_COMMON__EA__ACTION_TYPE__SEQUENCES__REPLAY__START_PLAYING`

The structure contains the following elements:

`SequenceHandle`
Handle of the sequence to be played

`Trigger_SoftwareOut_Write`
For action type `G_COMMON__EA__ACTION_TYPE__TRIGGER__SOFTWARE_OUT__WRITE`

The structure contains the following elements:

`Channel`
Software output number, starting with **1**

`Mode`
Write mode for software output

The following values are possible:

`G_COMMON_EA__TRIGGER__SOFTWARE_OUT__WRITE__MODE__SET`
The software output is set

`G_COMMON_EA__TRIGGER__SOFTWARE_OUT__WRITE__MODE__RESET`
The software output is reset

`G_COMMON_EA__TRIGGER__SOFTWARE_OUT__WRITE__MODE__TOGGLE`
The software output is toggled

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`PulseDuration`
0 : no pulse is generated, the trigger line is modified according to `Mode` only

not **0** : the trigger line is modified according to `Mode` and after the time `PulseDuration` the trigger line is toggled

Series61: resolution = **100ns** , max = **25500ns**

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_EA_AssignActionToEvent

G_Common_EA_AssignActionToEvent — Assign action to event

Description

Use this command to assign an action to an event

Action and event need to be initialized with G_Common_EA_InitAction and G_Common_EA_InitEvent first.

Definition

```
G_Error_t
G_Common_EA_AssignActionToEvent(
 const G_PortHandle_t  portHandle,
 const G_Common_EA_ActionHandle_t  actionHandle,
 const G_Common_EA_EventHandle_t  eventHandle
);
```

Parameters

`portHandle`
Handle to the communication port

`actionHandle`
Handle to action

`eventHandle`
Handle to event

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_EA_DeleteEvent

G_Common_EA_DeleteEvent — Delete event

Description

Use this command to delete an event. The event handle becomes invalid when the event is deleted.

Definition

```
G_Error_t
G_Common_EA_DeleteEvent(
 const G_PortHandle_t  portHandle,
 const G_Common_EA_DeleteEvent_CmdFlags_t  cmdFlags,
 const G_Common_EA_EventHandle_t  eventHandle
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_COMMON__EA__DELETE_EVENT__CMD_FLAG__NONE`
No flag is set

`G_COMMON__EA__DELETE_EVENT__CMD_FLAG__DELETE_ALL`
All events are deleted (the value of `eventHandle` is disregarded)

`eventHandle`
Handle to the event

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_EA_DeleteAction

G_Common_EA_DeleteAction — Delete action

Description

Use this command to delete an action. The action handle becomes invalid when the action is deleted.

Definition

```
G_Error_t
G_Common_EA_DeleteAction(
 const G_PortHandle_t  portHandle,
 const G_Common_EA_DeleteAction_CmdFlags_t  cmdFlags,
 const G_Common_EA_ActionHandle_t  actionHandle
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_COMMON__EA__DELETE_ACTION__CMD_FLAG__NONE`
No flag is set

`G_COMMON__EA__DELETE_ACTION__CMD_FLAG__DELETE_ALL`
All actions are deleted (the value of `actionHandle` is disregarded)

`actionHandle`
Handle to the action

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_EA_ExecuteAction

G_Common_EA_ExecuteAction — Execute an action

Description

Use this command to execute an action without beeing triggered by an event.

Definition

```
G_Error_t
G_Common_EA_ExecuteAction(
 const G_PortHandle_t  portHandle,
 const G_Common_EA_ActionHandle_t  actionHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

`actionHandle`
    Handle to the action

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_EA_CancelAction

G_Common_EA_CancelAction — Cancel action

Description

Use this command to cancel a running action.

Definition

```
G_Error_t
G_Common_EA_CancelAction(
 const G_PortHandle_t  portHandle,
 const G_Common_EA_ActionHandle_t  actionHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

`actionHandle`
: Handle to the action

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_EA_SetEventProperty

G_Common_EA_SetEventProperty — Set event property

Description

Use this command to set an event property

The event will not be triggered until its property G_COMMON__EA__EVENT_PROPERTY_ID_ _IS_ENABLED is set.

Definition

```
G_Error_t
G_Common_EA_SetEventProperty(
 const G_PortHandle_t  portHandle,
 const G_Common_EA_SetEventProperty_Cmd_t * const cmd
);
```

Parameters

portHandle
: Handle to the communication port

cmd
: Command parameters

  The structure contains the following elements:

  EventHandle
  : Event handle

  EventPropertyId
  : Event property ID

    The following values are possible:

    G_COMMON__EA__EVENT_PROPERTY_ID__IS_ENABLED
    : **0** = event is disabled

      1 = event is enabled

    G_COMMON__EA__EVENT_PROPERTY_ID__IS_INITIALIZED
    : **0** = event is not initialized

      1 = event is initialized

    G_COMMON__EA__EVENT_PROPERTY_ID__EVENT_COUNTER
    : Event counter value (is incremented by **1** when event occurred while it was enabled)

    G_COMMON__EA__EVENT_PROPERTY_ID__AUTO_DISABLE__IS_ENABLED
    : **0** = event is not automatically disabled

      1 = event is automatically disabled (see properties G_COMMON__EA__EVENT_PROPER-TY_ID__AUTO_DISABLE__ABSOLUTE and G_COMMON__EA__EVENT_PROPERTY_ID__ AUTO_DISABLE__RELATIVE)

    G_COMMON__EA__EVENT_PROPERTY_ID__AUTO_DISABLE__ABSOLUTE
    : Event is disabled automatically when event counter has reached this value and property G_COMMON__EA__EVENT_PROPERTY_ID__AUTO_DISABLE__IS_ENABLED is **1**

`G_COMMON__EA__EVENT_PROPERTY_ID__AUTO_DISABLE__RELATIVE`
Event is disabled automatically when event counter has reached the sum of the actual event counter value and this value. Additionally property `G_COMMON__EA__EVENT_PROPERTY_ID__AUTO_DISABLE__IS_ENABLED` has to be `1`.

`reserved1`
reserved parameter (must be initialized with `0`)

`reserved2`
reserved parameter (must be initialized with `0`)

`Value`
Event property value to be set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_EA_GetEventProperty

G_Common_EA_GetEventProperty — Get event property

Description

Use this command to query an event property

Definition

```
G_Error_t
G_Common_EA_GetEventProperty(
 const G_PortHandle_t  portHandle,
 const G_Common_EA_GetEventProperty_Cmd_t * const cmd,
 G_Common_EA_GetEventProperty_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`EventHandle`
Event handle

`EventPropertyId`
Event property ID

The following values are possible:

`G_COMMON__EA__EVENT_PROPERTY_ID__IS_ENABLED`
0 = event is disabled

1 = event is enabled

`G_COMMON__EA__EVENT_PROPERTY_ID__IS_INITIALIZED`
0 = event is not initialized

1 = event is initialized

`G_COMMON__EA__EVENT_PROPERTY_ID__EVENT_COUNTER`
Event counter value (is incremented by **1** when event occurred while it was enabled)

`G_COMMON__EA__EVENT_PROPERTY_ID__AUTO_DISABLE__IS_ENABLED`
0 = event is not automatically disabled

1 = event is automatically disabled (see properties `G_COMMON__EA__EVENT_PROPERTY_ID__AUTO_DISABLE__ABSOLUTE` and `G_COMMON__EA__EVENT_PROPERTY_ID__AUTO_DISABLE__RELATIVE`)

`G_COMMON__EA__EVENT_PROPERTY_ID__AUTO_DISABLE__ABSOLUTE`
Event is disabled automatically when event counter has reached this value and property `G_COMMON__EA__EVENT_PROPERTY_ID__AUTO_DISABLE__IS_ENABLED` is **1**

G_COMMON__EA__EVENT_PROPERTY_ID__AUTO_DISABLE__RELATIVE
Event is disabled automatically when event counter has reached the sum of the actual event counter value and this value. Additionally property G_COMMON__EA__EVENT_PROPERTY_ID__AUTO_DISABLE__IS_ENABLED has to be 1.

reserved1
reserved parameter (must be initialized with 0)

reserved2
reserved parameter (must be initialized with 0)

rsp
Returns response parameters

The structure contains the following elements:

EventHandle
Event handle

EventPropertyId
Event property ID

The following values are possible:

G_COMMON__EA__EVENT_PROPERTY_ID__IS_ENABLED
0 = event is disabled

1 = event is enabled

G_COMMON__EA__EVENT_PROPERTY_ID__IS_INITIALIZED
0 = event is not initialized

1 = event is initialized

G_COMMON__EA__EVENT_PROPERTY_ID__EVENT_COUNTER
Event counter value (is incremented by 1 when event occurred while it was enabled)

G_COMMON__EA__EVENT_PROPERTY_ID__AUTO_DISABLE__IS_ENABLED
0 = event is not automatically disabled

1 = event is automatically disabled (see properties G_COMMON__EA__EVENT_PROPERTY_ID__AUTO_DISABLE__ABSOLUTE and G_COMMON__EA__EVENT_PROPERTY_ID__AUTO_DISABLE__RELATIVE)

G_COMMON__EA__EVENT_PROPERTY_ID__AUTO_DISABLE__ABSOLUTE
Event is disabled automatically when event counter has reached this value and property G_COMMON__EA__EVENT_PROPERTY_ID__AUTO_DISABLE__IS_ENABLED is 1

G_COMMON__EA__EVENT_PROPERTY_ID__AUTO_DISABLE__RELATIVE
Event is disabled automatically when event counter has reached the sum of the actual event counter value and this value. Additionally property G_COMMON__EA__EVENT_PROPERTY_ID__AUTO_DISABLE__IS_ENABLED has to be 1.

reserved1
reserved parameter (must be initialized with 0)

reserved2
reserved parameter (must be initialized with 0)

Value
Returns the property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_EA_AllocateEventByHandle

G_Common_EA_AllocateEventByHandle — Allocate event by handle

Description

Use this command to allocate an event with a defined event handle.

This is especially useful when the event handle is used in previously recorded sequences.

Definition

```
G_Error_t
G_Common_EA_AllocateEventByHandle(
 const G_PortHandle_t  portHandle,
 const G_Common_EA_EventHandle_t  eventHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

`eventHandle`
: Event handle

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_EA_AllocateActionByHandle

G_Common_EA_AllocateActionByHandle — Allocate action by handle

Description

Use this command to allocate an action with a defined action handle.

This is especially useful when the action handle is used in previously recorded sequences.

Definition

```
G_Error_t
G_Common_EA_AllocateActionByHandle(
 const G_PortHandle_t  portHandle,
 const G_Common_EA_ActionHandle_t  actionHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

`actionHandle`
    Action handle

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_EA_SetActionProperty

G_Common_EA_SetActionProperty — Set action property

Description

Use this command to set an action property

Definition

```
G_Error_t
G_Common_EA_SetActionProperty(
 const G_PortHandle_t  portHandle,
 const G_Common_EA_SetActionProperty_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`ActionHandle`
Action handle

`ActionPropertyId`
Action property ID

The following values are possible:

`G_COMMON__EA__ACTION_PROPERTY_ID__IS_INITIALIZED`
0 = action is not initialized

1 = action is initialized

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Value`
Action property value to be set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_EA_GetActionProperty

G_Common_EA_GetActionProperty — Get action property

Description

Use this command to query an action property

Definition

```
G_Error_t
G_Common_EA_GetActionProperty(
 const G_PortHandle_t  portHandle,
 const G_Common_EA_GetActionProperty_Cmd_t * const cmd,
 G_Common_EA_GetActionProperty_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`ActionHandle`
Action handle

`ActionPropertyId`
Action property ID

The following values are possible:

`G_COMMON__EA__ACTION_PROPERTY_ID__IS_INITIALIZED`
0 = action is not initialized

1 = action is initialized

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`ActionHandle`
Action handle

`ActionPropertyId`
Action property ID

The following values are possible:

G_COMMON__EA__ACTION_PROPERTY_ID__IS_INITIALIZED
0 = action is not initialized

1 = action is initialized

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Value`
Returns the property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_EA_SimulateEvent

G_Common_EA_SimulateEvent — Simulate Event

Description

Use this command to trigger an event manually.

Definition

```
G_Error_t
G_Common_EA_SimulateEvent(
 const G_PortHandle_t  portHandle,
 const G_Common_EA_SimulateEvent_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_COMMON__EA__SIMULATE_EVENT__CMD_FLAG__NONE`
No flag is set

`EventHandle`
Event handle

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

4 Ethernet

These commands provide control over parameters of Ethernet devices.

G_Common_Ethernet_DHCP_Mode_Get

G_Common_Ethernet_DHCP_Mode_Get — Query the DHCP mode of the Ethernet device

Description

This command returns the DHCP mode of the Ethernet device.

DHCP stands for **Dynamic Host Configuration Protocol** and provides automatic configuration of the device specific network parameters (e.g. ip address, subnet mask) of a client by a DHCP server.

Definition

```
G_Error_t
G_Common_Ethernet_DHCP_Mode_Get(
 const G_PortHandle_t  portHandle,
 G_Common_Ethernet_DHCP_Mode_t * const mode
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
Returns the DHCP mode of the device

The following values are possible:

`G_COMMON__ETHERNET__DHCP__MODE__DISABLED`
The DHCP mode is disabled and the device specifc default network parameters are used.

`G_COMMON__ETHERNET__DHCP__MODE__ENABLED`
The DHCP mode is enabled and the network parameters can be configured by a DHCP server.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Ethernet_DHCP_Mode_Set

G_Common_Ethernet_DHCP_Mode_Set — Set the DHCP mode of the Ethernet device

Description

This command sets the DHCP mode of the Ethernet device.

DHCP stands for **Dynamic Host Configuration Protocol** and provides automatic configuration of the device specific network parameters (e.g. ip address, subnet mask) of a client by a DHCP server.

Per default, the DHCP mode is not enabled.

The device needs a power-on-reset for applying a changed DHCP mode.

Definition

```
G_Error_t
G_Common_Ethernet_DHCP_Mode_Set(
 const G_PortHandle_t  portHandle,
 const G_Common_Ethernet_DHCP_Mode_t  mode
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
DHCP mode

The following values are possible:

`G_COMMON__ETHERNET__DHCP__MODE__DISABLED`
The DHCP mode is disabled and the device specific default network parameters are used.

`G_COMMON__ETHERNET__DHCP__MODE__ENABLED`
The DHCP mode is enabled and the network parameters can be configured by a DHCP server.

If no DHCP server is present or the configuration failed, the device specific default network parameters are used.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Ethernet_IpAddress_Get

G_Common_Ethernet_IpAddress_Get — Query the ip address of the Ethernet device

Description

This command returns the ip address of the Ethernet device.

The ip address is needed for establishing a connection to the device via Ethernet. It can be dynamically assigned by a DHCP server or be a static address.

For using a dynamically assigned ip address, the DHCP mode of the device has to be enabled with G_Common_Ethernet_DHCP_Mode_Set .

Definition

```
G_Error_t
G_Common_Ethernet_IpAddress_Get(
 const G_PortHandle_t  portHandle,
 u8_t * const ipAddress
);
```

Parameters

`portHandle`
Handle to the communication port

`ipAddress`
Returns the current ip address of the device in an array of 4 bytes

The user has to ensure that the buffer `ipAddress` can hold at least 4 bytes of data.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

In this example, the ip address is queried and printed to the console.

```
u8_t  ipAddress[4];

…

rc =
 G_Common_Ethernet_IpAddress_Get(
  PortHandle1,
  ipAddress
 );

if (rc != G_NO_ERROR) {
 return rc;
}

printf(
 "Ip Address: %u.%u.%u.%u\n",
```

```
 ipAddress[0],
 ipAddress[1],
 ipAddress[2],
 ipAddress[3]
);
```

G_Common_Ethernet_IpAddress_Set

G_Common_Ethernet_IpAddress_Set — Set ip address of the Ethernet device

Description

This command sets the ip address of the Ethernet device.

The ip address is needed for establishing a connection to the device via Ethernet. It can be dynamically assigned by a DHCP server or be a static address.

The device needs a power-on-reset for applying a changed ip address.

If the ip address is changed and the DHCP mode is enabled, the changes will have no effect on the really used ip address, because the DHCP server will assign an ip address dynamically.

Definition

```
G_Error_t
G_Common_Ethernet_IpAddress_Set(
 const G_PortHandle_t  portHandle,
 const u8_t * const ipAddress
);
```

Parameters

`portHandle`
    Handle to the communication port

`ipAddress`
    Ip address to be set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
u8_t  ipAddress[4] = {192, 168, 1, 62};

…

rc =
 G_Common_Ethernet_IpAddress_Set(
  PortHandle1,
  ipAddress
 );

if (rc != G_NO_ERROR) {
 return rc;
}
```

G_Common_Ethernet_TcpPort_Get

G_Common_Ethernet_TcpPort_Get — Query the TCP Port of the device

Description

This command returns the TCP Port of the Ethernet device.

Definition

```
G_Error_t
G_Common_Ethernet_TcpPort_Get(
 const G_PortHandle_t  portHandle,
 u16_t * const tcpPort
);
```

Parameters

`portHandle`
    Handle to the communication port

`tcpPort`
    Returns the TCP Port of the device

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

In this example, the TCP Port is queried and printed to the console.

```
G_PortHandle_t  portHandle,
G_Error_t  rc;
u16_t  tcpPort;

…

rc =
 G_Common_Ethernet_TcpPort_Get(
  portHandle,
  &tcpPort
 );

if (rc != G_NO_ERROR) {
 return rc;
}

printf("TCP Port: %u\n", tcpPort);
```

G_Common_Ethernet_GetNetworkAdapterInfo

G_Common_Ethernet_GetNetworkAdapterInfo — Return information about available network adapters

Description

This command returns the IP address and the address mask for all available network adapters of the system.

Definition

```
G_Error_t
G_Common_Ethernet_GetNetworkAdapterInfo(
 u32_t * const numberOfAdapterInfos,
 G_Ethernet_AdapterInfo_t * const adapterInfo
);
```

Parameters

`numberOfAdapterInfos`
in : Number of adapter information `adapterInfo` can be filled with

Out: Number of available adapter information

`adapterInfo`
Returns IP addresses and address masks for all available network adapters

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

See G_Interface_GetInterfaceList_Ethernet for an example.

G_Common_Ethernet_FtpServer_AutoStart_Enable

G_Common_Ethernet_FtpServer_AutoStart_Enable — Enable auto start of FTP server

Description

Use this command to enable auto start of the FTP server on boot.

Use G_Common_Ethernet_FtpServer_AutoStart_GetState to query the current auto start behavior.

Definition

```
G_Error_t
G_Common_Ethernet_FtpServer_AutoStart_Enable(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Ethernet_FtpServer_AutoStart_Disable

G_Common_Ethernet_FtpServer_AutoStart_Disable — Disable auto start of FTP server

Description

Use this command to disable auto start of the FTP server on boot.

Use G_Common_Ethernet_FtpServer_AutoStart_GetState to query the current auto start behavior.

Definition

```
G_Error_t
G_Common_Ethernet_FtpServer_AutoStart_Disable(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Ethernet_FtpServer_AutoStart_GetState

G_Common_Ethernet_FtpServer_AutoStart_GetState — Get auto start state of FTP server

Description

Use this command to query the auto start state of the FTP server.

Use G_Common_Ethernet_FtpServer_AutoStart_Enable and G_Common_Ethernet_FtpServer_AutoStart_Disable to set or reset the auto start behavior.

Definition

```
G_Error_t
G_Common_Ethernet_FtpServer_AutoStart_GetState(
 const G_PortHandle_t  portHandle,
 u8_t * const autoStartState
);
```

Parameters

`portHandle`
Handle to the communication port

`autoStartState`
Returns FTP server auto start state

0 : FTP server is not started automatically

1 : FTP server is started automatically

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

5 Events

These commands provide control over **G-API** events.

G-API events are used mainly as notification objects for several purposes.

An example of using events in combination with a simple CAN monitor is provided in the **G-API samples directory** in your **G-API** installation directory.

To use a **G-API** event for notification purposes, it has to be allocated with function G_Common_Events_AllocateEvent . After the **G-API** notification object has been connected to an event as the reception of CAN monitor data with function G_Can_Monitor_BufferMode_EnableEvent), the wait function G_Common_Events_WaitForSingleEvent can be called to start waiting until the **G-API** event is signaled or the specified timeout occurred. When the **G-API** event is not needed anymore, it can be disconnected (in our example with function G_Can_Monitor_BufferMode_DisableEvent) and then be deallocated with function G_Common_Events_DeallocateEvent .

G_Common_Events_AllocateEvent

G_Common_Events_AllocateEvent — Allocate **G-API** event

Description

This command allocates a **G-API** event, that can be used for different purposes.

The allocated **G-API** event is specified by its handle which is returned via parameter `eventHandle`. This handle is used for all actions connected with this **G-API** event.

Definition

```
G_Error_t
G_Common_Events_AllocateEvent(
 const G_PortHandle_t  portHandle,
 G_Common_EventHandle_t * const eventHandle
);
```

Parameters

`portHandle`
Handle to the communication port

`eventHandle`
Returns handle to the allocated event

This handle is used for all further actions regarding the allocated event.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Events_DeallocateEvent

G_Common_Events_DeallocateEvent — Deallocate **G-API** event

Description

This command deallocates a **G-API** event.

After the **G-API** event has been deallocated, the event handle becomes invalid and the event can not be used anymore.

Definition

```
G_Error_t
G_Common_Events_DeallocateEvent(
 const G_PortHandle_t  portHandle,
 G_Common_EventHandle_t * const eventHandle
);
```

Parameters

`portHandle`
Handle to the communication port

`eventHandle`
Handle to the allocated event

This handle becomes invalid after the function has returned.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Events_WaitForSingleEvent

G_Common_Events_WaitForSingleEvent — Wait for event to be signaled

Description

This function waits for a single event specified by `eventHandle` to be signaled.

During this process, the calling process is blocked.

After the event is signaled, the event state is reset automatically.

If the event is not signaled within the time specified by `timeout`, the function returns with error code `G_ERROR__DLL__API__EVENT_TIMEOUT`.

Definition

```
G_Error_t
G_Common_Events_WaitForSingleEvent(
 const G_PortHandle_t  portHandle,
 const G_Common_EventHandle_t * const eventHandle,
 const u32_t  timeout
);
```

Parameters

`portHandle`
Handle to the communication port

`eventHandle`
Handle to the event

`timeout`
Timeout in milliseconds
- **0** : The function returns immediately if the event is not signaled.
- **0xFFFFFFFF** : The function will only return if the event is signaled.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
rc =
 G_Common_Events_WaitForSingleEvent(
  portHandle,
  eventHandle,
  5000
 );

switch (rc) {
 case G_NO_ERROR:
  printf("Event was signaled");
  break;

 case G_ERROR__DLL__API__EVENT_TIMEOUT:
  printf("Event was not signaled");
```

```
  break;

 default:
  printf("Error '0x%x'", rc);
  break;
}
```

The G_Common_Events_WaitForSingleEvent function is called with a timeout of 5 seconds. After the function has returned, the return code `rc` is evaluated and a corresponding message is printed.

6 Interface

These functions provide control over **G-API** interface data.

Each device can contain multiple controllers and each controller can contain multiple interfaces.

For accessing interface data, it is recommended to use the **GOEPEL electronic Hardware Explorer**.

These functions are intended to be used for accessing interface data from your application code.

G_Interface_GetInterfaceList

G_Interface_GetInterfaceList — Return a list with all available interfaces

Description

This function returns a list with all available interfaces and the corresponding interface data.

Definition

```
G_Error_t
G_Interface_GetInterfaceList(
 const G_Interface_GetInterfaceList_CmdFlags_t  cmdFlags,
 u32_t * const numberOfInterfaces,
 G_InterfaceData_t * const interfaceData,
 const u32_t  sizeOfInterfaceData
);
```

Parameters

`cmdFlags`
Command Flags

The following values are possible and can be combined:

`G_INTERFACE__GET_INTERFACE_LIST__CMD_FLAG__ZERO`
No flag set

`G_INTERFACE__GET_INTERFACE_LIST__CMD_FLAG__NO_ETHERNET`
Do not scan for Ethernet devices

This will speed up the process of building the interface list.

This flag is recommended if no **GOEPEL electronic** Ethernet devices are connected to the system.

`numberOfInterfaces`

- **in** : Number of interfaces the buffer `interfaceData` can be filled with
- **out** : Number of interfaces the buffer `interfaceData` contains

`interfaceData`
Returns interface data in the following structure:

`InterfaceType`
Interface type (e.g. CAN, LIN, ...)

`ControllerNumber`
Controller number on the device

`InterfaceNumber`
Interface number on the controller

`HostCommunicationType`
Host communication type (e.g. USB, PCI, Ethernet, ...)

`HostInterfaceType`
Host interface type (device type) (e.g. USB3052, smartCAR, ...)

`Serial`
Serial number of the device

`DeviceIndex`
Index (card number) of the device

`IpAddress of the device`
Union with IP address data (only available on Ethernet devices)

The union contains the following elements:

`Byte`
Structure with individual IP address bytes (**B1..B4**)

`All`
The IP address as an unsigned number

The IP address **192.168.1.62** using byte parameters would look like this:

```
IpAddress.Byte.B1 = 192;
IpAddress.Byte.B2 = 168;
IpAddress.Byte.B3 = 1;
IpAddress.Byte.B1 = 62;
```

Using the unsigned number parameter, the same IP address would look like this:

```
IpAddress.All = 0x3e01a8c0;
```

`TcpPort`
TCP port of the device (only available on Ethernet devices)

`reserved1`
Reserved byte

`reserved2`
Reserved byte

`sizeOfInterfaceData`
Size of type `G_InterfaceData_t` in bytes (necessary for compatibility reasons)

An example for building a list with all available interfaces and their names, called **GetInterfaceList** , can be found in the **samples directory** .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Interface_GetInterfaceList_PciPxi

G_Interface_GetInterfaceList_PciPxi — Return a list with all available PCI and PXI interfaces

Description

This function returns a list with all available PCI and PXI interfaces and the corresponding interface data.

Definition

```
G_Error_t
G_Interface_GetInterfaceList_PciPxi(
 u32_t * const numberOfInterfaces,
 G_InterfaceData_t * const interfaceData,
 const u32_t  sizeOfInterfaceData
);
```

Parameters

`numberOfInterfaces`

- **in** : Number of interfaces the buffer `interfaceData` can be filled with
- **out** : Number of interfaces the buffer `interfaceData` contains

`interfaceData`

Returns interface data in the following structure:

`InterfaceType`
Interface type (e.g. CAN, LIN, ...)

`ControllerNumber`
Controller number on the device

`InterfaceNumber`
Interface number on the controller

`HostCommunicationType`
Host communication type (e.g. USB, PCI, Ethernet, ...)

`HostInterfaceType`
Host interface type (device type) (e.g. PXI3052, PCI3060, ...)

`Serial`
Serial number of the device

`DeviceIndex`
Index (card number) of the device

`IpAddress of the device`
Union with IP address data (only available on Ethernet devices)

The union contains the following elements:

`Byte`
Structure with individual IP address bytes (**B1**..**B4**)

`All`
The IP address as an unsigned number

The IP address **192.168.1.62** using byte parameters would look like this:

```
IpAddress.Byte.B1 = 192;
IpAddress.Byte.B2 = 168;
IpAddress.Byte.B3 = 1;
IpAddress.Byte.B1 = 62;
```

Using the unsigned number parameter, the same IP address would look like this:

```
IpAddress.All = 0x3e01a8c0;
```

`TcpPort`
TCP port of the device (only available on Ethernet devices)

`reserved1`
Reserved byte

`reserved2`
Reserved byte

`sizeOfInterfaceData`
Size of type `G_InterfaceData_t` in bytes (necessary for compatibility reasons)

An example for building a list with all available interfaces and their names, called **GetInterfaceList** , can be found in the **samples directory** .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Interface_GetInterfaceList_Usb

G_Interface_GetInterfaceList_Usb — Return a list with all available USB interfaces

Description

This function returns a list with all available USB interfaces and the corresponding interface data.

Definition

```
G_Error_t
G_Interface_GetInterfaceList_Usb(
 u32_t * const numberOfInterfaces,
 G_InterfaceData_t * const interfaceData,
 const u32_t  sizeOfInterfaceData
);
```

Parameters

`numberOfInterfaces`

- **in** : Number of interfaces the buffer `interfaceData` can be filled with
- **out** : Number of interfaces the buffer `interfaceData` contains

`interfaceData`

Returns interface data in the following structure:

`InterfaceType`
Interface type (e.g. CAN, LIN, ...)

`ControllerNumber`
Controller number on the device

`InterfaceNumber`
Interface number on the controller

`HostCommunicationType`
Host communication type (e.g. USB, PCI, Ethernet, ...)

`HostInterfaceType`
Host interface type (device type) (e.g. USB3052, smartCAR, ...)

`Serial`
Serial number of the device

`DeviceIndex`
Index (card number) of the device

`IpAddress of the device`
Union with IP address data (only available on Ethernet devices)

The union contains the following elements:

`Byte`
Structure with individual IP address bytes (**B1**..**B4**)

`All`
The IP address as an unsigned number

The IP address **192.168.1.62** using byte parameters would look like this:

```
IpAddress.Byte.B1 = 192;
IpAddress.Byte.B2 = 168;
IpAddress.Byte.B3 = 1;
IpAddress.Byte.B1 = 62;
```

Using the unsigned number parameter, the same IP address would look like this:

```
IpAddress.All = 0x3e01a8c0;
```

`TcpPort`
TCP port of the device (only available on Ethernet devices)

`reserved1`
Reserved byte

`reserved2`
Reserved byte

`sizeOfInterfaceData`
Size of type `G_InterfaceData_t` in bytes (necessary for compatibility reasons)

An example for building a list with all available interfaces and their names, called **GetInterfaceList**, can be found in the **samples directory**.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Interface_GetInterfaceList_Ethernet

G_Interface_GetInterfaceList_Ethernet — Return a list with all available Ethernet interfaces for the selected Ethernet adapter

Description

This function returns a list with all available Ethernet interfaces and the corresponding interface data.

Definition

```
G_Error_t
G_Interface_GetInterfaceList_Ethernet(
 const G_Interface_GetInterfaceList_Ethernet_CmdFlags_t  cmdFlags,
 u32_t * const numberOfInterfaces,
 G_InterfaceData_t * const interfaceData,
 const u32_t  sizeOfInterfaceData,
 const G_Ethernet_AdapterInfo_t * const adapterInfo
);
```

Parameters

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_INTERFACE__GET_INTERFACE_LIST__ETHERNET__CMD_FLAG__NONE`
No command flag set

`G_INTERFACE__GET_INTERFACE_LIST__ETHERNET__CMD_FLAG__ALL`
All available network adapters will be scanned for Ethernet devices - `adapterInfo` will be disregarded

`numberOfInterfaces`
- **in** : Number of interfaces the buffer `interfaceData` can be filled with
- **out** : Number of interfaces the buffer `interfaceData` contains

`interfaceData`
Returns interface data in the following structure:

`InterfaceType`
Interface type (e.g. CAN, LIN, ...)

`ControllerNumber`
Controller number on the device

`InterfaceNumber`
Interface number on the controller

`HostCommunicationType`
Host communication type (e.g. USB, PCI, Ethernet, ...)

`HostInterfaceType`
Host interface type (device type) (e.g. USB3052, smartCAR, ...)

`Serial`
Serial number of the device

`DeviceIndex`
Index (card number) of the device

`IpAddress of the device`
Union with IP address data (only available on Ethernet devices)

The union contains the following elements:

`Byte`
Structure with individual IP address bytes (**B1..B4**)

`All`
The IP address as an unsigned number

The IP address **192.168.1.62** using byte parameters would look like this:

```
IpAddress.Byte.B1 = 192;
IpAddress.Byte.B2 = 168;
IpAddress.Byte.B3 = 1;
IpAddress.Byte.B1 = 62;
```

Using the unsigned number parameter, the same IP address would look like this:

```
IpAddress.All = 0x3e01a8c0;
```

`TcpPort`
TCP port of the device (only available on Ethernet devices)

`reserved1`
Reserved byte

`reserved2`
Reserved byte

`sizeOfInterfaceData`
Size of type `G_InterfaceData_t` in bytes (necessary for compatibility reasons)

`adapterInfo`
Network adapter info - providing IP address and address mask of the network adapter that will be scanned for Ethernet devices

Example

In this example, a list with all available network adapters is qeried and the first network adapter of this list is scanned for Ethernet devices.

```
G_Ethernet_AdapterInfo_t  adapterInfo[32];
u32_t  numberOfAdapterInfos;
G_InterfaceData_t  interfaceData[128];
u32_t  numberOfInterfaces;
const u32_t  sizeOfInterfaceData = sizeof(interfaceData[0]);
u32_t  i;
G_Error_t  rc;

numberOfAdapterInfos = 32;

rc =
```

```
 G_Common_Ethernet_GetNetworkAdapterInfo(
  &numberOfAdapterInfos,
  adapterInfo
 );

if (rc != G_NO_ERROR) {
 return rc;
}

if (numberOfAdapterInfos == 0) {
 return G_NO_ERROR;
}

numberOfInterfaces = 128;

rc =
 G_Interface_GetInterfaceList_Ethernet(
  G_INTERFACE__GET_INTERFACE_LIST__ETHERNET__CMD_FLAG__NONE,
  &numberOfInterfaces,
  interfaceData,
  sizeOfInterfaceData,
  adapterInfo[0]
 );

if (rc != G_NO_ERROR) {
 return rc;
}

printf("Number of interfaces: %u\n", numberOfInterfaces);

for (i = 0; i < numberOfInterfaces; i++) {
 printf("Interface type: %u\n", interfaceData[i].InterfaceType);
}

return G_NO_ERROR;
```

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Interface_Name_Get

G_Interface_Name_Get — Return interface name

Description

Use this function to query the name of an interface.

Definition

```
G_Error_t
G_Interface_Name_Get(
 const G_InterfaceData_t * const interfaceData,
 u32_t * const length,
 char * const name
);
```

Parameters

`interfaceData`
Interface data for selecting the interface

Can be queried with G_Interface_GetInterfaceList .

`length`
- **in** : Size of the buffer `name` in bytes
- **out** : Size of the name string in bytes

`name`
Returns name of the interface

If no name is assigned to the interface, the function will return with errorcode `G_ERROR__DLL__API__INVALID_INTERFACE`

An example for building a list with all available interfaces and their names, called **GetInterfaceList** , can be found in the **samples directory** .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Interface_Name_Set

G_Interface_Name_Set — Set interface name

Description

Set the logical name for an interface.

Definition

```
G_Error_t
G_Interface_Name_Set(
 const G_InterfaceData_t * const interfaceData,
 const char * const name
);
```

Parameters

`interfaceData`
Interface data for selecting the interface

Can be retrieved with G_Interface_GetInterfaceList .

`name`
String with interface name

The interface name has to start with a letter.

If the name is already assigned to another interface, the function will return with error-code `G_ERROR__DLL__API__OPERATION_NOT_ALLOWED`. In this case you need to change the name of this interface before calling G_Interface_Name_Set again.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

7 Monitor

Common monitor commands

These commands provide access to common monitor functionality.

In normal operation mode, you can only use one monitor instance per interface. If you want to use multiple monitor instances on the same interface, you can switch the current monitor instance with command G_Monitor_InstanceId_Set . All monitor related commands will use the instance ID defined by this command.

The default monitor instance ID is **0**.

G_Monitor_InstanceId_Set

G_Monitor_InstanceId_Set — Set monitor instance ID

Description

Use this command to set the value for the monitor instance ID.

The monitor instance ID is needed when multiple monitor instance on the same interface are needed.

Definition

```
G_Error_t
G_Monitor_InstanceId_Set(
 const G_PortHandle_t  portHandle,
 const u8_t  instanceId
);
```

Parameters

`portHandle`
Handle to the communication port

`instanceId`
Monitor instance ID to be set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Monitor_InstanceId_Get

G_Monitor_InstanceId_Get — Get current monitor instance ID

Description

Use this command to query the value of the current monitor instance ID.

The monitor instance ID is needed when multiple monitor instance on the same interface are needed.

Definition

```
u8_t
G_API
G_Monitor_InstanceId_Get(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

Returns the value of the current monitor instance ID.

8 Xcp - Universal Measurement and Calibration Protocol

These commands provide access to the XCP functionality of an interface.

In this part of the documentation a lot of XCP specific terms and abbreviations are used without further explanation. Please refer to the official XCP specification for further explanations.

All XCP commands require the specification of a **multisession channel**. If you want to use multiple XCP instances on one interface, this can be done by specifying different multisession channels. However in most cases, a single multisession channel will be adequate. The standard multisession channel value is **0**.

At the current state **CAN**, **FlexRay** and **Ethernet** interfaces are supported.

G_Common_Xcp_Reset

G_Common_Xcp_Reset — Reset all XCP Resources

Description

Use this command to reset all interface resources used by XCP.

Definition

```
G_Error_t
G_Common_Xcp_Reset(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Init_Can

G_Common_Xcp_Init_Can — Initialize XCP on CAN

Description

Use this command to initialize XCP on CAN interfaces.

Definition

```
G_Error_t
G_Common_Xcp_Init_Can(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Init_Can_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_COMMON__XCP__INIT__CAN__CMD__FLAG__NONE`
No flag is set

`G_COMMON__XCP__INIT__CAN__CMD__FLAG__MAX_DLC_REQUIRED`
When set: master to slave frames always have DLC = MAX_DLC = **8**

When not set: master to slave frames have a variable DLC

`CanIdMaster`
CAN identifier for TX (master sends to slave)

`CanIdSlave`
CAN identifier for RX (slave sends to master)

`TxTimeout`
Transmission timeout, in microseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Init_FlexRay

G_Common_Xcp_Init_FlexRay — Initialize XCP on FlexRay

Description

Use this command to initialize XCP on FlexRay interfaces.

Definition

```
G_Error_t
G_Common_Xcp_Init_FlexRay(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Init_FlexRay_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmd`
Command parameters

The stucture contains the following elements:

`Flags`
Command Flags

The following values are possible and can be combined:

`G_COMMON__XCP__INIT__FLEXRAY__CMD__FLAG__NONE`
No flag is set

`G_COMMON__XCP__INIT__FLEXRAY__CMD__FLAG__IGNORE_RX_NAX`
When not set: received PDUs with a wrong "Nax" (**N**ote **A**ddress for **X**CP) are ignored

When set: received PDUs with any "Nax" are processed

`TxTimeout`
Transmission timeout, in microseconds

`Nax`
Node address for XCP

`HeaderType`
Header type

The following values are possible:

`G_COMMON__XCP__FLEXRAY__HEADER_TYPE__NAX`
Byte0 = NAX

`G_COMMON__XCP__FLEXRAY__HEADER_TYPE__NAX_FILL`
Byte0 = NAX

Byte1 = FILL

`G_COMMON__XCP__FLEXRAY__HEADER_TYPE__NAX_CTR`
Byte0 = NAX

Byte1 = CTR

`G_COMMON__XCP__FLEXRAY__HEADER_TYPE__NAX_FILL3`
Byte0 = NAX

Byte1 = FILL

Byte2 = FILL

Byte3 = FILL

`G_COMMON__XCP__FLEXRAY__HEADER_TYPE__NAX_FILL2`
Byte0 = NAX

Byte1 = CTR

Byte2 = FILL

Byte3 = FILL

`G_COMMON__XCP__FLEXRAY__HEADER_TYPE__NAX_LEN`
Byte0 = NAX

Byte1 = LEN

`G_COMMON__XCP__FLEXRAY__HEADER_TYPE__NAX_CTR_LEN`
Byte0 = NAX

Byte1 = CTR

Byte2 = LEN

`G_COMMON__XCP__FLEXRAY__HEADER_TYPE__NAX_FILL2_LEN`
Byte0 = NAX

Byte1 = FILL

Byte2 = FILL

Byte3 = LEN

`G_COMMON__XCP__FLEXRAY__HEADER_TYPE__NAX_CTR_FILL_LEN`
Byte0 = NAX

Byte1 = CTR

Byte2 = FILL

Byte3 = LEN

`G_COMMON__XCP__FLEXRAY__HEADER_TYPE__UNKNOWN`
Unknown header type (this is only a placeholder)

`PacketAlignment`
Packet alignment

The following values are possible:

G_COMMON__XCP__FLEXRAY__PACKET_ALIGNMENT__8_BIT
The packets are 8 bit aligned

G_COMMON__XCP__FLEXRAY__PACKET_ALIGNMENT__16_BIT
The packets are 16 bit aligned

G_COMMON__XCP__FLEXRAY__PACKET_ALIGNMENT__32_BIT
The packets are 32 bit aligned

G_COMMON__XCP__FLEXRAY__PACKET_ALIGNMENT__UNKNOWN
The alignment is unknown (this is only a placeholder)

NumberOfPdus
Number of PDUs in Pdus

The maximum number of PDUs is defined by G_COMMON__XCP__INIT__FLEXRAY__PDU_BUFFER_SIZE

reserved1
reserved parameter (must be initialized with **0**)

reserved2
reserved parameter (must be initialized with **0**)

Pdus
Array with PDUs to be configured

The structure contains the following elements:

PduId
PDU ID

PacketType
Packet type

The following values are possible and can be combined:

G_COMMON__XCP__FLEXRAY__PACKET_TYPE__CMD
Packet type **CMD** (Command)

Master: TX

Slave: RX

G_COMMON__XCP__FLEXRAY__PACKET_TYPE__STIM
Packet type **STIM** (Data Stimulation Packet)

Master: TX

Slave: RX

G_COMMON__XCP__FLEXRAY__PACKET_TYPE__RES_ERR
Packet type **RES ERR** (Command Response Error Packet)

Master: RX

Slave: TX

G_COMMON__XCP__FLEXRAY__PACKET_TYPE__EV_SERV
Packet type **EV SERV** (Event Service Request Packet)

Master: RX

Slave: TX

`G_COMMON__XCP__FLEXRAY__PACKET_TYPE__DAQ`
Packet type **DAQ** (Data Acquisition Packet)

Master: RX

Slave: TX

`PduFlags`
PDU flags

The following values are possible and can be combined:

`G_COMMON__XCP__FLEXRAY__PDU_FLAGS__ZERO`
No flag is set

`G_COMMON__XCP__FLEXRAY__PDU_FLAG__TX_WITH_MAX_PDU_LENGTH`
When not set: transmit with variable PDU length

When set: transmit with maximum PDU length

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`reserved4`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Init_Ethernet_Udp4

G_Common_Xcp_Init_Ethernet_Udp4 — Initialize XCP for UDPv4 on Ethernet

Description

Use this command to initialize XCP for UDPv4 on Ethernet interfaces.

Definition

```
G_Error_t
G_Common_Xcp_Init_Ethernet_Udp4(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Init_Ethernet_Udp4_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_COMMON__XCP__INIT__ETHERNET__UDP4__CMD__FLAG__NONE`
No flag is set

`G_COMMON__XCP__INIT__ETHERNET__UDP4__CMD__FLAG__USE_VLAN_ID`
Use VLAN ID

A **V**irtual **L**ocal **A**rea **N**etwork can be configured with G_Ethernet_Node_Vlan_Config.

`VlanId`
VLAN ID

Only used if flag `G_COMMON__XCP__INIT__ETHERNET__UDP4__CMD__FLAG__USE_VLAN_ID` is set.

A **V**irtual **L**ocal **A**rea **N**etwork can be configured with G_Ethernet_Node_Vlan_Config.

`MasterIpAddress`
IP address of network master, in network byte order

`SlaveIpAddress`
IP address of network slave, in network byte order

`MasterPort`
Port of network master

`SlavePort`
Port of network slave

`TxTimeout`
Transmission timeout, in microseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Init_Ethernet_Udp6

G_Common_Xcp_Init_Ethernet_Udp6 — Initialize XCP for UDPv6 on Ethernet

Description

Use this command to initialize XCP for UDPv6 on Ethernet interfaces.

Definition

```
G_Error_t
G_Common_Xcp_Init_Ethernet_Udp6(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Init_Ethernet_Udp6_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_COMMON__XCP__INIT__ETHERNET__UDP6__CMD__FLAG__NONE`
No flag is set

`G_COMMON__XCP__INIT__ETHERNET__UDP6__CMD__FLAG__USE_VLAN_ID`
Use VLAN ID

A **V**irtual **L**ocal **A**rea **N**etwork can be configured with G_Ethernet_Node_Vlan_Config .

`VlanId`
VLAN ID

Only used if flag `G_COMMON__XCP__INIT__ETHERNET__UDP6__CMD__FLAG__USE_VLAN_ID` is set.

A **V**irtual **L**ocal **A**rea **N**etwork can be configured with G_Ethernet_Node_Vlan_Config .

`MasterIpAddress`
IP address of network master, in network byte order

`SlaveIpAddress`
IP address of network slave, in network byte order

`MasterPort`
Port of network master

`SlavePort`
Port of network slave

`TxTimeout`
Transmission timeout, in microseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Init_Ethernet_Tcp4

G_Common_Xcp_Init_Ethernet_Tcp4 — Initialize XCP for TCPv4 on Ethernet

Description

Use this command to initialize XCP for TCPv4 on Ethernet interfaces.

Definition

```
G_Error_t
G_Common_Xcp_Init_Ethernet_Tcp4(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Init_Ethernet_Tcp4_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_COMMON__XCP__INIT__ETHERNET__TCP4__CMD__FLAG__NONE`
No flag is set

`G_COMMON__XCP__INIT__ETHERNET__TCP4__CMD__FLAG__USE_VLAN_ID`
Use VLAN ID

A **V**irtual **L**ocal **A**rea **N**etwork can be configured with G_Ethernet_Node_Vlan_Config.

`VlanId`
VLAN ID

Only used if flag `G_COMMON__XCP__INIT__ETHERNET__TCP4__CMD__FLAG__USE_VLAN_ID` is set.

A **V**irtual **L**ocal **A**rea **N**etwork can be configured with G_Ethernet_Node_Vlan_Config.

`MasterIpAddress`
IP address of network master, in network byte order

`SlaveIpAddress`
IP address of network slave, in network byte order

`MasterPort`
Port of network master

`SlavePort`
Port of network slave

`TxTimeout`
Transmission timeout, in microseconds

`TcpConnectTimeout`
Timeout for TCP connection, in microseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Init_Ethernet_Tcp6

G_Common_Xcp_Init_Ethernet_Tcp6 — Initialize XCP for TCPv6 on Ethernet

Description

Use this command to initialize XCP for TCPv6 on Ethernet interfaces.

Definition

```
G_Error_t
G_Common_Xcp_Init_Ethernet_Tcp6(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Init_Ethernet_Tcp6_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_COMMON__XCP__INIT__ETHERNET__TCP6__CMD__FLAG__NONE`
No flag is set

`G_COMMON__XCP__INIT__ETHERNET__TCP6__CMD__FLAG__USE_VLAN_ID`
Use VLAN ID

A **V**irtual **L**ocal **A**rea **N**etwork can be configured with G_Ethernet_Node_Vlan_Config.

`VlanId`
VLAN ID

Only used if flag `G_COMMON__XCP__INIT__ETHERNET__TCP6__CMD__FLAG__USE_VLAN_ID` is set.

A **V**irtual **L**ocal **A**rea **N**etwork can be configured with G_Ethernet_Node_Vlan_Config.

`MasterIpAddress`
IP address of network master, in network byte order

`SlaveIpAddress`
IP address of network slave, in network byte order

`MasterPort`
Port of network master

`SlavePort`
Port of network slave

`TxTimeout`
Transmission timeout, in microseconds

`TcpConnectTimeout`
Timeout for TCP connection, in microseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_CommandWithResponse

G_Common_Xcp_CommandWithResponse — Send XCP command and wait for response

Description

Use this command to send an XCP command in standard mode and wait for the response.

This command is used for sending a CTO frame with one expected response frame. For DTO frames or frames where more than one response frames are expected, use the corresponding FIFO functions G_Common_Xcp_WriteCtoTxFifo and G_Common_Xcp_WriteDtoTxFifo !

Definition

```
G_Error_t
G_Common_Xcp_CommandWithResponse(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_CommandWithResponse_Cmd_t * const cmd,
 G_Common_Xcp_CommandWithResponse_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with `0`)

`cmd`
Command parameters

The structure contains the following elements:

`RxTimeout`
Receive timeout, in microseconds

`Flags`
Command flags

The following flags are possible and can be combined:

`G_COMMON__XCP__COMMAND_WITH_RESPONSE__CMD__FLAG__NONE`
No flag is set

`NumberOfRetries`
Number of send-retries when no "RES" (response) or "ERR" (error) is received

`Length`
Length of data to be transmitted

`Data`
Data to be transmitted

The maximum number of data bytes is defined by `G_COMMON__XCP__COMMAND__DATA_BUFFER_SIZE`

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_COMMON__XCP__COMMAND_WITH_RESPONSE__RSP_FLAG__NONE`
No flag is set

`G_COMMON__XCP__COMMAND_WITH_RESPONSE__RSP_FLAG__VALID`
This flag is set when `Length` is not zero, `Error` is not zero or flag `G_COMMON__XCP__COMMAND_WITH_RESPONSE__RSP_FLAG__RX_TIMEOUT` is set

`G_COMMON__XCP__COMMAND_WITH_RESPONSE__RSP_FLAG__RX_TIMEOUT`
When not set: no RX timeout occurred

When set: RX timeout occurred

`Error`
Returns XCP error number in case of error

`Length`
in : size of buffer `Data`, allocated by user

Out: size of returned data

`Data`
Returns data

This buffer needs to be allocated by the user! (see Example)

Example

```
G_Common_Xcp_CommandWithResponse_Cmd_t  cmd;
G_Common_Xcp_CommandWithResponse_Rsp_t  rsp;
G_Error_t  rc;

…

cmd.RxTimeout = 123456;
cmd.NumberOfRetries = 2;
cmd.Length = 8;

for (i = 0; i < cmd.Length; i++) {
 cmd.Data[i] = (u8_t) i;
}

cmd.Flags = G_COMMON__XCP__COMMAND_WITH_RESPONSE__CMD__FLAG__NONE;

rsp.Data =
 malloc(
  0xFF
 );

if (rsp.Data == NULL) {
 return -1;
}
```

```
rsp.Length = 0xFF;

rc =
 G_Common_Xcp_CommandWithResponse(
  portHandle,
  0,
  &cmd,
  &rsp
 );
```

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_GetState

G_Common_Xcp_GetState — Get XCP state

Description

Use this command to query the XCP state.

Definition

```
G_Error_t
G_Common_Xcp_GetState(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 G_Common_Xcp_GetState_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_COMMON__XCP__GET_STATE__RSP_FLAG__NONE`
No flag is set

`G_COMMON__XCP__GET_STATE__RSP_FLAG__RESPONSE_AVAILABLE`
A response is available

`G_COMMON__XCP__GET_STATE__RSP_FLAG__CTO_RX_FIFO_NOT_EMPTY`
The CTO RX Fifo is not empty

The FIFO can be queried with G_Common_Xcp_ReadCtoRxFifo

`G_COMMON__XCP__GET_STATE__RSP_FLAG__DTO_RX_FIFO_NOT_EMPTY`
The DTO RX FIFO is not empty

The FIFO can be queried with G_Common_Xcp_ReadDtoRxFifo

`G_COMMON__XCP__GET_STATE__RSP_FLAG__CTO_TX_FIFO_NOT_EMPTY`
The CTO TX FIFO is not empty

`G_COMMON__XCP__GET_STATE__RSP_FLAG__DTO_TX_FIFO_NOT_EMPTY`
The DTO TX FIFO is not empty

`CommandState`
XCP Command state

The following values are possible:

`G_COMMON__XCP__COMMAND__STATE__IDLE`
XCP idle state

`G_COMMON__XCP__COMMAND__STATE__SEND_COMMAND`
A transmission is in progress

`G_COMMON__XCP__COMMAND__STATE__WAIT_RESPONSE`
Waiting for response

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Property_GetById

G_Common_Xcp_Property_GetById — Get XCP property value

Description

Use this command to query an XCP property value.

Definition

```
G_Error_t
G_Common_Xcp_Property_GetById(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Property_GetById_Cmd_t * const cmd,
 G_Common_Xcp_Property_GetById_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmd`
Command parameters

The stucture contains the following elements:

`PropertyId`
Property ID

The following values are possible:

`G_COMMON__XCP__PROPERTY_ID__DTO_RX_FIFO_DEPTH`
The size of the DTO RX FIFO

`G_COMMON__XCP__PROPERTY_ID__CTO_RX_FIFO_DEPTH`
The size of the CTO RX FIFO

`G_COMMON__XCP__PROPERTY_ID__DTO_TX_FIFO_DEPTH`
The size of the DTO TX FIFO

`G_COMMON__XCP__PROPERTY_ID__CTO_TX_FIFO_DEPTH`
The size of the CTO TX FIFO

`G_COMMON__XCP__PROPERTY_ID__UNKNOWN`
Unknown property (this is only a placeholder)

`G_COMMON__XCP__PROPERTY_ID__DTO_RX_FIFO__USE_TIMESTAMP`
Use timestamp on DTO RX FIFO messages.

To query DTO RX FIFO messages with timestamp data, use command G_Common_Xcp_ReadDtoRxFifoTs .

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`PropertyId`
Property ID

The following values are possible:

`G_COMMON__XCP__PROPERTY_ID__DTO_RX_FIFO_DEPTH`
The size of the DTO RX FIFO

`G_COMMON__XCP__PROPERTY_ID__CTO_RX_FIFO_DEPTH`
The size of the CTO RX FIFO

`G_COMMON__XCP__PROPERTY_ID__DTO_TX_FIFO_DEPTH`
The size of the DTO TX FIFO

`G_COMMON__XCP__PROPERTY_ID__CTO_TX_FIFO_DEPTH`
The size of the CTO TX FIFO

`G_COMMON__XCP__PROPERTY_ID__UNKNOWN`
Unknown property (this is only a placeholder)

`G_COMMON__XCP__PROPERTY_ID__DTO_RX_FIFO__USE_TIMESTAMP`
Use timestamp on DTO RX FIFO messages.

To query DTO RX FIFO messages with timestamp data, use command G_Common_Xcp_ReadDtoRxFifoTs .

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Value`
Property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Property_SetById

G_Common_Xcp_Property_SetById — Set XCP property value

Description

Use this command to set an XCP property value.

Definition

```
G_Error_t
G_Common_Xcp_Property_SetById(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Property_SetById_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmd`
Command parameters

The stucture contains the following elements:

`PropertyId`
Property ID

The following values are possible:

`G_COMMON__XCP__PROPERTY_ID__DTO_RX_FIFO_DEPTH`
The size of the DTO RX FIFO

`G_COMMON__XCP__PROPERTY_ID__CTO_RX_FIFO_DEPTH`
The size of the CTO RX FIFO

`G_COMMON__XCP__PROPERTY_ID__DTO_TX_FIFO_DEPTH`
The size of the DTO TX FIFO

`G_COMMON__XCP__PROPERTY_ID__CTO_TX_FIFO_DEPTH`
The size of the CTO TX FIFO

`G_COMMON__XCP__PROPERTY_ID__UNKNOWN`
Unknown property (this is only a placeholder)

`G_COMMON__XCP__PROPERTY_ID__DTO_RX_FIFO__USE_TIMESTAMP`
Use timestamp on DTO RX FIFO messages.

To query DTO RX FIFO messages with timestamp data, use command G_Common_Xcp_ReadDtoRxFifoTs .

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Value`
Property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_ReadDtoRxFifo

G_Common_Xcp_ReadDtoRxFifo — Read DTO RX FIFO

Description

Use this command to get the frames of the DTO RX FIFO.

The DTO RX FIFO stores asynchronously received DTO frames.

If timestamp information is needed, use G_Common_Xcp_ReadDtoRxFifoTs .

Definition

```
G_Error_t
G_Common_Xcp_ReadDtoRxFifo(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 u32_t * const numberOfFrames,
 G_Common_Xcp_ReadDtoRxFifo_Frame_t * const frames
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`numberOfFrames`
in : number of frames that fit into `frames`

Out: number of frames returned

`frames`
Returns received DTO frames from RX FIFO

The structure contains the following elements:

`Flags`
Frame flags

The following values are possible and can be combined:

`G_COMMON__XCP__DTO_RX_FIFO__FRAME__FLAG__NONE`
No flag is set

`G_COMMON__XCP__DTO_RX_FIFO__FRAME__FLAG__FIFO_OVERFLOW`
A FIFO buffer overflow occurred

The size of the DTO RX FIFO can be configured with G_Common_Xcp_Property_SetById and property ID `G_COMMON__XCP__PROPERTY_ID__DTO_RX_FIFO_DEPTH`.

`reserved`
reserved parameter (must be initialized with **0**)

`Length`
Frame length in bytes

`Data`
Frame data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_ReadDtoRxFifoTs

G_Common_Xcp_ReadDtoRxFifoTs — Read DTO RX FIFO with timestamp data

Description

Use this command to get the frames of the DTO RX FIFO including timestamp data.

The DTO RX FIFO stores asynchronously received DTO frames.

In contrast to G_Common_Xcp_ReadDtoRxFifo the returned frames contain timestamp information.

To use this function, you need to activate the reception of timestamped frames with G_Common_Xcp_Property_SetById .

Definition

```
G_Error_t
G_Common_Xcp_ReadDtoRxFifoTs(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 u32_t * const numberOfFrames,
 G_Common_Xcp_ReadDtoRxFifoTs_Frame_t * const frames
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`numberOfFrames`
in : number of frames that fit into `frames`

Out: number of frames returned

`frames`
Returns received DTO frames from RX FIFO

The structure contains the following elements:

`Flags`
Frame flags

The following values are possible and can be combined:

`G_COMMON__XCP__DTO_RX_FIFO_TS__FRAME__FLAG__NONE`
No flag is set

`G_COMMON__XCP__DTO_RX_FIFO_TS__FRAME__FLAG__FIFO_OVERFLOW`
A FIFO buffer overflow occurred

The size of the DTO RX FIFO can be configured with G_Common_Xcp_Property_SetById and property ID `G_COMMON__XCP__PROPERTY_ID__DTO_RX_FIFO_DEPTH`.

`reserved`
reserved parameter (must be initialized with **0**)

`Length`
Frame length in bytes

`Timestamp`
Timestamp data in nanoseconds

`Data`
Frame data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_ReadCtoRxFifo

G_Common_Xcp_ReadCtoRxFifo — Read CTO RX FIFO

Description

Use this command to get the frames of the CTO RX FIFO.

The DTO RX FIFO stores asynchronously received DTO frames.

Definition

```
G_Error_t
G_Common_Xcp_ReadCtoRxFifo(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 u32_t * const numberOfFrames,
 G_Common_Xcp_ReadCtoRxFifo_Frame_t * const frames
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`numberOfFrames`
in : number of frames that fit into `frames`

Out: number of frames returned

`frames`
Returns received CTO frames from RX FIFO

The structure contains the following elements:

`Flags`
Frame flags

The following values are possible and can be combined:

`G_COMMON__XCP__CTO_RX_FIFO__FRAME__FLAG__NONE`
No flag is set

`G_COMMON__XCP__CTO_RX_FIFO__FRAME__FLAG__FIFO_OVERFLOW`
A FIFO buffer overflow occurred

The size of the CTO RX FIFO can be configured with G_Common_Xcp_Property_SetById and property ID `G_COMMON__XCP__PROPERTY_ID__CTO_RX_FIFO_DEPTH`.

`reserved`
reserved parameter (must be initialized with **0**)

`Length`
Frame length in bytes

`Data`
Frame data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_WriteDtoTxFifo

G_Common_Xcp_WriteDtoTxFifo — Write frames into DTO TX FIFO

Description

Use this command to send DTO frames via the DTO TX FIFO.

The response frame(s) to the sent DTO frame can be read with G_Common_Xcp_ReadDtoRxFifo .

Definition

```
G_Error_t
G_Common_Xcp_WriteDtoTxFifo(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u32_t  numberOfFrames,
 const G_Common_Xcp_WriteDtoTxFifo_Frame_t * const frames
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`numberOfFrames`
Number of frames to be written

`frames`
Array with frames to be written

The structure contains the following elements:

`Length`
Data length of the frame

The maximum data length of a DTO frame is **65535** bytes.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Data`
Frame data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_WriteCtoTxFifo

G_Common_Xcp_WriteCtoTxFifo — Write frames into CTO TX FIFO

Description

Use this command to send CTO frames via the CTO TX FIFO.

The response frame(s) to the sent CTO frame can be read with G_Common_Xcp_ReadCtoRxFifo .

Definition

```
G_Error_t
G_Common_Xcp_WriteCtoTxFifo(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u32_t  numberOfFrames,
 const G_Common_Xcp_WriteCtoTxFifo_Frame_t * const frames
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`numberOfFrames`
Number of frames to be written

`frames`
Array with frames to be written

The structure contains the following elements:

`Length`
Data length of the frame

The maximum data length of a CTO frame is **255** bytes.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Data`
Frame data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

9 XCP Cmd - XCP commands and command sequences

These commands provide control over common XCP functionality.

To start an XCP session, you need to initialize the selected interface by using G_Common_Xcp_Init_Can or G_Common_Xcp_Init_FlexRay .

In this part of the documentation a lot of XCP specific terms and abbreviations are used without further explanation. Please refer to the official XCP specification for further explanations.

All XCP commands require the specification of a **multisession channel** . If you want to use multiple XCP instances on one interface, this can be done by specifying different multisession channels. However in most cases, a single multisession channel will be adequate. The standard multisession channel value is **0** .

Currently only **CAN** and **FlexRay** interfaces are supported.

G_Common_Xcp_Cmd_Connect

G_Common_Xcp_Cmd_Connect — Set up connection with slave

Description

Use this command to set up an XCP connection with a slave device.

Definition

```
G_Error_t
G_Common_Xcp_Cmd_Connect(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Cmd_Connect_Cmd_t * const cmd,
 G_Common_Xcp_Cmd_Connect_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmd`
Command parameters

The structure contains the following elements:

`Mode`
Connect mode

The following values are possible:

`G_COMMON__XCP__CMD__CONNECT__MODE__NORMAL`
The master can start an XCP communication with the slave.

`G_COMMON__XCP__CMD__CONNECT__MODE__USER_DEFINED`
The master can start an XCP communication with the slave and at the same time tell the slave that it should go into a special (user defined) mode.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_COMMON__XCP__CMD__CONNECT__RSP_FLAG__NONE`
No flag is set

`G_COMMON__XCP__CMD__CONNECT__RSP_FLAG__CAL_PAG`
CALibration and PAGing

0 = calibration/ paging not available

1 = calibration/ paging available

`G_COMMON__XCP__CMD__CONNECT__RSP_FLAG__DAQ`
DAQ lists supported

0 = DAQ lists not available

1 = DAQ lists available

`G_COMMON__XCP__CMD__CONNECT__RSP_FLAG__STIM`
STIMulation

0 = stimulation not available

1 = stimulation available (data stimulation mode of a DAQ list available)

`G_COMMON__XCP__CMD__CONNECT__RSP_FLAG__PGM`
ProGraMming

0 = Flash programming not available

1 = Flash programming available

`G_COMMON__XCP__CMD__CONNECT__RSP_FLAG__SLAVE_BLOCK_MODE`
Indicates whether the Slave Block Mode is available

0 - Slave Block Mode is not available

1 - Slave Block Mode is available

`G_COMMON__XCP__CMD__CONNECT__RSP_FLAG__OPTIONAL`
Indicates whether additional information on supported types of Communication mode is available

0 - additional information on supported types of Communication mode is not available

1 - additional information on supported types of Communication mode is available

`ByteOrder`
Byte order used for transferring multi-byte parameters in an XCP Packet

The following values are possible:

`G_COMMON__XCP__CMD__CONNECT__BYTE_ORDER__INTEL`
Intel format

`G_COMMON__XCP__CMD__CONNECT__BYTE_ORDER__MOTOROLA`
Motorola format (MSB on lower address/position)

`AddressGranularity`
Address granularity

The address granularity indicates the size of an element contained at a single address. It is needed if the master has to do address calculation.

The following values are possible:

`G_COMMON__XCP__CMD__CONNECT__ADDRESS_GRANULARITY__BYTE`
Byte

`G_COMMON__XCP__CMD__CONNECT__ADDRESS_GRANULARITY__WORD`
Word

`G_COMMON__XCP__CMD__CONNECT__ADDRESS_GRANULARITY__DWORD`
DWord

`ProtocolLayerVersion`
XCP Protocol Layer Version Number (most significant byte only)

`TransportLayerVersion`
XCP Transport Layer Version Number (most significant byte only)

`MaxDto`
Maximum DTO size in bytes

`MaxCto`
Maximum CTO size in bytes

`reserved`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Cmd_Disconnect

G_Common_Xcp_Cmd_Disconnect — Disconnect from slave

Description

Use this command to disconnect from an XCP slave.

Definition

```
G_Error_t
G_Common_Xcp_Cmd_Disconnect(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Cmd_GetStatus

G_Common_Xcp_Cmd_GetStatus — Get current session status from slave

Description

Use this command to get the current session status from the XCP slave.

Definition

```
G_Error_t
G_Common_Xcp_Cmd_GetStatus(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 G_Common_Xcp_Cmd_GetStatus_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_COMMON__XCP__CMD__GET_STATUS__RSP_FLAG__NONE`
No flag is set

`G_COMMON__XCP__CMD__GET_STATUS__RSP_FLAG__STORE_CAL_REQ`
REQuest to STORE CALibration data

0 = STORE_CAL_REQ mode is reset

1 = STORE_CAL_REQ mode is set

`G_COMMON__XCP__CMD__GET_STATUS__RSP_FLAG__STORE_DAQ_REQ`
REQuest to STORE DAQ list

0 = STORE_DAQ_REQ mode is reset

1 = STORE_DAQ_REQ mode is set

`G_COMMON__XCP__CMD__GET_STATUS__RSP_FLAG__CLEAR_DAQ_REQ`
REQuest to CLEAR DAQ configuration

0 = CLEAR_DAQ_REQ is reset

1 = CLEAR_DAQ_REQ is set

`G_COMMON__XCP__CMD__GET_STATUS__RSP_FLAG__DAQ_RUNNING`
Data Transfer

0 = Data transfer is not running

1 = Data transfer is running

`G_COMMON__XCP__CMD__GET_STATUS__RSP_FLAG__RESUME`
RESUME Mode

0 = Slave is not in RESUME mode

1 = Slave is in RESUME mode

`G_COMMON__XCP__CMD__GET_STATUS__RSP_FLAG__CAL_PAG_PROTECTED`
CALibration/PAGing commands

0 = CALibration/PAGing commands are not protected with SEED & Key mechanism

1 = CALibration/PAGing commands are protected with SEED & Key mechanism

`G_COMMON__XCP__CMD__GET_STATUS__RSP_FLAG__DAQ_PROTECTED`
DAQ list commands (DIRECTION = DAQ)

0 = DAQ list commands are not protected with SEED & Key mechanism

1 = DAQ list commands are protected with SEED & Key mechanism

`G_COMMON__XCP__CMD__GET_STATUS__RSP_FLAG__STIM_PROTECTED`
DAQ list commands (DIRECTION = STIM)

0 = DAQ list commands are not protected with SEED & Key mechanism

1 = DAQ list commands are protected with SEED & Key mechanism

`G_COMMON__XCP__CMD__GET_STATUS__RSP_FLAG__PGM_PROTECTED`
ProGraMming commands

0 = ProGraMming commands are not protected with SEED & Key mechanism

1 = ProGraMming commands are protected with SEED & Key mechanism

`SessionConfigutationId`
The Session Configuration ID is used for DAQ.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Cmd_GetCommModeInfo

G_Common_Xcp_Cmd_GetCommModeInfo — Get communication mode info

Description

Use this command to get communication mode info from the XCP slave.

Definition

```
G_Error_t
G_Common_Xcp_Cmd_GetCommModeInfo(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 G_Common_Xcp_Cmd_GetCommModeInfo_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_COMMON__XCP__CMD__GET_COMM_MODE_INFO__RSP_FLAG__NONE`
No flag is set

`G_COMMON__XCP__CMD__GET_COMM_MODE_INFO__RSP_FLAG__MASTER_BLOCK_MODE`
Master Block Mode

0 = Master Block Mode not available

1 = Master Block Mode available

`G_COMMON__XCP__CMD__GET_COMM_MODE_INFO__RSP_FLAG__INTERLEAVED_MODE`
Interleaved Mode

0 = Interleaved Mode not available

1 = Interleaved Mode available

`MaxBs`
Indicates the maximum allowed block size as the number of consecutive command packets in a block sequence.

`MinSt`
Indicates the required minimum separation time between the packets of a block transfer from the master device to the slave device in units of 100 microseconds.

`QueueSize`
If Interleaved Mode is available (flag `G_COMMON__XCP__CMD__GET_COMM_MODE_INFO__RSP_FLAG__INTERLEAVED_MODE` set), `QueueSize` indicates the maximum number of consecutive command packets the master can send to the receipt queue of the slave.

`XcpDriverVersionNumber`
The XCP Driver Version Number indicates the version number of the XCP driver in the slave. The major driver version is the high nibble of the version number, the minor driver version is the low nibble.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Cmd_GetSeed

G_Common_Xcp_Cmd_GetSeed — Get seed for unlocking a protected resource

Description

Use this command to get a seed for unlocking a protected resource of the XCP slave.

Definition

```
G_Error_t
G_Common_Xcp_Cmd_GetSeed(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Cmd_GetSeed_Resource_t  resource,
 u32_t * const numberOfSeedBytes,
 u8_t * const seed
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`resource`
Resource whose seed is to be queried

The following values are possible:

`G_COMMON__XCP__CMD__GET_SEED__RESOURCE__CAL_PAG`
CALibration/PAGing commands

`G_COMMON__XCP__CMD__GET_SEED__RESOURCE__DAQ`
DAQ list commands (DIRECTION = DAQ)

`G_COMMON__XCP__CMD__GET_SEED__RESOURCE__STIM`
DAQ list commands (DIRECTION = STIM)

`G_COMMON__XCP__CMD__GET_SEED__RESOURCE__PGM`
ProGraMming commands

`numberOfSeedBytes`
in : size of buffer `seed` in bytes

out : size of returned seed in bytes

`seed`
Returns seed

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Cmd_Unlock

G_Common_Xcp_Cmd_Unlock — Send key for unlocking a protected resource

Description

Use this command to send a key for unlocking a protected resource of an XCP slave.

Definition

```
G_Error_t
G_Common_Xcp_Cmd_Unlock(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u8_t  numberOfBytes,
 const u8_t * const key
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with **0**)

`numberOfBytes`
: Number of key bytes

`key`
: Key bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Cmd_SetMta

G_Common_Xcp_Cmd_SetMta — Set Memory Transfer Address in slave

Description

This command will initialize a pointer (32Bit address + 8Bit extension) for following memory transfer commands.

Definition

```
G_Error_t
G_Common_Xcp_Cmd_SetMta(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Cmd_SetMta_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmd`
Command parameters

The structure contains the following elements:

`Address`
Address

`AddressExtension`
Address Extension

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Cmd_Download

G_Common_Xcp_Cmd_Download — Download data to slave (including SET_MTA, PROGRAM_PREPARE and DOWNLOAD_NEXT)

Description

Use this command to download data from master to slave, including XCP commands SET_MTA, PROGRAM_PREPARE and DOWNLOAD_NEXT.

Definition

```
G_Error_t
G_Common_Xcp_Cmd_Download(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Cmd_Download_Cmd_t * const cmd,
 const u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmd`
Command parameters

The structure contains the following elements:

`Address`
Address

`AddressExtension`
Address extension

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`CodeSize`
Size of data to be downloaded in bytes

`data`
Data to be downloaded

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Cmd_Upload

G_Common_Xcp_Cmd_Upload — Upload data from slave to master (including SET_MTA)

Description

Use this command to upload data from slave to master, including XCP command SET_MTA.

Definition

```
G_Error_t
G_Common_Xcp_Cmd_Upload(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Cmd_Upload_Cmd_t * const cmd,
 u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmd`
Command parameters

The structure contains the following elements:

`Address`
Address

`AddressExtension`
Address extension

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`NumberOfBytes`
Number of bytes to be uploaded

`data`
Returns uploaded data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Cmd_GetPgmProcessorInfo

G_Common_Xcp_Cmd_GetPgmProcessorInfo — Get general information on PGM processor

Description

Use this command to get general information on PGM processor.

Definition

```
G_Error_t
G_Common_Xcp_Cmd_GetPgmProcessorInfo(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 G_Common_Xcp_Cmd_GetPgmProcessorInfo_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_COMMON__XCP__CMD__GET_PGM_PROCESSOR_INFO__RSP_FLAG__NONE`
No flag is set

`G_COMMON__XCP__CMD__GET_PGM_PROCESSOR_INFO__RSP_FLAG__ABSOLUTE_MODE`
Absolute mode supported

`G_COMMON__XCP__CMD__GET_PGM_PROCESSOR_INFO__RSP_FLAG__FUNCTIONAL_MODE`
Functional mode supported

`G_COMMON__XCP__CMD__GET_PGM_PROCESSOR_INFO__RSP_FLAG__COMP_SUPPORTED`
Compression supported

`G_COMMON__XCP__CMD__GET_PGM_PROCESSOR_INFO__RSP_FLAG__COMP_REQUIRED`
Compression required

`G_COMMON__XCP__CMD__GET_PGM_PROCESSOR_INFO__RSP_FLAG__ENC_SUPPORTED`
Encryption supported

`G_COMMON__XCP__CMD__GET_PGM_PROCESSOR_INFO__RSP_FLAG__ENC_REQUIRED`
Encryption required

`G_COMMON__XCP__CMD__GET_PGM_PROCESSOR_INFO__RSP_FLAG__NON_SEQ_PGM_SUPPORTED`
None Sequential Programming supported

`G_COMMON__XCP__CMD__GET_PGM_PROCESSOR_INFO__RSP_FLAG__NON_SEQ_PGM_REQUIRED`
None Sequential Programming required

`MaxSector`
Total number of sectors in the slave device

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Cmd_GetSectorInfo

G_Common_Xcp_Cmd_GetSectorInfo — Get specific information for a sector

Description

Use this command to get specific information for a sector.

Definition

```
G_Error_t
G_Common_Xcp_Cmd_GetSectorInfo(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Cmd_GetSectorInfo_Cmd_t * const cmd,
 G_Common_Xcp_Cmd_GetSectorInfo_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmd`
Command parameters

The structure contains the following elements:

`SectorNumber`
Sector number

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`ClearSequenceNumber`
The Clear Sequence Number and Program Sequence Number describe, in which subsequential order the master has to clear and program flash memory sectors. Each sequence number must be unique. Sectors, which do not have to be programmed, can be skipped in the programming flow control.

`ProgramSequenceNumber`
The Clear Sequence Number and Program Sequence Number describe, in which subsequential order the master has to clear and program flash memory sectors. Each sequence number must be unique. Sectors, which do not have to be programmed, can be skipped in the programming flow control.

`ProgrammingMethod`
Programming method

`SectorNameLength`
Length of the sector name in bytes (without 0 termination), 0 if not available

`SectorStartAddress`
Start address for this sector

`SectorLength`
Length for this sector in bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Cmd_ProgramClear

G_Common_Xcp_Cmd_ProgramClear — Clear a part of non-volatile memory

Description

Use this command to clear a part of non-volatile memory.

Definition

```
G_Error_t
G_Common_Xcp_Cmd_ProgramClear(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Cmd_ProgramClear_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmd`
Command parameters

The structure contains the following elements:

`Mode`
Access mode

The following values are possible:

`G_COMMON__XCP__CMD__PROGRAM_CLEAR__MODE__ABSOLUTE`
The absolute access mode is active (default)

`G_COMMON__XCP__CMD__PROGRAM_CLEAR__MODE__FUNCTIONAL`
The functional access mode is active

`Flags`
Command flags (only used if `Mode` = `G_COMMON__XCP__CMD__PROGRAM_CLEAR__MODE__FUNCTIONAL`

The following values are possible and can be combined:

`G_COMMON__XCP__CMD__PROGRAM_CLEAR__CMD_FLAG__NONE`
No flag is set

`G_COMMON__XCP__CMD__PROGRAM_CLEAR__CMD_FLAG__CLEAR_CALIBRATION_AREA`
Clear all the calibration data area(s)

`G_COMMON__XCP__CMD__PROGRAM_CLEAR__CMD_FLAG__CLEAR_CODE_AREA`
Clear all the code area(s) (the boot area is not covered)

`G_COMMON__XCP__CMD__PROGRAM_CLEAR__CMD_FLAG__CLEAR_NVRAM_AREA`
Clear the NVRAM area(s)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`ClearRange`
The Clear Range indicates the length of the memory part to be cleared.

Only used if `Mode = G_COMMON__XCP__CMD__PROGRAM_CLEAR__MODE__ABSOLUTE`

`ProjectSpecific`
Project specific use-cases

Only used if `Mode = G_COMMON__XCP__CMD__PROGRAM_CLEAR__MODE__FUNCTIONAL`

mask = 0xFFFFFF00

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Cmd_BuildChecksum

G_Common_Xcp_Cmd_BuildChecksum — Build checksum over memory range

Description

Use this command to build a checksum over a memory range.

Definition

```
G_Error_t
G_Common_Xcp_Cmd_BuildChecksum(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Cmd_BuildChecksum_Cmd_t * const cmd,
 G_Common_Xcp_Cmd_BuildChecksum_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmd`
Command paraneters

The structure contains the following elements:

`BlockSize`
Defines the size of the area for building a checksum, in bytes

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`reserved4`
reserved parameter (must be initialized with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`Checksum type`
The following values are possible:

`G_COMMON__XCP__CHECKSUM_TYPE__UNKNOWN`
Unknown checksum type

`G_COMMON__XCP__CHECKSUM_TYPE__XCP_ADD_11`
Add BYTE into a BYTE checksum, ignore overflows

`G_COMMON__XCP__CHECKSUM_TYPE__XCP_ADD_12`
Add BYTE into a WORD checksum, ignore overflows

`G_COMMON__XCP__CHECKSUM_TYPE__XCP_ADD_14`
Add BYTE into a DWORD checksum, ignore overflows

`G_COMMON__XCP__CHECKSUM_TYPE__XCP_ADD_22`
Add WORD into a WORD checksum, ignore overflows, blocksize must be modulo 2

`G_COMMON__XCP__CHECKSUM_TYPE__XCP_ADD_24`
Add WORD into a DWORD checksum, ignore overflows, blocksize must be modulo 2

`G_COMMON__XCP__CHECKSUM_TYPE__XCP_ADD_44`
Add DWORD into DWORD, ignore overflows, blocksize must be modulo 4

`G_COMMON__XCP__CHECKSUM_TYPE__XCP_CRC_16`
CRC 16

`G_COMMON__XCP__CHECKSUM_TYPE__CRC_16_CITT`
CRC 16 CITT

`G_COMMON__XCP__CHECKSUM_TYPE__CRC_32`
CRC 32

`G_COMMON__XCP__CHECKSUM_TYPE__USER_DEFINED`
User defined algorithm, in externally calculated function

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Checksum`
Returns checksum value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Cmd_Program

G_Common_Xcp_Cmd_Program — Non-volatile memory programming (including PROGRAM_START, SET_MTA, PROGRAM_PREPARE and PROGRAM_NEXT)

Description

Use this command for non-volatile memory programming, including XCP commands PROGRAM_START, SET_MTA, PROGRAM_PREPARE and PROGRAM_NEXT.

Definition

```
G_Error_t
G_Common_Xcp_Cmd_Program(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Cmd_Program_Cmd_t * const cmd,
 const u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_COMMON__XCP__CMD__PROGRAM__CMD_FLAG__NONE`
No flag is set

`Address`
Address for non-volatile memory programming

`AddressExtension`
Address extension for non-volatile memory programming

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`CodeSize`
Size of data in bytes

`data`
Data to be programmed

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Cmd_ProgramReset

G_Common_Xcp_Cmd_ProgramReset — Indicate the end of a programming sequence

Description

Use this command to indicate the end of a programming sequence.

Definition

```
G_Error_t
G_Common_Xcp_Cmd_ProgramReset(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

10 Xcp Daq - Xcp Data Acquisition

Use these commands to set up a Xcp Data Acquisition (DAQ).

The following steps are required for setting up a DAQ on Xcp:

- Initialize Xcp on the selected interface with G_Common_Xcp_Init_Can or G_Common_Xcp_Init_FlexRay
- Specify the signals for the DAQ with G_Common_Xcp_Daq_SetSignals
- Start the DAQ with G_Common_Xcp_Daq_Start

Optionally the Xcp DAQ timeouts can be set with G_Common_Xcp_Daq_SetTimeouts and the current DAQ state can be queried with G_Common_Xcp_Daq_GetState .

The DAQ can be stopped with G_Common_Xcp_Daq_Stop and all DAQ resources are freed with G_Common_Xcp_Daq_Reset .

In this part of the documentation a lot of XCP specific terms and abbreviations are used without further explanation. Please refer to the official XCP specification for further explanations.

All XCP commands require the specification of a **multisession channel** . If you want to use multiple XCP instances on one interface, this can be done by specifying different multisession channels. However in most cases, a single multisession channel will be adequate. The standard multisession channel value is **0** .

Currently only **CAN** and **FlexRay** interfaces are supported.

G_Common_Xcp_Daq_GetState

G_Common_Xcp_Daq_GetState — Query DAQ state

Description

Use this command to query the current state of the DAQ.

Definition

```
G_Error_t
G_Common_Xcp_Daq_GetState(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 G_Common_Xcp_Daq_GetState_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_COMMON__XCP__DAQ__GET_STATE__RSP_FLAG__NONE`
No flag is set

`State`
Current DAQ state

The following values are possible:

`G_COMMON__XCP__DAQ__STATE__STOPPED`
The DAQ is not running

`G_COMMON__XCP__DAQ__STATE__RUNNING`
The DAQ is running

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Daq_SetTimeouts

G_Common_Xcp_Daq_SetTimeouts — Set DAQ timeouts

Description

Use this command to set DAQ timeouts.

The values for timeout T1..T7 can be found in the A2L file for the slave device.

Per default all timeouts are set to **100 milliseconds** .

Definition

```
G_Error_t
G_Common_Xcp_Daq_SetTimeouts(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Daq_SetTimeouts_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command Flags

The following values are possible and can be combined:

`G_COMMON__XCP__DAQ__SET_TIMEOUTS__CMD_FLAG__NONE`
No flag is set

`T1`
Timeout T1 in milliseconds

`T2`
Timeout T2 in milliseconds

`T3`
Timeout T3 in milliseconds

`T4`
Timeout T4 in milliseconds

`T5`
Timeout T5 in milliseconds

`T6`
Timeout T6 in milliseconds

T7
: Timeout T7 in milliseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Daq_SetSignals

G_Common_Xcp_Daq_SetSignals — Set DAQ signals

Description

Use this command to specify the signals for a DAQ.

The values for the signal parameters can be found in the **A2L file** of the slave device.

Definition

```
G_Error_t
G_Common_Xcp_Daq_SetSignals(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Daq_SetSignals_CmdFlags_t  cmdFlags,
 const u32_t  numberOfSignals,
 const G_Common_Xcp_Daq_Signal_t * const daqSignals
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_COMMON__XCP__DAQ__SET_SIGNALS__CMD_FLAG__NONE`
No flag is set

`numberOfSignals`
Number of DAQ signals to be set

`daqSignals`
Pointer to array with DAQ signal data

The structure contains the following elements:

`Flags`
Signal flags

The following values are possible and can be combined:

`G_COMMON__XCP__DAQ__SIGNAL__FLAG__NONE`
No flag is set

`BitOffset`
Signal bit offset

Allows definition of signals that represent the status of a single bit, e.g. for **Bit7** the value of `BitOffset` has to be **7**.

If the value for this signal is **0**, the value for the selected bit is **0**. If the value for this signal is **>0**, the value for the selected bit is **1**.

For normal data elements (no single bit values) `BitOffset` has to be **0xFF**.

`DataType`
DAQ signal data type

The following values are possible:

`G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__UBYTE`
Unsigned byte (1 byte)

`G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__SBYTE`
Signed byte (1 byte)

`G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__UWORD`
Unsigned word (2 bytes)

`G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__SWORD`
Signed word (2 bytes)

`G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__ULONG`
Unsigned long (4 bytes)

`G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__SLONG`
Signed long (4 bytes)

`G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__FLOAT32`
32 bit float (4 bytes)

`AddressExtension`
Address extension of DAQ signal

`reserved1`
reserved parameter (must be initialized with **0**)

`EventChannel`
Event channel number that determines the data transmission timing

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Address`
Signal address

`Handle`
Handle for identifying this particular DAQ signal

This parameter is optional and can be used for quickly assigning signal data returned by G_Common_Xcp_Daq_ReadRxSignals to a specific signal. Its value can be chosen freely by the user.

This parameter is optional.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Daq_Reset

G_Common_Xcp_Daq_Reset — Reset DAQ resources

Description

Use this command to reset all DAQ resources.

Definition

```
G_Error_t
G_Common_Xcp_Daq_Reset(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Daq_Reset_CmdFlags_t  cmdFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_COMMON__XCP__DAQ__RESET__CMD_FLAG__NONE`
No flag is set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Daq_Start

G_Common_Xcp_Daq_Start — Start DAQ

Description

Use this command to start the DAQ.

Definition

```
G_Error_t
G_Common_Xcp_Daq_Start(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Daq_Start_CmdFlags_t  cmdFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_COMMON__XCP__DAQ__START__CMD_FLAG__NONE`
No flag is set

`G_COMMON__XCP__DAQ__START__CMD_FLAG__USE_TIMESTAMP`
Use timestamp information

Flag is not set : Use G_Common_Xcp_Daq_ReadRxSignals to read received signals.

Flag is set : Use G_Common_Xcp_Daq_ReadRxSignalsTs to read received signals.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Daq_Start2

G_Common_Xcp_Daq_Start2 — Start DAQ with extended parameters

Description

Use this command to start the DAQ with extended parameters.

For starting a DAQ with a device that is protected by a **seed & key** mechanism, first call `G_Common_Xcp_Daq_Start2` with command flag `G_COMMON__XCP__DAQ__START_2__CMD_FLAG__UNLOCK` **not** set.

The function will return with error `G_ERROR__DLL__COMMON__XCP__DAQ_NOT_INITIALIZED`, a seed in response parameter `Seed` and response flag `G_COMMON__XCP__DAQ__START_2__RSP_FLAG__SEED_RETURNED` set.

You can now compute the corresponding key and call `G_Common_Xcp_Daq_Start2` again but this time with flag `G_COMMON__XCP__DAQ__START_2__CMD_FLAG__UNLOCK` **set** and the key value provided in command parameter `Key`.

If the DAQ has been started successfully, the function will return with `G_NO_ERROR`.

Definition

```
G_Error_t
G_Common_Xcp_Daq_Start2(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Daq_Start2_Cmd_t * const cmd,
 G_Common_Xcp_Daq_Start2_Rsp_t * const rsp
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with **0**)

`cmd`
: Command parameters

  The structure contains the following elements:

  `Flags`
  : Command flags

    The following values are possible and can be combined:

    `G_COMMON__XCP__DAQ__START_2__CMD_FLAG__NONE`
    : No flag is set

    `G_COMMON__XCP__DAQ__START_2__CMD_FLAG__USE_TIMESTAMP`
    : Use timestamp information

Flag is not set : Use G_Common_Xcp_Daq_ReadRxSignals to read received signals.

Flag is set : Use G_Common_Xcp_Daq_ReadRxSignalsTs to read received signals.

`G_COMMON__XCP__DAQ__START_2__CMD_FLAG__UNLOCK`
Used for unlocking a seed & key protected device (see example below)

`KeyLength`
Length of the key in bytes

(Only regarded when flag `G_COMMON__XCP__DAQ__START_2__CMD_FLAG__UNLOCK` is set)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Key`
Array with key bytes

(Only regarded when flag `G_COMMON__XCP__DAQ__START_2__CMD_FLAG__UNLOCK` is set)

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_COMMON__XCP__DAQ__START_2__RSP_FLAG__NONE`
No flag is set

`G_COMMON__XCP__DAQ__START_2__RSP_FLAG__SEED_RETURNED`
A seed is returned in parameter `Seed`

See example below for an example of seed & key mechanism.

`SeedLength`
Returns length of seed in bytes

(Only valid when flag `G_COMMON__XCP__DAQ__START_2__RSP_FLAG__SEED_RETURNED` is set)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Seed`
Returns seed bytes

(Only valid when flag G_COMMON__XCP__DAQ__START_2__RSP_FLAG__SEED_RETURNED is set)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

In the following example a DAQ is started with a seed & key mechanism.

```
G_Common_Xcp_Daq_Start2_Cmd_t  cmd;
G_Common_Xcp_Daq_Start2_Rsp_t  rsp;
const u8_t  channel = 0;
G_Error_t  rc;

cmd.Flags = G_COMMON__XCP__DAQ__START_2__CMD_FLAG__NONE;
cmd.KeyLength = 0;
cmd.reserved1 = 0;
cmd.reserved2 = 0;
cmd.reserved3 = 0;

rc =
 G_Common_Xcp_Daq_Start2(
  PortHandle,
  channel,
  &cmd,
  &rsp
 );

if (rc != G_NO_ERROR) {
 if ((rsp.Flags & G_COMMON__XCP__DAQ__START_2__RSP_FLAG__SEED_RETURNED) ==
 0) {
  // no seed returned -> an error happened
  return rc;
 }

 // a seed was returned -> compute the corresponding key value
 …

 // in this example the key is "0x11, 0x22, 0x33"
 cmd.Flags |= G_COMMON__XCP__DAQ__START_2__CMD_FLAG__UNLOCK;
 cmd.KeyLength = 3;
 cmd.Key[0] = 0x11;
 cmd.Key[1] = 0x22;
 cmd.Key[2] = 0x33;

 rc =
  G_Common_Xcp_Daq_Start2(
   PortHandle,
   channel,
   &cmd,
   &rsp
  );

 if (rc != G_NO_ERROR) {
  return rc;
 }
```

```
 // the device was unlocked successfully and the DAQ has been started
}

return G_NO_ERROR;
```

G_Common_Xcp_Daq_Stop

G_Common_Xcp_Daq_Stop — Stop DAQ

Description

Use this command to stop the DAQ.

The DAQ configuration (including configured DAQ signals) stays valid.

Definition

```
G_Error_t
G_Common_Xcp_Daq_Stop(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Daq_Stop_CmdFlags_t  cmdFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_COMMON__XCP__DAQ__STOP__CMD_FLAG__NONE`
No flag is set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Daq_ReadRxSignals

G_Common_Xcp_Daq_ReadRxSignals — Read DAQ RX signals

Description

Use this command to read DAQ signal values that were received after the start of the DAQ.

The DAQ functionality uses the **DtoRxFifo** for receiving signal data. It is strongly recommended to not use the **DtoRxFifo** for other tasks while the DAQ is running.

Definition

```
G_Error_t
G_Common_Xcp_Daq_ReadRxSignals(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Daq_ReadRxSignals_CmdFlags_t  cmdFlags,
 G_Common_Xcp_Daq_ReadRxSignals_RspFlags_t * const rspFlags,
 u32_t * const numberOfSignals,
 G_Common_Xcp_Daq_RxSignal_t * const daqSignals
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_COMMON__XCP__DAQ__READ_RX_SIGNALS__CMD_FLAG__NONE`
No flag is set

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_COMMON__XCP__DAQ__READ_RX_SIGNALS__RSP_FLAG__NONE`
No flag is set

`G_COMMON__XCP__DAQ__READ_RX_SIGNALS__RSP_FLAG__FIFO_OVERFLOW`
A FIFO overflow occurred

The DAQ uses the **DtoRxFifo**. If this FIFO is not big enough to store all received signal data, you can increase its size with G_Common_Xcp_Property_SetById using property `G_COMMON__XCP__PROPERTY_ID__DTO_RX_FIFO_DEPTH`.

The default FIFO size is **256**, therefore a total number of 256 DAQ frames can be stored.

`G_COMMON__XCP__DAQ__READ_RX_SIGNALS_TS__RSP_FLAG__FIFO_NOT_EMPTY`
DAQ FIFO not empty

There is still data in the DAQ FIFO. Execute the command again to query this data.

`numberOfSignals`
In : maximum number of signals the buffer `signals` can hold

Out : number of signals returned in buffer `signals`

`daqSignals`
Returns array with signal data of received DAQ signals

The structure contains the following elements:

`Handle`
The signal handle that was defined by the user (see G_Common_Xcp_Daq_SetSignals)

`Address`
Signal Address

`DataType`
DAQ signal data type

The following values are possible:

`G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__UBYTE`
Unsigned byte (1 byte)

`G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__SBYTE`
Signed byte (1 byte)

`G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__UWORD`
Unsigned word (2 bytes)

`G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__SWORD`
Signed word (2 bytes)

`G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__ULONG`
Unsigned long (4 bytes)

`G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__SLONG`
Signed long (4 bytes)

`G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__FLOAT32`
32 bit float (4 bytes)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Value`
DAQ signal value

Depending on the value of `DataType`, this value has to be casted to the correct type (see pseudo code below)

```
switch (signal->DataType) {
 case G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__UBYTE:
```

```
  u8_t  v = *((u8_t *) signal->Value));
  break;

 case G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__SBYTE:
  s8_t  v = *((s8_t *) signal->Value));
  break;

 case G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__UWORD:
  u16_t  v = *((u16_t *) signal->Value));
  break;

 case G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__SWORD:
  s16_t  v = *((s16_t *) signal->Value));
  break;

 case G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__ULONG:
  u32_t  v = *((u32_t *) signal->Value));
  break;

 case G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__SLONG:
  s32_t  v = *((s32_t *) signal->Value));
  break;

 case G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__FLOAT32:
  float32_t  v = *((float32_t *) signal->Value));
  break;
}
```

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Common_Xcp_Daq_ReadRxSignalsTs

G_Common_Xcp_Daq_ReadRxSignalsTs — Read DAQ RX signals with timestamp information

Description

Use this command to read DAQ signal values (with timestamp) that were received after the start of the DAQ.

The DAQ functionality uses the **DtoRxFifo** for receiving signal data. It is strongly recommended to not use the **DtoRxFifo** for other tasks while the DAQ is running.

When using timestamp information you need to start the DAQ with G_Common_Xcp_Daq_Start and command flag `G_COMMON__XCP__DAQ__START__CMD_FLAG__USE_TIMESTAMP` set.

Definition

```
G_Error_t
G_Common_Xcp_Daq_ReadRxSignalsTs(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Common_Xcp_Daq_ReadRxSignalsTs_CmdFlags_t  cmdFlags,
 G_Common_Xcp_Daq_ReadRxSignalsTs_RspFlags_t * const rspFlags,
 u32_t * const numberOfSignals,
 G_Common_Xcp_Daq_RxSignalTs_t * const signals
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_COMMON__XCP__DAQ__READ_RX_SIGNALS_TS__CMD_FLAG__NONE`
No flag is set

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_COMMON__XCP__DAQ__READ_RX_SIGNALS_TS__RSP_FLAG__NONE`
No flag is set

`G_COMMON__XCP__DAQ__READ_RX_SIGNALS_TS__RSP_FLAG__FIFO_OVERFLOW`
A FIFO overflow occurred

The DAQ uses the **DtoRxFifo**. If this FIFO is not big enough to store all received signal data, you can increase its size with G_Common_Xcp_Property_SetById using property `G_COMMON__XCP__PROPERTY_ID__DTO_RX_FIFO_DEPTH`.

The default FIFO size is **256**, therefore a total number of 256 DAQ frames can be stored.

`G_COMMON__XCP__DAQ__READ_RX_SIGNALS_TS__RSP_FLAG__FIFO_NOT_EMPTY`
DAQ FIFO not empty

There is still data in the DAQ FIFO. Execute the command again to query this data.

`numberOfSignals`
In : maximum number of signals the buffer `signals` can hold

Out : number of signals returned in buffer `signals`

`signals`
Returns array with signal data of received DAQ signals

The structure contains the following elements:

`Handle`
The signal handle that was defined by the user (see G_Common_Xcp_Daq_SetSignals)

`Address`
Signal Address

`DataType`
DAQ signal data type

The following values are possible:

`G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__UBYTE`
Unsigned byte (1 byte)

`G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__SBYTE`
Signed byte (1 byte)

`G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__UWORD`
Unsigned word (2 bytes)

`G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__SWORD`
Signed word (2 bytes)

`G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__ULONG`
Unsigned long (4 bytes)

`G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__SLONG`
Signed long (4 bytes)

`G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__FLOAT32`
32 bit float (4 bytes)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Timestamp`
Timestamp information in nanoseconds

`Value`
DAQ signal value

Depending on the value of `DataType`, this value has to be casted to the correct type (see pseudo code below)

```
switch (signal->DataType) {
 case G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__UBYTE:
  u8_t  v = *((u8_t *) signal->Value));
  break;

 case G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__SBYTE:
  s8_t  v = *((s8_t *) signal->Value));
  break;

 case G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__UWORD:
  u16_t  v = *((u16_t *) signal->Value));
  break;

 case G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__SWORD:
  s16_t  v = *((s16_t *) signal->Value));
  break;

 case G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__ULONG:
  u32_t  v = *((u32_t *) signal->Value));
  break;

 case G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__SLONG:
  s32_t  v = *((s32_t *) signal->Value));
  break;

 case G_COMMON__XCP__DAQ__SIGNAL__DATA_TYPE__FLOAT32:
  float32_t  v = *((float32_t *) signal->Value));
  break;
}
```

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

CAN Functions

1 CAN Overview

These **G-API** commands provide access to the CAN hardware functionality.

For using extended identifiers or the **CAN FD** functionality of the interface, you have to call G_Can_InitInterface with the command flags set accordingly.

When sending frames with **Extended Identifiers**, bit **31** (**0x80000000**) has to be set in the frame ID value.

When sending **CAN FD** frames, bit **30** (**0x40000000**) has to be set in the frame ID value.

Some commands use a `Dlc` parameter which stands for **Data Length Code** and specifies the length of a CAN message. The following table shows the relation between **Data Length Code** and **Data Length** in bytes.

DLC	Data Length (bytes)
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	12
10	16
11	20
12	24
13	32
14	48
15	64

Table 1 DLC to data length

2 CCP - Can Calibration Protocol

These commands are used to access the CCP functionality of the CAN interface.

Some parts of this description are taken from the official CCP specification which can be found at http://www.asam.net.

CCP stands for Can Calibration Protocol which is part of the ASAP standards.

It defines the communication of controllers with a master device using the CAN 2.0B (11-bit and 29-bit identifier), which includes 2.0A (11-bit identifier) for

1. data acquisition from the controllers,

2. memory transfers to and control functions in the controllers for calibration.

Providing these functions the CCP may be used in the areas of

- development of electronic control units (ECU),
- systems for functional and environmental tests of an ECU,
- test systems and test stands for the controlled devices (combustion engines, gearboxes, suspension systems, climatic control systems, body systems, anti-locking systems),
- on-board test and measurement systems of pre-series vehicles, and
- any non-automotive application of CAN-based distributed electronic control systems.

All CCP commands require the specification of a **multisession channel**. If you want to use multiple CCP instances on one CAN interface, this can be realised by specifying different multisession channels. However in most cases, a single multisession channel will be adequate. The standard multisession channel value is **0**.

Definitions and Abbreviations

Definitions and Abbreviations — The following descriptions and abbreviations will be used in the documentation of the CCP **G-API** commands.

CCP

Can Calibration protocol

CRO

Command Receive Object: message sent from the master device to the slave device(s)

CRM

Command Return Message : one type of message sent from the slave device to the master device containing command / error code and command counter

DAQ

Data Acquisition : definition of a procedure and messages sent from the slave device to the master device for fast data acquisition from an ECU

DTO

Data Transmission Object : message sent from the slave device to the master device (Command Return Message or Event Message or Data Acquisition Message)

ECU

Electronic Control Unit : an electronical device with a central processing unit performing programmed functions with its peripheral circuitry

Message Object

A message transmitted on the CAN, which is sent from a transmitting ECU to any receiving / listening ECU. The coding of the data contained in the message is "known" by the receiving ECUs. A message object's data can be 0 to 8 byte.

Message Frame

Message Frame is a synonymous name for Message Object in the recent literature on CAN

Master and slave devices

A set of controllers connected via CAN exchange Message Objects. An additional, 'external' controller, connected to the network, which communicates with one or more of these controllers and issues commands to them, here is called a master device (host). The controllers in the existing network receiving commands are called 'slave devices'.

ODT

Object Descriptor Table : list of elements (variables), used for organisation of data acquisition

G_Can_Ccp_Reset

G_Can_Ccp_Reset — Reset CCP configuration

Description

This command is used to reset the CCP configuration and free all CCP-related recources.

Definition

```
G_Error_t
G_Can_Ccp_Reset(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Init

G_Can_Ccp_Init — Initialize CCP

Description

Use this command to initialize the CCP for the selected interface.

This needs to be done prior to any other CCP command.

Definition

```
G_Error_t
G_Can_Ccp_Init(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Can_Ccp_Init_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`parameters`
Structure with command parameters

The structure contains the following elements:

`CmdFlags`
Command flags

The following values are possible and can be combined:

`G_CAN__CCP__INIT__CMD_FLAG__NONE`
No flag is set

`G_CAN__CCP__INIT__CMD_FLAG__BIG_ENDIAN`
The data on the bus will be transmitted in **big endian (Motorola)** format

`CroId`
CAN identifier of the **Command Receive Object** (CRO)

`DtoId`
CAN identifier of the **Data Transmission Object** (DTO)

`TxTimeout`
Timeout for the CRO to be transmitted, in microseconds

Depending on factors like the bus load, the transmission of a CRO can be delayed. When it was not transmitted within `TxTimeout` microseconds, the transmission is aborted.

`ResponseFifoDepth`
Maximum number of CRMs (Command Return Messages) that can be buffered

When a CRM is received, it is stored in an internal buffer. By calling G_Can_Ccp_GetResponse the first received message is returned and removed from the buffer.

When using G_Can_Ccp_CommandWithResponse , this is done automatically after a CRO has been sent.

If the internal buffer is full and another message is received, this message is lost and a buffer overrun is signaled at the former received message.

`DaqMessageFifoDepth`
Maximum number of DAQs that can be buffered

When querying all received DAQ messages with G_Can_Ccp_GetDaqMessages , this value represents the smallest possible buffer size for parameter `numberOfMessages`.

`EventMessageFifoDepth`
Maximum number of event messages that can be buffered

When querying all received event messages with G_Can_Ccp_GetEventMessages , this value represents the smallest possible buffer size for parameter `numberOfMessages`.

`StationAddress`
Slave station address (always in **little endian (Intel)** format)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command

G_Can_Ccp_Command — Send a CRO

Description

Use this command to send a CRO (Command Receive Object) to the slave device(s)

For sending a CRO and reading the related CRM automatically, use G_Can_Ccp_CommandWithResponse .

Definition

```
G_Error_t
G_Can_Ccp_Command(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Can_Ccp_Command_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`parameters`
Structure with command parameters

The structure contains the following elements:

`CmdFlags`
Command flags

The following values are possible and can be combined:

`G_CAN__CCP__COMMAND__CMD_FLAG__NONE`
No flag is set

`G_CAN__CCP__COMMAND__CMD_FLAG__AUTOMATIC_CTR`
The command counter is increased automatically with each CRO sent

`AckTimeout`
Acknowledge timeout for CRO, in microseconds

The slave has to respond to the CRO within `AckTimeout` microseconds, otherwise transmission will be repeated `NumberOfRetries` times.

`NumberOfRetries`
Number of retries for sending the CRO

If the slave does not respond to the CRO after it has been sent `NumberOfRetries` times, a response timeout error will occur.

`reserved1`
reserved parameter (must be initialized with `0`)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Data`
Data array with CRO data bytes

A maximum number of **8 bytes** is possible.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_GetCommandState

G_Can_Ccp_GetCommandState — Query the state of a CRO transmission

Description

Use this command to query the state of the current transmission of a CRO.

Definition

```
G_Error_t
G_Can_Ccp_GetCommandState(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 G_Can_Ccp_GetCommandState_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`rsp`
Returns structure with response data

The structure contains the following elements:

`Flags`
Command state flags

The following values are possible and can be combined:

`G_CAN__CCP__GET_COMMAND_STATE__RSP_FLAG__NONE`
No flag is set

`G_CAN__CCP__GET_COMMAND_STATE__RSP_FLAG__BUSY`
The transmission has not yet been finished or acknowledged

`G_CAN__CCP__GET_COMMAND_STATE__RSP_FLAG__TX_IN_PROGRESS`
The transmission is currently in progress

`G_CAN__CCP__GET_COMMAND_STATE__RSP_FLAG__WAITING_FOR_ACK`
Currently waiting for an acknowledge

`CcpError`
Returns the **CCP error** value

The following values are possible:

`G_CAN__CCP__ERROR__NO_ERROR`
No error

`G_CAN__CCP__ERROR__UNKNOWN`
Unknown CCP error

`G_CAN__CCP__ERROR__NEGATIVE_ACK`
A negative acknowledge was received

`G_CAN__CCP__ERROR__INVALID_DAQ_LIST_NUMBER`
The DAQ list number is invalid

`G_CAN__CCP__ERROR__RESPONSE_TIMEOUT`
A response timeout occurred

`G_CAN__CCP__ERROR__TRANSMIT_TIMEOUT`
A transmit timeout occurred

`G_CAN__CCP__ERROR__UNKNOWN_STATE`
The protocol is in an unknown state

`G_CAN__CCP__ERROR__INTERNAL`
An internal error occurred

`NumberOfRetries`
Number of retries for sending the CRO

`RetryCounter`
Number of times the sending of the CRO has already been repeated

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_GetResponse

G_Can_Ccp_GetResponse — Query a CRM

Description

A Command Receive Object **CRO** is sent from the master device to one of the slave devices. The slave device answers with a Data Transmission Object **DTO** containing a Command Return Message **CRM**.

Use this command to query the last received CRM.

For sending a CRO and reading the related CRM automatically, use G_Can_Ccp_CommandWithResponse .

Definition

```
G_Error_t
G_Can_Ccp_GetResponse(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 G_Can_Ccp_GetResponse_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`rsp`
Returns structure with response data

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_CAN__CCP__GET_RESPONSE__RSP_FLAG__NONE`
No flag is set

`G_CAN__CCP__GET_RESPONSE__RSP_FLAG__BUSY`
A reception of a CRM is currently in progress

`G_CAN__CCP__GET_RESPONSE__RSP_FLAG__VALID`
The returned CRM is valid

`G_CAN__CCP__GET_RESPONSE__RSP_FLAG__BUFFER_OVERRUN`
An overrun of the internal buffer for responses (CRMs) occurred

The buffer was not big enough to handle all received CRM. Data was lost.

`G_CAN__CCP__GET_RESPONSE__RSP_FLAG__BUFFER_NOT_EMPTY`
The internal buffer is not empty

Call G_Can_Ccp_GetResponse subsequently to get the remaining CRMs.

`G_CAN__CCP__GET_RESPONSE__RSP_FLAG__UNEXPECTED_CTR`
The command counter of this CRM does not match the expected command counter for the sent CRO

`CcpError`
CCP error value

CCP errors specify protocol based errors. A description for a CCP error code can be retrieved with G_Can_Ccp_GetCcpErrorDescription .

`Dlc`
Data length code of the CRM (see DLC to data length)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Data`
Data bytes of the CRM

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_GetEventMessages

G_Can_Ccp_GetEventMessages — Query event messages

Description

Use this command to query all event messages from the internal event message buffer.

Event messages are used to report internal slave status changes in order to invoke error recovery or other services.

Definition

```
G_Error_t
G_Can_Ccp_GetEventMessages(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 G_Can_Ccp_GetEventMessages_RspFlags_t * const rspFlags,
 u32_t * const numberOfMessages,
 G_Can_Ccp_EventMessage_t * const messages
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_CAN__CCP__GET_EVENT_MESSAGES__RSP_FLAG__NONE`
No flag is set

`G_CAN__CCP__GET_EVENT_MESSAGES__RSP_FLAG__BUFFER_OVERRUN`
An overrun of the internal event buffer occurred - data was lost

`numberOfMessages`

- **in** : Size of buffer `messages`, in number of messages

  The minimum size you may provide equals the value of `EventMessageFifoDepth` of G_Can_Ccp_Init .

- **out** : Number of returned messages in `messages`

`messages`
Returns array with event messages

An event message contains the following elements:

`Dlc`
Data length code (see DLC to data length)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Data`
Event message data bytes (8)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

In this example, the CCP is initialized and event messages are queried.

```
G_Can_Ccp_Init_Parameters_t  param;
G_Can_Ccp_GetEventMessages_RspFlags_t  rspFlags;
u32_t  numberOfMessages;
G_Can_Ccp_EventMessage_t  messages[100];
u32_t  i;
u32_t  j;
G_Error_t  rc;

// init CCP
memset(&param, 0, sizeof(param));

param.CroId = 0x123;
param.DtoId = 0x666;
param.StationAddress = 0x200;
param.TxTimeout = 111111;
param.ResponseFifoDepth = 100;
param.DaqMessageFifoDepth = 100;
param.EventMessageFifoDepth = 100;

rc =
 G_Can_Ccp_Init(
  PortHandle,
  0,
  &param
 );

ErrorHandler(rc);

// CCP communication is taking place and event messages are received
…

// query event messages
numberOfMessages = 100;

rc =
 G_Can_Ccp_GetEventMessages(
  PortHandle,
  channel,
  &rspFlags,
  &numberOfMessages,
  messages
 );

ErrorHandler(rc);

// print received messages
for (i = 0; i < numberOfMessages; i++) {
 printf("\n------------------------\n");
 printf("Dlc: %u\n", messages[i].Dlc);
 printf("Data: ");

 for (j = 0; j < messages[i].Dlc; j++) {
  printf(" 0x%02x", messages[i].Data[j]);
```

```
  }
}
```

G_Can_Ccp_GetDaqMessages

G_Can_Ccp_GetDaqMessages — Query DAQ messages

Description

Use this command to query all DAQ messages from the internal DAQ message buffer.

Definition

```
G_Error_t
G_Can_Ccp_GetDaqMessages(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 G_Can_Ccp_GetDaqMessages_RspFlags_t * const rspFlags,
 u32_t * const numberOfMessages,
 G_Can_Ccp_DaqMessage_t * const messages
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_CAN__CCP__GET_DAQ_MESSAGES__RSP_FLAG__NONE`
No flag is set

`G_CAN__CCP__GET_DAQ_MESSAGES__RSP_FLAG__BUFFER_OVERRUN`
An overrun of the internal DAQ buffer occurred - data was lost

`numberOfMessages`

- **In** : Size of buffer `messages`, in number of messages

  The minimum size you may provide equals the value of `DaqMessageFifoDepth` of G_Can_Ccp_Init .

- **Out** : Number of returned messages in `messages`

`messages`
Returns array with DAQ messages

A DAQ message contains the following elements:

`Dlc`
Data length code (see DLC to data length)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Data`
DAQ message data bytes (8)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

In this example, the CCP is initialized and DAQ messages are queried.

```
G_Can_Ccp_Init_Parameters_t  param;
G_Can_Ccp_GetDaqMessages_RspFlags_t  rspFlags;
u32_t  numberOfMessages;
G_Can_Ccp_DaqtMessage_t  messages[100];
u32_t  i;
u32_t  j;
G_Error_t  rc;

// init CCP
memset(&param, 0, sizeof(param));

param.CroId = 0x123;
param.DtoId = 0x666;
param.StationAddress = 0x200;
param.TxTimeout = 111111;
param.ResponseFifoDepth = 100;
param.DaqMessageFifoDepth = 100;
param.EventMessageFifoDepth = 100;

rc =
 G_Can_Ccp_Init(
  PortHandle,
  0,
  &param
 );

ErrorHandler(rc);

// CCP communication is taking place and DAQ messages are received
…

// query DAQ messages
numberOfMessages = 100;

rc =
 G_Can_Ccp_GetDaqMessages(
  PortHandle,
  channel,
  &rspFlags,
  &numberOfMessages,
  messages
 );

ErrorHandler(rc);

// print received messages
for (i = 0; i < numberOfMessages; i++) {
 printf("\n------------------------\n");
 printf("Dlc: %u\n", messages[i].Dlc);
 printf("Data: ");

 for (j = 0; j < messages[i].Dlc; j++) {
  printf(" 0x%02x", messages[i].Data[j]);
```

```
  }
}
```

G_Can_Ccp_CommandWithResponse

G_Can_Ccp_CommandWithResponse — Send a CRO and wait for the related CRM

Description

Use this command to send a CRO (Command Receive Object) and wait for the related CRM (Command Return Message).

This command is a high level version of G_Can_Ccp_Command and G_Can_Ccp_GetResponse . After G_Can_Ccp_Command has been executed successfully, G_Can_Ccp_GetResponse is called as long as the protocol is busy or the buffer is not empty. This means that the returned CRM is the CRM that was received last.

Definition

```
G_Error_t
G_Can_Ccp_CommandWithResponse(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Can_Ccp_Command_Parameters_t * const parameters,
 G_Can_Ccp_GetResponse_Rsp_t * const rsp
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel

`parameters`
: Structure with command parameters

  These parameters are equal to the command parameters of G_Can_Ccp_Command .

`rsp`
: Returns response parameters

  These parameters are equal to the response parameters of G_Can_Ccp_GetResponse .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

A CRO is sent and the corresponding CRM is returned including CCP error handling.

```
G_Can_Ccp_Command_Parameters_t  param;
G_Can_Ccp_GetResponse_Rsp_t  rsp;
G_Error_t  rc;

memset(&param, 0, sizeof(param));

param.AckTimeout = 20000;
param.NumberOfRetries = 2;
param.Data[0] = 0x01;
param.Data[0] = 0x00;
param.Data[0] = 0x00;
param.Data[0] = 0x02;

rc =
 G_Can_Ccp_CommandWithResponse(
  PortHandle,
  channel,
  &param,
  &rsp
 );

if (rc != G_NO_ERROR) {
 return rc;
}

// print error description
printf("%s\n", G_Can_Ccp_GetCcpErrorDescription(&rsp));
```

G_Can_Ccp_GetCcpErrorDescription

G_Can_Ccp_GetCcpErrorDescription — Query description for CCP errors

Description

Use this command to query a description of a CCP error that was returned by G_Can_Ccp_GetResponse or G_Can_Ccp_CommandWithResponse .

Definition

```
char *
G_Can_Ccp_GetCcpErrorDescription(
 const G_Can_Ccp_GetResponse_Rsp_t * const rsp
);
```

Parameters

`rsp`
: Structure with response data, returned by G_Can_Ccp_GetResponse or G_Can_Ccp_CommandWithResponse

Return Value

String with error description

G_Can_Ccp_Command_Connect

G_Can_Ccp_Command_Connect — Connect

Description

This command establishes a continuous **logical point-to-point connection** with the selected slave station for the master-slave command protocol. All following protocol commands refer to this station only, until another station is selected.

A CONNECT command to another station temporarily disconnects the active station (see DISCONNECT). A CONNECT command to an already connected station is acknowledged. A slave device does not respond to any commands unless being addressed by a prior CONNECT command with the correct station address. The **station address** is specified as a number in little-endian byte order (Intel format, low byte first).

Definition

```
G_Error_t
G_Can_Ccp_Command_Connect(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u16_t  stationAddress

);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel

`stationAddress`
: Station address (Intel format)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_ExchangeId

G_Can_Ccp_Command_ExchangeId — Exchange station identifications

Description

The CCP master and slave stations exchange IDs for automatic session configuration. This might include automatic assignment of a data acquisition setup file depending on the slave's returned identifier (Plug'n'Play).

Definition

```
G_Error_t
G_Can_Ccp_Command_ExchangeId(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u8_t * const masterDeviceId,
 G_Can_Ccp_Command_ExchangeId_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`masterDeviceId`
CCP master device identifier information (optional and implementation specific)

`rsp`
Returns structure with response data

The structure contains the following elements:

`Length`
Length of slave device identifier in bytes

`DataTypeQualifier`
Data type qualifier of slave device identifier (optional and implementation specific)

`ResourceAvailabilityFlags`
Resource Availability flags - returns available resources

The following values are possible and can be combined:

`G_CAN__CCP__RESOURCE_FLAG__NONE`
No flag is set

`G_CAN__CCP__RESOURCE_FLAG__CAL`
Calibration

`G_CAN__CCP__RESOURCE_FLAG__DAQ`
DAQ

`G_CAN__CCP__RESOURCE_FLAG__PGM`
Memory programming

`ResourceProtectionFlags`
Resource Protection flags - returns protection status for available resources (see Resource Flags)

If you want to access a protected resource, it needs to be unlocked with G_Can_Ccp_Command_GetSeed and G_Can_Ccp_Command_Unlock .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_GetSeed

G_Can_Ccp_Command_GetSeed — Get seed for key

Description

Returns "seed" data for a seed&key algorithm that is used for computing the "key" to unlock the requested function for authorized access (see G_Can_Ccp_Command_Unlock below).

Definition

```
G_Error_t
G_Can_Ccp_Command_GetSeed(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u8_t  function,
 G_Can_Ccp_Command_GetSeed_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`function`
Requested slave resource or function (see Resource Flags)

Only one resource or function may be requested. If more than one resource is requested, the command together with the following G_Can_Ccp_Command_Unlock command has to be performed multiple times.

`rsp`
Returns structure with repsonse data

The structure contains the following elements:

`ProtectionStatus`
Protection status (1 = TRUE, 0 = FALSE)

If Protection status = FALSE, it is not required to unlock the requested function.

`SeedData`
Seed data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_Unlock

G_Can_Ccp_Command_Unlock — Unlock protection

Description

Unlocks the slave device security protection (if applicable) using a "key" computed from "seed". See G_Can_Ccp_Command_GetSeed above.

Definition

```
G_Error_t
G_Can_Ccp_Command_Unlock(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u8_t * const key,
 G_Can_Ccp_ResourceFlags_t * const privilegeStatus
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel

`key`
: Key value computed from `SeedData` of command G_Can_Ccp_Command_GetSeed

`privilegeStatus`
: Returns current privilege status (updated resource flags after unlock operation (see Resource Flags))

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

For unlocking the **Calibration** function of an ECU, a seed is requested, the key is computed and the function is unlocked.

```
G_Can_Ccp_Command_GetSeed_Rsp_t  rsp;
G_Error_t  rc;

// request seed
rc =
 G_Can_Ccp_Command_GetSeed(
  PortHandle,
  0,
  G_CAN__CCP__RESOURCE_FLAG__CAL,
  &rsp
 );

if (rc != G_NO_ERROR) {
 return rc;
}

if (rsp.ProtectionStatus != 0) {
 // the calibration function is protected
 u8_t  key[4];
 u32_t  j;
 G_Can_Ccp_ResourceFlags_t  privilegeStatus;

 // compute key value with top secret algorithm
 for (j = 0; j < 4; j++) {
  key[j] = ~seed[j];
 }

 // unlock calibration
 rc =
  G_Can_Ccp_Command_Unlock(
   PortHandle,
   0,
   key,
   &privilegeStatus
  );

 if (rc != G_NO_ERROR) {
  return rc;
 }

 assert(privilegeStatus & G_CAN__CCP__RESOURCE_FLAG__CAL);
}
```

G_Can_Ccp_Command_SetMta

G_Can_Ccp_Command_SetMta — Set Memory Transfer Address

Description

This command will initialize a base pointer (32Bit + extension) for all following memory transfers. The address extension depends on the slave controller's organization and may identify a switchable memory bank or a memory segment.

The **MTA number** (handle) is used to identify different transfer address locations (pointers).

MTA0 is used by `G_Can_Ccp_Command_Download`, `G_Can_Ccp_Command_Upload`, `G_Can_Ccp_Command_Download6`, `G_Can_Ccp_Command_SelectCalPage`, `G_Can_Ccp_Command_ClearMemory`, `G_Can_Ccp_Command_Program` and `G_Can_Ccp_Command_Program6`. **MTA1** is used by `G_Can_Ccp_Command_Move`.

Definition

```
G_Error_t
G_Can_Ccp_Command_SetMta(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Can_Ccp_MtaNumber_t  mtaNumber,
 const u8_t  addressExtension,
 const u32_t  address
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`mtaNumber`
MTA number for selecting MTA

The following values are possible:

`G_CAN__CCP__MTA_NUMBER__0`
MTA0

`G_CAN__CCP__MTA_NUMBER__1`
MTA1

`addressExtension`
Address extension

`address`
Address

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_Download

G_Can_Ccp_Command_Download — Data download

Description

The data block of the specified length (`dataSize`) contained in the CRO will be copied into memory, starting at the current Memory Transfer Address 0 (MTA0). The MTA0 pointer will be post-incremented by the value of `dataSize`.

Definition

```
G_Error_t
G_Can_Ccp_Command_Download(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u8_t  dataSize,
 const u8_t * const  data,
 G_Can_Ccp_Command_Download_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`dataSize`
Size of data block to follow in bytes

`data`
Data to be transferred (up to 5 bytes)

`rsp`
Returns structure with response data

The structure contains the following elements:

`MTA0Extension`
MTA0 extension (after post-increment)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`MTA0Address`
MTA0 address (after post-increment)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_Download6

G_Can_Ccp_Command_Download6 — Data download 6 bytes

Description

The data block with the fixed length of **6 bytes** contained in the CRO will be copied into memory, starting at the current Memory Transfer Address 0 (MTA0). The MTA0 pointer will be post-incremented by the value **6**.

Definition

```
G_Error_t
G_Can_Ccp_Command_Download6(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u8_t * const  data,
 G_Can_Ccp_Command_Download6_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`data`
Data to be transferred (up to 5 bytes)

`rsp`
Returns structure with response data

The structure contains the following elements:

`MTA0Extension`
MTA0 extension (after post-increment)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`MTA0Address`
MTA0 address (after post-increment)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_Upload

G_Can_Ccp_Command_Upload — Data upload

Description

A data block of the specified length (`size`), starting at current MTA0, will be copied into the corresponding DTO data field. The MTA0 pointer will be post-incremented by the value of `size`.

Definition

```
G_Error_t
G_Can_Ccp_Command_Upload(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u8_t  size,
 u8_t * const  data
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`size`
Size of data block to be uploaded in bytes

`data`
Returns requested data bytes

You have to provide a buffer that can handle `size` number of bytes!

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_ShortUp

G_Can_Ccp_Command_ShortUp — Short upload

Description

A data block of the specified length (`size`), starting at `address` will be copied into the corresponding DTO data field. The MTA0 pointer remains unchanged.

Definition

```
G_Error_t
G_Can_Ccp_Command_ShortUp(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u8_t  size,
 const u8_t  addressExtension,
 const u32_t  address,
 u8_t * const  data
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel

`size`
: Size of data block to be uploaded in bytes (1..5)

`addressExtension`
: Address extension

`address`
: Address

`data`
: Returns requested data bytes

You have to provide a buffer that can handle `size` number of bytes!

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_SelectCalPage

G_Can_Ccp_Command_SelectCalPage — Select calibration data page

Description

This command's function depends on the ECU implementation. The previously initialized MTA0 points to the start of the calibration data page that is selected as the currently active page by this command.

Using two blocks of ECU memory for calibration, G_Can_Ccp_Command_SetMta and G_Can_Ccp_Command_SelectCalPage can be used for a kind of "emergency interrupt" from the master device in order to bring the slave system into a "safe state" by preparing the change of these two memory blocks with G_Can_Ccp_Command_SetMta and executing the change immediately with G_Can_Ccp_Command_SelectCalPage .

Definition

```
G_Error_t
G_Can_Ccp_Command_SelectCalPage(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_GetDaqSize

G_Can_Ccp_Command_GetDaqSize — Get size of DAQ list

Description

Returns the size of the specified DAQ list as the number of available Object Descriptor Tables (ODTs) and clears the current list. If the specified list number is not available, `DaqListSize` = 0 should be returned. The DAQ list is initialized and data acquisition by this list is stopped.

An individual CAN Identifier may be assigned to a DAQ list to configure multi ECU data acquisition. This feature is optional. If the given identifier isn't possible, an error code is returned. 29 bit CAN identifiers are marked by the most significant bit set.

Definition

```
G_Error_t
G_Can_Ccp_Command_GetDaqSize(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u8_t  daqListNumber,
 const u32_t  dtoId,
 G_Can_Ccp_Command_GetDaqSize_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`daqListNumber`
DAQ list number (0, 1, ...)

`dtoId`
CAN Identifier of DTO dedicated to list number

`rsp`
Returns structure with response data

The structure contains the following elements:

`DaqListSize`
DAQ list size (= number of ODTs in this list)

`FirstPid`
First PID of DAQ list

The PID of a specific ODT of a DAQ list is determined by: **PID = First PID of DAQ list + ODT number**

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_SetDaqPtr

G_Can_Ccp_Command_SetDaqPtr — Set DAQ list pointer

Description

Initializes the DAQ list pointer for a subsequent write to a DAQ list.

After the DAQ list pointer is set, G_Can_Ccp_Command_WriteDaq can be used to set the data elements in the selected ODT.

Definition

```
G_Error_t
G_Can_Ccp_Command_SetDaqPtr(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u8_t  daqListNumber,
 const u8_t  odtNumber,
 const u8_t  elementNumber
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel

`daqListNumber`
: DAQ list number (0, 1, ...)

`odtNumber`
: Object Descriptor Table ODT number (0, 1, ...)

`elementNumber`
: Element number within ODT (0, 1, ...)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_WriteDaq

G_Can_Ccp_Command_WriteDaq — Write DAQ list entry

Description

Writes one entry (description of single DAQ element) to a DAQ list defined by the DAQ list pointer (see G_Can_Ccp_Command_SetDaqPtr). The following DAQ element sizes are defined: 1 byte , 2 bytes (type word), 4 bytes (type long / Float).

An ECU may not support individual address extensions for each element and 2 or 4 byte element sizes. It is the responsibility of the master device to care for the ECU limitations. The limitations may be defined in the slave device description file (e.g. ASAP2).

It is the responsibility of the slave device, that all bytes of a DAQ element are consistent upon transmission.

Definition

```
G_Error_t
G_Can_Ccp_Command_WriteDaq(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Can_Ccp_Command_WriteDaq_Size_t  sizeOfDaqElement,
 const u8_t  addressExtension,
 const u32_t  address
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`sizeOfDaqElement`
Size of DAQ element

The following values are possible:

`G_CAN__CCP__COMMAND__WRITE_DAQ__SIZE__1_BYTE`
Size of DAQ element = 1 byte

`G_CAN__CCP__COMMAND__WRITE_DAQ__SIZE__2_BYTES`
Size of DAQ element = 2 bytes

`G_CAN__CCP__COMMAND__WRITE_DAQ__SIZE__4_BYTES`
Size of DAQ element = 4 bytes

`addressExtension`
Address extension of DAQ element

`address`
Address of DAQ element

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_StartStop

G_Can_Ccp_Command_StartStop — Start / stop data transmission

Description

This command is used to start or to stop the data acquisition or to prepare a synchronized start of the specified DAQ list. The **Last ODT number** specifies which ODTs (From 0 to Last ODT number) of this DAQ list should be transmitted. The **Event Channel No.** specifies the generic signal source that effectively determines the data transmission timing.

To allow reduction of the desired transmission rate, a **prescaler** may be applied to the Event Channel. The prescaler value factor must be greater than or equal to 1.

The ECU specific properties of event channels and DAQ lists may be described in the slave device description (ASAP2).

Definition

```
G_Error_t
G_Can_Ccp_Command_StartStop(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Can_Ccp_Command_StartStop_Mode_t  mode,
 const u8_t  daqListNumber,
 const u8_t  lastOdtNumber,
 const u8_t  eventChannelNumber,
 const u16_t  transmissionRatePrescaler
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel

`mode`
: Transmission mode

  The following values are possible:

  `G_CAN__CCP__COMMAND__START_STOP__MODE__START`
  : The specified DAQ list will be started

  `G_CAN__CCP__COMMAND__START_STOP__MODE__STOP`
  : The specified DAQ list will be stopped

  `G_CAN__CCP__COMMAND__START_STOP__MODE__PREPARE`
  : Prepare specified DAQ list for synchronized start

    The DAQ list is configured with the provided parameters but does not start the data acquisition. This parameter is used for a synchronized start of all configured DAQ lists.

`daqListNumber`
: DAQ list number

`lastOdtNumber`
: Last ODT number

`eventChannelNumber`
Event Channel number

`transmissionRatePrescaler`
Transmission rate prescaler

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_Disconnect

G_Can_Ccp_Command_Disconnect — Disconnect

Description

Disconnects the slave device. The disconnection can be temporary, setting the slave device in an "off line" state or with parameter mode = `G_CAN__CCP__COMMAND__DISCONNECT__MODE__END_OF_SESSION` terminating the calibration session.

Terminating the session invalidates all state information and resets the slave protection status.

A temporary disconnection doesn't stop the transmission of DAQ messages. The MTA values, the DAQ setup, the session status and the protection status are unaffected by the temporary disconnection and remain unchanged.

If the ECU supports the resume feature and the resume bit was set by G_Can_Ccp_Command_SetSessionStatus , the DAQ related functions behave like in a temporary disconnection. The protection status for DAQ remains unlocked.

Definition

```
G_Error_t
G_Can_Ccp_Command_Disconnect(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Can_Ccp_Command_Disconnect_Mode_t  mode,
 const u16_t  stationAddress
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`mode`
Disconnect mode

The following values are possible:

`G_CAN__CCP__COMMAND__DISCONNECT__MODE__TEMPORARY`
Disconnect temporarily

`G_CAN__CCP__COMMAND__DISCONNECT__MODE__END_OF_SESSION`
End of session

`stationAddress`
Station address (Intel format)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_SetSessionStatus

G_Can_Ccp_Command_SetSessionStatus — Set session status

Description

Keeps the slave node informed about the current state of the calibration session.

The session status bits of the slave device are cleared on power-up, on session log-off and in applicable fault conditions.

Definition

```
G_Error_t
G_Can_Ccp_Command_SetSessionStatus(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Can_Ccp_SessionStatusFlags_t  statusFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`statusFlags`
Session status flags to be set

The following values are possible and can be combined:

`G_CAN__CCP__SESSION_STATUS_FLAG__NONE`
No flag is set

`G_CAN__CCP__SESSION_STATUS_FLAG__CAL`
Calibration data initialized

`G_CAN__CCP__SESSION_STATUS_FLAG__DAQ`
DAQ list(s) initialized

`G_CAN__CCP__SESSION_STATUS_FLAG__RESUME`
Request to save DAQ setup during shutdown in ECU

The ECU automatically restarts DAQ after startup.

`G_CAN__CCP__SESSION_STATUS_FLAG__STORE`
Request to save calibration data during shutdown in ECU

`G_CAN__CCP__SESSION_STATUS_FLAG__RUN`
Session in progress

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_GetSessionStatus

G_Can_Ccp_Command_GetSessionStatus — Get session status

Description

Return session status of slave device

Definition

```
G_Error_t
G_Can_Ccp_Command_GetSessionStatus(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 G_Can_Ccp_Command_GetSessionStatus_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`rsp`
Returns structure with response data

The structure contains the following elements:

`StatusFlags`
Session status (see Session Status Flags)

`StatusInfoQualifier`
Additional status information qualifier

The use of **additional status information** is manufacturer and / or project specific, it is not part of this protocol specification. For example, additional status information could contain an incremental checksum result, that keeps track of the current session activities.

If the return information does not contain additional status information, the additional status information qualifier has to be FALSE (0). If the additional status information is not FALSE, it may be used to determine the type of additional status information.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`StatusInfo`
Additional status information (optional)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_BuildChksum

G_Can_Ccp_Command_BuildChksum — Build checksum

Description

Returns a checksum result of the memory block that is defined by MTA0 (memory transfer area start address) and `size`. The checksum algorithm may be manufacturer and / or project specific.

Definition

```
G_Error_t
G_Can_Ccp_Command_BuildChksum(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u32_t  size,
 G_Can_Ccp_Command_BuildChksum_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`size`
Block size in bytes

`rsp`
Returns structure with response data

The structure contains the following elements:

`Size`
Size of checksum data in bytes

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Data`
Checksum data (implementation specific) (up to 4 bytes)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_ClearMemory

G_Can_Ccp_Command_ClearMemory — Clear Memory

Description

This command may be used to erase FLASH EPROM prior to reprogramming. The MTA0 pointer points to the memory location to be erased.

Definition

```
G_Error_t
G_Can_Ccp_Command_ClearMemory(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u32_t  size
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel

`size`
: Size of memory to be erased, in bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_Program

G_Can_Ccp_Command_Program — Program

Description

The data block of the specified length `size` contained in the CRO will be programmed into non-volatile memory (FLASH, EEPROM), starting at current MTA0. The MTA0 pointer will be post-incremented by the value of `size`.

Definition

```
G_Error_t
G_Can_Ccp_Command_Program(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u8_t  size,
 const u8_t * const data,
 G_Can_Ccp_Command_Program_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`size`
Size of data block to follow (bytes)

`data`
Data to be programmed (max. 5 bytes)

`rsp`
Returns structure with response data

The structure contains the following elements:

`MTA0Extension`
MTA0 extension (after post-increment)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`MTA0Address`
MTA0 address (after post-increment)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_Program6

G_Can_Ccp_Command_Program6 — Program 6 bytes

Description

The data block with the length of 6 bytes contained in the CRO will be programmed into non-volatile memory (FLASH, EEPROM), starting at current MTA0. The MTA0 pointer will be post-incremented by 6.

Definition

```
G_Error_t
G_Can_Ccp_Command_Program6(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u8_t * const data,
 G_Can_Ccp_Command_Program6_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`data`
Data to be programmed (6 bytes)

`rsp`
Returns structure with response data

The structure contains the following elements:

`MTA0Extension`
MTA0 extension (after post-increment)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`MTA0Address`
MTA0 address (after post-increment)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_Move

G_Can_Ccp_Command_Move — Move memory block

Description

A data block of the specified length `size` will be copied from the address defined by MTA 0 (source pointer) to the address defined by MTA 1 (destination pointer).

Definition

```
G_Error_t
G_Can_Ccp_Command_Move(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u32_t  size
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel

`size`
: Size (number of bytes) of data block to be moved

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_DiagService

G_Can_Ccp_Command_DiagService — Diagnostic Service

Description

The slave device carries out the requested service and automatically sets the Memory Transfer Address MTA0 to the location from which the CCP master device (host) may subsequently upload the requested diagnostics service return information.

To upload the requested service information, use G_Can_Ccp_Command_Upload

Definition

```
G_Error_t
G_Can_Ccp_Command_DiagService(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u16_t  serviceNumber,
 const u8_t * const parameters,
 G_Can_Ccp_Command_DiagService_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`serviceNumber`
Diagnostic service number

`parameters`
Optional parameters (up to 4 bytes)

`rsp`
Returns structure with response data

The structure contains the following elements:

`Length`
Length of return information in bytes

`DataType`
Data type qualifier of return information

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_ActionService

G_Can_Ccp_Command_ActionService — Action service

Description

The slave device carries out the requested service and automatically sets the Memory Transfer Address MTA0 to the location from which the CCP master device may subsequently upload the requested action service return information (if applicable).

To upload the requested service information, use G_Can_Ccp_Command_Upload

Definition

```
G_Error_t
G_Can_Ccp_Command_ActionService(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u16_t  serviceNumber,
 const u8_t * const parameters,
 G_Can_Ccp_Command_ActionService_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`serviceNumber`
Action service number

`parameters`
Optional parameters (up to 4 bytes)

`rsp`
Returns structure with response data

The structure contains the following elements:

`Length`
Length of return information in bytes

`DataType`
Data type qualifier of return information

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_Test

G_Can_Ccp_Command_Test — Test availability

Description

This command is used to test whether the slave with the specified station address is available for CCP communication or not. This command does not establish a logical connection nor does it trigger any activities in the specified slave station.

If the slave station is not available for CCP communication, the command will return with an error.

Definition

```
G_Error_t
G_Can_Ccp_Command_Test(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u16_t  stationAddress
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel

`stationAddress`
: Station address (Intel format)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_StartStopAll

G_Can_Ccp_Command_StartStopAll — Start / stop synchronized data transmission

Description

This command is used to start the periodic transmission of all DAQ lists configured with a previously sent G_Can_Ccp_Command_StartStop (`modus = G_CAN__CCP__COMMAND__START_STOP__MODE__PREPARE`) as "prepared to start" in a synchronized manner. The command is also used to stop the periodic transmission of **all** DAQ lists, including those not started synchronized.

Definition

```
G_Error_t
G_Can_Ccp_Command_StartStopAll(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Can_Ccp_Command_StartStopAll_Mode_t  mode
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`mode`
Transmission mode

The following values are possible:

`G_CAN__CCP__COMMAND__START_STOP_ALL__MODE__STOP`
Stops data transmission

`G_CAN__CCP__COMMAND__START_STOP_ALL__MODE__START`
Starts data transmission

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_GetActiveCalPage

G_Can_Ccp_Command_GetActiveCalPage — Get currently active Calibration Page

Description

This command returns the start address of the calibration page that is currently active in the slave device.

Definition

```
G_Error_t
G_Can_Ccp_Command_GetActiveCalPage(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 G_Can_Ccp_Command_GetActiveCalPage_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`rsp`
Returns structure with response parameters

The structure contains the following parameters:

`AddressExtension`
Address extension

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Address`
Address of the calibration page that is currently active in the slave device

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Ccp_Command_GetCcpVersion

G_Can_Ccp_Command_GetCcpVersion — Get implemented version of CCP

Description

This command performs a mutual identification of the protocol version used in the master and in the slave device to agree on a common protocol version. This command is expected to be executed prior to G_Can_Ccp_Command_ExchangeId .

Definition

```
G_Error_t
G_Can_Ccp_Command_GetCcpVersion(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u8_t  desiredMainVersion,
 const u8_t  desiredReleaseVersion,
 G_Can_Ccp_Command_GetCcpVersion_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`desiredMainVersion`
Main Protocol version (desired)

`desiredReleaseVersion`
Release within version (desired)

`rsp`
Returns structure with response data

The structure contains the following elements:

`MainVersion`
Main Protocol version as implemented in the slave device

`ReleaseVersion`
Release number within version number as implemented in the slave device

`reserved1`
reserved parameter (must be initialized with `0`)

`reserved2`
reserved parameter (must be initialized with `0`)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

The master sends G_Can_Ccp_Command_GetCcpVersion to the slave device. The desired main protocol version is **2** and the desired release is **1**, i.e. the desired protocol version is **CCP 2.1**.

```
G_Can_Ccp_Command_GetCcpVersion_Rsp_t  rsp;
G_Error_t  rc;

rc =
 G_Can_Ccp_Command_GetCcpVersion(
  PortHandle,
  0,
  2,
  1,
  &rsp
 );

if (rc != G_NO_ERROR) {
 return rc;
}

printf(
 "Protocol version of the slave device: %u.%u.\n",
 rsp.MainVersion,
 rsp.ReleaseVersion
);
```

3 Common CAN commands

These commands provide control over common CAN properties.

G_Can_InitInterface

G_Can_InitInterface — Interface initialization

Description

This command resets the selected CAN interface without software reset into the initial state.

Definition

```
G_Error_t
G_Can_InitInterface(
 const G_PortHandle_t portHandle,
 const G_Can_IdMode_t idMode,
 const G_Can_InitInterface_CmdFlags_t cmdFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`idMode`
Variable of type **G_Can_IdMode_t**

The following values are available:

`G_CAN__ID_MODE__STANDARD`
11 bit identifiers

`G_CAN__ID_MODE__EXTENDED`
29 bit identifiers

`G_CAN__ID_MODE__MIXED`
11 bit identifiers and 29 bit identifiers

For using extended identifiers or the **CAN FD** functionality of the interface, you have to call G_Can_InitInterface with the command flags set accordingly.

When sending frames with **Extended Identifiers**, bit **31** (**0x80000000**) has to be set in the frame ID value.

When sending **CAN FD** frames, bit **30** (**0x40000000**) has to be set in the frame ID value.

`cmdFlags`
Command flags

The following flags are available and can be combined:

`G_CAN__INIT_INTERFACE__CMD_FLAG__NONE`
No flag is set

`G_CAN__INIT_INTERFACE__CMD_FLAG__ENABLE_ANALYZER`
Analyzer Mode (no transmitting possible)

`G_CAN__INIT_INTERFACE__CMD_FLAG__ENABLE_BLINKING`
Blinking of the LEDs activated

`G_CAN__INIT_INTERFACE__CMD_FLAG__DISABLE_NO_ACK_PAUSES`
No intermission if no acknowledge is received

Use this flag to deactivate the compliance of intermissions if no dominant bit is received in the acknowledge slot of the sent CAN frames.

If this flag is not set, an intermission of 100 milliseconds will be kept after sending a CAN frame with not received acknowledge for 100 milliseconds uninterruptedly.

If this flag is set, the intermissions (waiting phases) will be suppressed. A CAN frame is sent as long as an acknowledge bit (dominant bit in the acknowledge slot) is received for this CAN frame.

`G_CAN__INIT_INTERFACE__CMD_FLAG__FD_MODE__FD`
Receive and transmit frames in **CAN** or **CAN FD** format

`G_CAN__INIT_INTERFACE__CMD_FLAG__FD_MODE__TX_FD`
Receive frames in **CAN** or **CAN FD** format, transmit frames in **CAN FD** format only, baud rate switch disabled

`G_CAN__INIT_INTERFACE__CMD_FLAG__FD_MODE__TX_FD_BRS`
Receive frames in **CAN** or **CAN FD** format, transmit frames in **CAN FD** format only, baud rate switch enabled

`G_CAN__INIT_INTERFACE__CMD_FLAG__FD_MODE__TX_NO_FD`
Receive frames in **CAN** or **CAN FD** format transmit frames in **CAN** format only

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

4 CyclicMsgs

These commands are used to control the cyclic messages, including multiplexed bytes and ramps.

G_Can_CyclicMsgs_DefineMsg

G_Can_CyclicMsgs_DefineMsg — Message definition

Description

This command defines the message indicated by `id`.

Definition

```
G_Error_t
G_Can_CyclicMsgs_DefineMsg(
 const G_PortHandle_t portHandle,
 const u32_t id,
 const u16_t cycleTime,
 const G_Can_CyclicMsgs_SendState_t sendState,
 const u8_t group,
 const u8_t msgCount,
 const u8_t * const data,
 const u8_t dlc
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Identifier of the message to be defined

For using extended identifiers or the **CAN FD** functionality of the interface, you have to call G_Can_InitInterface with the command flags set accordingly.

When sending frames with **Extended Identifiers**, bit **31** (**0x80000000**) has to be set in the frame ID value.

When sending **CAN FD** frames, bit **30** (**0x40000000**) has to be set in the frame ID value.

`cycleTime`
Cycle time in milliseconds

`sendState`
Message send state

The following values are possible:

`G_CAN__CYCLIC_MSGS__SEND_STATE__DONT_SEND_MSG`
Do not send the message

`G_CAN__CYCLIC_MSGS__SEND_STATE__SEND_MSG`
Send the message

`group`
Group ID of the message

In the recent version, only group IDs `0` and `1` are supported.

All messages belonging to the same group can be started at the same time with G_Can_CyclicMsgs_StartGroup and stopped with G_Can_CyclicMsgs_StopGroup .

If the group ID is `0`, the message does not have any group affiliation.

`msgCount`
Number of transmit repetitions of the message
- `0` - Always send the message (unlimited)
- 1..255 - Send the message 1..255 times

`data`
Pointer to the buffer for data bytes

`dlc`
Data length code (see DLC to data length)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_DeleteMsg

G_Can_CyclicMsgs_DeleteMsg — Delete the message

Description

This command deletes the message indicated by `id` and already defined by G_Can_CyclicMsgs_DefineMsg.

Please consider that after calling the command not only the sending of this message indicated by `id` is stopped. Additionally, the message itself is removed from the internal administration. Sending it again is only possible by the command G_Can_CyclicMsgs_DefineMsg .

Definition

```
G_Error_t
G_Can_CyclicMsgs_DeleteMsg(
 const G_PortHandle_t portHandle,
 const u32_t id
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier of the message to be deleted

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_SetMsgBreakCount

G_Can_CyclicMsgs_SetMsgBreakCount — Suspend transmission

Description

Suspend the transmission of the message for a specified number of times.

Definition

```
G_Error_t
G_Can_CyclicMsgs_SetMsgBreakCount(
 const G_PortHandle_t  portHandle,
 const u32_t  id,
 const u16_t  breakCount
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier of the message

`breakCount`
: Number of times the transmission of the message will be suspended

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_SetMsgBreakTime

G_Can_CyclicMsgs_SetMsgBreakTime — Suspend transmission

Description

Suspend the transmission of the message for a specified time.

Definition

```
G_Error_t
G_Can_CyclicMsgs_SetMsgBreakTime(
 const G_PortHandle_t  portHandle,
 const u32_t  id,
 const u16_t  breakTime
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier of the message

`breakTime`
: Time period the transmission of the message will be suspended, in milliseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_SetCalculationBreakCount

G_Can_CyclicMsgs_SetCalculationBreakCount — Suspend calculation of a ramp or a checksum

Description

Use this command to suspend the calculation of a ramp or a checksum.

Definition

```
G_Error_t
G_Can_CyclicMsgs_SetCalculationBreakCount(
 const G_PortHandle_t  portHandle,
 const u32_t  id,
 const G_Common_Signal_t signal,
 const G_Can_CyclicMsgs_SetCalculationBreakCount_Mode_t  mode,
 const u16_t  breakCount
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Identifier of the message

`signal`
Signal definition of the ramp or checksum (see G_Common_Signal_t)

`mode`
Specifies whether the calculation of a ramp or a checksum is suspended

The following values are possible:

`G_CAN__CYCLIC_MSGS__SET_CALCULATION_BREAK_COUNT__MODE__RAMP`
The calculation of a ramp is suspended

`G_CAN__CYCLIC_MSGS__SET_CALCULATION_BREAK_COUNT__MODE__CHECKSUM`
The calculation of a checksum is suspended

`breakCount`
Number of times the calculation will be suspended

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_DefineDisturbance

G_Can_CyclicMsgs_DefineDisturbance — Set temporary failures

Description

This command offers the possibility to simulate temporary failures of the CAN message defined by `id` divergent of the normal bus behavior:

The CAN message defined by `id` is sent `count` times with the data length defined by `dlc` and the data defined by `data` in accordance with the set mask byte bit positions defined by `mask`.

Definition

```
G_Error_t
G_Can_CyclicMsgs_DefineDisturbance(
 const G_PortHandle_t portHandle,
 const u32_t id,
 const u8_t  dlc,
 const u16_t  count,
 const G_Can_Data_t  mask,
 const G_Can_Data_t  data
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier of CAN message

`dlc`
: Data length code while the disturbance is active (0..64)

DLC	Data Length (bytes)
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	12
10	16
11	20
12	24
13	32
14	48

DLC	Data Length (bytes)
15	64

Table 2 DLC to data length

`count`
Number of disturbed messages

`mask`
Mask bytes (0..64)

`data`
Data bytes while the disturbance is active (0..64)

Data is assumed in accordance with the set mask byte bit positions. If no mask byte bits are set, the original data is assumed.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_ChangeMsg

G_Can_CyclicMsgs_ChangeMsg — Change message parameters

Description

This command enables to change several parameters of an already defined message.

Definition

```
G_Error_t
G_Can_CyclicMsgs_ChangeMsg(
 const G_PortHandle_t portHandle,
 const G_Can_CyclicMsgs_ChangeMsg_Parameters_t \
 * const changeMsgParameters
);
```

Parameters

`portHandle`
Handle to the communication port

`changeMsgParameters`
Message parameters to change

The structure consists of the following items:

`Id`
Identifier of the message whose parameters have to be changed

`Flags`
Command flags

The following values are possible and can be combined:

`G_CAN__CYCLIC_MSGS__CHANGE_MSG__CMD_FLAG__NONE`
No flag is set

`G_CAN__CYCLIC_MSGS__CHANGE_MSG__CMD_FLAG__CHANGE_SEND_STATE`
Change the send state of the message

`G_CAN__CYCLIC_MSGS__CHANGE_MSG__CMD_FLAG__CHANGE_GROUP`
Change the group affiliation of the message

see also G_Can_CyclicMsgs_ChangeGroupNumber

`G_CAN__CYCLIC_MSGS__CHANGE_MSG__CMD_FLAG__CHANGE_MSG_COUNT`
Change message count

`G_CAN__CYCLIC_MSGS__CHANGE_MSG__CMD_FLAG__CHANGE_DLC`
Change the data length of the message

`G_CAN__CYCLIC_MSGS__CHANGE_MSG__CMD_FLAG__CHANGE_DATA`
Change the data content of the message

`G_CAN__CYCLIC_MSGS__CHANGE_MSG__CMD_FLAG__CHANGE_CYCLE_TIME`
Change message cycle time

`G_CAN__CYCLIC_MSGS__CHANGE_MSG__CMD_FLAG__USE_EXTENDED_MASK_AND_DATA`
Use extended mask and data (for **CAN FD** messages with more than 8 data bytes)

`SendState`
Send state of the message

The following values are possible:

`G_CAN__CYCLIC_MSGS__SEND_STATE__DONT_SEND_MSG`
Do not send the message

`G_CAN__CYCLIC_MSGS__SEND_STATE__SEND_MSG`
Send the message

`Group`
New group ID of the message

In the current version, the group IDs `0` and `1` only are supported. (There is no group affiliation for the message if group ID = `0`).

`MsgCount`
Number of the transmit repetitions of the message
- `0` - Always send the message (unlimited)
- 1..255 - Send the message 1..255 times

`Dlc`
Data length code (see DLC to data length)

`Data`
Structure for message data

The structure consists of the following items:

`Flags`
Message data flags

The following flags are possible and can be combined:

`G_CAN__CYCLIC_MSGS__CHANGE_MSG__DATA__CMD_FLAG__NONE`
No flag is set

`G_CAN__CYCLIC_MSGS__CHANGE_MSG__DATA__CMD_FLAG__USE_MIN_DELAY`
Use the minimum delay time between messages

`G_CAN__CYCLIC_MSGS__CHANGE_MSG__DATA__CMD_FLAG__SAVE_CURRENT_DA-TA`
Data part of the current message is saved

`G_CAN__CYCLIC_MSGS__CHANGE_MSG__DATA__CMD_FLAG__RESTORE_SAVED_DA-TA`
Saved data part of a message is restored

`Mask`
Data mask bytes

`Data`
Data bytes

`MinimumDelay`
Minimum delay between the previous and the current transmission of the message

`NumberOfMsgsBeforeRestore`
Number of the transmitted messages before restoring the saved data part

`CycleTime`
Structure for message cycle time

The structure consists of the following items:

`Flags`
Message cycle time flags

The following flags are possible and can be combined:

`G_CAN__CYCLIC_MSGS__CHANGE_MSG__CYCLE_TIME__CMD_FLAG__NONE`
No flag is set

`G_CAN__CYCLIC_MSGS__CHANGE_MSG__CYCLE_TIME__CMD_FLAG__USE_MIN_DELAY`
Use the minimum delay time between messages

`G_CAN__CYCLIC_MSGS__CHANGE_MSG__CYCLE_TIME__CMD_FLAG__SAVE_CURRENT`
The current cycle time of the message is saved

`G_CAN__CYCLIC_MSGS__CHANGE_MSG__CYCLE_TIME__CMD_FLAG__RESTORE_SAVED`
The saved cycle time of a message is restored

`G_CAN__CYCLIC_MSGS__CHANGE_MSG__CYCLE_TIME__CMD_FLAG__USE_BOTH_TIMES`
This flag is relevant if flag `G_CAN__CYCLIC_MSGS__CHANGE_MSG__CYCLE_TIME__CMD_FLAG__RESTORE_SAVED` is set.

The stored cycle time and the new cycle time run in parallel

Set this flag for AUTOSAR mixed transmission mode:

ComTxMode = MIXED

ComTxModeTimePeriod = stored cycle time

ComTxModeRepetitionPeriod = `CycleTime`

ComTxModeNumberOfRepetitions = `NumberOfMessagesBeforeRestore` + 1

`G_CAN__CYCLIC_MSGS__CHANGE_MSG__CYCLE_TIME__CMD_FLAG__ALLOW_SHIFT`
This flag is relevant if flag `G_CAN__CYCLIC_MSGS__CHANGE_MSG__CYCLE_TIME__CMD_FLAG__USE_BOTH_TIMES` is set:

The stored cycle time may be shifted due to minimum delay time, e.g. the stored cycle time is **10**, then transmission is on time: 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, ...

With a change on time **28**, `MinimumDelayTime` = 3, `CycleTime` = 0, `NumberOfMessagesBeforeRestore` = 0: 0, 10, 20, 28, 38, 48, 58, 68, 78, 88, 98, ...

With a change on time **22**, `MinimumDelayTime` = 3, `CycleTime` = 0, `NumberOfMessagesBeforeRestore` = 0: 0, 10, 20, 23, 30, 40, 50, 60, 70, 80, 90, ...

With a change on time **29**, `MinimumDelayTime` = 2, `CycleTime` = 4, `NumberOfMessagesBeforeRestore` = 2: 0, 10, 20, 29, 31, 33, 37, 41, 51, 61, 71, ...

`G_CAN__CYCLIC_MSGS__CHANGE_MSG__CYCLE_TIME__CMD_FLAG__NO_OVERWRITE`
This flag should always be set, because it doesn't save actual cycle time as first cycle time when a restore of the first cycle time has not been finished (second cycle time is active). So the first cycle time is not overwritten.

G_CAN__CYCLIC_MSGS__CHANGE_MSG__CYCLE_TIME__CMD_FLAG__CANCEL_RESTORE
The saved cycle time value will not be restored.

CycleTime
New message cycle time

MinimumDelay
Minimum delay between the previous and the current transmission of the message

NumberOfMsgsBeforeRestore
Number of the transmitted messages before restoring the saved cycle time

reserved1
reserved parameter (must be initialized with **0**)

reserved2
reserved parameter (must be initialized with **0**)

ExtendedMaskAndData
Structure for extended mask and data bytes

This part is used for defining mask and data bytes of messages with more than 8 data bytes. The first 8 data bytes are set in the Data structure and all following bytes (9..64) are set here.

G_CAN__CYCLIC_MSGS__CHANGE_MSG__CMD_FLAG__USE_EXTENDED_MASK_AND_DATA needs to be set to use this structure.

The structure consists of the following items:

Length
Length of extended mask and data (in bytes)

reserved1
reserved parameter (must be initialized with **0**)

reserved2
reserved parameter (must be initialized with **0**)

reserved3
reserved parameter (must be initialized with **0**)

Mask
Buffer with mask bytes (0..56)

Data
Buffer with data bytes (0..56)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_ChangeMsgData

G_Can_CyclicMsgs_ChangeMsgData — Change the message

Description

This command changes the data of the message given by `id`.

Definition

```
G_Error_t
G_Can_CyclicMsgs_ChangeMsgData(
 const G_PortHandle_t portHandle,
 const u32_t id,
 const u8_t mask[64],
 const u8_t dlc,
 const u8_t data[64],
 const G_Can_CyclicMsgs_ChangeMsgData_ChangeMode_t changeMode,
 const u16_t minimumDelay
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier of the message whose parameters have to be changed

`mask`
: Buffer with mask bytes

`dlc`
: Data length code (see DLC to data length)

`data`
: Buffer with data bytes

`changeMode`
: Change mode

  The following values are possible:

  `G_CAN__CYCLIC_MSGS__CHANGE_MSG_DATA__CHANGE_MODE__IN_CYCLE`
  : Change data in the next cycle

  `G_CAN__CYCLIC_MSGS__CHANGE_MSG_DATA__CHANGE_MODE__IMMEDIATELY`
  : Change data immediately

  `G_CAN__CYCLIC_MSGS__CHANGE_MSG_DATA__CHANGE_MODE__WITH_MINIMUM_DELAY`
  : Change data after minimum delay time specified in `minimumDelay` (the message is sent at the earliest possible time after processing the minimum delay following the previous transmitting of this message)

`minimumDelay`
: Minimum delay between the previous and the repeated transmission of the CAN message with the identifier `id`

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_ChangeMsgDataAndMsgCount

G_Can_CyclicMsgs_ChangeMsgDataAndMsgCount — Change the message

Description

This command changes the data and the number of transmit repetitions of the message given by `id`.

Definition

```
G_Error_t
G_Can_CyclicMsgs_ChangeMsgDataAndMsgCount(
 const G_PortHandle_t portHandle,
 const u32_t id,
 const u8_t msgCount,
 const u8_t mask[8],
 const u8_t dlc,
 const u8_t data[8],
 const G_Can_CyclicMsgs_ChangeMsgData_ChangeMode_t changeMode,
 const u16_t minimumDelay
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Identifier of the message

`msgCount`
Number of transmit repetitions
- `0` - Always send the message
- **`1..255`** - Send the message 1..255 times

`mask`
Mask bytes

`dlc`
Data length code (see DLC to data length)

`data`
Data bytes

`changeMode`
How to change the message parameters

The following values are possible:

`G_CAN__CYCLIC_MSGS__CHANGE_MSG_DATA__CHANGE_MODE__IN_CYCLE`
Parameters are changed in the next transmit cycle

`G_CAN__CYCLIC_MSGS__CHANGE_MSG_DATA__CHANGE_MODE__IMMEDIATELY`
Parameters are changed immediately

`G_CAN__CYCLIC_MSGS__CHANGE_MSG_DATA__CHANGE_MODE__WITH_MINIMUM_DELAY`
Parameters are changed after the minimum delay time

`minimumDelay`
Minimum delay between the previous and the current transmitting of the CAN message with the identifier `id`

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_ChangeGroupNumber

G_Can_CyclicMsgs_ChangeGroupNumber — Change the group affiliation of a message

Description

This command changes the group affiliation of an already defined message.

All messages belonging to the same group can be started at the same time with G_Can_CyclicMsgs_StartGroup and stopped with G_Can_CyclicMsgs_StopGroup .

If the group number is `0`, the message does not have any group affiliation.

Definition

```
G_Error_t
G_Can_CyclicMsgs_ChangeGroupNumber(
 const G_PortHandle_t portHandle,
 const u32_t id,
 const u8_t group
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Identifier of a message whose group affiliation has to be changed

`group`
New group number of the message

In the recent version, the group IDs `0` and `1` only are supported.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
rc =
G_Can_CyclicMsgs_ChangeGroupNumber(
 portHandle,
 0x123,
 1
);
```

In this example, the group number of the message with the identifier `0x123` is changed by **G_Can_CyclicMsgs_ChangeGroupNumber** to the value `1`.

G_Can_CyclicMsgs_ChangeCycleTime

G_Can_CyclicMsgs_ChangeCycleTime — Change message cycle time

Description

Use this command to change the cycle time of a cyclic message.

Definition

```
G_Error_t
G_Can_CyclicMsgs_ChangeCycleTime(
 const G_PortHandle_t portHandle,
 const u32_t id,
 const u16_t cycleTime
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier of the message whose cycle time has to be changed

`cycleTime`
: New message cycle time in milliseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_StartMsg

G_Can_CyclicMsgs_StartMsg — Starting the message

Description

This command sends the message given by `id`.

Definition

```
G_Error_t
G_Can_CyclicMsgs_StartMsg(
 const G_PortHandle_t portHandle,
 const u32_t id
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier of the message to be transmitted

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_StopMsg

G_Can_CyclicMsgs_StopMsg — Stop the message

Description

This command stops the sending operation of the message given by `id`.

Definition

```
G_Error_t
G_Can_CyclicMsgs_StopMsg(
 const G_PortHandle_t portHandle,
 const u32_t id
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier of the message to be stopped

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_GetMsgState

G_Can_CyclicMsgs_GetMsgState — Query the message state

Description

Use this command to query the state of a message.

Definition

```
G_Error_t
G_Can_CyclicMsgs_GetMsgState(
 const G_PortHandle_t portHandle,
 const u32_t id,
 G_Can_CyclicMsgs_GetMsgState_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Identifier of the message

`rsp`
Pointer to response structure

The structure contains the following elements:

`Id`
Identifier of the message

`CycleTime`
Cycle time in milliseconds

`Group`
Message group number

`Mode`
Message mode

The following values are possible:

`G_CAN__CYCLIC_MSGS__MSG_MODE__STOPPED`
Message is stopped

`G_CAN__CYCLIC_MSGS__MSG_MODE__STARTED`
Message is started

`MsgCount`
0 : message is sent endlessly

1..N..255 : message is sent **N** times

`Dlc`
Data length code (0..64)

DLC	Data Length (bytes)
0	0

DLC	Data Length (bytes)
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	12
10	16
11	20
12	24
13	32
14	48
15	64

Table 3 DLC to data length

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Data`
Message data bytes (0..64)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_StartGroup

G_Can_CyclicMsgs_StartGroup — Start the group

Description

This command sends all messages contained in the group given by `group`.

Please consider that the group `0` cannot be started. Messages with this group number do not have any group affiliation.

Definition

```
G_Error_t
G_Can_CyclicMsgs_StartGroup(
 const G_PortHandle_t portHandle,
 const u8_t group
);
```

Parameters

`portHandle`
Handle to the communication port

`group`
Group number of the messages

In the recent version, the group IDs `0` and `1` only are supported.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_StopGroup

G_Can_CyclicMsgs_StopGroup — Stop the group

Description

This command stops the sending operation of all messages contained in the group given by `group`.

Please consider that the group `0` cannot be stopped. Messages with this group number do not have any group affiliation.

Definition

```
G_Error_t
G_Can_CyclicMsgs_StopGroup(
 const G_PortHandle_t portHandle,
 const u8_t group
);
```

Parameters

`portHandle`
Handle to the communication port

`group`
Group number of the messages

In the recent version, the group IDs `0` and `1` only are supported.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_MultiplexBytes_Define

G_Can_CyclicMsgs_MultiplexBytes_Define — Define multiplex bytes

Description

This command defines multiplex bytes for a CAN message.

Multiplexing the data bytes of a CAN message serves to transfer more information as usually possible with only eight data bytes. Generally the multiplex code located fixed in a certain data byte explaines the meaning of the remaining data bytes. This way time multiplexing is carried out.

The multiplex code itself is defined together with the actual multiplex data within parameter `Data`.

Definition

```
G_Error_t
G_Can_CyclicMsgs_MultiplexBytes_Define(
 const G_PortHandle_t portHandle,
 const u32_t  id,
 const u8_t  updateRate,
 const u8_t  numberOfIndices,
 const G_Can_Data_t * const mask,
 const G_Can_Data_t * const data
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: CAN Identifier

`updateRate`
: Update rate of the Multiplexbytes

  1. With each CAN message

  2. With every second CAN message

  3. With every third CAN message
  etc.

`numberOfIndices`
: Number of multiplex indices (1..16)

`mask`
: Pointer to array with mask bytes (0..64)

  (Data is assumed in accordance with the set mask byte bit positions)

  Mask bytes have no indices, therefore the same mask is applied to every data byte index.

`data`
: Pointer to array with data bytes

  The elements of the array represent the indices of multiplex bytes.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
G_Can_Data_t  mask;
G_Can_Data_t  data;
G_Error_t  rc;

mask.U64.High = 0xFFFFFF;
mask.U64.Low = 0;

data.U8[0] = 0x42;
data.U8[1] = 0x11;
data.U8[2] = 0x22;
data.U8[3] = 0x33;
data.U8[4] = 0x44;
data.U8[5] = 0x55;
data.U8[6] = 0x66;
data.U8[7] = 0x77;

rc =
 G_Can_CyclicMsgs_MultiplexBytes_Define(
  portHandle,
  0x123,
  1,
  1,
  &mask,
  data
 );
```

Defining multiplex bytes with a byte mask for a CAN message with identifier **0x123**

G_Can_CyclicMsgs_MultiplexBytes_Stop

G_Can_CyclicMsgs_MultiplexBytes_Stop — Stop multiplexing bytes

Description

The multiplexed bytes of the CAN message defined by `id` are stopped by this command. After stopping the multiplexed bytes, the defined data is assumed in accordance with the defined byte mask.

Definition

```
G_Error_t
G_Can_CyclicMsgs_MultiplexBytes_Stop(
 const G_PortHandle_t portHandle,
 const u32_t  id,
 const G_Can_Data_t * const mask,
 const G_Can_Data_t * const data
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier

`mask`
: Mask bytes (0..7)

  (Data is assumed in accordance with the set mask byte bit positions)

`data`
: Data bytes (0..7)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_Ramp_Define

G_Can_CyclicMsgs_Ramp_Define — Ramp definition

Description

Use this command to define and start a new ramp or to change the mode of a running ramp within a CAN message.

A ramp serves to create alternately increasing / decreasing signal courses.

Definition

```
G_Error_t
G_Can_CyclicMsgs_Ramp_Define(
 const G_PortHandle_t portHandle,
 const G_Can_CyclicMsgs_Ramp_Define_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Ramp parameters

The structure contains the following elements:

`Id`
Identifier of the CAN message

`Signal`
Signal definition of the ramp (see G_Common_Signal_t)

`Mode`
Ramp mode

The following values are possible:

`G_CAN__CYCLIC_MSGS__RAMP__MODE__TRIANGULAR`
Triangular ramp

`G_CAN__CYCLIC_MSGS__RAMP__MODE__SAWTOOTH`
Sawtooth ramp

`G_CAN__CYCLIC_MSGS__RAMP__MODE__SAWTOOTH_WITH_OVERFLOW_BIT`
Sawtooth ramp with overflow bit

`G_CAN__CYCLIC_MSGS__RAMP__MODE__STOP`
The ramp is stopped

`Direction`
Starting direction

The following values are possible:

`G_CAN__CYCLIC_MSGS__RAMP__DIRECTION__DOWN`
Ramp starts downwards

`G_CAN__CYCLIC_MSGS__RAMP__DIRECTION__UP`
Ramp starts upwards

`MinValue`
Minimum value of the ramp

`MaxValue`
Maximum value of the ramp

`UpdateRate`
Updating the ramp

1 : with each CAN message

2 : with every second CAN message

etc.

`Increment`
Increment of the ramp

`RampSum`
Relative ramp sum (sum of the partial increments starting with the execution of this command, therefore relative)

(0 = No limitation)

`NumberOfFlanks`
Number of flanks to be used

(0 = No limitation)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

The ramp starts using the current signal value. To avoid unexpected behavior, it is recommended to ensure that the current signal value is within a valid range.

`NumberOfFlanks` and `RampSum` are abort conditions. When both values do equal **0** , the ramp runs endlessly, otherwise the ramp stops when meeting one of these conditions.

When a ramp runs, the partial increments are added continuously with a stepwidth defined with **Increment** .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_Ramp_Delete

G_Can_CyclicMsgs_Ramp_Delete — Delete ramp

Description

Use this command to delete a ramp.

Definition

```
G_Error_t
G_Can_CyclicMsgs_Ramp_Delete(
 const G_PortHandle_t portHandle,
 const u32_t  id,
 const G_Common_Signal_t  signal
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier of the CAN message

`signal`
: Signal definition of the ramp (see G_Common_Signal_t)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_Ramp_ChangeMode

G_Can_CyclicMsgs_Ramp_ChangeMode — Change ramp mode

Description

Use this command to change the mode of one or more ramps.

Definition

```
G_Error_t
G_Can_CyclicMsgs_Ramp_ChangeMode(
 const G_PortHandle_t portHandle,
 const u32_t  numberOfRamps,
 const G_Can_CyclicMsgs_Ramp_ChangeMode_Ramps_t * const  ramps
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfRamps`
Number of ramps whose mode is to be changed

`ramps`
Pointer to structure array with ramp mode parameters

Each array elements represents the parameters for one ramp.

The structure contains the following elements:

`Id`
Identifier of the CAN message

`Signal`
Signal definition of the ramp (see G_Common_Signal_t)

`Mode`
New ramp mode (see Ramp Mode)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_Ramp_GetState

G_Can_CyclicMsgs_Ramp_GetState — Query ramp state

Description

Use this command to query the current state of a ramp.

Definition

```
G_Error_t
G_Can_CyclicMsgs_Ramp_GetState(
 const G_PortHandle_t portHandle,
 const u32_t  id,
 const G_Common_Signal_t  signal,
 G_Can_CyclicMsgs_Ramp_State_Flags_t * const rampState,
 G_Can_CyclicMsgs_Ramp_State_Flags_t * const incRampState,
 u32_t * const rampSum,
 u16_t * const incRampFlanks
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Identifier of the CAN message

`signal`
Signal definition of the ramp (see G_Common_Signal_t)

`rampState`
Returned ramp state flags

The following values are possible and can be combined:

`G_CAN__CYCLIC_MSGS__RAMP__STATE__FLAG__NONE`
No flag set

`G_CAN__CYCLIC_MSGS__RAMP__STATE__FLAG__RUNNING`
The ramp is currently running.

`G_CAN__CYCLIC_MSGS__RAMP__STATE__FLAG__DIRECTION_IS_UP`
The ramp's running direction is **upwards** .

If this flag is not set, the ramp's running direction is **downwards** .

`incRampState`
Returned state of increment ramp (see `rampState` for possible values)

`rampSum`
Current internal ramp sum

`incRampFlanks`
Number of already executed increment ramp flanks

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_Ramp_DefineIncRamp

G_Can_CyclicMsgs_Ramp_DefineIncRamp — Define increment ramp

Description

Use this command to define an incement ramp.

An increment ramp is required when a ramp shall not operate with a constant increment but rather the increment itself shall run in a ramp.

The command offers the possibility to directly influence the data of a further CAN message synchronous to the increment ramp. This influence takes place as a ramp with a constant increment.

Definition

```
G_Error_t
G_Can_CyclicMsgs_Ramp_DefineIncRamp(
 const G_PortHandle_t portHandle,
 const G_Can_CyclicMsgs_Ramp_DefineIncRamp_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Parameter structure

The structure contains the following elements:

`Id`
Identifier of the CAN message

`Signal`
Signal definition of the ramp (see G_Common_Signal_t)

`MinValue`
Minimum value of the increment ramp

`MaxValue`
Maximum value of the increment ramp

`CycleTime`
Cycle time of the increment ramp (from flank to flank)

`NumberOfFlanks`
Number of increment flanks

The value `0` defines an infinite execution.

`Direction`
Starting direction (see Ramp Direction)

`reserved1`
reserved parameter (must be initialized with `0`)

`reserved2`
reserved parameter (must be initialized with `0`)

`reserved3`
reserved parameter (must be initialized with **0**)

`SyncRamp`
Pointer to sync ramp definition

If this pointer equals **NULL**, no sync ramp is defined.

The sync ramp definition contains the following elements:

`Id`
Identifier of the CAN message

`Signal`
Signal definition of the ramp (see G_Common_Signal_t)

`Mode`
Ramp mode (see Ramp Mode)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`MinValue`
Minimum value of the sync ramp

`MaxValue`
Maximum value of the sync ramp

By defining the same value for `MinValue` and `MaxValue`, the increment ramp is deleted and this value is assumed as the `increment` value of the basis ramp.

If a ramp that includes an increment ramp is defined repeatedly with command G_Can_CyclicMsgs_Ramp_Define, the increment ramp for this ramp is deleted.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_Ramp_ResetRampSum

G_Can_CyclicMsgs_Ramp_ResetRampSum — Reset ramp sum

Description

Use this command to reset the internal ramp sum of a ramp.

Definition

```
G_Error_t
G_Can_CyclicMsgs_Ramp_ResetRampSum(
 const G_PortHandle_t portHandle,
 const u32_t  id,
 const G_Common_Signal_t  signal
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier of the CAN message

`signal`
: Signal definition of the ramp (see G_Common_Signal_t)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_Parity_Define

G_Can_CyclicMsgs_Parity_Define — Activate parity bit

Description

Use this command to define and activate a parity bit.

Definition

```
G_Error_t
G_Can_CyclicMsgs_Parity_Define(
 const G_PortHandle_t portHandle,
 const u32_t  id,
 const G_Can_CyclicMsgs_Parity_Mode_t  mode,
 const G_Common_Signal_t  signal,
 const G_Can_Data_t  mask
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Identifier of the CAN message

`mode`
Parity bit mode

The following values are possible:

`G_CAN__CYCLIC_MSGS__PARITY__MODE__EVEN`
Even parity

`G_CAN__CYCLIC_MSGS__PARITY__MODE__ODD`
Odd parity

`signal`
Signal definition of the parity bit (see G_Common_Signal_t)

`mask`
Mask bytes (0..7)

The Mask bytes are used to define the bits of the data bytes to be used for the calculation of the parity. Therefore the parity bit itself should be deleted in the Mask.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_Parity_Delete

G_Can_CyclicMsgs_Parity_Delete — Deactivate parity bit

Description

Use this command to deactivate a parity bit.

Definition

```
G_Error_t
G_Can_CyclicMsgs_Parity_Delete(
 const G_PortHandle_t portHandle,
 const u32_t  id,
 const G_Common_Signal_t  signal
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier of the CAN message

`signal`
: Signal definition of the parity bit (see G_Common_Signal_t)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_Checksum_Define

G_Can_CyclicMsgs_Checksum_Define — Checksum definition

Description

Use this command to define a checksum.

Definition

```
G_Error_t
G_Can_CyclicMsgs_Checksum_Define(
 const G_PortHandle_t portHandle,
 const G_Can_CyclicMsgs_Checksum_Mode_t  mode,
 const u32_t  id,
 const G_Common_Signal_t  signal,
 const G_Can_Data_t * const mask
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
Checksum mode

The following values are possible:

`G_CAN__CYCLIC_MSGS__CHECKSUM__MODE__XOR`
XOR checksum

`G_CAN__CYCLIC_MSGS__CHECKSUM__MODE__CRC8_J1850`
CRC8 checksum according to **J1850**

`G_CAN__CYCLIC_MSGS__CHECKSUM__MODE__ADD_AND_COMPLEMENT_TO_ONE`
'Add bytes and complement to **1**' checksum

`id`
Identifier of the CAN message

`signal`
Signal definition of the checksum (see G_Common_Signal_t)

`mask`
Mask bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_Checksum_Define_XorWithUserBytes

G_Can_CyclicMsgs_Checksum_Define_XorWithUserBytes — Define **XOR** checksum with user bytes

Description

Use this command to define a **XOR** checksum with user bytes.

Definition

```
G_Error_t
G_Can_CyclicMsgs_Checksum_Define_XorWithUserBytes(
 const G_PortHandle_t portHandle,
 const u32_t  id,
 const G_Common_Signal_t  signal,
 const G_Can_Data_t * const mask,
 const u8_t  numberOfBytes,
 const u8_t * const bytes
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier of the CAN message

`signal`
: Signal definition of the checksum (see G_Common_Signal_t)

`mask`
: Mask bytes

`numberOfBytes`
: Number of user bytes

`bytes`
: Pointer to array with user bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_Checksum_Define_GeneralCrcWithUserBytes

G_Can_CyclicMsgs_Checksum_Define_GeneralCrcWithUserBytes — Define **General Crc** checksum with user bytes

Description

Use this command to define a **General Crc** checksum with user bytes.

Definition

```
G_Error_t
G_Can_CyclicMsgs_Checksum_Define_GeneralCrcWithUserBytes(
 const G_PortHandle_t portHandle,
 const u32_t  id,
 const G_Common_Signal_t  signal,
 const G_Can_Data_t * const mask,
 const G_Can_CyclicMsgs_Checksum_Define_GeneralCrc_Parameters_t * const
 parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Identifier of the CAN message

`signal`
Signal definition of the checksum (see G_Common_Signal_t)

`mask`
Mask bytes

`parameters`
Structure with **General Crc** parameters

The structure contains the following elements:

`Order`
Checksum order (**8**, **16**, ...)

`NumberOfUserBytesBefore`
Number of user bytes before checksum calculation over data bytes

`NumberOfUserBytesAfter`
Number of user bytes after checksum calculation over data bytes

`reserved`
reserved parameter (must be initialized with **0**)

`Polynomial`
Polynomial value (e.g. **0x1D** for **CRC8-J1850**)

`InitialValue`
Initial value (e.g. **0xFF** for **CRC8-J1850**)

`FinalXorValue`
Final **XOR** value (e.g. **0xFF** for **CRC8-J1850**)

`UserBytesBefore`
Pointer to array with user bytes before checksum calculation over data bytes

`UserBytesAfter`
Pointer to array with user bytes after checksum calculation over data bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_Checksum_Define_Custom12

G_Can_CyclicMsgs_Checksum_Define_Custom12 — Define **Custom12** checksum

Description

Calculate CRC over CAN data bytes and one byte of `DataArray`. The byte is selected (`DataArray` is indexed) by the value of the signal defined by `DataArray_IndexSignal`.

Definition

```
G_Error_t
G_Can_CyclicMsgs_Checksum_Define_Custom12(
 const G_PortHandle_t portHandle,
 const u32_t  id,
 const G_Common_Signal_t  signal,
 const G_Can_Data_t * const mask,
 const G_Can_CyclicMsgs_Checksum_Define_Custom12_Parameters_t *
  const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Identifier of the CAN message

`signal`
Signal definition of the checksum (see G_Common_Signal_t)

`mask`
Mask bytes

`parameters`
Structure with **General Crc** parameters

The structure contains the following elements:

`Order`
Checksum order (**8**, **16**, ...)

`DataArray_Size`
Size of `DataArray`, in bytes (e.g. 16)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Polynomial`
Polynomial value (e.g. **0x2F**)

`InitialValue`
Initial value (e.g. **0xFF**)

`FinalXorValue`
Final **XOR** value (e.g. **0xFF**)

`DataArray_IndexSignal`
Signal description for index into `DataArray`

`DataArray`
Pointer to array with user bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_Checksum_Define_AddBytes

G_Can_CyclicMsgs_Checksum_Define_AddBytes — Define **Add Bytes** checksum

Description

Calculate checksum over CAN data as a sum of data bytes.

Definition

```
G_Error_t
G_Can_CyclicMsgs_Checksum_Define_AddBytes(
 const G_PortHandle_t portHandle,
 const u32_t  id,
 const G_Common_Signal_t  signal,
 const G_Can_Data_t * const mask,
 const G_Can_CyclicMsgs_Checksum_Define_AddBytes_Parameters_t *
  const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Identifier of the CAN message

`signal`
Signal definition of the checksum (see G_Common_Signal_t)

`mask`
Mask bytes

`parameters`
Structure with **General Crc** parameters

The structure contains the following elements:

`InputSignal`
Additional input signal. Set member "Size" to zero to indicate that an additional input signal is not used.

`InitialValue`
Initial Value (optional parameter, 0 if not present)

`FinalXorValue`
Final Final **XOR** value (optional parameter e.g. **0xFF** , 0 if not present)

`InputSignalShiftLeft`
Value by which the signal is to be shifted to the left (optional parameter, 0 if not present)

`reserved`
optional parameter (should not be present)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_Checksum_Delete

G_Can_CyclicMsgs_Checksum_Delete — Delete checksum

Description

Use this command to delete all checksums of a message.

Definition

```
G_Error_t
G_Can_CyclicMsgs_Checksum_Delete(
 const G_PortHandle_t portHandle,
 const u32_t  id,
 const G_Common_Signal_t  signal
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier of the CAN message

`signal`
: Signal definition of the checksum (see G_Common_Signal_t)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_Property_SetById

G_Can_CyclicMsgs_Property_SetById — Set cyclic message property value

Description

Use this command to set property values of a cyclic message.

Definition

```
G_Error_t
G_Can_CyclicMsgs_Property_SetById(
 const G_PortHandle_t portHandle,
 const G_Can_CyclicMsgs_Property_SetById_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`CanId`
CAN Identifier of the message

`PropertyId`
ID of the property to be set

The following values are possible:

`G_CAN__CYCLIC_MSGS__PROPERTY_ID__MSG_FLAG__BRS`
use type **u8_t**

Bit Rate Switch bit for CAN FD frames

0 = bit rate switch is disabled

1 = bit rate switch is enabled

`G_CAN__CYCLIC_MSGS__PROPERTY_ID__MSG_FLAG__ESI`
use type **u8_t**

Error state indicator bit for CAN FD frames

0 = transmitting node is error active

1 = transmitting node is error passive

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`PropertyValue`
Value of the property to be set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_CyclicMsgs_Property_GetById

G_Can_CyclicMsgs_Property_GetById — Get cyclic message property value

Description

Use this command to query property values of a cyclic message.

Definition

```
G_Error_t
G_Can_CyclicMsgs_Property_GetById(
 const G_PortHandle_t portHandle,
 const G_Can_CyclicMsgs_Property_GetById_Cmd_t * const cmd,
 G_Can_CyclicMsgs_Property_GetById_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`CanId`
CAN Identifier of the message

`PropertyId`
ID of the property to be set

The following values are possible:

`G_CAN__CYCLIC_MSGS__PROPERTY_ID__MSG_FLAG__BRS`
use type **u8_t**

Bit Rate Switch bit for CAN FD frames

0 = bit rate switch is disabled

1 = bit rate switch is enabled

`G_CAN__CYCLIC_MSGS__PROPERTY_ID__MSG_FLAG__ESI`
use type **u8_t**

Error state indicator bit for CAN FD frames

0 = transmitting node is error active

1 = transmitting node is error passive

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`CanId`
CAN Identifier of the message

`PropertyId`
ID of the property to be set

The following values are possible:

`G_CAN__CYCLIC_MSGS__PROPERTY_ID__MSG_FLAG__BRS`
use type **u8_t**

Bit Rate Switch bit for CAN FD frames

0 = bit rate switch is disabled

1 = bit rate switch is enabled

`G_CAN__CYCLIC_MSGS__PROPERTY_ID__MSG_FLAG__ESI`
use type **u8_t**

Error state indicator bit for CAN FD frames

0 = transmitting node is error active

1 = transmitting node is error passive

`reserved1`
reserved parameter

`reserved2`
reserved parameter

`PropertyValue`
Value of the property

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

5 Diag

These commands are used to control the diagnostics.

G_Can_Diag_GetState

G_Can_Diag_GetState — Query the diagnostics status

Description

This command queries the current diagnostics status.

Definition

```
G_Error_t
G_Can_Diag_GetState(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Can_Diag_GetState_CmdFlags_t cmdFlags,
 G_Error_t * const lastErrorCode,
 G_Can_Diag_Type_t * const type,
 G_Can_Diag_State_t * const state,
 G_Can_Diag_GetState_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`cmdFlags`
Variable of type **G_Can_Diag_GetState_CmdFlags_t**

The following flags are available:

- `G_CAN__DIAG__GET_STATE__CMD_FLAG__NONE`

  No flag, if no other has to be transferred
- `G_CAN__DIAG__GET_STATE__CMD_FLAG__RESET_LAST_ERROR`

  Reset the error code of the last error that has occurred in the firmware

`lastErrorCode`
Pointer to a variable of type **G_Error_t**. The error code of the last error that has occurred in the firmware (or `G_NO_ERROR` = no error) is returned in this variable. For getting an error description, you can call G_Common_GetFirmwareErrorDescription.

`type`
Variable of type **G_Can_Diag_Type_t**. The diagnostics type is returned in this variable.

The following values are available:

- `G_CAN__DIAG__TYPE__OFF` - No diagnostics
- `G_CAN__DIAG__TYPE__KW2000_TP_1_6` - KWP2000 on TP1.6
- `G_CAN__DIAG__TYPE__KW2000_TP_2_0` - KWP2000 on TP2.0
- `G_CAN__DIAG__TYPE__KW2000_ISOTP` - KWP2000 on ISOTP
- `G_CAN__DIAG__TYPE__GMLAN` - GMLAN
- `G_CAN__DIAG__TYPE__UDS_ISOTP` - UDS on ISOTP
- `G_CAN__DIAG__TYPE__J1939` - J1939

`state`
Pointer to a variable of type **G_Can_Diag_State_t**. The current diagnostics status is returned in this variable.

The following values are available:

- `G_CAN__DIAG__STATE__NOT_INITIALIZED`

  Not initialized
- `G_CAN__DIAG__STATE__DISCONNECTED`

  Disconnected
- `G_CAN__DIAG__STATE__CONNECT_IN_PROGRESS`

  The connecting is in progress
- `G_CAN__DIAG__STATE__CONNECTED`

  Connected
- `G_CAN__DIAG__STATE__DISCONNECT_IN_PROGRESS`

  The disconnection is in progress

`rspFlags`
Pointer to a variable of type **G_Can_Diag_GetState_RspFlags_t**

The following return flags are available:

- `G_CAN__DIAG__GET_STATE__RSP_FLAG__NONE`

  No flag was set
- `G_CAN__DIAG__GET_STATE__RSP_FLAG__BUSY`

  A request has not been responded yet/ successfully sent
- `G_CAN__DIAG__GET_STATE__RSP_FLAG__SYNC_RX_BUFFER_NOT_EMPTY`

  Synchronous receiving buffer is not empty
- `G_CAN__DIAG__GET_STATE__RSP_FLAG__ASYNC_RX_BUFFER_NOT_EMPTY`

  Asynchronous receiving buffer is not empty
- `G_CAN__DIAG__GET_STATE__RSP_FLAG__UUDT_RX_BUFFER_NOT_EMPTY`

  UUDT receiving buffer is not empty (UUDT = **U**nacknowledged **U**nsegmented **D**ata **T**ransfer)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Diag_GetState_Async

G_Can_Diag_GetState_Async — Query the diagnostics status (by asynchronous query)

Description

This command starts the asynchronous query of the current diagnostics status. If a response is available, the callback function given with G_Can_Diag_GetState_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_API_CAN_DLL G_Error_t
G_Can_Diag_GetState_Async(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Can_Diag_GetState_CmdFlags_t  cmdFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`cmdFlags`
Variable of type **G_Can_Diag_GetState_CmdFlags_t**

The following flags are available:

- `G_CAN__DIAG__GET_STATE__CMD_FLAG__NONE`

  No flag, if no other has to be transferred
- `G_CAN__DIAG__GET_STATE__CMD_FLAG__RESET_LAST_ERROR`

  Reset the error code of the last error that has occurred in the firmware

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Diag_GetState_Async_Callback

G_Can_Diag_GetState_Async_Callback — Callback function for the asynchronous query of the current diagnostics status

Description

This function will be automatically called if a response for the asynchronous query of the current diagnostics status with G_Can_Diag_GetState_Async is available. It must be entered by the user and set with G_Can_Diag_GetState_Async_AddCallback .

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Can_Diag_GetState_Async_Callback(
 const G_PortHandle_t  portHandle,
 const u8_t channel,
 const G_Error_t  lastErrorCode,
 const G_Can_Diag_Type_t  type,
 const G_Can_Diag_State_t  state,
 const G_Can_Diag_GetState_RspFlags_t  rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`lastErrorCode`
Error code of the last error that has occurred in the firmware. For getting an error description, you can call G_Common_GetFirmwareErrorDescription.

`type`
Diagnostic type

The following values are available:

- `G_CAN__DIAG__TYPE__OFF` - No diagnostics
- `G_CAN__DIAG__TYPE__KW2000_TP_1_6` - KWP2000 on TP1.6
- `G_CAN__DIAG__TYPE__KW2000_TP_2_0` - KWP2000 on TP2.0
- `G_CAN__DIAG__TYPE__KW2000_ISOTP` - KWP2000 on ISOTP
- `G_CAN__DIAG__TYPE__GMLAN` - GMLAN
- `G_CAN__DIAG__TYPE__UDS_ISOTP` - UDS on ISOTP
- `G_CAN__DIAG__TYPE__J1939` - J1939

`state`
Current diagnostics status

The following values are available:
- `G_CAN__DIAG__STATE__NOT_INITIALIZED`

  Not initialized
- `G_CAN__DIAG__STATE__DISCONNECTED`

  Disconnected
- `G_CAN__DIAG__STATE__CONNECT_IN_PROGRESS`

  Connecting is in progress
- `G_CAN__DIAG__STATE__CONNECTED`

  Connected
- `G_CAN__DIAG__STATE__DISCONNECT_IN_PROGRESS`

  Disconnecting is in progress

`rspFlags`
Return flags

The following return flags are available:
- `G_CAN__DIAG__GET_STATE__RSP_FLAG__NONE`

  No flag was set
- `G_CAN__DIAG__GET_STATE__RSP_FLAG__BUSY`

  A request has not been responded yet/ successfully sent
- `G_CAN__DIAG__GET_STATE__RSP_FLAG__SYNC_RX_BUFFER_NOT_EMPTY`

  Synchronous receiving buffer is not empty
- `G_CAN__DIAG__GET_STATE__RSP_FLAG__ASYNC_RX_BUFFER_NOT_EMPTY`

  Asynchronous receiving buffer is not empty
- `G_CAN__DIAG__GET_STATE__RSP_FLAG__UUDT_RX_BUFFER_NOT_EMPTY`

  UUDT receiving buffer is not empty (UUDT = **U**nacknowledged **U**nsegmented **D**ata **T**ransfer)

G_Can_Diag_GetState_Async_AddCallback

G_Can_Diag_GetState_Async_AddCallback — Set the callback function for the asynchronous query of the current diagnostics status

Description

This command defines a callback function for the asynchronous query of the current diagnostics status with G_Can_Diag_GetState_Async.

Definition

```
G_API_CAN_DLL G_Error_t
G_Can_Diag_GetState_Async_AddCallback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Can_Diag_GetState_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`callback`
Function pointer of the callback function (see G_Can_Diag_GetState_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Diag_GetState_Async_RemoveCallback

G_Can_Diag_GetState_Async_RemoveCallback — Remove the callback function for the asynchronous query of the current diagnostics status

Description

This command removes the callback function for the asynchronous query of the current diagnostics status.

Definition

```
G_API_CAN_DLL G_Error_t
G_Can_Diag_GetState_Async_RemoveCallback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Diag_Config_GmLan

G_Can_Diag_Config_GmLan — Configure the diagnostics

Description

This command configures the diagnostics for GMLAN.

Definition

```
G_Error_t
G_Can_Diag_Config_GmLan(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Can_Diag_Config_Mode_t mode,
 const u32_t globalTimeout,
 const G_Can_Diag_Config_CmdFlags_t cmdFlags,
 const G_Can_Diag_Config_GmLan_Parameters_t * const gmLanParameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`mode`
Variable of type **G_Can_Diag_Config_Mode_t**

The following values are available:

- `G_CAN__DIAG__CONFIG__MODE__SET_DEFAULT_PARAMS`

  Set standard parameters, with initialization
- `G_CAN__DIAG__CONFIG__MODE__SET_PARAMS`

  Set parameters, with initialization
- `G_CAN__DIAG__CONFIG__MODE__SET_FLAGS_ONLY_WITHOUT_INIT`

  Only global timeout and set flags, without initialization
- `G_CAN__DIAG__CONFIG__MODE__SET_PARAMS_WITHOUT_INIT`

  Set parameters, without initialization

`globalTimeout`
Global timeout in milliseconds (starts directly before sending the request and stops after complete receiving of the response or after sending the request successfully if no response is expected, e.g. 10000 milliseconds)

`cmdFlags`
Variable of type **G_Can_Diag_Config_CmdFlags_t**

The following flags are available:

- `G_CAN__DIAG__CONFIG__CMD_FLAG__NONE`

  No flag, if no other has to be transferred

- G_CAN__DIAG__CONFIG__CMD_FLAG__DISABLE_21_HANDLING

  The negative response BusyRepeatRequest is not treated
- G_CAN__DIAG__CONFIG__CMD_FLAG__DISABLE_23_HANDLING

  The negative response RoutineNotComplete is not treated
- G_CAN__DIAG__CONFIG__CMD_FLAG__DISABLE_78_HANDLING

  The negative response RequestCorrectlyReceivedResponsePending is not treated
- G_CAN__DIAG__CONFIG__CMD_FLAG__ALL_RESPONSES_AS_SYNC

  Also unexpectedly received diagnostics responses are written to the normal diagnostics receiving buffer

gmLanParameters
Pointer to a variable of type **G_Can_Diag_Config_GmLan_Parameters_t**

The structure **G_Can_Diag_Config_GmLan_Parameters_t** consists of the following items:

- P2max
  Timeout given in milliseconds (maximum time between the end of the request and the beginning of the response, e.g. 200 milliseconds)
- P3max
  Timeout given in milliseconds

  (maximum time between the end of the request and the beginning of the response while 0x78: RequestCorrectlyReceivedResponsePending, e.g. 5100 milliseconds)
- Repetitions
  Number of request repetitions if the control unit does not react within the timeout time (P2max or P3max), e.g. 2
- reserved1
  Reserved byte, must be completed with 0
- reserved2
  Reserved byte, must be completed with 0
- TesterPresent
  Structure of tester present

  The structure consists of the following items:
  - Mode
    Variable of type **G_Can_Diag_TesterPresent_Mode_t**

    indicating the tester present mode

    The following values are available:
    - G_CAN__DIAG__TESTER_PRESENT__MODE__OFF - Deactivated
    - G_CAN__DIAG__TESTER_PRESENT__MODE__PHYSICAL - Physical
    - G_CAN__DIAG__TESTER_PRESENT__MODE__FUNCTIONAL - Functional
  - ResponseMode
    Variable of type **G_Can_Diag_TesterPresent_RspMode_t** indicating the response mode

    The following values are available:
    - G_CAN__DIAG__TESTER_PRESENT__RSP_MODE__NO_RSP_REQUIRED

      No response required
    - G_CAN__DIAG__TESTER_PRESENT__RSP_MODE__RSP_REQUIRED

      Response required
  - CycleTime
    Cycle time in milliseconds, e.g. 5100 milliseconds
  - reserved1
    Reserved byte, must be initialized with 0

- `reserved2`
  Reserved byte, must be initialized with `0`
- `reserved3`
  Reserved byte, must be initialized with `0`
- `DataLength`
  Number of data bytes
- `Data`
  Buffer with data bytes (0..7). When declaring a variable of type **G_Can_Diag_Config_GmLan_Parameters_t** this buffer is set in size of `8` bytes.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Diag_Config_J1939

G_Can_Diag_Config_J1939 — Configuration

Description

This command configures the diagnostics for J1939.

Definition

```
G_Error_t
G_Can_Diag_Config_J1939(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Can_Diag_Config_Mode_t mode,
 const u32_t globalTimeout,
 const G_Can_Diag_Config_CmdFlags_t cmdFlags,
 const G_Can_Diag_Config_J1939_Parameters_t * const j1939Parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`mode`
Variable of type **G_Can_Diag_Config_Mode_t**

The following values are available:

- `G_CAN__DIAG__CONFIG__MODE__SET_DEFAULT_PARAMS`

  Set standard parameters, with initialization
- `G_CAN__DIAG__CONFIG__MODE__SET_PARAMS`

  Set parameters, with initialization
- `G_CAN__DIAG__CONFIG__MODE__SET_FLAGS_ONLY_WITHOUT_INIT`

  Only global timeout and set flags, without initialization
- `G_CAN__DIAG__CONFIG__MODE__SET_PARAMS_WITHOUT_INIT`

  Set parameters, without initialization

`globalTimeout`
Global timeout in milliseconds
(starts directly before sending the request and stops after complete receiving of the response or after sending the request successfully if no response is expected, e.g. 10000 milliseconds)

`cmdFlags`
Variable of type **G_Can_Diag_Config_CmdFlags_t**

The following flags are available:

- `G_CAN__DIAG__CONFIG__CMD_FLAG__NONE`

  No flag, if no other has to be transferred
- `G_CAN__DIAG__CONFIG__CMD_FLAG__DISABLE_21_HANDLING`

The negative response BusyRepeatRequest is not treated
- G_CAN__DIAG__CONFIG__CMD_FLAG__DISABLE_23_HANDLING

The negative response RoutineNotComplete is not treated
- G_CAN__DIAG__CONFIG__CMD_FLAG__DISABLE_78_HANDLING

The negative response RequestCorrectlyReceivedResponsePending is not treated
- G_CAN__DIAG__CONFIG__CMD_FLAG__ALL_RESPONSES_AS_SYNC

Also unexpectedly received diagnostics responses are written to the normal diagnostics receiving buffer

j1939Parameters
Pointer to a variable of type **G_Can_Diag_Config_J1939_Parameters_t**

The structure **G_Can_Diag_Config_J1939_Parameters_t** consists of the following items:
- T1Timeout
  Timeout given in microseconds (maximum time between the end of the request and the beginning of the response, e.g. 500000 µs)
- T2Timeout
  Timeout given in microseconds (time between the reception of a "Busy"-response and the resending of this request, e.g. 250000 µs)
- Repetitions
  Number of request repetitions, if the control unit does not react within the timeout time (e.g. 2)
- reserved1
  Reserved byte, must be initialized with 0
- reserved2
  Reserved byte, must be initialized with 0
- TesterPresent
  Structure for tester present

  The structure consists of the following items:
  - Mode
    Variable of type **G_Can_Diag_TesterPresent_Mode_t** indicating the tester present mode

    The following values are available:
    - G_CAN__DIAG__TESTER_PRESENT__MODE__OFF - Deactivated
    - G_CAN__DIAG__TESTER_PRESENT__MODE__PHYSICAL - Physical
    - G_CAN__DIAG__TESTER_PRESENT__MODE__FUNCTIONAL - Functional
  - ResponseMode
    Variable of type **G_Can_Diag_TesterPresent_RspMode_t** indicating the response mode

    The following values are available:
    - G_CAN__DIAG__TESTER_PRESENT__RSP_MODE__NO_RSP_REQUIRED

      No response required
    - G_CAN__DIAG__TESTER_PRESENT__RSP_MODE__RSP_REQUIRED

      Response required
  - DataLength
    Number of data bytes
  - reserved1
    Reserved byte, must be initialized with 0
  - CycleTime
    Cycle time in microseconds
  - Data
    Buffer with data bytes (0..10). When declaring a variable of type **G_Can_Diag_Config_J1939_Parameters_t** this buffer is set in size of 11 bytes.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Diag_Config_Kw2000_IsoTp

G_Can_Diag_Config_Kw2000_IsoTp — Configure the diagnostics

Description

This command configures the diagnostics for KWP2000 on ISOTP.

Definition

```
G_Error_t
G_Can_Diag_Config_Kw2000_IsoTp(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Can_Diag_Config_Mode_t mode,
 const u32_t globalTimeout,
 const G_Can_Diag_Config_CmdFlags_t cmdFlags,
 const G_Can_Diag_Config_Kw2000_IsoTp_Parameters_t \
 * const kw2000_IsoTpParameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`mode`
Variable of type **G_Can_Diag_Config_Mode_t**

The following values are available:

- `G_CAN__DIAG__CONFIG__MODE__SET_DEFAULT_PARAMS`

  Set standard parameters, with initialization
- `G_CAN__DIAG__CONFIG__MODE__SET_PARAMS`

  Set parameters, with initialization
- `G_CAN__DIAG__CONFIG__MODE__SET_FLAGS_ONLY_WITHOUT_INIT`

  Only global timeout and set flags, without initialization
- `G_CAN__DIAG__CONFIG__MODE__SET_PARAMS_WITHOUT_INIT`

  Set parameters, without initialization

`globalTimeout`
Global timeout in milliseconds (starts directly before sending the request and stops after complete receiving of the response or after sending the request successfully if no response is expected, e.g. 10000 milliseconds)

`cmdFlags`
Variable of type **G_Can_Diag_Config_CmdFlags_t**

The following flags are available:
- `G_CAN__DIAG__CONFIG__CMD_FLAG__NONE`

  No flag, if no other has to be transferred
- `G_CAN__DIAG__CONFIG__CMD_FLAG__DISABLE_21_HANDLING`

  The negative response `BusyRepeatRequest` is not treated
- `G_CAN__DIAG__CONFIG__CMD_FLAG__DISABLE_23_HANDLING`

  The negative response `RoutineNotComplete` is not treated
- `G_CAN__DIAG__CONFIG__CMD_FLAG__DISABLE_78_HANDLING`

  The negative response `RequestCorrectlyReceivedResponsePending` is not treated
- `G_CAN__DIAG__CONFIG__CMD_FLAG__ALL_RESPONSES_AS_SYNC`

  Also unexpectedly received diagnostics responses are written to the normal diagnostics receiving buffer

`kw2000_IsoTpParameters`
Pointer to a variable of type **G_Can_Diag_Config_Kw2000_IsoTp_Parameters_t**

The structure **G_Can_Diag_Config_Kw2000_IsoTp_Parameters_t** consists of the following items:
- `P2max`
  Timeout given in milliseconds (maximum time between the end of the request and the beginning of the response, e.g. 200 milliseconds)
- `P3max`
  Timeout given in milliseconds (maximum time between the end of the request and the beginning of the response while `0x78: RequestCorrectlyReceivedResponsePending`, e.g. 5000 milliseconds)
- `Repetitions`
  Number of request repetitions if the control unit does not react within the timeout time (P2max or P3max) (e.g. 2)
- `reserved1`
  Reserved byte, must be initialized with `0`
- `reserved2`
  Reserved byte, must be initialized with `0`
- `TesterPresent`
  Structure for tester present

  The structure consists of the following items:
  - `Mode`
    Variable of type **G_Can_Diag_TesterPresent_Mode_t** indicating the tester present mode

    The following values are available:
    - `G_CAN__DIAG__TESTER_PRESENT__MODE__OFF` - Deactivated
    - `G_CAN__DIAG__TESTER_PRESENT__MODE__PHYSICAL` - Physical
    - `G_CAN__DIAG__TESTER_PRESENT__MODE__FUNCTIONAL` - Functional
  - `ResponseMode`
    Variable of type **G_Can_Diag_TesterPresent_RspMode_t** indicating the response mode

    The following values are available:
    - `G_CAN__DIAG__TESTER_PRESENT__RSP_MODE__NO_RSP_REQUIRED`

      No response required
    - `G_CAN__DIAG__TESTER_PRESENT__RSP_MODE__RSP_REQUIRED`

      Response required

- `CycleTime`
  Cycle time in milliseconds, e.g. 1000 milliseconds
- `reserved1`
  Reserved byte, must be initialized with `0`
- `reserved2`
  Reserved byte, must be initialized with `0`
- `reserved3`
  Reserved byte, must be initialized with `0`
- `DataLength`
  Number of data bytes
- `Data`
  Buffer with data bytes (0..7). When declaring a variable of type **G_Can_Diag_Config_Kw2000_IsoTp_Parameters_t**, this buffer is set in size of `8` bytes.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Diag_Config_Kw2000_Tp_1_6

G_Can_Diag_Config_Kw2000_Tp_1_6 — Configure the diagnostics

Description

This command configures the diagnostics for KWP2000 on TP1.6.

Definition

```
G_Error_t
G_Can_Diag_Config_Kw2000_Tp_1_6(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Can_Diag_Config_Mode_t mode,
 const u32_t globalTimeout,
 const G_Can_Diag_Config_CmdFlags_t cmdFlags,
 const G_Can_Diag_Config_Kw2000_Tp_1_6_Parameters_t \
 * const kw2000_Tp_1_6Parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`mode`
Variable of type **G_Can_Diag_Config_Mode_t**

The following values are available:

- `G_CAN__DIAG__CONFIG__MODE__SET_DEFAULT_PARAMS`

  Set standard parameters, with initialization
- `G_CAN__DIAG__CONFIG__MODE__SET_PARAMS`

  Set parameters, with initialization
- `G_CAN__DIAG__CONFIG__MODE__SET_FLAGS_ONLY_WITHOUT_INIT`

  Only global timeout and set flags, without initialization
- `G_CAN__DIAG__CONFIG__MODE__SET_PARAMS_WITHOUT_INIT`

  Set parameters, without initialization

`globalTimeout`
Global timeout in milliseconds (starts directly before sending the request and stops after complete receiving of the response or after sending the request successfully if no response is expected, e.g. 10000 milliseconds)

`cmdFlags`
Variable of type **G_Can_Diag_Config_CmdFlags_t**

The following flags are available:

- `G_CAN__DIAG__CONFIG__CMD_FLAG__NONE`

  No flag, if no other has to be transferred
- `G_CAN__DIAG__CONFIG__CMD_FLAG__DISABLE_21_HANDLING`

  The negative response `BusyRepeatRequest` is not treated
- `G_CAN__DIAG__CONFIG__CMD_FLAG__DISABLE_23_HANDLING`

  The negative response `RoutineNotComplete` is not treated
- `G_CAN__DIAG__CONFIG__CMD_FLAG__DISABLE_78_HANDLING`

  The negative response `RequestCorrectlyReceivedResponsePending` is not treated
- `G_CAN__DIAG__CONFIG__CMD_FLAG__ALL_RESPONSES_AS_SYNC`

  Also unexpectedly received diagnostics responses are written to the normal diagnostics receiving buffer

`kw2000_Tp_1_6Parameters`
Pointer to a variable of type **G_Can_Diag_Config_Kw2000_Tp_1_6_Parameters_t**

The structure **G_Can_Diag_Config_Kw2000_Tp_1_6_Parameters_t** consists of the following items:

- `P2max`
  Timeout given in milliseconds (maximum time between the end of the request and the beginning of the response, e.g. 200 milliseconds)
- `Repetitions`
  Number of request repetitions if the control unit does not react within the timeout time (P2max) (e.g. 2)
- `AddressWord`
  Address word for excitation (control unit address + parity bit)
- `TargetAddress`
  Target address
- `SourceAddress`
  Source address
- `reserved1`
  Reserved byte, must be initialized with `0`
- `TesterPresentCycle`
  Cycle time for tester present in milliseconds (only this parameter of the tester present service can be modified, e.g. 2000 milliseconds)
- `reserved2`
  Reserved byte, must be initialized with `0`
- `reserved3`
  Reserved byte, must be initialized with `0`

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Diag_Config_Kw2000_Tp_2_0

G_Can_Diag_Config_Kw2000_Tp_2_0 — Configure the diagnostics

Description

This command configures the diagnostics for KWP2000 on TP2.0.

Definition

```
G_Error_t
G_Can_Diag_Config_Kw2000_Tp_2_0(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Can_Diag_Config_Mode_t mode,
 const u32_t globalTimeout,
 const G_Can_Diag_Config_CmdFlags_t cmdFlags,
 const u16_t p2max,
 const u16_t repetitions
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`mode`
Variable of type **G_Can_Diag_Config_Mode_t**

The following values are available:

- `G_CAN__DIAG__CONFIG__MODE__SET_DEFAULT_PARAMS`

  Set standard parameters, with initialization
- `G_CAN__DIAG__CONFIG__MODE__SET_PARAMS`

  Set parameters, with initialization
- `G_CAN__DIAG__CONFIG__MODE__SET_FLAGS_ONLY_WITHOUT_INIT`

  Only global timeout and set flags, without initialization
- `G_CAN__DIAG__CONFIG__MODE__SET_PARAMS_WITHOUT_INIT`

  Set parameters, without initialization

`globalTimeout`
Global timeout in milliseconds (starts directly before sending the request and stops after complete receiving of the response or after sending the request successfully if no response is expected, e.g. 10000 milliseconds)

`cmdFlags`
Variable of type **G_Can_Diag_Config_CmdFlags_t**

The following flags are available:

- `G_CAN__DIAG__CONFIG__CMD_FLAG__NONE`

  No flag, if no other has to be transferred
- `G_CAN__DIAG__CONFIG__CMD_FLAG__DISABLE_21_HANDLING`

  The negative response `BusyRepeatRequest` is not treated
- `G_CAN__DIAG__CONFIG__CMD_FLAG__DISABLE_23_HANDLING`

  The negative response `RoutineNotComplete` is not treated
- `G_CAN__DIAG__CONFIG__CMD_FLAG__DISABLE_78_HANDLING`

  The negative response `RequestCorrectlyReceivedResponsePending` is not treated
- `G_CAN__DIAG__CONFIG__CMD_FLAG__ALL_RESPONSES_AS_SYNC`

  Also unexpectedly received diagnostics responses are written to the normal diagnostics receiving buffer

`p2max`
Timeout given in milliseconds (maximum time between the end of the request and the beginning of the response, e.g. 600 milliseconds)

`repetitions`
Number of request repetitions if the control unit does not react within the timeout time (p2max) (e.g. 2)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Diag_Config_Uds_IsoTp

G_Can_Diag_Config_Uds_IsoTp — Configure the diagnostics

Description

This command configures the diagnostics for UDS on ISOTP.

Definition

```
G_Error_t
G_Can_Diag_Config_Uds_IsoTp(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Can_Diag_Config_Mode_t mode,
 const u32_t globalTimeout,
 const G_Can_Diag_Config_CmdFlags_t cmdFlags,
 const G_Can_Diag_Config_Uds_IsoTp_Parameters_t \
 * const uds_IsoTpParameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`mode`
Variable of type **G_Can_Diag_Config_Mode_t**

The following values are available:

- `G_CAN__DIAG__CONFIG__MODE__SET_DEFAULT_PARAMS`

  Set standard parameters, with initialization
- `G_CAN__DIAG__CONFIG__MODE__SET_PARAMS`

  Set parameters, with initialization
- `G_CAN__DIAG__CONFIG__MODE__SET_FLAGS_ONLY_WITHOUT_INIT`

  Only global timeout and set flags, without initialization
- `G_CAN__DIAG__CONFIG__MODE__SET_PARAMS_WITHOUT_INIT`

  Set parameters, without initialization

`globalTimeout`
Global timeout in milliseconds (starts directly before sending the request and stops after complete receiving of the response or after sending the request successfully if no response is expected, e.g. **10000 milliseconds**)

`cmdFlags`
Variable of type **G_Can_Diag_Config_CmdFlags_t**

The following flags are available:

- `G_CAN__DIAG__CONFIG__CMD_FLAG__NONE`

  No flag, if no other has to be transferred

- G_CAN__DIAG__CONFIG__CMD_FLAG__DISABLE_21_HANDLING

  The negative response BusyRepeatRequest is not treated
- G_CAN__DIAG__CONFIG__CMD_FLAG__DISABLE_23_HANDLING

  The negative response RoutineNotComplete is not treated
- G_CAN__DIAG__CONFIG__CMD_FLAG__DISABLE_78_HANDLING

  The negative response RequestCorrectlyReceivedResponsePending is not treated
- G_CAN__DIAG__CONFIG__CMD_FLAG__ALL_RESPONSES_AS_SYNC

  Also unexpectedly received diagnostics responses are written to the normal diagnostics receiving buffer

uds_IsoTpParameters
Pointer to a variable of type **G_Can_Diag_Config_Uds_IsoTp_Parameters_t**

The structure **G_Can_Diag_Config_Uds_IsoTp_Parameters_t** consists of the following items:
- P2max
  Timeout given in milliseconds

  (maximum time between the end of the request and the beginning of the response, e.g. 200 milliseconds)
- P3max
  Timeout given in milliseconds

  (maximum time between the end of the request and the beginning of the response while 0x78: RequestCorrectlyReceivedResponsePending, e.g. 5100 milliseconds)
- Repetitions
  Number of request repetitions if the control unit does not react within the timeout time (P2max or P3max), e.g. 2
- reserved1
  Reserved byte, must be completed with 0
- reserved2
  Reserved byte, must be completed with 0
- TesterPresent
  Structure for tester present

  The structure consists of the following items:
  - Mode
    Variable of type **G_Can_Diag_TesterPresent_Mode_t** indicating the tester present mode

    The following values are available:
    - G_CAN__DIAG__TESTER_PRESENT__MODE__OFF - Deactivated
    - G_CAN__DIAG__TESTER_PRESENT__MODE__PHYSICAL - Physical
    - G_CAN__DIAG__TESTER_PRESENT__MODE__FUNCTIONAL - Functional
  - ResponseMode
    Variable of type **G_Can_Diag_TesterPresent_RspMode_t** indicating the response mode

    The following values are available:
    - G_CAN__DIAG__TESTER_PRESENT__RSP_MODE__NO_RSP_REQUIRED

      No response required
    - G_CAN__DIAG__TESTER_PRESENT__RSP_MODE__RSP_REQUIRED

      Response required
  - reserved1
    Reserved byte, must be initialized with 0
  - reserved2
    Reserved byte, must be initialized with 0

- `reserved3`
  Reserved byte, must be initialized with `0`
- `DataLength`
  Number of data bytes
- `Data`
  Buffer with data bytes (0..7). When declaring a variable of type **G_Can_Diag_Config_Uds_IsoTp_Parameters_t** this buffer is set in size of `8` bytes.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Diag_Config_Off

G_Can_Diag_Config_Off — Deactivate the diagnostics

Description

This command deactivates the diagnostics.

Definition

```
G_Error_t
G_Can_Diag_Config_Off(
 const G_PortHandle_t portHandle,
 const u8_t channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

6 Monitor

These commands are used to control the monitor.

G_Can_Monitor_DefineFilter

G_Can_Monitor_DefineFilter — Definition of the receiving filter

Description

This command defines the monitor receiving filter.

This command enables to detect selected identifiers by means of the CAN monitor. If the filter has been activated, all identifiers between `startId` and `endId` are filtered. In the case of a deactivated filter, all identifiers "go through". In the case that only ONE identifier has to be filtered, please use for `mode` = `G_CAN__MONITOR__DEFINE_FILTER__MODE__RANGE` `startId` equal `endId`.

Definition

```
G_Error_t
G_Can_Monitor_DefineFilter(
 const G_PortHandle_t portHandle,
 const G_Can_Monitor_DefineFilter_Mode_t mode,
 const u32_t startId,
 const u32_t endId
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
Variable of type **G_Can_Monitor_DefineFilter_Mode_t** indicating the filter mode

The following values are available:

- `G_CAN__MONITOR__DEFINE_FILTER__MODE__OFF`

  No filter
- `G_CAN__MONITOR__DEFINE_FILTER__MODE__RANGE`

  Filter the range
- `G_CAN__MONITOR__DEFINE_FILTER__MODE__ADD_RANGE`

  Add the range
- `G_CAN__MONITOR__DEFINE_FILTER__MODE__REMOVE_RANGE`

  Remove the range
- `G_CAN__MONITOR__DEFINE_FILTER__MODE__REMOVE_ALL`

  Remove all filter ranges

`startId`
Identifier with that the range begins

`endId`
Identifier with that the range ends

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Monitor_BufferMode_Start

G_Can_Monitor_BufferMode_Start — Start the monitor

Description

This command starts the monitor in the "BufferMode".

In the mode "BufferMode", the messages come in in succession into an internal ring buffer after passing the monitor filter as sent on the bus/ received by the bus. The size of this internal ring buffer is given by `bufferSize`.

Definition

```
G_Error_t
G_Can_Monitor_BufferMode_Start(
 const G_PortHandle_t portHandle,
 const G_Can_Monitor_BufferMode_t mode,
 const u32_t bufferSize
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
Variable of type **G_Can_Monitor_BufferMode_Mode_t** indicating the messages coming in into the ring buffer

The following values are available:

- `G_CAN__MONITOR__BUFFER_MODE__MODE__NOTHING`

  No messages
- `G_CAN__MONITOR__BUFFER_MODE__MODE__RX`

  RX (Received messages)
- `G_CAN__MONITOR__BUFFER_MODE__MODE__TX`

  TX (Transmitted messages)
- `G_CAN__MONITOR__BUFFER_MODE__MODE__TX_AND_RX`

  RX + TX (Received and transmitted messages)
- `G_CAN__MONITOR__BUFFER_MODE__MODE__ERROR_FRAME`

  Error frames
- `G_CAN__MONITOR__BUFFER_MODE__MODE__ERROR_FRAME_AND_RX`

  Error frames + RX
- `G_CAN__MONITOR__BUFFER_MODE__MODE__ERROR_FRAME_AND_TX`

  Error frames + TX
- `G_CAN__MONITOR__BUFFER_MODE__MODE__ERROR_FRAME_AND_TX_AND_RX`

  Error frames + TX + RX

`bufferSize`
Buffer size of the ring buffer in bytes (e.g. 20000), or **`0`** for default size

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Monitor_BufferMode_Start2

G_Can_Monitor_BufferMode_Start2 — Start CAN monitor in buffer mode

Description

Use this command to start the CAN monitor in buffer mode.

This command provides extended features over G_Can_Monitor_BufferMode_Start .

Definition

```
G_Error_t
G_Can_Monitor_BufferMode_Start2(
 const G_PortHandle_t portHandle,
 const G_Can_Monitor_BufferMode_Start2_CmdFlags_t  cmdFlags,
 const u32_t numberOfItems
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_CAN__MONITOR__BUFFER_MODE__START_2__CMD_FLAG__NONE`
No flag is set

`G_CAN__MONITOR__BUFFER_MODE__START_2__CMD_FLAG__RX`
Monitor received frames

`G_CAN__MONITOR__BUFFER_MODE__START_2__CMD_FLAG__TX`
Monitor sent frames

`G_CAN__MONITOR__BUFFER_MODE__START_2__CMD_FLAG__ERROR_FRAME`
Monitor error frames

`G_CAN__MONITOR__BUFFER_MODE__START_2__CMD_FLAG__LARGE_ITEMS`
Monitor large items (message data > 8 bytes)

If this flag is set, use G_Can_Monitor_BufferMode_GetItems_Large to query monitor items.

`G_CAN__MONITOR__BUFFER_MODE__START_2__CMD_FLAG__FD`
Monitor **CAN FD** frames

`G_CAN__MONITOR__BUFFER_MODE__START_2__CMD_FLAG__LARGE_ITEMS` has to be set.

`G_CAN__MONITOR__BUFFER_MODE__START_2__CMD_FLAG__DIRECT_READ`
The monitor items are read directly from the device memory and therefore are not buffer in G-API layer.

Use this mode if you experience excessive bus traffic because of error frames that renders the device unresponsible.

`numberOfItems`
Size of the internal monitor buffer in number of items

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Monitor_BufferMode_Stop

G_Can_Monitor_BufferMode_Stop — Stop the monitor

Description

This command stops the monitor in the “BufferMode”.

Definition

```
G_Error_t
G_Can_Monitor_BufferMode_Stop(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Monitor_BufferMode_GetItems

G_Can_Monitor_BufferMode_GetItems — Request of the buffer entries

Description

This command requests the monitor buffer entries.

Definition

```
G_Error_t
G_Can_Monitor_BufferMode_GetItems(
 const G_PortHandle_t portHandle,
 const u32_t bufferSize,
 u32_t * const numberOfItems,
 G_Can_Monitor_BufferItem_t * const bufferItems
);
```

Parameters

`portHandle`
Handle to the communication port

`bufferSize`
Size of buffer `bufferItems` in bytes

`numberOfItems`
Returns the number of items returned in `bufferItems`

`bufferItems`
Returns buffer items

All entries of this type consist of the following items:

- `Timestamp`
  Time stamp in ns
- `Id`
  Identifier
- `Flags`
  Variable of type **G_Can_Monitor_BufferItem_Flags_t**

  The following return flags are available:
  - `G_CAN__MONITOR__BUFFER_ITEM__FLAG__NONE`

    No flag was set
  - `G_CAN__MONITOR__BUFFER_ITEM__FLAG__EXTENDED_ID`

    29 bit identifiers
  - `G_CAN__MONITOR__BUFFER_ITEM__FLAG__TRANSMIT`

    Transmitted message (TX)
  - `G_CAN__MONITOR__BUFFER_ITEM__FLAG__ERROR_FRAME`

    Error frame
  - `G_CAN__MONITOR__BUFFER_ITEM__FLAG__EVENT`

    Event
  - `G_CAN__MONITOR__BUFFER_ITEM__FLAG__BUFFER_OVERRUN`

    Monitor buffer overrun of the firmware

- `G_CAN__MONITOR__BUFFER_ITEM__FLAG__DLL_BUFFER_OVERRUN`

  Monitor buffer overrun in the API

  

  If this flag is set, the buffer size given by G_Can_Monitor_BufferMode_Start was too small for the incoming entries.

- `Dlc`
  Data length code (see DLC to data length)
- `DataLength`
  Data length in bytes
- `Data`
  Buffer with data bytes - sized of `8` bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Query of entries with **G_Can_Monitor_BufferMode_GetItems**

```
u32_t numberOfItems;
G_Can_Monitor_BufferItem_t bufferItems[102];
u32_t bufferSize;

…

bufferSize = sizeof(bufferItems);

rc =
 G_Can_Monitor_BufferMode_GetItems(
  portHandle,
  bufferSize,
  &numberOfItems,
  bufferItems
 );

if (rc == G_NO_ERROR) {

…

}
```

G_Can_Monitor_BufferMode_GetItems_Large

G_Can_Monitor_BufferMode_GetItems_Large — Get large monitor items

Description

Use this command to query large monitor items from the buffer monitor.

To use large buffer monitor items, the monitor has to be started with G_Can_Monitor_BufferMode_Start2 and flag `G_CAN__MONITOR__BUFFER_MODE__START_2__CMD_FLAG__LARGE_ITEMS` has to be set.

Definition

```
G_Error_t
G_Can_Monitor_BufferMode_GetItems_Large(
 const G_PortHandle_t  portHandle,
 G_Can_Monitor_BufferMode_GetItems_Large_RspFlags_t * const  rspFlags,
 u32_t  * const numberOfItems,
 G_Can_Monitor_BufferItem_Large_t  * const items
);
```

Parameters

`portHandle`
Handle to the communication port

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_CAN__MONITOR__BUFFER_MODE__GET_ITEMS__LARGE__RSP_FLAG__NONE`
No flag is set

`G_CAN__MONITOR__BUFFER_MODE__GET_ITEMS__LARGE__RSP_FLAG__NOT_EMPTY`
There are still items left in the monitor buffer

`numberOfItems`
in : size of `items` in number of items

out : number of returned items

`items`
Returns monitor items

The structure contains the following elements:

`Timestamp`
Timestamp in nanoseconds

`Id`
CAN identifier of the frame

`Flags`
Item flags

The following values are possible and can be combined:

`G_CAN__MONITOR__BUFFER_ITEM__LARGE__FLAG__NONE`
No flag is set

`G_CAN__MONITOR__BUFFER_ITEM__LARGE__FLAG__EXTENDED_ID`
The frame has an extendend identifier

`G_CAN__MONITOR__BUFFER_ITEM__LARGE__FLAG__TRANSMIT`
This is a transmitted frame

`G_CAN__MONITOR__BUFFER_ITEM__LARGE__FLAG__ERROR_FRAME`
This is an error frame

`G_CAN__MONITOR__BUFFER_ITEM__LARGE__FLAG__FDF`
The frame is a **CAN FD Frame**

`G_CAN__MONITOR__BUFFER_ITEM__LARGE__FLAG__EVENT`
This is a monitor event

`G_CAN__MONITOR__BUFFER_ITEM__LARGE__FLAG__BRS`
The **Bit Rate Switch** bit in the frame header is set

`G_CAN__MONITOR__BUFFER_ITEM__LARGE__FLAG__ESI`
The **Error State Indicator** bit in the frame header is set

`G_CAN__MONITOR__BUFFER_ITEM__LARGE__FLAG__BUFFER_OVERRUN`
A buffer overrun occurred (frames were lost due to a too small buffer)

`reserved`
reserved parameter (must be initialized with **0**)

`Dlc`
Data length code of the frame (see DLC to data length)

`DataLength`
Data length of the frame in bytes

`Data`
Frame data bytes (0..64)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Monitor_BufferMode_EnableEvent

G_Can_Monitor_BufferMode_EnableEvent — Enable monitor event

Description

This command enables the monitor to signal the event specified by `eventHandle` as soon as monitor data is received.

For general notes on **G-API** event handling, see Section 5, "Events".

Definition

```
G_Error_t
G_Can_Monitor_BufferMode_EnableEvent(
 const G_PortHandle_t portHandle,
 const G_Common_EventHandle_t * const eventHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

`eventHandle`
: Handle to the event

  Each time monitor data is received, the specified handle is signaled.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Monitor_BufferMode_DisableEvent

G_Can_Monitor_BufferMode_DisableEvent — Disable monitor event

Description

This command disables a monitor event.

After this function has been executed, no event will be signaled when new monitor data is received.

For general notes on **G-API** event handling, see Section 5, "Events".

Definition

```
G_Error_t
G_Can_Monitor_BufferMode_DisableEvent(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Monitor_ListMode_Start

G_Can_Monitor_ListMode_Start — Start the monitor

Description

This command starts the monitor in “ListMode”.

In the case of “ListMode” a list entry exists for each identifier. This list entry is updated when receiving/transmitting the identifier. At any time it can be requested specificly by giving the identifier.

Currently the “List reception” is available only for 11 bit identifiers.

Definition

```
G_Error_t
G_Can_Monitor_ListMode_Start(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Monitor_ListMode_Stop

G_Can_Monitor_ListMode_Stop — Stop the monitor

Description

This command stops the monitor in the “ListMode”.

Definition

```
G_Error_t
G_Can_Monitor_ListMode_Stop(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Monitor_ListMode_GetItem

G_Can_Monitor_ListMode_GetItem — Request of the list entry

Description

This command requests a list entry for the identifier given by `id`.

Definition

```
G_Error_t
G_Can_Monitor_ListMode_GetItem(
 const G_PortHandle_t portHandle,
 const u32_t id,
 u64_t * const timestamp,
 u32_t * const msgCount,
 G_Can_Monitor_ListMode_GetItem_RspFlags_t * const rspFlags,
 u8_t * const dlc,
 u8_t data[8]
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Identifier of the message whose list entry has to be requested

`timestamp`
Time stamp in ns

`msgCount`
Returns how often the requested identifier was sent or received

`rspFlags`
Pointer to a variable of type **G_Can_Monitor_ListMode_GetItem_RspFlags_t**

The following return flags are available:

- `G_CAN__MONITOR__LIST_MODE__GET_ITEM__RSP_FLAG__NONE`

  No flag was set
- `G_CAN__MONITOR__LIST_MODE__GET_ITEM__RSP_FLAG__TRANSMIT`

  Transmitted message (TX)

`dlc`
Pointer to the variable `dlc`. The data length (0..8) is returned in this variable.

`data`
Pointer to the variable `data`. The data bytes (0..7) are returned in this variable.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Monitor_ListMode_GetItem_Async

G_Can_Monitor_ListMode_GetItem_Async — Request of the list entry (by asynchronous request)

Description

This command starts the asynchronous request of a list entry for the identifier given by `id`. As soon as a response is available, a callback function set with G_Can_Monitor_ListMode_GetItem_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Can_Monitor_ListMode_GetItem_Async(
 const G_PortHandle_t portHandle,
 const u32_t id
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier of the message whose list entry has to be requested

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Monitor_ListMode_GetItem_Async_Callback

G_Can_Monitor_ListMode_GetItem_Async_Callback — Callback function for the asynchronous request of the monitor list entries

Description

This function will be called if a response for the asynchronous request of monitor list entries with G_Can_Monitor_ListMode_GetItem_Async is available. It must be entered by the user and set with G_Can_Monitor_ListMode_GetItem_Async_AddCallback.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Can_Monitor_ListMode_GetItem_Async_Callback(
 const G_PortHandle_t  portHandle,
 const u32_t id,
 const u64_t  timestamp,
 const u32_t  msgCount,
 const G_Can_Monitor_ListMode_GetItem_RspFlags_t  rspFlags,
 const u8_t  dlc,
 const u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Identifier of the message

`timestamp`
Time stamp in ns

`msgCount`
Returns how often the requested identifier was sent or received

`rspFlags`
Return flags

The following return flags are available:

- `G_CAN__MONITOR__LIST_MODE__GET_ITEM__RSP_FLAG__NONE`

  No flag was set
- `G_CAN__MONITOR__LIST_MODE__GET_ITEM__RSP_FLAG__TRANSMIT`

  Transmitted message (TX)

`dlc`
Data length of the message (0..8)

`data`
Pointer to the data bytes of the message (0..7)

G_Can_Monitor_ListMode_GetItem_Async_AddCallback

G_Can_Monitor_ListMode_GetItem_Async_AddCallback — Set the callback function for the asynchronous request of monitor list entries

Description

This command defines a callback function for the asynchronous request of monitor list entries with G_Can_Monitor_ListMode_GetItem_Async.

Definition

```
G_API_CAN_DLL G_Error_t
G_Can_Monitor_ListMode_GetItem_Async_AddCallback(
 const G_PortHandle_t  portHandle,
 const G_Can_Monitor_ListMode_GetItem_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`callback`
Function pointer of the callback function (see G_Can_Monitor_ListMode_GetItem_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Monitor_ListMode_GetItem_Async_RemoveCallback

G_Can_Monitor_ListMode_GetItem_Async_RemoveCallback — Remove the callback function for the asynchronous request of monitor list entries

Description

This command removes a callback function for the asynchronous request of monitor list entries with G_Can_Monitor_ListMode_GetItem_Async.

Definition

```
G_Error_t
G_Can_Monitor_ListMode_GetItem_Async_RemoveCallback(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Monitor_WriteUserEvent

G_Can_Monitor_WriteUserEvent — Write user event into monitor buffer

Description

Use this command to write a user event into the monitor buffer.

A user event consists of an event identifier, event data bytes and is identified by a CAN identifier of **0xFFFFFFFE** .

Definition

```
G_Error_t
G_Can_Monitor_WriteUserEvent(
 const G_PortHandle_t portHandle,
 const u8_t  userEvent,
 const u8_t  length,
 const u8_t * const data
);
```

Parameters

`portHandle`
: Handle to the communication port

`userEvent`
: User Event Identifier (0x80..0xFF)

  The User Event Identifier is the first byte of the frame.

`length`
: Length of User Event Data in bytes (0..7)

`data`
: User Event data bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Monitor_MonitorTime_Get

G_Can_Monitor_MonitorTime_Get — Query timestamps of specific monitor properties

Description

Use this function to query timestamps of specific monitor properties, like the start time of the monitor or the current monitor time.

Definition

```
G_Error_t
G_API
G_Can_Monitor_MonitorTime_Get(
 const G_PortHandle_t portHandle,
 G_Can_Monitor_MonitorTime_Get_RspFlags_t * const rspFlags,
 u64_t * const startTime,
 u64_t * const currentTime
);
```

Parameters

`portHandle`
Handle to the communication port

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_CAN__MONITOR__MONITOR_TIME__GET__RSP_FLAG__NONE`
No flag is set

`G_CAN__MONITOR__MONITOR_TIME__GET__RSP_FLAG__MONITOR_STARTED`
The monitor has been started

`startTime`
Returns the start time of the monitor, in nanoseconds

`currentTime`
Returns the current monitor time, in nanoseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

7 NM

These commands are used to control the **N**etwork **M**anagement (**NM**).

G_Can_Nm_Config_Off

G_Can_Nm_Config_Off — Disable the network management

Description

This command disables the network management.

Definition

```
G_Error_t
G_Can_Nm_Config_Off(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Nm_Config_Osek

G_Can_Nm_Config_Osek — Configure OSEK network management

Description

This command configures the OSEK network management.

Definition

```
G_Error_t
G_Can_Nm_Config_Osek(
 const G_PortHandle_t  portHandle,
 const G_Can_Nm_Config_Osek_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Pointer to structure with parameters

The structure contains the following elements:

- `CmdFlags`

  Command flags for enhanced network management configuration.

  With no command flag set, the default configuration for the specific network management type defined by `Type` is used and the corresponding elements of this structure are disregarded.

  The following values are posible and can be combined:
  - `G_CAN__NM__CONFIG__OSEK__CMD_FLAG__NONE`

    No command flag set
  - `G_CAN__NM__CONFIG__OSEK__CMD_FLAG__SET_DLC`

    Set data length of all network nodes defined by `Dlc`
  - `G_CAN__NM__CONFIG__OSEK__CMD_FLAG__SET_DESTINATION_BYTE_OFFSET`

    Set byte position of target address defined by `DestinationByteOffset`
  - `G_CAN__NM__CONFIG__OSEK__CMD_FLAG__SET_RANGE`

    Set NM identifier range defined by structure `Range`
  - `G_CAN__NM__CONFIG__OSEK__CMD_FLAG__SET_PROTOCOL_BITS`

    Set protocol bits defined by structure `ProtocolBits`
  - `G_CAN__NM__CONFIG__OSEK__CMD_FLAG__SET_RING_STABLE_BIT`

    Set ring stable bit defined by structure `RingStableBit`
  - `G_CAN__NM__CONFIG__OSEK__CMD_FLAG__SET_TIMES`

    Set NM timing parameters defined by structure `Time`
- `Type` Type of network management

  The following values are possible:
  - `G_CAN__NM__CONFIG__OSEK__TYPE__VAG` **VAG** network management
  - `G_CAN__NM__CONFIG__OSEK__TYPE__DAIMLER` **DAIMLER** network management

- `G_CAN__NM__CONFIG__OSEK__TYPE__PORSCHE` **PORSCHE** network management
- `G_CAN__NM__CONFIG__OSEK__TYPE__BMW` **BMW** network management
- `G_CAN__NM__CONFIG__OSEK__TYPE__FORD` **FORD** network management

- `Dlc`

  Data length of network nodes
- `DestinationByteOffset`

  Byte position of target address
- `reserved`
- `Range`

  Structure with identifier range configuration parameters

  The structure contains the following elements:
  - `IdBase`

    ID basis address of NM (e.g. **0x400**)
  - `IdMask`

    ID mask (e.g. **0x3F**)
  - `NumberOfNodes`

    Maximum number of control units

    (power of two numbers required, e.g. **64**)
  - `reserved1`
  - `reserved2`
- `ProtocolBits` Structure with protocol bit configuration parameters

  The structure contains the following elements:
  - `MessageTypeMask`

    Complete mask for the NM Message Type (e.g. **0x0700**)
  - `AliveCode`

    Code for the NM Alive Message (e.g. **0x0200**)
  - `RingCode`

    Code for the NM Ring Message (e.g. **0x0100**)
  - `LimpHomeCode`

    Code for the NM Emergency Message (e.g. **0x0400**)
  - `SleepIndMask`

    Mask for the Sleep Indication Bit (e.g. **0x1000**)
  - `SleepIndCodeTrue`

    Code for the set Sleep Indication Bit (e.g. **0x1000**)
  - `SleepIndCodeFalse`

    Code for the reset Sleep Indication Bit (e.g. **0x0000**)
  - `SleepAckMask`

    Mask for the Sleep Acknowledge Bit (e.g. **0x2000**)
  - `SleepAckCodeTrue`

    Code for the set Sleep Acknowledge Bit (e.g. **0x2000**)
  - `SleepAckCodeFalse`

    Code for the reset Sleep Acknowledge Bit (e.g. **0x0000**)

- `RingStableBit` Structure with ring stable bit configuration parameters

  The Structure contains the following elements:
  - `RingStableMask`

    Mask for the Ring Stable Bit (e.g. **0x4000**)
  - `RingStableCodeTrue`

    Code for the set Ring Stable Bit (e.g. **0x4000**)
  - `RingStableCodeFalse`

    Code for the reset Ring Stable Bit (e.g. **0x0000**)
- `Times`

  Structure with timing parameters

  The structure contains the following elements:
  - `Typ`

    Typical Ring Time in milliseconds (e.g. **100**)
  - `Max`

    Maximum Ring Time in milliseconds (e.g. **260**)
  - `Error`

    Emergency run time in milliseconds (e.g. **1000**)
  - `WaitBusSleep`

    Waiting time prior to transition into **BusSleep** mode in milliseconds (e.g. **1500**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Configure NM for **BMW** type and set data length to **4**

```
G_Can_Nm_Config_Osek_Parameters_t  parameters;

parameters.Type = G_CAN__NM__CONFIG__OSEK__TYPE__BMW;
parameters.CmdFlags = G_CAN__NM__CONFIG__OSEK__CMD_FLAG__SET_DLC;
parameters.Dlc = 4;
parameters.reserved = 0;

return
G_Can_Nm_Config_Osek(
 portHandle,
 &parameters
);
```

G_Can_Nm_Config_SetData

G_Can_Nm_Config_SetData — Set data bytes of all NM nodes

Description

This command sets the data bytes of all NM nodes.

Definition

```
G_Error_t
G_Can_Nm_Config_SetData(
 const G_PortHandle_t  portHandle,
 const u8_t  mask[8],
 const u8_t  data[8]
);
```

Parameters

`portHandle`
: Handle to the communication port

`mask`
: Mask bytes (0..7)

`data`
: Data bytes (0..7)

  Data is assumed in accordance with the set mask byte bit positions.

  In the case no mask byte bits are set, the original data is assumed.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Set bytes **1** and **2** to **0x11** and **0x22** and leave all other bytes with their original data.

```
const u8_t  mask[8] = {0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
const u8_t  data[8] = {0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88};

return
G_Can_Nm_Config_SetData(
 portHandle,
 mask,
 data
);
```

G_Can_Nm_Config_Autosar

G_Can_Nm_Config_Autosar — Configure AUTOSAR network management parameters

Description

This function configures the AUTOSAR network management.

Definition

```
G_Error_t
G_Can_Nm_Config_Autosar(
 const G_PortHandle_t  portHandle,
 const G_Can_Nm_Config_Autosar_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

parameters
Structure with configuration parameters

The structure contains the following elements:

`Type`
AUTOSAR network management type

The following types are possible:

`G_CAN__NM__CONFIG__AUTOSAR__TYPE__DEFAULT`
Use this type for a common AUTOSAR network management when none of the other types fits your needs. The network management will run with default AUTOSAR parameters.

`G_CAN__NM__CONFIG__AUTOSAR__TYPE__VAG_NMH_1`
VAG NM High 1.x network management

`G_CAN__NM__CONFIG__AUTOSAR__TYPE__VAG_NMH_2`
VAG NM High 2.x network management

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`CmdFlags`
Command flags

The following values are possible and can be combined:

`G_CAN__NM__CONFIG__AUTOSAR__CMD_FLAG__NONE`
No flag is set

`G_CAN__NM__CONFIG__AUTOSAR__CMD_FLAG__SET_TIMES`
Set Timing parameters specified by structure `Times`

`G_CAN__NM__CONFIG__AUTOSAR__CMD_FLAG__SET_RANGE`
Set node identifier range parameters specified by structure `Range`

`Times`
Structure with timing parameters (used when flag `G_CAN__NM__CONFIG__AUTOSAR__CMD_FLAG__SET_TIMES` is set)

The structure contains the following elements:

`NmTimeout`
NM-Timeout Time in milliseconds

`WaitBusSleep`
Wait Bus-Sleep Time in milliseconds

`MsgCycle`
Message-Cycle Time in milliseconds

`RepeatMsg`
Repeat Message Time in milliseconds

`Range`
Structure with node identifier range parameters (used when flag `G_CAN__NM__CONFIG__AUTOSAR__CMD_FLAG__SET_RANGE` is set)

The structure contains the following elements:

`IdBase`
ID basis address of NM (e.g. **0x400**)

`NumberOfNodes`
Maximum number of control units (power of two required, e.g. **64**)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

For specifying more advanced parameters, use G_Can_Nm_Config_Autosar_Global after initializing.

For specifying parameters for a specific node, use G_Can_Nm_Config_Autosar_Node .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Nm_Config_Autosar_Global

G_Can_Nm_Config_Autosar_Global — Configure global AUTOSAR network management parameters

Description

Use this command to configure specific AUTOSAR NM parameters after initializing the NM with function G_Can_Nm_Config_Autosar .

Definition

```
G_Error_t
G_Can_Nm_Config_Autosar_Global(
 const G_PortHandle_t  portHandle,
 const G_Can_Nm_Config_Autosar_Global_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Structure with configuration parameters

The structure contains the following elements:

`CmdFlags`
Command flags

The following values are possible and can be combined:

`G_CAN__NM__CONFIG__AUTOSAR__GLOBAL__CMD_FLAG__NONE`
No flag is set

`G_CAN__NM__CONFIG__AUTOSAR__GLOBAL__CMD_FLAG__USE_BUS_LOAD_REDUCTION`
Enable **Bus Load Reduction**

`G_CAN__NM__CONFIG__AUTOSAR__GLOBAL__CMD_FLAG__USE_NODE_ID_IN_PDU`
Insert **Node Identifier** into PDU at position specified by parameter `PduNidPosition`

`G_CAN__NM__CONFIG__AUTOSAR__GLOBAL__CMD_FLAG__USE_CBV_IN_PDU`
Insert **Control Bit Vector** into PDU at position specified by parameter `PduCbvPosition`

`G_CAN__NM__CONFIG__AUTOSAR__GLOBAL__CMD_FLAG__USE_STATE_IN_PDU`
Insert **PDU State** into PDU at position specified by parameter `PduStatePosition`

`G_CAN__NM__CONFIG__AUTOSAR__GLOBAL__CMD_FLAG__SET_PDU_DATA`
Set or change **PDU Data** according to parameters `PduLength` and `PduData`

`Times`
Structure with timing parameters

The structure contains the following elements:

`NmTimeout`
NM-Timeout Time in microseconds

`WaitBusSleep`
Wait Bus-Sleep Time in microseconds

`MsgCycle`
Message-Cycle Time in microseconds

`RepeatMsg`
Repeat Message Time in microseconds

`PduNidPosition`
Specifies the position of the **Node Identifier** within the PDUs when flag `G_CAN__NM__CONFIG__AUTOSAR__GLOBAL__CMD_FLAG__USE_NODE_ID_IN_PDU` is set

`PduCbvPosition`
Specifies the position of the **Control Bit Vector** within the PDUs when flag `G_CAN__NM__CONFIG__AUTOSAR__GLOBAL__CMD_FLAG__USE_CBV_IN_PDU` is set

`PduStatePosition`
Specifies the position of the **PDU State** within the PDUs when flag `G_CAN__NM__CONFIG__AUTOSAR__GLOBAL__CMD_FLAG__USE_STATE_IN_PDU` is set

`PduLength`
Specifies the length of the PDUs in bytes when flag `G_CAN__NM__CONFIG__AUTOSAR__GLOBAL__CMD_FLAG__SET_PDU_DATA` is set

`PduData`
Specifies the data bytes of the PDUs when flag `G_CAN__NM__CONFIG__AUTOSAR__GLOBAL__CMD_FLAG__SET_PDU_DATA` is set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Nm_Config_Autosar_Node

G_Can_Nm_Config_Autosar_Node — Configure AUTOSAR network management parameters for a specific node

Description

Use this command to configure AUTOSAR NM parameters for a specific node.

Definition

```
G_Error_t
G_Can_Nm_Config_Autosar_Node(
 const G_PortHandle_t  portHandle,
 const G_Can_Nm_Config_Autosar_Node_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Structure with configuration parameters

The structure contains the following elements:

`CmdFlags`
Command flags

The following values are possible and can be combined:

`G_CAN__NM__CONFIG__AUTOSAR__NODE__CMD_FLAG__NONE`
No flag is set

`G_CAN__NM__CONFIG__AUTOSAR__NODE__CMD_FLAG__SET_MSG_CYCLE_OFFSET`
Set **Message Cycle Offset Time** specified by `MsgCycleOffset`

`G_CAN__NM__CONFIG__AUTOSAR__NODE__CMD_FLAG__SET_MSG_REDUCED_TIME`
Set **Message Reduced Time** specified by `MsgReducedTime`

`G_CAN__NM__CONFIG__AUTOSAR__NODE__CMD_FLAG__SET_PDU_DATA`
Set **PDU Data** specified by `PduLength` and `PduData`

`G_CAN__NM__CONFIG__AUTOSAR__NODE__CMD_FLAG__SET_IMMED_TRANS`
Enable immediate transmission

The PDU is sent immediately for `NumberOfImmediateNmTransmissions` with a cycle time of `ImmediateNmCycleTime` microseconds.

`NodeId`
Node identifier of the network node to be configured

`PduLength`
Specifies the length of the PDU in bytes when flag `G_CAN__NM__CONFIG__AUTOSAR__NODE__CMD_FLAG__SET_PDU_DATA` is set

`NumberOfImmediateNmTransmissions`
Number of immediate transmissions

Only used if flag G_CAN__NM__CONFIG__AUTOSAR__NODE__CMD_FLAG__SET_IMMED_TRANS is set.

reserved2
reserved parameter (must be initialized with **0**)

MsgCycleOffset
Specifies the **Message Cycle Offset Time** in microseconds when flag G_CAN__NM__CONFIG__AUTOSAR__NODE__CMD_FLAG__SET_MSG_CYCLE_OFFSET is set

MsgReducedTime
Specifies the **Message Reduced Time** when flag G_CAN__NM__CONFIG__AUTOSAR__NODE__CMD_FLAG__SET_MSG_REDUCED_TIME is set

PduData
Structure that specifies the **PDU Data** when flag G_CAN__NM__CONFIG__AUTOSAR__NODE__CMD_FLAG__SET_PDU_DATA is set

The structure contains the following elements:

Mask
Specifies the data mask of the PDU

For every bit set, the corresponding bit in Data will be set.

Data
Specifies the data bytes of the PDU according to Mask

ImmediateNmCycleTime
Immediate NM cycle time

Cycle time of the PDU if immediate transmission is enabled

Only used if flag G_CAN__NM__CONFIG__AUTOSAR__NODE__CMD_FLAG__SET_IMMED_TRANS is set.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Nm_Autosar_Node_GetState

G_Can_Nm_Autosar_Node_GetState — Query the state of an AUTOSAR network node

Description

Use this function to query the state of an AUTOSAR network node.

Definition

```
G_Error_t
G_Can_Nm_Autosar_Node_GetState(
 const G_PortHandle_t  portHandle,
 const u8_t  nodeId,
 G_Can_Nm_Autosar_Node_GetState_RspFlags_t * const rspFlags,
 G_Can_Nm_Autosar_NetworkMode_t * const networkMode,
 G_Can_Nm_Autosar_NodeState_t * const nodeState
);
```

Parameters

`portHandle`
Handle to the communication port

`nodeId`
Identifier of the network node whose state should be returned

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_CAN__NM__AUTOSAR__NODE__GET_STATE__RSP_FLAG__NONE`
No flag is set

`G_CAN__NM__AUTOSAR__NODE__GET_STATE__RSP_FLAG__IS_STARTED`
The node is started

`G_CAN__NM__AUTOSAR__NODE__GET_STATE__RSP_FLAG__IS_NETWORK_REQUESTED`
Signals that the node needs to communicate on the network

`networkMode`
Returns the network mode of the node

The following values are possible:

`G_CAN__NM__AUTOSAR__NETWORK_MODE__UNKNOWN`
The node is in an unknown mode

`G_CAN__NM__AUTOSAR__NETWORK_MODE__BUS_SLEEP`
The node is in mode **Bus Sleep**

`G_CAN__NM__AUTOSAR__NETWORK_MODE__PREPARE_BUS_SLEEP`
The node is in mode **Prepare Bus Sleep**

`G_CAN__NM__AUTOSAR__NETWORK_MODE__NETWORK`
The node is in mode **Network**

`nodeState`
Returns the current state of the node

The following values are possible:

`G_CAN__AUTOSAR__NODE_STATE__UNKNOWN`
The node state is unknown

`G_CAN__AUTOSAR__NODE_STATE__NOT_STARTED`
The node is **not started**

`G_CAN__AUTOSAR__NODE_STATE__BUS_SLEEP`
The node is in **Bus Sleep** state

`G_CAN__AUTOSAR__NODE_STATE__PREPARE_BUS_SLEEP`
The node is in **Prepare Bus Sleep** state

`G_CAN__AUTOSAR__NODE_STATE__READY_SLEEP`
The node is in **Ready Sleep** state

`G_CAN__AUTOSAR__NODE_STATE__NORMAL_OPERATION`
The node is in **Normal Operation** state

`G_CAN__AUTOSAR__NODE_STATE__REPEAT_MESSAGE`
The node is in **Repeat Message** state

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Nm_Autosar_RepeatMessageRequest

G_Can_Nm_Autosar_RepeatMessageRequest — Trigger a network node to send a **Repeat Message** request

Description

This command triggers a network node to send a **Repeat Message** request.

Definition

```
G_Error_t
G_Can_Nm_Autosar_RepeatMessageRequest(
 const G_PortHandle_t  portHandle,
 const u8_t  nodeId
);
```

Parameters

`portHandle`
: Handle to the communication port

`nodeId`
: Identifier of the network node

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Nm_Autosar_RequestBusSynchronization

G_Can_Nm_Autosar_RequestBusSynchronization — Trigger a network node to request **Bus Synchronization**

Description

This command triggers a network node to request **Bus Synchronization** .

Definition

```
G_Error_t
G_Can_Nm_Autosar_RequestBusSynchronization(
 const G_PortHandle_t  portHandle,
 const u8_t  nodeId
);
```

Parameters

`portHandle`
: Handle to the communication port

`nodeId`
: Identifier of the network node

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Nm_Nodes_Start

G_Can_Nm_Nodes_Start — Start NM nodes

Description

This command starts the specified NM nodes.

Definition

```
G_Error_t
G_Can_Nm_Nodes_Start(
 const G_PortHandle_t  portHandle,
 const u8_t  numberOfNodes,
 const u8_t * const nodes
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfNodes`
Number of nodes to be started

`nodes`
Pointer to array with node numbers

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Start nodes **1** and **2**.

```
const u8_t  nodes[] = {0x01, 0x02};

return
G_Can_Nm_StartNodes(
 PortHandle,
 2,
 nodes
);
```

G_Can_Nm_Nodes_Stop

G_Can_Nm_Nodes_Stop — Stop NM nodes

Description

This command stops the specified NM nodes.

Definition

```
G_Error_t
G_Can_Nm_Nodes_Stop(
 const G_PortHandle_t  portHandle,
 const u8_t  numberOfNodes,
 const u8_t * const nodes
);
```

Parameters

`portHandle`
: Handle to the communication port

`numberOfNodes`
: Number of nodes to be stopped

`nodes`
: Pointer to array with node numbers

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Nm_Nodes_GoToMode_Awake

G_Can_Nm_Nodes_GoToMode_Awake — Trigger nodes to go to awake mode

Description

This command triggers the transition of all nodes indicated by `nodes` to awake mode.

Definition

```
G_Error_t
G_Can_Nm_Nodes_GoToMode_Awake(
 const G_PortHandle_t  portHandle,
 const u8_t  numberOfNodes,
 const u8_t * const nodes
);
```

Parameters

`portHandle`
: Handle to the communication port

`numberOfNodes`
: Number of nodes

`nodes`
: Pointer to array with node numbers

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Nm_Nodes_GoToMode_BusSleep

G_Can_Nm_Nodes_GoToMode_BusSleep — Trigger nodes to go to bus sleep mode

Description

This command triggers the transition of all nodes indicated by `nodes` to go to bus sleep mode.

Definition

```
G_Error_t
G_Can_Nm_Nodes_GoToMode_BusSleep(
 const G_PortHandle_t  portHandle,
 const u8_t  numberOfNodes,
 const u8_t * const nodes
);
```

Parameters

`portHandle`
: Handle to the communication port

`numberOfNodes`
: Number of nodes

`nodes`
: Pointer to array with node numbers

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Nm_Nodes_ChangeGroup

G_Can_Nm_Nodes_ChangeGroup — Change group number

Description

This command changes the group number of the selected nodes.

Definition

```
G_Error_t
G_Can_Nm_Nodes_ChangeGroup(
 const G_PortHandle_t  portHandle,
 const u8_t  group,
 const u8_t  numberOfNodes,
 const u8_t * const nodes
);
```

Parameters

`portHandle`
Handle to the communication port

`group`
Desired group ID

0 : all nodes

1..N : group **1..N**

In the recent version, only group IDs `0` and `1` are supported.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Nm_Group_Start

G_Can_Nm_Group_Start — Start group of NM nodes

Description

This command starts a group of NM nodes.

Definition

```
G_Error_t
G_Can_Nm_Group_Start(
 const G_PortHandle_t  portHandle,
 const u8_t  group
);
```

Parameters

`portHandle`
Handle to the communication port

`group`
Group ID of nodes

0 : all nodes

1..N : nodes belonging to group **1..N**

In the recent version, only group IDs `0` and `1` are supported.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Nm_Group_Stop

G_Can_Nm_Group_Stop — Stop group of NM nodes

Description

This command stops a group of NM nodes.

Definition

```
G_Error_t
G_Can_Nm_Group_Stop(
 const G_PortHandle_t  portHandle,
 const u8_t  group
);
```

Parameters

`portHandle`
Handle to the communication port

`group`
Group ID of nodes

0 : all nodes

1..N : nodes belonging to group **1..N**

In the recent version, only group IDs `0` and `1` are supported.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Nm_Group_GoToMode_Awake

G_Can_Nm_Group_GoToMode_Awake — Trigger group of nodes to go to awake mode

Description

This command triggers a group of nodes to go to awake mode.

Definition

```
G_Error_t
G_Can_Nm_Group_GoToMode_Awake(
 const G_PortHandle_t  portHandle,
 const u8_t  group
);
```

Parameters

`portHandle`
Handle to the communication port

`group`
Group ID of nodes

0 : all nodes

1..N : nodes belonging to group **1..N**

In the recent version, only group IDs `0` and `1` are supported.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Nm_Group_GoToMode_BusSleep

G_Can_Nm_Group_GoToMode_BusSleep — Trigger group of nodes to go to bus sleep mode

Description

This command triggers a group of nodes to go to bus sleep mode.

Definition

```
G_Error_t
G_Can_Nm_Group_GoToMode_BusSleep(
 const G_PortHandle_t  portHandle,
 const u8_t  group
);
```

Parameters

`portHandle`
Handle to the communication port

`group`
Group ID of nodes

0 : all nodes

1..N : nodes belonging to group **1..N**

In the recent version, only group IDs `0` and `1` are supported.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

8 Node

These commands are used to control and configure the CAN node.

G_Can_Node_Baudrate_Set

G_Can_Node_Baudrate_Set — Set the baud rate

Description

Use this command to set the baud rate and optionally extended parameters.

Definition

```
G_Error_t
G_Can_Node_Baudrate_Set(
 const G_PortHandle_t portHandle,
 const u32_t  baudrate,
 const G_Can_Node_Baudrate_Set_ExtendedParam_t * const extendedParam,
 G_Can_Node_Baudrate_ActualValues_t * const actualValues
);
```

Parameters

`portHandle`
Handle to the communication port

`baudrate`
Baud rate in Baud (e.g. **500000** for **500kBaud**)

`extendedParam`
Pointer to extended baud rate parameters

If `extendedParam` equals **NULL** , no extended parameters are set.

The extended parameter structure contains the following elements:

`SamplePoint_Min`
Minimum sample point

`SamplePoint_Max`
Maximum sample point

`NumberOfTimeQuanta_Min`
Minimum number of time quanta 1 + TSeg1 + TSeg2 = 8..25)

`NumberOfTimeQuanta_Max`
Maximum number of time quanta (1 + TSeg1 + TSeg2 = 8..25)

`TSeg1_Min`
Minimum number of time quanta before sample point minus one (3..16)

`TSeg1_Max`
Maximum number of time quanta before sample point minus one (3..16)

`TSeg2_Min`
Minumum number of time quanta after sample point minus one (2..8)

`TSeg2_Max`
Maximum number of time quanta after sample point minus one (2..8)

`Sjw_Min`
Minimum resynchronization jump width (1..4)

`Sjw_Max`
Maximum resynchronization jump width (1..4)

Use **0** if you do not want to specify a specific parameter of this structure.

`actualValues`
Pointer to response structure with currently set baud rate values

The structure contains the following elements:

`Baudrate`
Baud rate in Baud (e.g. **500000** for **500kBaud**)

`CanControllerClock`
CAN controller clock frequency in Hz

`SamplePoint`
Sample point

`NumberOfTimeQuanta`
Number of time quanta (1 + TSeg1 + TSeg2)

`TSeg1`
Number of time quanta before sample point minus one

`TSeg2`
Number of time quanta after sample point

`Sjw`
resynchronization jump width

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
const u32_t  baudRate = 500000;
G_Can_Node_Baudrate_ActualValues_t  actualValues;
G_Error_t  rc;

rc =
 G_Can_Node_Baudrate_Set(
  PortHandle1,
  baudRate,
  NULL,
  &actualValues
 );
```

In this example, the baud rate is set without extended parameters since the pointer to the extended parameter structure is **NULL** .
The currently set baud rate values can be read in structure `actualValues`.

Example

```
const u32_t  baudRate = 500000;
G_Can_Node_Baudrate_Set_ExtendedParam_t  extendedParam;
G_Can_Node_Baudrate_ActualValues_t  actualValues;
G_Error_t  rc;

extendedParam.TSeg1_Min = 10;
extendedParam.TSeg1_Max = 14;
extendedParam.TSeg2_Min = 2;
extendedParam.TSeg2_Max = 4;
extendedParam.Sjw_Min = 1;
extendedParam.Sjw_Max = 2;

extendedParam.NumberOfTimeQuanta_Min = 0;
extendedParam.NumberOfTimeQuanta_Max = 0;
extendedParam.SamplePoint_Min = 0;
extendedParam.SamplePoint_Max = 0;
extendedParam.reserved1 = 0;
extendedParam.reserved2 = 0;

rc =
 G_Can_Node_Baudrate_Set(
  PortHandle1,
  baudRate,
  &extendedParam,
  &actualValues
 );
```

In this second example, the baud rate is set including some extended parameters. Note that unused extended parameters are set to **0**.
The currently set baud rate values can be read in structure `actualValues`.

G_Can_Node_Baudrate_Get

G_Can_Node_Baudrate_Get — Get actual baud rate values

Description

Use this command to query the actual baud rate values of the hardware.

Definition

```
G_Error_t
G_Can_Node_Baudrate_Get(
 const G_PortHandle_t portHandle,
 G_Can_Node_Baudrate_ActualValues_t * const actualValues
);
```

Parameters

`portHandle`
: Handle to the communication port

`actualValues`
: Pointer to the response structure with the currently set baud rate values (see Baudrate Values)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Node_BaudrateFd_Set

G_Can_Node_BaudrateFd_Set — Set baud rate for CAN FD node

Description

Use this command to set the baud rate for a CAN FD node.

Definition

```
G_Error_t
G_Can_Node_BaudrateFd_Set(
 const G_PortHandle_t portHandle,
 const G_Can_Node_BaudrateFd_Set_Cmd_t * const cmd,
 G_Can_Node_BaudrateFd_Set_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following parameters are possible and can be combined:

`G_CAN__NODE__BAUDRATE_FD__SET__CMD_FLAG__NONE`
No flag is set

`G_CAN__NODE__BAUDRATE_FD__SET__CMD_FLAG__CHANGE_TDC`
not set : don't regard flag `G_CAN__NODE__BAUDRATE_FD__SET__CMD_FLAG__TDC` and parameter `TdcOffset`

set : change flag `G_CAN__NODE__BAUDRATE_FD__SET__CMD_FLAG__TDC` and parameter `TdcOffset` according to their values

`G_CAN__NODE__BAUDRATE_FD__SET__CMD_FLAG__TDC`
not set : disable transceiver delay compensation

set : enable transceiver delay compensation

`BaudRate`
Baud rate in baud (e.g. 500000 for 500kBaud)

`SamplePoint_Min`
Minimum sample point, use **0** if you don't want to specify this parameter

`SamplePoint_Max`
Maximum sample point, use **0** if you don't want to specify this parameter

`NumberOfTimeQuanta_Min`
Minimum number of time quanta (1 + `TSeg1` + `TSeg2`), use **0** if you don't want to specify this parameter

`NumberOfTimeQuanta_Max`
Maximum number of time quanta (1 + `TSeg1` + `TSeg2`), use **0** if you don't want to specify this parameter

`TSeg1_Min`
Minimum number of time quanta before sample point minus one, use **0** if you don't want to specify this parameter

`TSeg1_Max`
Maximum number of time quanta before sample point minus one, use **0** if you don't want to specify this parameter

`TSeg2_Min`
Minimum number of time quanta after sample point, use **0** if you don't want to specify this parameter

`TSeg2_Max`
Maximum number of time quanta after sample point, use **0** if you don't want to specify this parameter

`Sjw_Min`
Minimum resyncronization jump width (**>= 1**), use **0** if you don't want to specify this parameter

`Sjw_Max`
Maximum resyncronization jump width (**>= 1**), use **0** if you don't want to specify this parameter

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`TdcOffset`
Transceiver delay compensation offset, in nanoseconds

`rsp`
Returns current baud rate values

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_CAN__NODE__BAUDRATE_FD__SET__RSP_FLAG__NONE`
No flag is set

`G_CAN__NODE__BAUDRATE_FD__SET__RSP_FLAG__FD`
not set : CAN operation according **ISO11898-1**

set : CAN FD operation

`G_CAN__NODE__BAUDRATE_FD__SET__RSP_FLAG__TDC`
not set : transceiver delay compensation is disabled

set : transceiver delay compensation is enabled

`BaudRate`
Baud rate in baud (e.g. 500000 for 500kBaud)

`CanControllerClock`
CAN controller clock frequency in Hz

`SamplePoint`
Sample point

`TSeg1`
Number of time quanta before sample point minus one

`TSeg2`
Number of time quanta after sample point

`Sjw`
Resyncronization jump width

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`TdcOffset`
Transceiver delay compensation offset, in nanoseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Node_BaudrateFd_Get

G_Can_Node_BaudrateFd_Get — Query current baud rate values

Description

Use this command to query current baud rate values

Definition

```
G_Error_t
G_Can_Node_BaudrateFd_Get(
 const G_PortHandle_t portHandle,
 const G_Can_Node_BaudrateFd_Get_CmdFlags_t  cmdFlags,
 G_Can_Node_BaudrateFd_Get_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_CAN__NODE__BAUDRATE_FD__GET__CMD_FLAG__NONE`
No flag is set

`rsp`
Returns current baud rate values

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_CAN__NODE__BAUDRATE_FD__GET__RSP_FLAG__NONE`
No flag is set

`G_CAN__NODE__BAUDRATE_FD__GET__RSP_FLAG__FD`
not set : CAN operation according **ISO11898-1**

set : CAN FD operation

`G_CAN__NODE__BAUDRATE_FD__GET__RSP_FLAG__TDC`
not set : transceiver delay compensation is disabled

set : transceiver delay compensation is enabled

`BaudRate`
Baud rate in baud (e.g. 500000 for 500kBaud)

`CanControllerClock`
CAN controller clock frequency in Hz

`SamplePoint`
Sample point

`TSeg1`
Number of time quanta before sample point minus one

`TSeg2`
Number of time quanta after sample point

`Sjw`
Resyncronization jump width

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`TdcOffset`
Transceiver delay compensation offset, in nanoseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Node_ErrorFrameCounter_Get

G_Can_Node_ErrorFrameCounter_Get — Get error frame counter

Description

Use this command to query the error frame counter value.

Definition

```
G_Error_t
G_Can_Node_ErrorFrameCounter_Get(
 const G_PortHandle_t portHandle,
 u32_t * const errorFrameCounter
);
```

Parameters

`portHandle`
: Handle to the communication port

`errorFrameCounter`
: Error frame counter value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Node_ErrorFrameCounter_Reset

G_Can_Node_ErrorFrameCounter_Reset — Reset error frame counter

Description

Use this command to reset the error frame counter.

Definition

```
G_Error_t
G_Can_Node_ErrorFrameCounter_Reset(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Node_GetState

G_Can_Node_GetState — Get node state

Description

Use this command to query the state of a CAN node.

Definition

```
G_Error_t
G_Can_Node_GetState(
 const G_PortHandle_t portHandle,
 const G_Can_Node_GetState_Mode_t  mode,
 G_Can_Node_GetState_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
Query mode

The following values are possible:

`G_CAN__NODE__GET_STATE__MODE__STANDARD`
Standard query

`rsp`
Pointer to response structure

The structure contains the following elements:

`RxErrorCounter`
Receive error counter

`TxErrorCounter`
Transmit error counter

`ErrorWarningLevel`
CAN error warning level

`InitState`
Initialization state

The following values are possible:

`G_CAN__NODE__INIT_STATE__OK`
All is OK

`G_CAN__NODE__INIT_STATE__ERROR_WARNING`
Error warning

`G_CAN__NODE__INIT_STATE__NO_ACK_WAIT`
Waiting after not receiving a CAN acknowledge

`G_CAN__NODE__INIT_STATE__BUS_OFF`
Bus off

G_CAN__NODE__INIT_STATE__BUS_OFF_WAIT
Waiting after bus off

Flags
Response flags

The following values are possible and can be combined:

G_CAN__NODE__GET_STATE__RSP_FLAG__NONE
No response flag set

G_CAN__NODE__GET_STATE__RSP_FLAG__ERROR_WARNING
Error warning

G_CAN__NODE__GET_STATE__RSP_FLAG__BUS_OFF
Bus off

TransceiverPin_Enable
State of **enable** transceiver pin

The following values are possible:

G_CAN__NODE__PIN_STATE__LOW
Pin state is low

G_CAN__NODE__PIN_STATE__HIGH
Pin state is high

G_CAN__NODE__PIN_STATE__NOT_AVAILABLE
Pin state is not available

TransceiverPin_NotStandby
State of **not standby** pin (see Node Pin State)

TransceiverPin_NotWake
State of **not wake** pin (see Node Pin State)

TransceiverPin_NotError
State of **not error** pin (see Node Pin State)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Node_Transceiver_SetMode

G_Can_Node_Transceiver_SetMode — Set transceiver mode

Description

Use this command to set or modify the transceiver mode.

Definition

```
G_Error_t
G_API
G_Can_Node_Transceiver_SetMode(
 const G_PortHandle_t portHandle,
 const G_Can_Node_Transceiver_Mode_t  mode
);
```

Parameters

`portHandle`
: Handle to the communication port

`mode`
: Transceiver mode

  The following values are possible:

  `G_CAN__NODE__TRANSCEIVER__MODE__NORMAL`
  : Normal mode

  `G_CAN__NODE__TRANSCEIVER__MODE__SLEEP`
  : Sleep mode

  `G_CAN__NODE__TRANSCEIVER__MODE__WAKE_UP`
  : Wakeup mode

  `G_CAN__NODE__TRANSCEIVER__MODE__HIGH_SPEED`
  : Highspeed transmission mode

  `G_CAN__NODE__TRANSCEIVER__MODE__SINGLE_WIRE_ON_CAN_HIGH`
  : Single wire mode on CAN high

  `G_CAN__NODE__TRANSCEIVER__MODE__SINGLE_WIRE_ON_CAN_LOW`
  : Single wire mode on CAN low

Not all transceiver modes can be set for all transceiver types (see table below).

	PCA 82C251	**TJA 1041(A)**	**TJA 1054**	**AU 5790**	**B 100115**
Transceiver type	high speed	high speed	low speed	single wire	truck and trailer
NORMAL	yes	yes	yes	yes	yes
SLEEP	no	yes	yes	yes	yes
WAKE_UP	no	no	no	yes	no
HIGH_SPEED	no	no	no	yes	no
SINGLE_WIRE_ON_CAN_HIGH	no	no	no	no	yes
SINGLE_WIRE_ON_CAN_LOW	no	no	no	no	yes

Table 4 Transceiver Types and Modes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Node_TxPath_Control

G_Can_Node_TxPath_Control — Control hardware transmit path

Description

Use this command to control the transmit path of the hardware.

Definition

```
G_Error_t
G_Can_Node_TxPath_Control(
 const G_PortHandle_t portHandle,
 const G_Can_Node_TxPath_Mode_t  mode
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
TX path mode

The following values are possible:

`G_CAN__NODE__TX_PATH__MODE__DISABLE`
Switch off transmit path (no sending possible)

This mode allows monitoring of CAN messages without sending a dominant acknowledge bit, even if a recessive acknowledge bit is received.

`G_CAN__NODE__TX_PATH__MODE__ENABLE`
Enable transmit path

`G_CAN__NODE__TX_PATH__MODE__NORMAL_OPERATION`
Normal operation of transmit path

`G_CAN__NODE__TX_PATH__MODE__RECESSIVE_LEVEL`
Set transmit path to a recessive level

`G_CAN__NODE__TX_PATH__MODE__DOMINANT_LEVEL`
Set transmit path to a dominant level

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Node_BusTermination_Enable

G_Can_Node_BusTermination_Enable — Enable internal bus termination

Description

Use this command to enable the internal bus termination.

The internal bus termination is enabled by default. Use G_Can_Node_BusTermination_Disable to disable the internal bus termination.

Internal bus termination is only available on dedicated devices.

If not supported by the device, the command will return with a firmware error code.

Definition

```
G_Error_t
G_Can_Node_BusTermination_Enable(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Node_BusTermination_Disable

G_Can_Node_BusTermination_Disable — Disable internal bus termination

Description

Use this command to disable the internal bus termination.

For re-enabling the internal bus termination, use G_Can_Node_BusTermination_Enable.

Internal bus termination is only available on dedicated devices.

If not supported by the device, the command will return with a firmware error code.

Definition

```
G_Error_t
G_Can_Node_BusTermination_Disable(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Node_InternalVBat_Enable

G_Can_Node_InternalVBat_Enable — Enable internal transceiver supply voltage

Description

Use this command to enable the internal transceiver supply voltage.

The internal transceiver supply voltage is enabled by default. Use G_Can_Node_InternalVBat_Disable to disable the internal transceiver supply voltage.

Internal transceiver supply voltage is only available on dedicated devices.

The internal transceiver supply voltage setting is stored permanently.

When connecting an external transceiver supply voltage, the internal transceiver supply voltage must be disabled.

Definition

```
G_Error_t
G_Can_Node_InternalVBat_Enable(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Node_InternalVBat_Disable

G_Can_Node_InternalVBat_Disable — Disable internal transceiver supply voltage

Description

Use this command to disable the internal transceiver supply voltage.

Use G_Can_Node_InternalVBat_Enable to re-enable the internal transceiver supply voltage.

Internal transceiver supply voltage is only available on dedicated devices.

The internal transceiver supply voltage setting is stored permanently.

Definition

```
G_Error_t
G_Can_Node_InternalVBat_Disable(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Node_BusIdleTime_Get

G_Can_Node_BusIdleTime_Get — Return Bus Idle Time

Description

This command returns the time since the last detected bus action (message transmission, message reception, error frame reception) in milliseconds.

Definition

```
G_Error_t
G_Can_Node_BusIdleTime_Get(
 const G_PortHandle_t portHandle,
 u32_t * const busIdleTime
);
```

Parameters

`portHandle`
: Handle to the communication port

`busIdleTime`
: Returns the time since the last detected bus action (message transmission, message reception, error frame reception) in milliseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Node_FdState_Get

G_Can_Node_FdState_Get — Get CAN FD state of CAN node

Description

Use this command to query the CAN FD state of the CAN node.

Definition

```
G_Error_t
G_Can_Node_FdState_Get(
 const G_PortHandle_t portHandle,
 const G_Can_Node_FdState_Get_CmdFlags_t  cmdFlags,
 G_Can_Node_FdState_Get_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_CAN__NODE__FD_STATE__GET__CMD_FLAG__NONE`
No flag is set

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_CAN__NODE__FD_STATE__GET__RSP_FLAG__NONE`
No flags is set

`G_CAN__NODE__FD_STATE__GET__RSP_FLAG__FD_SUPPORTED`
not set : CAN FD is not supported

set : CAN FD is supported

`FdMode`
CAN FD mode

The following values are possible:

`G_CAN__FD_MODE__NO_FD`
The node can only receive and transmit frames in **CAN** format

`G_CAN__FD_MODE__FD`
The node can receive and transmit frames in **CAN** or **CAN FD** format

G_CAN__FD_MODE__TX_FD
The node can receive frames in **CAN** or **CAN FD** format and transmit frames in **CAN FD** format only with **Bit Rate Switch** **disabled**

G_CAN__FD_MODE__TX_FD_BRS
The node can receive frames in **CAN** or **CAN FD** format and transmit frames in **CAN FD** format only with **Bit Rate Switch** **enabled** .

G_CAN__FD_MODE__TX_NO_FD
The node can receive frames in **CAN** or **CAN FD** format and transmit frames in **CAN** format only

TxBrs_Global
Global value for TX **Bit Rate Switch**

The following values are possible:

G_CAN__TX_BRS__GLOBAL__NONE
All FD frames are transmitted without **Bit Rate Switch**

G_CAN__TX_BRS__GLOBAL__ALL
All FD frames are transmitted with **Bit Rate Switch**

G_CAN__TX_BRS__GLOBAL__SPECIFIC
FD frames are transmitted with **Bit Rate Switch** if the **Bit Rate Switch** of the corresponding transmission object is set

This property is available in FdMode == FD only

TxBrs_InitValue
Initial value for TX **Bit Rate Switch**

Transmission objects initialized as FD messages derive their BRS bit from this property. This is necessary, because most commands don't have a command parameter to specify the BRS bit.

The following values are possible:

G_CAN__TX_BRS__INIT_VALUE__DISABLED
TX **Bit Rate Switch** is disabled

G_CAN__TX_BRS__INIT_VALUE__ENABLED
TX **Bit Rate Switch** is enabled

This property is available in FdMode == FD only

TxEsi_Global
Global value of TX error state indicator

The following values are possible:

G_CAN__TX_ESI__GLOBAL__DEPENDS_ON_ERROR_PASSIVE_FLAG
Esi value depends on error passive flag

G_CAN__TX_ESI__GLOBAL__IS_SET
Esi is set

G_CAN__TX_ESI__GLOBAL__DEPENDS_ON_ERROR_PASSIVE_FLAG_AND_ESI_BIT
Esi value depends on error passive flag and esi bit

TxEsi_InitValue
Initial value of TX error state indicator

Transmission objects initialized as FD messages derive their ESI bit from this property. This is necessary, because most commands don't have a command parameter to specify the ESI bit.

The following values are possible:

`G_CAN__TX_ESI__INIT_VALUE__ERROR_ACTIVE`
Esi initial value is "error active"

`G_CAN__TX_ESI__INIT_VALUE__ERROR_PASSIVE`
Esi initial value is "error passive"

`FdFrameFormatSpec`
Specification of the CAN FD format that is used

The following values are possible:

`G_CAN__FD_FRAME_FORMAT_SPEC__ISO11898_1`
ISO 11898 1

`G_CAN__FD_FRAME_FORMAT_SPEC__BOSCH_CAN_FD_SPEC_1_0`
Bosch CAN FD spec 1.0

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`BaudRateFd`
CAN controller baud rate in baud for CAN FD frames with **Bit Rate Switch** (e.g. 2000000 for 2 MBaud)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Node_FdState_Set

G_Can_Node_FdState_Set — Set CAN FD parameters of CAN node

Description

Use this command to set CAN FD parameters of the CAN node.

Definition

```
G_Error_t
G_Can_Node_FdState_Set(
 const G_PortHandle_t portHandle,
 const G_Can_Node_FdState_Set_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The command flags define which parameters are set. If a command flag is not set, the corresponding parameter is disregarded.

The following values are possible and can be combined:

`G_CAN__NODE__FD_STATE__SET__CMD_FLAG__NONE`
No flags is set

`G_CAN__NODE__FD_STATE__SET__CMD_FLAG__SET_TX_BRS_GLOBAL`
Set TX Bit Rate Switch global value

`G_CAN__NODE__FD_STATE__SET__CMD_FLAG__SET_TX_BRS_INIT_VALUE`
Set TX Bit Rate Switch initial value

`G_CAN__NODE__FD_STATE__SET__CMD_FLAG__SET_TX_ESI_GLOBAL`
Set TX Error State Indicator global value

`G_CAN__NODE__FD_STATE__SET__CMD_FLAG__SET_TX_ESI_INIT_VALUE`
Set TX Error State Indicator initial value

`G_CAN__NODE__FD_STATE__SET__CMD_FLAG__SET_FD_FRAME_FORMAT_SPEC`
Set CAN FD frame format specification

`TxBrs_Global`
Global value for TX **Bit Rate Switch**

The following values are possible:

`G_CAN__TX_BRS__GLOBAL__NONE`
All FD frames are transmitted without **Bit Rate Switch**

G_CAN__TX_BRS__GLOBAL__ALL
All FD frames are transmitted with **Bit Rate Switch**

G_CAN__TX_BRS__GLOBAL__SPECIFIC
FD frames are transmitted with **Bit Rate Switch** if the **Bit Rate Switch** of the corresponding transmission object is set

This property is available in FdMode == FD only

TxBrs_InitValue
Initial value for TX **Bit Rate Switch**

Transmission objects initialized as FD messages derive their BRS bit from this property. This is necessary, because most commands don't have a command parameter to specify the BRS bit.

The following values are possible:

G_CAN__TX_BRS__INIT_VALUE__DISABLED
TX **Bit Rate Switch** is disabled

G_CAN__TX_BRS__INIT_VALUE__ENABLED
TX **Bit Rate Switch** is enabled

This property is available in FdMode == FD only

TxEsi_Global
Global value of TX error state indicator

The following values are possible:

G_CAN__TX_ESI__GLOBAL__DEPENDS_ON_ERROR_PASSIVE_FLAG
Esi value depends on error passive flag

G_CAN__TX_ESI__GLOBAL__IS_SET
Esi is set

G_CAN__TX_ESI__GLOBAL__DEPENDS_ON_ERROR_PASSIVE_FLAG_AND_ESI_BIT
Esi value depends on error passive flag and esi bit

TxEsi_InitValue
Initial value of TX error state indicator

Transmission objects initialized as FD messages derive their ESI bit from this property. This is necessary, because most commands don't have a command parameter to specify the ESI bit.

The following values are possible:

G_CAN__TX_ESI__INIT_VALUE__ERROR_ACTIVE
Esi initial value is "error active"

G_CAN__TX_ESI__INIT_VALUE__ERROR_PASSIVE
Esi initial value is "error passive"

FdFrameFormatSpec
Specification of the CAN FD format that is used

The following values are possible:

G_CAN__FD_FRAME_FORMAT_SPEC__ISO11898_1
ISO 11898 1

G_CAN__FD_FRAME_FORMAT_SPEC__BOSCH_CAN_FD_SPEC_1_0
Bosch CAN FD spec 1.0

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`BaudRateFd`
CAN controller baud rate in baud for CAN FD frames with **Bit Rate Switch** (e.g. 2000000 for 2 MBaud)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

9 Tar

These commands are used to control and configure the **TAR** functionality.

TAR stands for "**T** ransmit **A** fter **R** eception" and means sending one or more CAN Messages triggered by the reception of a specific CAN Message with specific data.

G_Can_Tar_Config_Simple

G_Can_Tar_Config_Simple — Simple **TAR** configuration

Description

This command is used to set the simple **TAR** configuration.

In contrast to G_Can_Tar_Config_Standard , the IDs of the trigger and sending messages (`rxId` and `txId`) are fixed.

When running **TAR** in simple configuration, the data bytes of sent messages are defined by command G_Can_Tar_AddTxItems_Simple .

Definition

```
G_Error_t
G_Can_Tar_Config_Simple(
 const G_PortHandle_t  portHandle,
 const u8_t channel,
 const u32_t  rxId,
 const u32_t  txId
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
TAR channel

`rxId`
Identifier of trigger message (0..0x7FF or 0..0x1FFFFFFF)

`txId`
Identifier of sending message (0..0x7FF or 0..0x1FFFFFFF)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tar_Config_Standard

G_Can_Tar_Config_Standard — Standard **TAR** configuration

Description

This command is used to set the standard **TAR** configuration. Identifier, data length, data bytes and their corresponding byte masks for the trigger message are defined. The command offers more detailed control over **TAR** than G_Can_Tar_Config_Simple .

When running **TAR** in standard configuration mode, you can define messages that are sent after a trigger message is received by command G_Can_Tar_AddTxItems_Standard .

Definition

```
G_Error_t
G_Can_Tar_Config_Standard(
 const G_PortHandle_t  portHandle,
 const u8_t channel,
 const u32_t  rxId,
 const u32_t  rxIdMask,
 const u8_t  rxDlc,
 const u8_t  rxDlcMask,
 const u8_t  rxData[8],
 const u8_t  rxDataMask[8]
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
TAR channel

`rxId`
Identifier of trigger message (0..0x7FF or 0..0x1FFFFFFF)

`rxIdMask`
Identifier mask of trigger message (e.g. 0xFFFFFFFF)

`rxDlc`
Data length of trigger message (0..8)

`rxDlcMask`
Data length mask of trigger message (0x00..0xFF)

0x00 : the received data length is not considered

0xFF : the received data length must comply with `rxDlc`

`rxData`
Data bytes of trigger message (0..7)

`rxDataMask`
Data mask of trigger message (0..7)

(set bits denote that the corresponding received bits must comply with the corresponding bits of `rxData`)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tar_Config_Off

G_Can_Tar_Config_Off — Reset **TAR** configuration

Description

This command is used to reset **TAR** configuration.

Definition

```
G_Error_t
G_Can_Tar_Config_Off(
 const G_PortHandle_t  portHandle,
 const u8_t channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: **TAR** channel

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tar_AddTxItems_Simple

G_Can_Tar_AddTxItems_Simple — Add **TAR** transmit items

Description

This command adds **TAR** transmit items for the selected channel.

The **TAR** on the selected channel has to be configured with G_Can_Tar_Config_Simple before adding simple **TAR** items.

Definition

```
G_Error_t
G_Can_Tar_AddTxItems_Simple(
 const G_PortHandle_t  portHandle,
 const u8_t channel,
 const u16_t  numberOfItems,
 const G_Can_Tar_TxItem_Simple_t * const items
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
TAR channel

`numberOfItems`
Number of simple **TAR** items to add

`items`
Pointer to structure array with simple **TAR** items

The structure contains the following elements:

- `Delay`

  This parameter has a double meaning:
  In the case the `G_CAN__TAR__TX_ITEM__FLAG__BREAK` flag is set, `Delay` indicates the number of trigger conditions to be ignored (that means, nothing is sent for `Delay` times).
  If the `G_CAN__TAR__TX_ITEM__FLAG__BREAK` flag is NOT set, `Delay` indicates the time in milliseconds between the trigger condition and the sending of the CAN message of this transmit entry.
- `Flags`

  TAR item flags

  The following values are possible and can be combined:
  - `G_CAN__TAR__TX_ITEM__FLAG__NONE`

    No item flag set
  - `G_CAN__TAR__TX_ITEM__FLAG__SEND_NEXT`

    Send next item immediately without waiting for trigger message
  - `G_CAN__TAR__TX_ITEM__FLAG__RESTART`

    After executing this entry the first added transmit entry is executed again

- `G_CAN__TAR__TX_ITEM__FLAG__BREAK`

  The trigger condition is ignored `Delay` times (nothing is sent)
- `Dlc`

  Data length (0..8)
- `Data`

  Data bytes (0..7)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tar_AddTxItems_Simple2

G_Can_Tar_AddTxItems_Simple2 — Add simple **TAR** transmit items with support for large data (e.g. CAN FD)

Description

This command adds **TAR** simple transmit items for the selected channel.

In contrast to command G_Can_Tar_AddTxItems_Simple this command provides support for large data payload (e.g. CAN FD).

The **TAR** on the selected channel has to be configured with G_Can_Tar_Config_Simple before adding simple **TAR** items.

Definition

```
G_Error_t
G_Can_Tar_AddTxItems_Simple2(
 const G_PortHandle_t  portHandle,
 const u8_t channel,
 const u16_t  numberOfItems,
 const G_Can_Tar_TxItem_Simple2_t * const items
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
TAR channel

`numberOfItems`
Number of simple **TAR** items to add

`items`
Pointer to structure array with simple **TAR** items

The structure contains the following elements:

- `Delay`

  This parameter has a double meaning:
  In the case the `G_CAN__TAR__TX_ITEM__FLAG__BREAK` flag is set, `Delay` indicates the number of trigger conditions to be ignored (that means, nothing is sent for `Delay` times).
  If the `G_CAN__TAR__TX_ITEM__FLAG__BREAK` flag is NOT set, `Delay` indicates the time in milliseconds between the trigger condition and the sending of the CAN message of this transmit entry.
- `Flags`

  TAR item flags

  The following values are possible and can be combined:
  - `G_CAN__TAR__TX_ITEM__FLAG__NONE`

    No item flag set
  - `G_CAN__TAR__TX_ITEM__FLAG__SEND_NEXT`

Send next item immediately without waiting for trigger message

- `G_CAN__TAR__TX_ITEM__FLAG__RESTART`

  After executing this entry the first added transmit entry is executed again

- `G_CAN__TAR__TX_ITEM__FLAG__BREAK`

  The trigger condition is ignored `Delay` times (nothing is sent)

- `Dlc`

  Data length code (0..15)

DLC	Data Length (bytes)
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	12
10	16
11	20
12	24
13	32
14	48
15	64

Table 5 DLC to data length

- `Data`

  Data bytes (0..63)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tar_AddTxItems_Standard

G_Can_Tar_AddTxItems_Standard — Add **TAR** transmit items

Description

This command adds **TAR** transmit items for the selected channel.

The **TAR** on the selected channel has to be configured with G_Can_Tar_Config_Standard before adding standard **TAR** items.

Definition

```
G_Error_t
G_Can_Tar_AddTxItems_Standard(
 const G_PortHandle_t  portHandle,
 const u8_t channel,
 const u16_t  numberOfItems,
 const G_Can_Tar_TxItem_Standard_t * const items
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
TAR channel

`numberOfItems`
Number of standard **TAR** items to add

`items`
Pointer to structure array with standard **TAR** items

The structure contains the following elements:

- `Id`

  Transmit identifier (0..0x7FF or 0..0x1FFFFFFF)
- `Delay`

  This parameter has a double meaning:
  In the case the `G_CAN__TAR__TX_ITEM__FLAG__BREAK` flag is set, `Delay` indicates the number of trigger conditions to be ignored (that means, nothing is sent for `Delay` times).
  If the `G_CAN__TAR__TX_ITEM__FLAG__BREAK` flag is NOT set, `Delay` indicates the time in milliseconds between the trigger condition and the sending of the CAN message of this transmit entry.
- `Flags`

  TAR item flags

  The following values are possible and can be combined:
  - `G_CAN__TAR__TX_ITEM__FLAG__NONE`

    No item flag set
  - `G_CAN__TAR__TX_ITEM__FLAG__SEND_NEXT`

    Send next item immediately without waiting for trigger message

- `G_CAN__TAR__TX_ITEM__FLAG__RESTART`

  After executing this entry the first added transmit entry is executed again
- `G_CAN__TAR__TX_ITEM__FLAG__BREAK`

  The trigger condition is ignored `Delay` times (nothing is sent)

- `Dlc`

  Data length (0..8)
- `Data`

  Data bytes (0..7)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tar_AddTxItems_Standard2

G_Can_Tar_AddTxItems_Standard2 — Add standard **TAR** transmit items with support for large data (e.g. CAN FD)

Description

This command adds standard **TAR** transmit items for the selected channel.

In contrast to command G_Can_Tar_AddTxItems_Standard this command provides support for large data payload (e.g. CAN FD).

The **TAR** on the selected channel has to be configured with G_Can_Tar_Config_Standard before adding standard **TAR** items.

Definition

```
G_Error_t
G_Can_Tar_AddTxItems_Standard2(
 const G_PortHandle_t  portHandle,
 const u8_t channel,
 const u16_t  numberOfItems,
 const G_Can_Tar_TxItem_Standard2_t * const items
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
TAR channel

`numberOfItems`
Number of standard **TAR** items to add

`items`
Pointer to structure array with standard **TAR** items

The structure contains the following elements:

- `Id`

  Transmit identifier (0..0x7FF or 0..0x1FFFFFFF)
- `Delay`

  This parameter has a double meaning:
  In the case the `G_CAN__TAR__TX_ITEM__FLAG__BREAK` flag is set, `Delay` indicates the number of trigger conditions to be ignored (that means, nothing is sent for `Delay` times).
  If the `G_CAN__TAR__TX_ITEM__FLAG__BREAK` flag is NOT set, `Delay` indicates the time in milliseconds between the trigger condition and the sending of the CAN message of this transmit entry.
- `Flags`

  TAR item flags

  The following values are possible and can be combined:
  - `G_CAN__TAR__TX_ITEM__FLAG__NONE`

No item flag set

- `G_CAN__TAR__TX_ITEM__FLAG__SEND_NEXT`

  Send next item immediately without waiting for trigger message
- `G_CAN__TAR__TX_ITEM__FLAG__RESTART`

  After executing this entry the first added transmit entry is executed again
- `G_CAN__TAR__TX_ITEM__FLAG__BREAK`

  The trigger condition is ignored `Delay` times (nothing is sent)

- `Dlc`

  Data length code (0..15)

DLC	Data Length (bytes)
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	12
10	16
11	20
12	24
13	32
14	48
15	64

Table 6 DLC to data length

- `Data`

  Data bytes (0..63)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tar_GetState

G_Can_Tar_GetState — Query **TAR** status

Description

This command is used to query the status of **TAR** functionality for the **TAR** channel defined by `channel`.

Definition

```
G_Error_t
G_Can_Tar_GetState(
 const G_PortHandle_t  portHandle,
 const u8_t channel,
 G_Can_Tar_GetState_Type_t * const type,
 G_Can_Tar_GetState_RspFlags_t * const rspFlags,
 u32_t * const numberOfTxItems,
 u32_t * const currentTxItem
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
TAR channel

`type`
Pointer to variable for **TAR** type

The following response values are possible:

- `G_CAN__TAR__GET_STATE__TYPE__UNKNOWN`

  Unknown **TAR** configuration
- `G_CAN__TAR__GET_STATE__TYPE__OFF`

  TAR deactivated
- `G_CAN__TAR__GET_STATE__TYPE__SIMPLE`

  Simple **TAR** configuration

  (fixed receiving ID, fixed sending ID)
- `G_CAN__TAR__GET_STATE__TYPE__STANDARD`

  Standard **TAR** configuration

  (receiving ID mask, receiving data mask, any sending ID)

`rspFlags`
Response flags

The following response values are possible and can be combined:

- `G_CAN__TAR__GET_STATE__RSP_FLAG__NONE`

  No response flag set
- `G_CAN__TAR__GET_STATE__RSP_FLAG__STARTED`

TAR functionality has been started

- `G_CAN__TAR__GET_STATE__RSP_FLAG__BUSY`

  TAR is currently responding to a trigger message

`numberOfTxItems`
Total number of **TAR** transmit entries

`currentTxItem`
Number of the **TAR** transmit entry executed next when meeting the trigger condition (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tar_Start

G_Can_Tar_Start — Start **TAR**

Description

This command is used to start **TAR** functionality for the **TAR** channel defined by `channel`.

Definition

```
G_Error_t
G_Can_Tar_Start(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Can_Tar_Start_Mode_t  mode
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
TAR channel

`mode`
Start mode

The following values are possible:

- `G_CAN__TAR__START__MODE__RESTART`

  Start with the first **TAR** entry
- `G_CAN__TAR__START__MODE__RESUME`

  Start with the successor of the **TAR** transmit entry executed last

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tar_Stop

G_Can_Tar_Stop — Stop **TAR**

Description

This command is used to stop **TAR** functionality for the **TAR** channel defined by `channel`.

Definition

```
G_Error_t
G_Can_Tar_Stop(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: **TAR** channel

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

10 TP

These commands are used to control the **T**ransport **P**rotocol (**TP**).

G_Can_Tp_GetChannel

G_Can_Tp_GetChannel — Request the multisession channel

Description

This command requests a multisession channel.

The command is used for the dynamic administration of multisession channels. A multisession channel is requested and the corresponding concrete channel number is returned into the `channel`.

This multisession channel administration may be required if several applications or software threads operate with the firmware and with the same multisession channels.

If always only one application works with the firmware, a multisession channel administration of the application is sufficient and no administration by the firmware is necessary.

Definition

```
G_Error_t
G_Can_Tp_GetChannel(
 const G_PortHandle_t portHandle,
 u8_t * const channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Pointer to the variable `channel`. The number of the assigned multisession channel (starting with 0) is returned in this variable.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tp_GetChannel_Async

G_Can_Tp_GetChannel_Async — Request of a multisession channel (by asynchronous request)

Description

This command starts the asynchronous request for a multisession channel. If a response is available, the callback function set with G_Can_Tp_GetChannel_Async_AddCallback will be called.

The command is used for the dynamic administration of multisession channels. A multisession channel is requested and the corresponding concrete channel number is returned in the callback function with `channel`.

This multisession channel administration may be required if several applications or software threads operate with the firmware and with the same multisession channels.

If always only one application works with the firmware, a multisession channel administration of the application is sufficient and no administration by the firmware is necessary.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_API_CAN_DLL G_Error_t
G_Can_Tp_GetChannel_Async(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tp_GetChannel_Async_Callback

G_Can_Tp_GetChannel_Async_Callback — Callback function for the asynchronous request of a multi-session channel

Description

This function will be automatically called if a response for the asynchronous request of a multisession channel with G_Can_Tp_GetChannel_Async is available. It must be entered by the user and set with G_Can_Tp_GetChannel_Async_AddCallback.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Can_Tp_GetChannel_Async_Callback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Assigned multisession channel

G_Can_Tp_GetChannel_Async_AddCallback

G_Can_Tp_GetChannel_Async_AddCallback — Set the callback function for the asynchronous request of a multisession channel

Description

This command defines a callback function for the asynchronous request of a multisession channel with G_Can_Tp_GetChannel_Async.

Definition

```
G_API_CAN_DLL G_Error_t
G_Can_Tp_GetChannel_Async_AddCallback(
 const G_PortHandle_t  portHandle,
 const G_Can_Tp_GetChannel_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`callback`
Function pointer of the callback function (see G_Can_Tp_GetChannel_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tp_GetChannel_Async_RemoveCallback

G_Can_Tp_GetChannel_Async_RemoveCallback — Remove the callback function for the asynchronous request of a multisession channel

Description

This command removes a callback function for the asynchronous request of a multisession channel.

Definition

```
G_API_CAN_DLL G_Error_t
G_Can_Tp_GetChannel_Async_RemoveCallback(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tp_FreeChannel

G_Can_Tp_FreeChannel — Release the multisession channel

Description

This command releases the multisession channel given by `channel`.

The command is used for the dynamic administration of multisession channels. The multisession channel defined under `channel` is released.

This multisession channel administration may be required if several applications or software threads operate with the firmware and with the same multisession channels.

If always only one application works with the firmware, a multisession channel administration of the application is sufficient and no administration by the firmware is necessary.

Definition

```
G_Error_t
G_Can_Tp_FreeChannel(
 const G_PortHandle_t portHandle,
 const u8_t channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tp_Config_GmLan

G_Can_Tp_Config_GmLan — Configuring the multisession channel

Description

This command configures the multisession channel given by `channel` for the transport protocol "GM-LAN".

Definition

```
G_Error_t
G_Can_Tp_Config_GmLan(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Can_Tp_Config_GmLan_Parameters_t * const gmLanParameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`gmLanParameters`
Pointer to a variable of type **G_Can_Tp_Config_GmLan_Parameters_t**

The containing "reserved" bytes must be completed with `0`.

The structure **G_Can_Tp_Config_GmLan_Parameters_t** consists of the following items:

- `USDT`
  Structure for USDT (**U** nacknowledged **S** egmented **D** ata **T** ransfer)

  The structure consists of the following items:
  - `PhysicalSourceAddress`
    Own physical control unit address

    (not relevant in the case of `PhysicalAddressingFormat`

    `=` `G_CAN__TP__CONFIG__GMLAN__ADDRESSING_FORMAT__NORMAL`

)
  - `PhysicalTargetAddress`
    Physical control unit address of the test object

    (not relevant in the case of `PhysicalAddressingFormat`

    `=` `G_CAN__TP__CONFIG__GMLAN__ADDRESSING_FORMAT__NORMAL`

)
  - `FunctionalSourceAddress`
    Own functional control unit address

    (only for simulating ECUs, not relevant in the case of `FunctionalAddressing-Format`

    `=` `G_CAN__TP__CONFIG__GMLAN__ADDRESSING_FORMAT__NORMAL`

)

- `FunctionalTargetAddress`
  Functional control unit address of the test object

  (disregarded if `FunctionalAddressingFormat`

  `= G_CAN__TP__CONFIG__GMLAN__ADDRESSING_FORMAT__NORMAL`

)
- `PhysicalRequestId`
  Physical request identifier
- `PhysicalResponseId`
  Physical response identifier
- `AllNodeFunctionalRequestId`
  Functional request identifier (e.g. 0x101)
- `PhysicalAddressingFormat`
  Variable of type **G_Can_Tp_Config_GmLan_AddressingFormat_t**

  Physical addressing format (in general

  `G_CAN__TP__CONFIG__GMLAN__ADDRESSING_FORMAT__NORMAL`)

  The following return values are available:
  - `G_CAN__TP__CONFIG__GMLAN__ADDRESSING_FORMAT__NORMAL`
  - `G_CAN__TP__CONFIG__GMLAN__ADDRESSING_FORMAT__EXTENDED`
  - `G_CAN__TP__CONFIG__GMLAN__ADDRESSING_FORMAT__MIXED`
- `FunctionalAddressingFormat`
  Variable of type **G_Can_Tp_Config_GmLan_AddressingFormat_t**

  Functional addressing format (in general

  `G_CAN__TP__CONFIG__GMLAN__ADDRESSING_FORMAT__EXTENDED`)

  The following return values are available:
  - `G_CAN__TP__CONFIG__GMLAN__ADDRESSING_FORMAT__NORMAL`
  - `G_CAN__TP__CONFIG__GMLAN__ADDRESSING_FORMAT__EXTENDED`
  - `G_CAN__TP__CONFIG__GMLAN__ADDRESSING_FORMAT__MIXED`
- `BlockSize`
  Block size (e.g. 8)

  `0`: Between the `ConsecutiveFrames`

  no `FlowControl` frames are expected

  1..255 : After sending of **1..255** `ConsecutiveFrames`

  a `FlowControl` frame is expected
- `SeparationTime`
  Time between CAN frames to be met by the communication partner,

  in milliseconds, e.g. 0 milliseconds
- `Flags`
  Variable of type **G_Can_Tp_Config_GmLan_CmdFlags_t**

  The following return flags are available:
  - `G_CAN__TP__CONFIG__GMLAN__CMD_FLAG__NONE`

    No flag, if no other has to be transferred
  - `G_CAN__TP__CONFIG__GMLAN__CMD_FLAG__USE_OWN_SEPARATION_TIME`

For segmented sending, the time between the CAN messages which is indicated under `OwnSeparationTime` is met.

If this flag is not set, for the segmented sending the value of the `SeparationTime` received from the communication partner will be met.

- `OwnSeparationTime`
  Self-meeting time between two CAN messages within a segmented data exchange,

  in milliseconds, e.g. 10 milliseconds
- `reserved1`
  Reserved byte, must be initialized with `0`
- `reserved2`
  Reserved byte, must be initialized with `0`
- `reserved3`
  Reserved byte, must be initialized with `0`
- `TimeoutAs`
  Sending timeout send-site,

  in milliseconds, e.g. 250 milliseconds
- `TimeoutAr`
  Sending timeout receive-site,

  in milliseconds, e.g. 250 milliseconds
- `TimeoutBs`
  `FlowControl` Frame receiving timeout send-site,

  in milliseconds, e.g. 250 milliseconds
- `TimeoutCr`
  `ConsecutiveFrame` receiving timeout receive-site,

  in milliseconds, e.g. 250 milliseconds

- `UUDT`
  Structure for UUDT

  The structure consists of the following items:
  - `ResponseId`
    Response Identifier (UUDT = **U** nacknowledged **U** nsegmented **D** ata **T** ransfer)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tp_Config_Iso15765

G_Can_Tp_Config_Iso15765 — Config transport protocol ISO 15765

Description

Use this command to configure the transport protocol ISO 15765.

Definition

```
G_Error_t
G_Can_Tp_Config_Iso15765(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Can_Tp_Config_Iso15765_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_CAN__TP__CONFIG__ISO_15765__CMD_FLAG__NONE`
No flag is set

`G_CAN__TP__CONFIG__ISO_15765__CMD_FLAG__DISABLE_TX_SIDE__PHYS`
0 : enable physical sender side

1 : disable physical sender side

`G_CAN__TP__CONFIG__ISO_15765__CMD_FLAG__DISABLE_RX_SIDE__PHYS`
0 : enable physical receiver side

1 : disable physical receiver side

`G_CAN__TP__CONFIG__ISO_15765__CMD_FLAG__DISABLE_TX_SIDE__FUNC`
0 : enable functional sender side

1 : disable functional sender side

`G_CAN__TP__CONFIG__ISO_15765__CMD_FLAG__ENABLE_RX_SIDE__FUNC`
0 : disable functional receiver side

1 : enable functional receiver side

`G_CAN__TP__CONFIG__ISO_15765__CMD_FLAG__START_SN_WITH_ZERO`
Start sequence number with 0

The SequenceNumber (SN) starts with `0` in the first **Consecutive Frame** (CF) after a **Flow Control Frame** (FC).

If this flag is not set, the SequenceNumber (SN) starts with `1` in the first **Consecutive Frame** (CF) after a **Flow Control Frame** (FC).

`G_CAN__TP__CONFIG__ISO_15765__CMD_FLAG__SEP_TIME_BETWEEN_FC_AND_CF`
Use **Separation Time** derived from the communication partner between **Flow Control Frame** and **Consecutive Frame** .

`G_CAN__TP__CONFIG__ISO_15765__CMD_FLAG__SET_DLC_TO_NUM_OF_USED_BYTES`
Set DLC to number of used bytes

The data length of the CAN message is set to the number of the used data bytes (the CAN message contains the used data bytes only).

`G_CAN__TP__CONFIG__ISO_15765__CMD_FLAG__USE_PADDING_BYTE`
Use value specified by `PaddingByte` for initialization of unused bytes.

`G_CAN__TP__CONFIG__ISO_15765__CMD_FLAG__IGNORE_CF_WITH_WRONG_SN`
Ignore a received **CF** (consecutive frame) if the **SN** (sequence number) is not expected.

`G_CAN__TP__CONFIG__ISO_15765__CMD_FLAG__IGNORE_RECEIVED_ST_MIN`
Ignore a received **Separation Time Minimum** .

`G_CAN__TP__CONFIG__ISO_15765__CMD_FLAG__USE_TX_DL__PHYSICAL`
Use `TxDl_Physical` to configure the maximum usable payload length in bytes on physical sender side.

`G_CAN__TP__CONFIG__ISO_15765__CMD_FLAG__USE_TX_DL__FUNCTIONAL`
Use `TxDl_Functional` to configure the maximum usable payload length in bytes on functional sender side.

`LocalAddress_Physical`
Physical local address

`RemoteAddress_Physical`
Physical remote address

`LocalAddress_Functional`
Functional local address

`RemoteAddress_Functional`
Functional remote address

`LocalCanId_Physical`
Physical local CAN identifier

`RemoteCanId_Physical`
Physical remote CAN identifier

`LocalCanId_Functional`
Functional local CAN identifier

`RemoteCanId_Functional`
Functional remote CAN identifier

`AddressingFormat_Physical`
Physical addressing format

The following values are possible:

G_CAN__TP__ISO_15765__ADDRESSING_FORMAT__NORMAL
Normal addressing format (regular CAN identifiers are used)

G_CAN__TP__ISO_15765__ADDRESSING_FORMAT__EXTENDED
Extended addressing format (extended CAN identifiers are used)

G_CAN__TP__ISO_15765__ADDRESSING_FORMAT__MIXED
Mixed addressing format (regular and extended CAN identifiers are used)

AddressingFormat_Functional
Functional addressing format

The following values are possible:

G_CAN__TP__ISO_15765__ADDRESSING_FORMAT__NORMAL
Normal addressing format (regular CAN identifiers are used)

G_CAN__TP__ISO_15765__ADDRESSING_FORMAT__EXTENDED
Extended addressing format (extended CAN identifiers are used)

G_CAN__TP__ISO_15765__ADDRESSING_FORMAT__MIXED
Mixed addressing format (regular and extended CAN identifiers are used)

BS
ISO 15765-2 name: BS

BlockSize , this value is sent to the remote entity

see ISO 15765-2:2016(E) 9.6.5.3 BlockSize (BS) parameter definition

STmin
ISO 15765-2 name: STmin

SeparationTime minimum , this value is sent to the remote entity

see ISO 15765-2:2016(E) 9.6.5.4 SeparationTime minimum (STmin) parameter definition

TimeoutAs
[micro seconds] tx timeout sender side (e.g. 100000)

TimeoutAr
[micro seconds] tx timeout receiver side (e.g. 100000)

TimeoutBs
[micro seconds] FC rx timeout sender side (e.g. 100000)

TimeoutCr
[micro seconds] CF rx timeout receiver side (e.g. 100000)

TimeBrBetweenFfAndFc
[micro seconds] time between reception of a "first frame" and transmission of a "flow control". (e.g. 0)

TimeBrBetweenCfAndFc
[micro seconds] time between reception of a "consecutive frame" and transmission of a "flow control". (e.g. 0)

TimeCs
[micro seconds] time until transmission of the next "consecutive frame" (e.g. 0)

The time can be longered by the received "STmin" parameter if flag G_CAN__TP__CONFIG__ISO_15765__CMD_FLAG__IGNORE_RECEIVED_ST_MIN is not set.

`PaddingByte`
For initialization of unused bytes.

This parameter is used only if flag `G_CAN__TP__CONFIG__ISO_15765__CMD_FLAG__USE_PADDING_BYTE` is set.

`TxDl_Physical`
This parameter is used only if flag `G_CAN__TP__CONFIG__ISO_15765__CMD_FLAG__USE_TX_DL__PHYSICAL` is set.

Configures the maximum usable payload length in bytes on physical sender side (8, 12, 16, 20, 24, 32, 48, 64).

default value: 8

`TxDl_Functional`
This parameter is used only if flag G_CAN__TP__CONFIG__ISO_15765__CMD_FLAG__USE_TX_DL__FUNCTIONAL is set.

Configures the maximum usable payload length in bytes on functional sender side (8, 12, 16, 20, 24, 32, 48, 64).

The default value depends on FDF bit (bit 30) in `LocalCanId_Functional`:

default value: **8** if FDF bit is not set, **64** if FDF bit is set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tp_Config_IsoTp

G_Can_Tp_Config_IsoTp — Configure the multisession channel

Description

This command configures the multisession channel indicated by `channel` for the transport protocol "ISOTP".

Definition

```
G_Error_t
G_Can_Tp_Config_IsoTp(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Can_Tp_Config_IsoTp_Parameters_t * const isoTpParameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`isoTpParameters`
Pointer to a variable of type **G_Can_Tp_Config_IsoTp_Parameters_t**

The structure **G_Can_Tp_Config_IsoTp_Parameters_t** consists of the following items:

- `PhysicalSourceAddress`
  Own physical control unit address

  (not relevant in the case of `PhysicalAddressingFormat`

  `= G_CAN__TP__CONFIG__ISOTP__ADDRESSING_FORMAT__NORMAL`

)
- `PhysicalTargetAddress`
  Test object related physical address of the control unit

  (not relevant in the case of `PhysicalAddressingFormat`

  `= G_CAN__TP__CONFIG__ISOTP__ADDRESSING_FORMAT__NORMAL`

)
- `FunctionalSourceAddress`
  Own functional control unit address

  (only for simulating ECUs, not relevant in the case of `FunctionalAddressingFormat`

  `= G_CAN__TP__CONFIG__ISOTP__ADDRESSING_FORMAT__NORMAL`

)
- `FunctionalTargetAddress`
  Functional control unit address of the test object

  (there is no need for in the case of `FunctionalAddressingFormat`

  `= G_CAN__TP__CONFIG__ISOTP__ADDRESSING_FORMAT__NORMAL`

)

- `PhysicalSourceId`
  Own physical identifier
- `PhysicalTargetId`
  Physical test object identifier
- `FunctionalSourceId`
  Own functional identifier
- `FunctionalTargetId`
  Functional test object identifier
- `PhysicalAddressingFormat`
  Variable of type **G_Can_Tp_Config_IsoTp_AddressingFormat_t**. Physical addressing format (in general `G_CAN__TP__CONFIG__ISOTP__ADDRESSING_FORMAT__NORMAL`)

  The following values are available:
  - `G_CAN__TP__CONFIG__ISOTP__ADDRESSING_FORMAT__NORMAL`
  - `G_CAN__TP__CONFIG__ISOTP__ADDRESSING_FORMAT__EXTENDED`
  - `G_CAN__TP__CONFIG__ISOTP__ADDRESSING_FORMAT__MIXED`
- `FunctionalAddressingFormat`
  Variable of type **G_Can_Tp_Config_IsoTp_AddressingFormat_t**. Functional addressing format (in general `G_CAN__TP__CONFIG__ISOTP__ADDRESSING_FORMAT__EXTENDED`)

  The following values are available:
  - `G_CAN__TP__CONFIG__ISOTP__ADDRESSING_FORMAT__NORMAL`
  - `G_CAN__TP__CONFIG__ISOTP__ADDRESSING_FORMAT__EXTENDED`
  - `G_CAN__TP__CONFIG__ISOTP__ADDRESSING_FORMAT__MIXED`
- `BlockSize`
  Block size (e.g. 8)

  `0`: Between the `ConsecutiveFrames` no `FlowControl` frames are expected

  1..255 : After sending of **1..255** `ConsecutiveFrames` a `FlowControl` frame is expected
- `SeparationTime`
  Time between CAN frames to be met by the communication partner,

  Given in milliseconds, e.g. 0 milliseconds
- `Flags`
  Variable of type **G_Can_Tp_Config_IsoTp_CmdFlags_t**. In general all flags are `0`.

  The following flags are available:
  - `G_CAN__TP__CONFIG__ISOTP__CMD_FLAG__NONE`

    No flag, if no other has to be transferred
  - `G_CAN__TP__CONFIG__ISOTP__CMD_FLAG__USE_OWN_SEPARATION_TIME`

    For segmented sending the time given under `OwnSeparationTime` between the CAN messages is met

    

    If this flag is not set, the value of the `SeparationTime` received from the communication partner will be met for the segmented sending.
  - `G_CAN__TP__CONFIG__ISOTP__CMD_FLAG__START_SN_WITH_ZERO`

    The SequenceNumber (SN) starts with `0` in the first `ConsecutiveFrame` (CF) after a `FlowControl` frame (FC).

    

    If this flag is not set, the SequenceNumber (SN) starts with `1` in the first `ConsecutiveFrame` (CF) after a `FlowControl` frame (FC).
  - `G_CAN__TP__CONFIG__ISOTP__CMD_FLAG__SEPARATION_TIME_BETWEEN_FC_AND_CF`

Between a FlowControl frame and a ConsecutiveFrame the value of the SeparationTime received from the communication partner is met.

- G_CAN__TP__CONFIG__ISOTP__CMD_FLAG__SET_DLC_TO_NUMBER_OF_USED_BYTES

  The data length of the CAN message is set to the number of the used data bytes (the CAN message contains the used data bytes only)
- G_CAN__TP__ISOTP__FLAG__USE_TIME_BR_BETWEEN_FF_AND_FC

  Use TimeBrBetweenFfAndFc to define the time between reception of a "first frame" and transmission of a "flow control".
- G_CAN__TP__ISOTP__FLAG__USE_TIME_BR_BETWEEN_CF_AND_FC

  Use TimeBrBetweenCfAndFc to define the time between reception of a "consecutive frame" and transmission of a "flow control".
- G_CAN__TP__ISOTP__FLAG__USE_PADDING_BYTE

  Use PaddingByte for initialization of unused bytes.
- G_CAN__TP__ISOTP__FLAG__IGNORE_CF_WITH_WRONG_SN

  Ignore a received CF (consecutive frame) if the SN (sequence number) is not expected.
- G_CAN__TP__ISOTP__FLAG__USE_PHYSICAL_TX_DL

  Use PhysicalTxDl to configure the maximum usable payload length in bytes on physical sender side.
- G_CAN__TP__ISOTP__FLAG__USE_FUNCTIONAL_TX_DL

  Use FunctionalTxDl to configure the maximum usable payload length in bytes on functional sender side;

- OwnSeparationTime
  Self-meeting time between two CAN messages within a segmented data exchange, given in milliseconds, e.g. 10 milliseconds
- reserved1
  Reserved byte, must be initialized with 0
- reserved2
  Reserved byte, must be initialized with 0
- reserved3
  Reserved byte, must be initialized with 0
- TimeoutAs
  Sending timeout send-site,

  Given in milliseconds, e.g. 250 milliseconds
- TimeoutAr
  Sending timeout receive-site,

  Given in milliseconds, e.g. 250 milliseconds
- TimeoutBs
  FlowControl Frame receiving timeout send-site,

  Given in milliseconds, e.g. 250 milliseconds
- TimeoutCr
  ConsecutiveFrame Receiving timeout receive-site,

  Given in milliseconds, e.g. 250 milliseconds
- TimeBrBetweenFfAndFc
  Time between reception of a "first frame" and transmission of a "flow control" in milliseconds. This time is used only if flag G_CAN__TP__ISOTP__FLAG__USE_TIME_BR_BETWEEN_FF_AND_FC is set.

- `TimeBrBetweenCfAndFc`
  Time between reception of a "consecutive frame" and transmission of a "flow control" in milliseconds. This time is used only if flag `G_CAN__TP__ISOTP__FLAG__USE_TIME_BR_BETWEEN_CF_AND_FC` is set.
- `PaddingByte`
  For initialization of unused bytes.

  This parameter is used only if flag `G_CAN__TP__ISOTP__FLAG__USE_PADDING_BYTE` is set.
- `PhysicalTxDl`
  Configures the maximum usable payload length in bytes on physical sender side (8, 12, 16, 20, 24, 32, 48, 64).

  Default value: 8

  This parameter is used only if flag `G_CAN__TP__ISOTP__FLAG__USE_PHYSICAL_TX_DL` is set.
- `FunctionalTxDl`
  Configures the maximum usable payload length in bytes on functional sender side (8, 12, 16, 20, 24, 32, 48, 64).

  The default value depends on FDF bit (bit 30) in `FunctionalSourceId`:

  8 if FDF bit is not set

  64 if FDF bit is set

  This parameter is used only if flag `G_CAN__TP__ISOTP__FLAG__USE_FUNCTIONAL_TX_DL` is set.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tp_Config_J1939

G_Can_Tp_Config_J1939 — Configure the multisession channel

Description

Configuration of the multisession channel given by `channel` for the transport protocol "J1939".

Definition

```
G_Error_t
G_Can_Tp_Config_J1939(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Can_Tp_Config_J1939_Parameters_t * const j1939Parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`j1939Parameters`
Pointer to a variable of type **G_Can_Tp_Config_J1939_Parameters_t**

The structure **G_Can_Tp_Config_J1939_Parameters_t** consists of the following items:

- `SourceAddress`
  Own physical control unit address
- `DestinationAddress`
  Physical control unit address of the test object
- `reserved1`
  Reserved byte, must be initialized with `0`
- `reserved2`
  Reserved byte, must be initialized with `0`
- `TxTimeout`
  Sending timeout,

  Given in microseconds, e.g. 250000 µs
- `RxTimeout`
  Receiving timeout,

  Given in microseconds, e.g. 250000 µs
- `DelayTime`
  Intermission between individual `Data Transfer` messages

  Given in microseconds, e.g. 5000 µs
- `TimeoutTr`
  Timeout for sending `Data Transfer` messages

  Given in microseconds (shall be greater than `DelayTime`, e.g. 200000 µs)
- `TimeoutTh`
  Timeout for the temporary transmission interruption. At the latest after processing of **Th**, the transmission restarts or a new request for transmission interruption is sent.

  Given in microseconds, e.g. 500000 µs.

- `TimeoutT1`
  Timeout for receiving a `Data Transfer` message

  Given in microseconds, e.g. 750000 µs
- `TimeoutT2`
  Timeout for receiving the first `Data Transfer` message after the initialization of a point-to-point transmission or the restart of the transmission,

  Given in microseconds, e.g. 1250000 µs
- `TimeoutT3`
  Timeout for receiving an acknowledgment after sending a connecting request or after sending the last `Data Transfer` message

  Given in microseconds, e.g. 1250000 µs
- `TimeoutT4`
  Timeout after temporary transmission interruption

  Given in microseconds, e.g. 1050000 µs

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tp_Config_Tp_1_6

G_Can_Tp_Config_Tp_1_6 — Configure the multisession channel

Description

Configuration of the multisession channel given by `channel` for the transport protocol "TP1.6".

Definition

```
G_Error_t
G_Can_Tp_Config_Tp_1_6(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Can_Tp_Config_Tp_1_6_Parameters_t * const tp_1_6Parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`tp_1_6Parameters`
Pointer to a variable of type **G_Can_Tp_Config_Tp_1_6_Parameters_t**

The structure **G_Can_Tp_Config_Tp_1_6_Parameters_t** consists of the following items:

- `SourceAddress`
  Own control unit address
- `TargetAddress`
  Control unit address of the test object
- `BlockSize`
  Block size (0..15)
- `reserved`
  reserved parameter (must be initialized with **0**)
- `SourceSetupId`
  Own identifier for the ChannelSetup (e.g. 0x200 + `SourceAddress`)
- `TargetSetupId`
  Test object identifier for the ChannelSetup (e.g. 0x200 + `TargetAddress`)
- `SourceChannelId`
  Own identifier for data exchange
- `TargetChannelId`
  Test object identifier for data exchange
- `TimeoutT1`
  Time T1 in milliseconds (acknowledgment timeout for data telegrams, e.g. 45 milliseconds)
- `TimeoutT2`
  Time T2 in milliseconds (maximum time between two sending blocks, e.g. 450 milliseconds)
- `TimeoutT3`
  Time T3 in milliseconds (minimum time between two telegrams, e.g. 5 milliseconds)
- `TimeoutT4`
  Time T4 in milliseconds (Channel timeout if nothing is sent, e.g. 1000 milliseconds)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tp_Config_Tp_2_0

G_Can_Tp_Config_Tp_2_0 — Configuring the multisession channel

Description

This command configures the multisession channel given by `channel` for the transport protocol "TP2.0".

Definition

```
G_Error_t
G_Can_Tp_Config_Tp_2_0(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Can_Tp_Config_Tp_2_0_Parameters_t * const tp_2_0Parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`tp_2_0Parameters`
Pointer to a variable of type **G_Can_Tp_Config_Tp_2_0_Parameters_t**

The structure **G_Can_Tp_Config_Tp_2_0_Parameters_t** consists of the following items:

- `SourceAddress`
  Own control unit address
- `TargetAddress`
  Control unit address of the test object
- `BlockSize`
  Block size (0..15)
- `ApplicationType`
  Variable of type **G_Can_Tp_Config_Tp_2_0_ApplicationType_t** (Indication of the application type)

  The following return values are available:
  - `G_CAN__TP__CONFIG__TP_2_0__APPLICATION_TYPE__DIAG`

    Diagnostics
  - `G_CAN__TP__CONFIG__TP_2_0__APPLICATION_TYPE__INFO`

    Infotainment communication
  - `G_CAN__TP__CONFIG__TP_2_0__APPLICATION_TYPE__APP`

    Application protocol
  - `G_CAN__TP__CONFIG__TP_2_0__APPLICATION_TYPE__WFS_WIV`

    WFS/ WIV
- `SourceSetupId`
  Own identifier for ChannelSetup (e.g. 0x200 + `SourceAddress`)

  Has to be set to `0xFFFFFFFF` in the case of static channels
- `TargetSetupId`
  Test object identifier for ChannelSetup (e.g. 0x200 + `TargetAddress`)

Has to be set to `0xFFFFFFFF` in the case of static channels.

- `SourceChannelId`
  Own identifier for data exchange (is completely stipulated by the communication partner in the case of dynamic channels)
- `TargetChannelId`
  Test object identifier for data exchange
- `TimeoutT1`
  Time T1 in milliseconds (Acknowledgment timeout for data telegrams, e.g. 100 milliseconds)
- `TimeoutT3`
  Time T3 in milliseconds (Minimum time between two telegrams, e.g. 5 milliseconds)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tp_Config_Off

G_Can_Tp_Config_Off — Deactivate the configuration

Description

This command deactivates the current configuration of the multisession channel given by `channel`.

Definition

```
G_Error_t
G_Can_Tp_Config_Off(
 const G_PortHandle_t portHandle,
 const u8_t channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tp_Broadcast_StartTransmission

G_Can_Tp_Broadcast_StartTransmission — Start the transmission

Description

This command starts the transmission of a broadcast message (also multiple times).

Definition

```
G_Error_t
G_Can_Tp_Broadcast_StartTransmission(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Can_Tp_BroadCast_TransmissionMode_t mode,
 const u8_t * const data,
 const u32_t dataLength
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`mode`
Variable of type **G_Can_Tp_BroadCast_TransmissionMode_t** indicating the transmission mode

Broadcast requests can be sent repeatedly with `mode = G_CAN__TP__BROADCAST__TRANSMISSION_MODE__REQUEST_WITH_RETRIGGER`. Then repetition is deactivated by the command G_Can_Tp_Broadcast_StopTransmission .

The following values are available:

- `G_CAN__TP__BROADCAST__TRANSMISSION_MODE__REQUEST`

  Request
- `G_CAN__TP__BROADCAST__TRANSMISSION_MODE__REQUEST_WITH_RETRIGGER`

  Request with retriggering
- `G_CAN__TP__BROADCAST__TRANSMISSION_MODE__RESPONSE`

  Response

`data`
Pointer to the data buffer for data bytes of the broadcast message

`dataLength`
Number of the data bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tp_Broadcast_StopTransmission

G_Can_Tp_Broadcast_StopTransmission — Stop the transmission

Description

This command stops the repeated transmission of a broadcast message.

Definition

```
G_Error_t
G_Can_Tp_Broadcast_StopTransmission(
 const G_PortHandle_t portHandle,
 const u8_t channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tp_Broadcast_GetData

G_Can_Tp_Broadcast_GetData — Get the data of a received broadcast message

Description

This command requests data of a received broadcast message.

Definition

```
G_Error_t
G_Can_Tp_Broadcast_GetData(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 u8_t * const responseData,
 u32_t * const responseLength
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`responseData`
Pointer to the response data buffer

`responseLength`
Pointer to the variable `reponseLength`

Content when invoked: Size of response data buffer

Returned content: Size of response data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tp_Broadcast_GetData_Async

G_Can_Tp_Broadcast_GetData_Async — Request the data of a received broadcast message (by asynchronous request)

Description

This command starts the data request of a received broadcast message. If a response is available, the callback function set with G_Can_Tp_Broadcast_GetData_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Can_Tp_Broadcast_GetData_Async(
 const G_PortHandle_t portHandle,
 const u8_t channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tp_Broadcast_GetData_Async_Callback

G_Can_Tp_Broadcast_GetData_Async_Callback — Callback function for the asynchronous data request of a received broadcast message

Description

This function will be automatically called if a response for the asynchronous data request of a received broadcast message with G_Can_Tp_Broadcast_GetData_Async is available. It must be entered by the user and set with G_Can_Tp_Broadcast_GetData_Async_AddCallback.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Can_Tp_Broadcast_GetData_Async_Callback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u8_t * const responseData,
 const u32_t  responseLength
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

`responseData`
: Pointer to the response data buffer

`responseLength`
: Length of the response data in byte

G_Can_Tp_Broadcast_GetData_Async_AddCallback

G_Can_Tp_Broadcast_GetData_Async_AddCallback — Set the callback function for the asynchronous data request of a received broadcast message

Description

This command defines a callback function for the asynchronous data request of a received broadcast message with G_Can_Tp_Broadcast_GetData_Async.

Definition

```
G_API_CAN_DLL G_Error_t
G_Can_Tp_Broadcast_GetData_Async_AddCallback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Can_Tp_Broadcast_GetData_Async_Callback_t  callback
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

`callback`
: Function pointer of the callback function (see G_Can_Tp_Broadcast_GetData_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tp_Broadcast_GetData_Async_RemoveCallback

G_Can_Tp_Broadcast_GetData_Async_RemoveCallback — Remove the callback function for the asynchronous data request of a received broadcast message

Description

This command removes a callback function for the asynchronous data request of a received broadcast message.

Definition

```
G_API_CAN_DLL G_Error_t
G_Can_Tp_Broadcast_GetData_Async_RemoveCallback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Tp_SetMonitorFilter

G_Can_Tp_SetMonitorFilter — Set the monitor filter

Description

This command defines the monitor filter for the multisession channel given with `channel` and configured as TP channel.

The monitor filter is defined automatically that way that CAN messages used by this TP channel pass it. (Previously, all unwanted CAN messages should be blocked by G_Can_Monitor_DefineFilter .)

Definition

```
G_Error_t
G_Can_Tp_SetMonitorFilter(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Can_Tp_SetMonitorFilter_Mode_t mode
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`mode`
Variable of type **G_Can_Tp_SetMonitorFilter_Mode_t**

The following values are available:

- `G_CAN__TP__SET_MONITOR_FILTER__MODE__DISABLE`

  Deactivate the monitor filter setting
- `G_CAN__TP__SET_MONITOR_FILTER__MODE__ENABLE`

  Activate the monitor filter setting

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

11 Trigger

These commmands are used to control the **CAN Trigger** functionality.

The CAN Trigger can be configured for several applications, including the transmission of message groups or generating monitor events when configured trigger conditions are met.

It can also be used to generate trigger signals when specific CAN messages are transmitted or received.

G_Can_Trigger_InputMode_Set_SendMsgGroup

G_Can_Trigger_InputMode_Set_SendMsgGroup — Send message group when configured trigger conditions are met

Description

Use this command to configure the CAN Trigger to send a message group when the specified trigger conditions are met.

Definition

```
G_Error_t
G_Can_Trigger_InputMode_Set_SendMsgGroup(
 const G_PortHandle_t portHandle,
 const G_Can_Trigger_InputMode_Set_SendMsgGroup_CmdFlags_t  cmdFlags,
 const u8_t  group,
 const G_Can_Trigger_InputLevelSelection_t  inputLevelSelection
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_CAN__TRIGGER__INPUT_MODE__SET__SMG__CMD_FLAG__NONE`
No flag is set

`group`
Number of the message group whose transmission is started when the trigger conditions are met

A message group can be defined by calling G_Can_CyclicMsgs_DefineMsg with parameter `group` set accordingly.

`inputLevelSelection`
Required input level for triggering transmission of the message group

The following parameters are possible:

`G_CAN__TRIGGER__INPUT_LEVEL_SELECTION__LOW`
Low input level

`G_CAN__TRIGGER__INPUT_LEVEL_SELECTION__HIGH`
High input level

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Trigger_InputMode_Set_WriteEvent

G_Can_Trigger_InputMode_Set_WriteEvent — Write trigger event into monitor buffer when configured trigger conditions are met

Description

Use this command to write a trigger event into the monitor buffer when the specified trigger conditions are met.

Definition

```
G_Error_t
G_Can_Trigger_InputMode_Set_WriteEvent(
 const G_PortHandle_t portHandle,
 const G_Can_Trigger_InputMode_Set_WriteEvent_CmdFlags_t  cmdFlags,
 const G_Can_Trigger_InputEdgeSelection_t  inputEdgeSelection
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_CAN__TRIGGER__INPUT_MODE__SET__WE__CMD_FLAG__NONE`
No flag is set

`inputEdgeSelection`
Input signal edge that will trigger the write action

The following values are possible:

`G_CAN__TRIGGER__INPUT_EDGE_SELECTION__RISING`
A rising edge will trigger the write action

`G_CAN__TRIGGER__INPUT_EDGE_SELECTION__FALLING`
A falling edge will trigger the write action

`G_CAN__TRIGGER__INPUT_EDGE_SELECTION__BOTH`
Rising and falling edges will trigger the write action

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Trigger_InputMode_Reset

G_Can_Trigger_InputMode_Reset — Reset CAN Trigger Input Mode

Description

Use this command to reset the **CAN Trigger Input Mode** .

Definition

```
G_Error_t
G_Can_Trigger_InputMode_Reset(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Trigger_OutputMode_Set

G_Can_Trigger_OutputMode_Set — Set CAN Trigger Output Mode

Description

Use this command to set the **CAN Trigger Output Mode .**

The CAN trigger can be configured to manipulate a trigger output when a specific message is received or transmitted on the CAN bus.

Definition

```
G_Error_t
G_Can_Trigger_OutputMode_Set(
 const G_PortHandle_t portHandle,
 const G_Can_Trigger_OutputMode_Set_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`CmdFlags`
Command flags

The following values are possible and can be combined:

`G_CAN__TRIGGER__OUTPUT_MODE__SET__CMD_FLAG__NONE`
No flag is set

`Mode`
Output mode

The following values are possible:

`G_CAN__TRIGGER__OUTPUT_MODE__OFF`
The CAN trigger output is disabled

`G_CAN__TRIGGER__OUTPUT_MODE__RX`
The CAN trigger output is manipulated when the specified message is received

`G_CAN__TRIGGER__OUTPUT_MODE__TX`
The CAN trigger output is manipulated when the specified message is transmitted

`G_CAN__TRIGGER__OUTPUT_MODE__RX_TX`
The CAN trigger output is manipulated when the specified message is received or transmitted

`TriggerAction`
Defines the way the CAN trigger output is manipulated when the trigger conditions are met

The following values are possible:

G_IO__TRIGGER__OUTPUT_ACTION__POSITIVE_SPIKE
A **positive spike** will be generated when the trigger conditions are met

G_IO__TRIGGER__OUTPUT_ACTION__NEGATIVE_SPIKE
A **negative spike** will be generated when the trigger conditions are met

G_IO__TRIGGER__OUTPUT_ACTION__TOGGLE
The trigger output will be **toggled** when the trigger conditions are met

Type
Trigger type

The following values are possible:

G_IO__TRIGGER__OUTPUT_TYPE__CHECK_ID
The trigger condition is met when a CAN message with a specific **identifier** is received

G_IO__TRIGGER__OUTPUT_TYPE__CHECK_ID_AND_DATA
The trigger condition is met when a CAN message with a specific **identifier** and specific **message data** is received

reserved
reserved parameter (must be initialized with **0**)

Id
Specifies the identifier of the CAN message that triggers the action

The structure contains the following elements:

RxId
Specifies the identifier of the CAN message that triggers the action at its **reception**

When triggering on message reception, Mode has to be set to G_CAN__TRIGGER__OUTPUT_MODE__RX or G_CAN__TRIGGER__OUTPUT_MODE__RX_TX.

RxIdMask
Bit mask for RxId

The received message identifier has to match RxId in every bit defined by RxIdMask to trigger the action.

TxId
Specifies the identifier of the CAN message that triggers the action at its **transmission**

When triggering on message transmission, Mode has to be set to G_CAN__TRIGGER__OUTPUT_MODE__TX or G_CAN__TRIGGER__OUTPUT_MODE__RX_TX.

TxIdMask
Bit mask for TxId

The transmitted message identifier has to match TxId in every bit defined by TxIdMask to trigger the action.

Data
Specifies the data part of the CAN message that triggers the action

The structure contains the following elements:

RxData
Specifies the data bytes of the CAN message that triggers the action at its **reception** (0..8 bytes)

`RxDataMask`
Bit mask for `RxData` (0..8 bytes)

The received message data has to match `RxData` in every bit defined by `RxDataMask` to trigger the action.

`TxData`
Specifies the data bytes of the CAN message that triggers the action at its **transmission** (0..8 bytes)

`TxDataMask`
Bit mask for `TxData` (0..8 bytes)

The transmitted message data has to match `TxData` in every bit defined by `TxDataMask` to trigger the action.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

The trigger output is configured to generate a **positive spike** when a CAN message with identifier **0x123** is received.

```
G_PortHandle_t  portHandle;
G_Can_Trigger_OutputMode_Set_Parameters_t  param;
G_Error_t  rc;

…

param.CmdFlags = G_CAN__TRIGGER__OUTPUT_MODE__SET__CMD_FLAG__NONE;
param.Mode = G_CAN__TRIGGER__OUTPUT_MODE__RX;
param.TriggerAction = G_IO__TRIGGER__OUTPUT_ACTION__POSITIVE_SPIKE;
param.Type = G_IO__TRIGGER__OUTPUT_TYPE__CHECK_ID;
param.reserved = 0;

param.Id.RxId = 0x123;
param.Id.RxIdMask = 0xFFFFFFFF;

rc =
 G_Can_Trigger_OutputMode_Set(
  portHandle,
  &param
 );
```

G_Can_Trigger_InputState_Get

G_Can_Trigger_InputState_Get — Query state of CAN trigger input

Description

Use this command to query the state of the CAN trigger input

Definition

```
G_Error_t
G_Can_Trigger_InputState_Get(
 const G_PortHandle_t portHandle,
 const G_Can_Trigger_InputState_Get_Parameters_t * const parameters,
 G_Can_Trigger_InputState_Get_Response_t * const response
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Mode`
Query mode

The following values are possible:

`G_CAN__TRIGGER__INPUT_STATE__GET__MODE__UNKNOWN`
The query mode is unknown (not valid on function call)

`G_CAN__TRIGGER__INPUT_STATE__GET__MODE__STANDARD`
Standard query mode

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`response`
Returns response parameters

The structure contains the following elements:

`Mode`
Query mode (see G_Can_Trigger_InputState_Get_Mode_t)

`Level`
Current level of CAN trigger input

The following values are possible:

`G_CAN__TRIGGER__INPUT_STATE__GET__LEVEL__UNKNOWN`
Unknown level (obviously spelled wrong)

`G_CAN__TRIGGER__INPUT_STATE__GET__LEVEL__LOW`
The trigger input is **low**

`G_CAN__TRIGGER__INPUT_STATE__GET__LEVEL__HIGH`
The trigger input is **high**

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`EventCounter`
Number of trigger events that occurred at the trigger input

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Trigger_OutputState_Get

G_Can_Trigger_OutputState_Get — Query state of CAN trigger output

Description

Use this command to query the state of the CAN trigger output

Definition

```
G_Error_t
G_Can_Trigger_OutputState_Get(
 const G_PortHandle_t portHandle,
 const G_Can_Trigger_OutputState_Get_Parameters_t * const parameters,
 G_Can_Trigger_OutputState_Get_Response_t * const response
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Mode`
Query mode

The following values are possible:

`G_CAN__TRIGGER__OUTPUT_STATE__GET__MODE__UNKNOWN`
The query mode is unknown (not valid on function call)

`G_CAN__TRIGGER__OUTPUT_STATE__GET__MODE__STANDARD`
Standard query mode

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`response`
Returns response parameters

The structure contains the following elements:

`Mode`
Query mode (see G_Can_Trigger_OutputState_Get_Mode_t)

`Level`
Current level of CAN trigger output

The following values are possible:

`G_CAN__TRIGGER__OUTPUT_STATE__GET__LEVEL__UNKNOWN`
Unknown level

`G_CAN__TRIGGER__OUTPUT_STATE__GET__LEVEL__LOW`
The trigger output is **low**

`G_CAN__TRIGGER__OUTPUT_STATE__GET__LEVEL__HIGH`
The trigger output is **high**

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`RxEventCounter`
Number of trigger events that occurred due to the **reception** of the CAN message, defined by G_Can_Trigger_OutputMode_Set

`TxEventCounter`
Number of trigger events that occurred due to the **transmission** of the CAN message, defined by G_Can_Trigger_OutputMode_Set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Trigger_Input_Property_SetById

G_Can_Trigger_Input_Property_SetById — Set CAN trigger input properties

Description

Use this command to set CAN trigger **input properties** .

Definition

```
G_Error_t
G_Can_Trigger_Input_Property_SetById(
 const G_PortHandle_t portHandle,
 const G_Can_Trigger_Input_PropertyId_t  id,
 const u32_t  value
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Input property id

The following values are possible:

`G_CAN__TRIGGER__INPUT__PROPERTY_ID__UNKNOWN`
The property ID is unknown

`G_CAN__TRIGGER__INPUT__PROPERTY_ID__SOFTWARE_IN__CHANNEL`
Number of the software input that is used as trigger input (starting with **1**)

A trigger unit's source ("digital in", "trigger bus line in",...) can be routed to a trigger unit's target ("digital out", "trigger bus line out",...) with G_Io_Trigger_Source_Set .

`value`
Property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Trigger_Input_Property_GetById

G_Can_Trigger_Input_Property_GetById — Query CAN trigger input properties

Description

Use this command to query CAN trigger **input properties .**

Definition

```
G_Error_t
G_Can_Trigger_Input_Property_GetById(
 const G_PortHandle_t portHandle,
 const G_Can_Trigger_Input_PropertyId_t  id,
 u32_t * const value
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Input property ID (see G_Can_Trigger_Input_PropertyId_t)

`value`
Returns property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Trigger_Output_Property_SetById

G_Can_Trigger_Output_Property_SetById — Set CAN trigger output properties

Description

Use this command to set CAN trigger **output properties** .

Definition

```
G_Error_t
G_Can_Trigger_Output_Property_SetById(
 const G_PortHandle_t portHandle,
 const G_Can_Trigger_Output_PropertyId_t  id,
 const u32_t  value
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Output property id

The following values are possible:

`G_CAN__TRIGGER__OUTPUT__PROPERTY_ID__UNKNOWN`
The property ID is unknown

`G_CAN__TRIGGER__OUTPUT__PROPERTY_ID__SOFTWARE_OUT__CHANNEL`
Number of the software output that is used as trigger output (starting with **1**)

A trigger unit's source ("digital in", "trigger bus line in",...) can be routed to a trigger unit's target ("digital out", "trigger bus line out",...) with G_Io_Trigger_Source_Set .

`value`
Property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_Trigger_Output_Property_GetById

G_Can_Trigger_Output_Property_GetById — Query CAN trigger output properties

Description

Use this command to query CAN trigger **output properties .**

Definition

```
G_Error_t
G_Can_Trigger_Output_Property_GetById(
 const G_PortHandle_t portHandle,
 const G_Can_Trigger_Output_PropertyId_t  id,
 u32_t * const value
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Output property ID (see G_Can_Trigger_Output_PropertyId_t)

`value`
Returns property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

12 TxFifo

These commmands are used to control the sending FIFO functionality.

G_Can_TxFifo_GetState

G_Can_TxFifo_GetState — Get the status

Description

This command queries the current status of the sending FIFO.

Definition

```
G_Error_t
G_Can_TxFifo_GetState(
 const G_PortHandle_t portHandle,
 u32_t * const usedEntries,
 u32_t * const freeEntries
);
```

Parameters

`portHandle`
: Handle to the communication port

`usedEntries`
: Pointer to the variable `usedEntries`. In this variable the number of the used transmit entries is returned.

`freeEntries`
: Pointer to the variable `freeEntries`. The number of the free transmit entries is returned in this variable.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_TxFifo_GetState_Async

G_Can_TxFifo_GetState_Async — Query the TXFifo status (by asynchronous query)

Description

This command starts the asynchronous processing of the TXFifo state query. As soon as the response is available, the callback function being set with G_Can_TxFifo_GetState_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_API_CAN_DLL G_Error_t
G_Can_TxFifo_GetState_Async(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_TxFifo_GetState_Async_Callback

G_Can_TxFifo_GetState_Async_Callback — Callback function for the asynchronous query of the TXFifo status

Description

This function will be called as soon as a response for the asynchronous processing of the TXFifo status query with G_Can_TxFifo_GetState_Async is available. It must be entered by the user and set with G_Can_TxFifo_GetState_Async_AddCallback.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Can_TxFifo_GetState_Async_Callback(
 const G_PortHandle_t  portHandle,
 const u32_t  usedEntries,
 const u32_t  freeEntries
);
```

Parameters

`portHandle`
: Handle to the communication port

`usedEntries`
: Number of the used transmit entries

`freeEntries`
: Number of the free transmit entries

G_Can_TxFifo_GetState_Async_AddCallback

G_Can_TxFifo_GetState_Async_AddCallback — Set the callback function for the asynchronous query of the TXFifo status

Description

This command defines a callback function for the asynchronous processing of the TXFifo status query G_Can_TxFifo_GetState_Async.

Definition

```
G_API_CAN_DLL G_Error_t
G_Can_TxFifo_GetState_Async_AddCallback(
 const G_PortHandle_t  portHandle,
 const G_Can_TxFifo_GetState_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`callback`
Function pointer of the callback function (see G_Can_TxFifo_GetState_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_TxFifo_GetState_Async_RemoveCallback

G_Can_TxFifo_GetState_Async_RemoveCallback — Remove the callback function for the asynchronous query of the TXFifo status

Description

This command removes a callback function for the asynchronous processing of the TXFifo status query with G_Can_TxFifo_GetState_Async.

Definition

```
G_Error_t
G_Common_GetFirmwareVersion_Async_RemoveCallback(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_TxFifo_Reset

G_Can_TxFifo_Reset — Reset the sending FIFO

Description

This command resets the sending FIFO.

Prior to using the Sending FIFO functionality, this command should be executed.

Additionally the reset is required in special cases, e.g. after a bus short.

Definition

```
G_Error_t
G_Can_TxFifo_Reset(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_TxFifo_QueueMsg

G_Can_TxFifo_QueueMsg — Queue a message

Description

This command initiates the transmission of a CAN message using the FIFO functionality.

Please consider that on function return it cannot be ensured that the corresponding message has been transmitted by the hardware. The message is only queued into the hardware FIFO.

Definition

```
G_Error_t
G_Can_TxFifo_QueueMsg(
 const G_PortHandle_t portHandle,
 const u32_t id,
 const u8_t data[64],
 const u8_t dlc
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier

`data`
: Pointer to a buffer for data bytes

`dlc`
: Data length (see DLC to data length)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_TxFifo_QueueMsgs

G_Can_TxFifo_QueueMsgs — Send messages

Description

This command initiates the transmission of CAN messages using the FIFO functionality.

Please consider that on function return it cannot be ensured that the corresponding messages have already been transmitted by the hardware. The messages are only queued into the hardware FIFO.

If you want to queue items with more than 8 data bytes, use command G_Can_TxFifo_QueueMsgs2 .

Definition

```
G_Error_t
G_Can_TxFifo_QueueMsgs(
 const G_PortHandle_t portHandle,
 const u32_t timeout,
 const u32_t numberOfItems,
 const G_Can_TxFifo_MsgItem_t * const msgItems
);
```

Parameters

`portHandle`
Handle to the communication port

`timeout`
Queue timeout in milliseconds if there is no other empty transmit entry

`numberOfItems`
Number of the CAN messages to be transmitted

`msgItems`
Pointer to a buffer of type **G_Can_TxFifo_MsgItem_t** transferring the messages to be transmitted

All messages of this type consist of the following items:

- `Id`
  Identifier
- `Dlc`
  Data length (0..8)
- `reserved1`
  Reserved byte, must be initialized with `0`
- `reserved2`
  Reserved byte, must be initialized with `0`
- `reserved3`
  Reserved byte, must be initialized with `0`
- `Data`
  Buffer with data bytes sized `8` bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_TxFifo_QueueMsgs2

G_Can_TxFifo_QueueMsgs2 — Queue messages

Description

Use this command to add CAN messages to the transmission queue.

When the command call returns, the messages may still be in the hardware TX FIFO of the device waiting to be transmitted.

Definition

```
G_Error_t
G_Can_TxFifo_QueueMsgs2(
 const G_PortHandle_t portHandle,
 const G_Can_TxFifo_QueueMsgs2_CmdFlags_t  cmdFlags,
 const u32_t timeout,
 const u32_t numberOfItems,
 const G_Can_TxFifo_MsgItem2_t * const msgItems
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_CAN__TX_FIFO__QUEUE_MSGS_2__CMD_FLAGS__NONE`
No flag is set

`timeout`
Timeout for queuing messages, in milliseconds

`numberOfItems`
Number of items to be queued

`msgItems`
Array with message items to be queued

The strucure contains the following elements:

`Id`
CAN identifier

`Dlc`
Data length code (see DLC to data length)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Data`
Data bytes of the message (0..64)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

13 Uart Gateway

These commands are used to control the UART Gateway.

The UART Gateway is used to gate CAN messages to UART and vice versa.

G_Can_UartGateway_Init

G_Can_UartGateway_Init — Initialize UART Gateway

Description

Use this command to initialize the UART Gateway for gating CAN messages to UART and vice versa.

The UART Gateway can be reset with G_Can_UartGateway_Reset .

Definition

```
G_Error_t
G_Can_UartGateway_Init(
 const G_PortHandle_t portHandle,
 const G_Can_UartGateway_Init_Cmd_t * const cmd,
 G_Can_UartGateway_Init_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_CAN__UART_GATEWAY__INIT__CMD_FLAG__NONE`
No flag is set

`G_CAN__UART_GATEWAY__INIT__CMD_FLAG__ACK_ENABLE`
0 : disable acknowledge handling

1 : enable acknowledge handling

`G_CAN__UART_GATEWAY__INIT__CMD_FLAG__UART_PARITY_CHECK_ENABLE`
0 : ignore parity bit within received UART frames

1 : enable parity check within received UART frames: frames with a wrong parity bit are discarded

`G_CAN__UART_GATEWAY__INIT__CMD_FLAG__UART_SW_LOOP_ENABLE`
0 : normal gateway behavior

1 : received CAN messages are ignored, received UART messages are not gated to CAN, received UART messages are looped back to UART (UART reception -> convertion to CAN format -> convertion to UART format -> UART transmission)

`G_CAN__UART_GATEWAY__INIT__CMD_FLAG__LENGTH_INCLUSIVE_LENGTH_BYTE`
0 : the length byte contains the number of bytes after the length byte

1 : the length byte contains the number of bytes after the length byte plus the length byte itself

`MaxRetries`
Specifies the maximum TX retries when no acknowledge is received.

If the specified maximum has been reached, the frame will be discarded.

`AckTimeout`
Specifies the time to wait for an acknowledge before message is repeated, in microseconds

`UartInstanceType`
UART instance type

The following values are possible:

`G_CAN__UART_GATEWAY__UART_INSTANCE_TYPE__LVDS_CONTROL_CHANNEL`
LVDS control channel UART

In addition to setting the UART instance type, you need to set the LVDS data mode with G_Lvds_Common_Data_Property_SetById .

Use property id `G_LVDS__COMMON__DATA__PROPERTY_ID__DATA_MODE` and data mode `G_LVDS__COMMON__DATA__DATA_MODE__UART_TO_DESERIALIZER` for a **deserializer** or data mode **G_LVDS__COMMON__DATA__DATA_MODE__UART_TO_SERIALIZER** for a serializer.

`UartInstanceId`
Instance ID of the UART instance, starting with 0

`UartParity`
UART parity

The following values are possible:

`G_CAN__UART_GATEWAY__UART_PARITY__NONE`
No parity

`G_CAN__UART_GATEWAY__UART_PARITY__EVEN`
Even parity

`G_CAN__UART_GATEWAY__UART_PARITY__ODD`
Odd parity

`reserved`
reserved parameter (must be initialized with **0**)

`UartBaudRate`
UART baud rate, in baud

`rsp`
Returns response parameters

The structure contains the following elements:

`UartBaudRate_Desired`
Desired UART baud rate value, in baud

`UartBaudRate_Actual`
Actual UART baud rate value, in baud

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Can_UartGateway_Reset

G_Can_UartGateway_Reset — Reset UART Gateway

Description

Use this command to reset the UART Gateway.

Definition

```
G_Error_t
G_Can_UartGateway_Reset(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

CAN STM Functions

1 CAN STM Overview

These **G-API** commands provide access to the CAN STM hardware functionality.

CAN STM stands for **Stress** and **Trigger Module** which can be used to manipulate CAN and CAN FD communication.

External resources can be triggered by specific frames as well as the **CAN STM** can be triggered by external resources.

2 Disturbance

These commands provide control over CAN STM disturbance functionality.

G_CanStm_Disturbance_Properties_SetById

G_CanStm_Disturbance_Properties_SetById — Set property values

Description

Use this command to set disturbance property values by ID.

Definition

```
G_Error_t
G_CanStm_Disturbance_Properties_SetById(
 const G_PortHandle_t  portHandle,
 const G_CanStm_Disturbance_Properties_SetById_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`NumberOfProperties`
Number of properties to be set

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Properties`
Pointer to array with properties

The structure contains the following elements:

`Id`
Property ID

The following values are possible:

`G_CAN_STM__DISTURBANCE__PROPERTY_ID__DISTURBANCE_ENABLE`
0 : disturbance is disabled

1 : disturbance is enabled

`G_CAN_STM__DISTURBANCE__PROPERTY_ID__DISTURBANCE_MODE`
Disturbance mode

The following values are possible:

`G_CAN_STM__DISTURBANCE__DISTURBANCE_MODE__CONTINUOUS`
Continuous disturbance

G_CAN_STM__DISTURBANCE__DISTURBANCE_MODE__LIMITED
The disturbance is limited to a number of cycles

G_CAN_STM__DISTURBANCE__PROPERTY_ID__DISTURBANCES_PER_CYCLE
1 .. 255 : number of disturbances per disturbance cycle

G_CAN_STM__DISTURBANCE__PROPERTY_ID__DISTURBANCE_CYCLES
1 .. 65535 : number of disturbance cycles

G_CAN_STM__DISTURBANCE__PROPERTY_ID__DISTURBANCE_PAUSE
1 .. 65535 : pause duration in milliseconds between two disturbance cycles

G_CAN_STM__DISTURBANCE__PROPERTY_ID__DISTURBANCE_LENGTH
1 .. 16 : length of disturbance (dominant level) as number of bits

`Value`
Property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_CanStm_Disturbance_Properties_GetById

G_CanStm_Disturbance_Properties_GetById — Get property values

Description

Use this command to query disturbance property values by ID.

Definition

```
G_Error_t
G_CanStm_Disturbance_Properties_GetById(
 const G_PortHandle_t  portHandle,
 const G_CanStm_Disturbance_Properties_SetById_Cmd_t * const cmd,
 G_CanStm_Disturbance_Properties_GetById_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`NumberOfProperties`
Number of properties to be queried

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Ids`
Pointer to array with property IDs

The following values are possible:

`G_CAN_STM__DISTURBANCE__PROPERTY_ID__DISTURBANCE_ENABLE`
0 : disturbance is disabled

1 : disturbance is enabled

`G_CAN_STM__DISTURBANCE__PROPERTY_ID__DISTURBANCE_MODE`
Disturbance mode

The following values are possible:

`G_CAN_STM__DISTURBANCE__DISTURBANCE_MODE__CONTINUOUS`
Continuous disturbance

`G_CAN_STM__DISTURBANCE__DISTURBANCE_MODE__LIMITED`
The disturbance is limited to a number of cycles

`G_CAN_STM__DISTURBANCE__PROPERTY_ID__DISTURBANCES_PER_CYCLE`
1 .. 255 : number of disturbances per disturbance cycle

`G_CAN_STM__DISTURBANCE__PROPERTY_ID__DISTURBANCE_CYCLES`
1 .. 65535 : number of disturbance cycles

`G_CAN_STM__DISTURBANCE__PROPERTY_ID__DISTURBANCE_PAUSE`
1 .. 65535 : pause duration in milliseconds between two disturbance cycles

`G_CAN_STM__DISTURBANCE__PROPERTY_ID__DISTURBANCE_LENGTH`
1 .. 16 : length of disturbance (dominant level) as number of bits

`rsp`
Returns response parameters

The structure contains the following elements:

`NumberOfProperties`
Number of properties

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Values`
Pointer to buffer for with returned values

The pointer needs to be initialized before function call.

Be sure to provide a buffer big enough for all returned property values!

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

3 Node

These commands provide control over common CAN STM node properties.

G_CanStm_Node_BaudRate_Set

G_CanStm_Node_BaudRate_Set — Set CAN baud rate

Description

Use this command to set the CAN baud rate.

Definition

```
G_Error_t
G_CanStm_Node_BaudRate_Set(
 const G_PortHandle_t  portHandle,
 const G_CanStm_Node_BaudRate_Set_Cmd_t * const cmd,
 G_CanStm_Node_BaudRate_Set_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_CAN_STM__NODE__BAUD_RATE__SET__CMD_FLAG__NONE`
No flag is set

`SamplePoint_Min`
Minimum sample point

Use `0` if you don't want to specify this parameter.

`BaudRate`
Baud rate in baud (e.g. 500000 for 500 KBaud)

`SamplePoint_Max`
Maximum sample point

Use `0` if you don't want to specify this parameter.

`Ntq_Min`
Minimum number of time quanta (1 + TSeg1 + TSeg2)

Use `0` if you don't want to specify this parameter.

`Ntq_Max`
Maximum number of time quanta (1 + TSeg1 + TSeg2)

Use `0` if you don't want to specify this parameter.

`TSeg1_Min`
Minimum number of time quanta before sample point minus one

Use **0** if you don't want to specify this parameter.

`TSeg1_Max`
Maximum number of time quanta before sample point minus one

Use **0** if you don't want to specify this parameter.

`TSeg2_Min`
Minimum number of time quanta after sample point

Use **0** if you don't want to specify this parameter.

`TSeg2_Max`
Maximum number of time quanta after sample

Use **0** if you don't want to specify this parameter.

`Sjw_Min`
Minimum resyncronization jump width (>= 1)

Use **0** if you don't want to specify this parameter.

`Sjw_Max`
Maximum resyncronization jump width (>= 1)

Use **0** if you don't want to specify this parameter.

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_CAN_STM__NODE__BAUD_RATE__SET__RSP_FLAG__NONE`
No flag is set

`CanControllerClock`
CAN controller clock frequency in Hz

`BaudRate`
Baud rate in baud (e.g. 500000 for 500 KBaud)

`SamplePoint`
Sample point

`Ntq`
Number of time quanta (1 + TSeg1 + TSeg2)

`TSeg1`
Number of time quanta before sample point minus one

`TSeg2`
Number of time quanta after sample point

`Sjw`
Resyncronization jump width

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_CanStm_Node_BaudRate_Get

G_CanStm_Node_BaudRate_Get — Query baud rate values

Description

Use this command to query CAN baud rate values.

Definition

```
G_Error_t
G_CanStm_Node_BaudRate_Get(
 const G_PortHandle_t  portHandle,
 const G_CanStm_Node_BaudRate_Get_Cmd_t * const cmd,
 G_CanStm_Node_BaudRate_Get_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_CAN_STM__NODE__BAUD_RATE__GET__CMD_FLAG__NONE`
No flag is set

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_CAN_STM__NODE__BAUD_RATE__GET__RSP_FLAG__NONE`
No flag is set

`CanControllerClock`
CAN controller clock frequency in Hz

`BaudRate`
Baud rate in baud (e.g. 500000 for 500 KBaud)

`SamplePoint`
Sample point

`Ntq`
Number of time quanta (1 + TSeg1 + TSeg2)

`TSeg1`
Number of time quanta before sample point minus one

`TSeg2`
Number of time quanta after sample point

`Sjw`
Resyncronization jump width

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_CanStm_Node_BaudRateFd_Set

G_CanStm_Node_BaudRateFd_Set — Set CAN FDbaud rate

Description

Use this command to set the CAN FD baud rate.

Definition

```
G_Error_t
G_CanStm_Node_BaudRateFd_Set(
 const G_PortHandle_t  portHandle,
 const G_CanStm_Node_BaudRateFd_Set_Cmd_t * const cmd,
 G_CanStm_Node_BaudRateFd_Set_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_CAN_STM__NODE__BAUD_RATE_FD__SET__CMD_FLAG__NONE`
No flag is set

`G_CAN_STM__NODE__BAUD_RATE_FD__SET__CMD_FLAG__CHANGE_TDC`
See flag `G_CAN_STM__NODE__BAUD_RATE_FD__SET__CMD_FLAG__TDC` and parameter `TdcOffset`

0 : don't change flag `G_CAN_STM__NODE__BAUD_RATE_FD__SET__CMD_FLAG__TDC` and parameter `TdcOffset`

1 : change flag `G_CAN_STM__NODE__BAUD_RATE_FD__SET__CMD_FLAG__TDC` and parameter `TdcOffset`

`G_CAN_STM__NODE__BAUD_RATE_FD__SET__CMD_FLAG__TDC`
0 : disable transmitter delay compensation

1 : enable transmitter delay compensation

`SamplePoint_Min`
Minimum sample point

Use **0** if you don't want to specify this parameter.

`SamplePoint_Max`
Maximum sample point

Use **0** if you don't want to specify this parameter.

`Ntq_Min`
Minimum number of time quanta (1 + TSeg1 + TSeg2)

Use **0** if you don't want to specify this parameter.

`Ntq_Max`
Maximum number of time quanta (1 + TSeg1 + TSeg2)

Use **0** if you don't want to specify this parameter.

`BaudRate`
Baud rate in baud (e.g. 2000000 for 2 MBaud)

`TSeg1_Min`
Minimum number of time quanta before sample point minus one

Use **0** if you don't want to specify this parameter.

`TSeg1_Max`
Maximum number of time quanta before sample point minus one

Use **0** if you don't want to specify this parameter.

`TSeg2_Min`
Minimum number of time quanta after sample point

Use **0** if you don't want to specify this parameter.

`TSeg2_Max`
Maximum number of time quanta after sample

Use **0** if you don't want to specify this parameter.

`Sjw_Min`
Minimum resyncronization jump width (>= 1)

Use **0** if you don't want to specify this parameter.

`Sjw_Max`
Maximum resyncronization jump width (>= 1)

Use **0** if you don't want to specify this parameter.

`TdcOffset`
Transmitter delay compensation offset [ns]

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_CAN_STM__NODE__BAUD_RATE_FD__SET__RSP_FLAG__NONE`
No flag is set

`G_CAN_STM__NODE__BAUD_RATE_FD__SET__RSP_FLAG__FD`
0 : classic CAN operation

1 : CAN FD operation

`G_CAN_STM__NODE__BAUD_RATE_FD__SET__RSP_FLAG__TDC`
0 : transmitter delay compensation is disabled

1 : transmitter delay compensation is enabled

`CanControllerClock`
CAN controller clock frequency in Hz

`BaudRate`
Baud rate in baud (e.g. 2000000 for 2 MBaud)

`SamplePoint`
Sample point

`Ntq`
Number of time quanta (1 + TSeg1 + TSeg2)

`TSeg1`
Number of time quanta before sample point minus one

`TSeg2`
Number of time quanta after sample point

`Sjw`
Resyncronization jump width

`TdcOffset`
Transmitter delay compensation offset [ns]

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_CanStm_Node_BaudRateFd_Get

G_CanStm_Node_BaudRateFd_Get — Query baud rate values

Description

Use this command to query CAN FD baud rate values.

Definition

```
G_Error_t
G_CanStm_Node_BaudRateFd_Get(
 const G_PortHandle_t  portHandle,
 const G_CanStm_Node_BaudRateFd_Get_Cmd_t * const cmd,
 G_CanStm_Node_BaudRateFd_Get_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_CAN_STM__NODE__BAUD_RATE_FD__GET__CMD_FLAG__NONE`
No flag is set

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_CAN_STM__NODE__BAUD_RATE_FD__GET__RSP_FLAG__NONE`
No flag is set

`G_CAN_STM__NODE__BAUD_RATE_FD__GET__RSP_FLAG__FD`
0 : classic CAN operation

1 : CAN FD operation

`G_CAN_STM__NODE__BAUD_RATE_FD__GET__RSP_FLAG__TDC`
0 : transmitter delay compensation is disabled

1 : transmitter delay compensation is enabled

`CanControllerClock`
CAN controller clock frequency in Hz

`BaudRate`
Baud rate in baud (e.g. 2000000 for 2 MBaud)

`SamplePoint`
Sample point

`Ntq`
Number of time quanta (1 + TSeg1 + TSeg2)

`TSeg1`
Number of time quanta before sample point minus one

`TSeg2`
Number of time quanta after sample point

`Sjw`
Resyncronization jump width

`TdcOffset`
Transmitter delay compensation offset [ns]

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_CanStm_Node_Property_SetById

G_CanStm_Node_Property_SetById — Set CAN STM node property

Description

Use this command to set a CAN STM node property.

Definition

```
G_Error_t
G_CanStm_Node_Property_SetById(
 const G_PortHandle_t  portHandle,
 const G_CanStm_Node_Property_SetById_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Id`
Property ID

The following values are possible:

`G_CAN_STM__NODE__PROPERTY_ID__CAN_CONTROLLER_CLOCK`
CAN controller clock frequency in Hz

`G_CAN_STM__NODE__PROPERTY_ID__FD__NON_ISO`
0 : CAN FD frame format according to ISO11898-1:2015

1 : CAN FD frame format according to Bosch CAN FD Specification V1.0

`G_CAN_STM__NODE__PROPERTY_ID__CAN_ACK_ENABLE`
0 : don't send a dominant level in CAN acknowledge slot

1 : send a dominant level in CAN acknowledge slot

`G_CAN_STM__NODE__PROPERTY_ID__TRANSCEIVER_SELECTION`
0 : no transceiver selected

1 : transceiver CAN1

2 : transceiver CAN2

3 : transceiver CAN3

4 : transceiver CAN4

Setting this property to a supported transceiver also sets properties `G_CAN_STM__NODE__PROPERTY_ID__ENABLE_BUS_TERMINATION` and `G_CAN_STM__NODE__PROPERTY_ID__ENABLE_INTERNAL_VBAT` to its default. values

`G_CAN_STM__NODE__PROPERTY_ID__ENABLE_BUS_TERMINATION`
This property controls the CAN bus termination.

0 : the bus termination is deactivated

1 : the bus termination is activated (default after transceiver selection)

`G_CAN_STM__NODE__PROPERTY_ID__ENABLE_INTERNAL_VBAT`
This property controls the VBat voltage of the CAN transceiver.

0 : the internal VBat voltage of the CAN transceiver is disabled

1 : the internal VBat voltage of the CAN transceiver is enabled (default)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Value`
Property value to be set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_CanStm_Node_Property_GetById

G_CanStm_Node_Property_GetById — Get CAN STM node property value

Description

Use this command to query a CAN STM node property value.

Definition

```
G_Error_t
G_CanStm_Node_Property_GetById(
 const G_PortHandle_t  portHandle,
 const G_CanStm_Node_Property_GetById_Cmd_t * const cmd,
 G_CanStm_Node_Property_GetById_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Id`
Property ID

The following values are possible:

`G_CAN_STM__NODE__PROPERTY_ID__CAN_CONTROLLER_CLOCK`
CAN controller clock frequency in Hz

`G_CAN_STM__NODE__PROPERTY_ID__FD__NON_ISO`
0 : CAN FD frame format according to ISO11898-1:2015

1 : CAN FD frame format according to Bosch CAN FD Specification V1.0

`G_CAN_STM__NODE__PROPERTY_ID__CAN_ACK_ENABLE`
0 : don't send a dominant level in CAN acknowledge slot

1 : send a dominant level in CAN acknowledge slot

`G_CAN_STM__NODE__PROPERTY_ID__TRANSCEIVER_SELECTION`
0 : no transceiver selected

1 : transceiver CAN1

2 : transceiver CAN2

3 : transceiver CAN3

4 : transceiver CAN4

Setting this property to a supported transceiver also sets properties `G_CAN_STM__NODE__PROPERTY_ID__ENABLE_BUS_TERMINATION` and `G_CAN_STM__NODE__PROPERTY_ID__ENABLE_INTERNAL_VBAT` to its default. values

G_CAN_STM__NODE__PROPERTY_ID__ENABLE_BUS_TERMINATION
This property controls the CAN bus termination.

0 : the bus termination is deactivated

1 : the bus termination is activated (default after transceiver selection)

G_CAN_STM__NODE__PROPERTY_ID__ENABLE_INTERNAL_VBAT
This property controls the VBat voltage of the CAN transceiver.

0 : the internal VBat voltage of the CAN transceiver is disabled

1 : the internal VBat voltage of the CAN transceiver is enabled (default)

reserved1
reserved parameter (must be initialized with **0**)

reserved2
reserved parameter (must be initialized with **0**)

rsp
Returns response parameters

The structure contains the following elements:

Id
Property ID

The following values are possible:

G_CAN_STM__NODE__PROPERTY_ID__CAN_CONTROLLER_CLOCK
CAN controller clock frequency in Hz

G_CAN_STM__NODE__PROPERTY_ID__FD__NON_ISO
0 : CAN FD frame format according to ISO11898-1:2015

1 : CAN FD frame format according to Bosch CAN FD Specification V1.0

G_CAN_STM__NODE__PROPERTY_ID__CAN_ACK_ENABLE
0 : don't send a dominant level in CAN acknowledge slot

1 : send a dominant level in CAN acknowledge slot

G_CAN_STM__NODE__PROPERTY_ID__TRANSCEIVER_SELECTION
0 : no transceiver selected

1 : transceiver CAN1

2 : transceiver CAN2

3 : transceiver CAN3

4 : transceiver CAN4

Setting this property to a supported transceiver also sets properties G_CAN_STM__NODE__PROPERTY_ID__ENABLE_BUS_TERMINATION and G_CAN_STM__NODE__PROPERTY_ID__ENABLE_INTERNAL_VBAT to its default. values

G_CAN_STM__NODE__PROPERTY_ID__ENABLE_BUS_TERMINATION
This property controls the CAN bus termination.

0 : the bus termination is deactivated

1 : the bus termination is activated (default after transceiver selection)

`G_CAN_STM__NODE__PROPERTY_ID__ENABLE_INTERNAL_VBAT`
This property controls the VBat voltage of the CAN transceiver.

0 : the internal VBat voltage of the CAN transceiver is disabled

1 : the internal VBat voltage of the CAN transceiver is enabled (default)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Value`
Returns property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

4 Trigger

These commands provide control over CAN STM trigger functionality.

G_CanStm_Trigger_Properties_SetById

G_CanStm_Trigger_Properties_SetById — Set property values

Description

Use this command to set trigger property values by ID.

Definition

```
G_Error_t
G_CanStm_Trigger_Properties_SetById(
 const G_PortHandle_t  portHandle,
 const G_CanStm_Trigger_Properties_SetById_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`NumberOfProperties`
Number of properties to be set

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Properties`
Pointer to array with properties

The structure contains the following elements:

`Id`
Property ID

The following values are possible:

`G_CAN_STM__TRIGGER__PROPERTY_ID__TRIGGER_ENABLE`
0 : trigger is disabled

1 : trigger is enabled

`G_CAN_STM__TRIGGER__PROPERTY_ID__TRIGGER_IN__ENABLE`
0 : trigger input is not used

1 : trigger input is used

`G_CAN_STM__TRIGGER__PROPERTY_ID__TRIGGER_IN__MODE`
The following values are possible:

G_CAN_STM__TRIGGER__TRIGGER_IN__MODE__AS_TRIGGER_ENABLE
Trigger input is used as trigger enable:

If property G_CAN_STM__TRIGGER__PROPERTY_ID__TRIGGER_IN__ENABLE is 1 and the trigger input level is equal to property G_CAN_STM__TRIGGER__PROPERTY_ID__TRIGGER_IN__POLARITY the trigger is enabled.

G_CAN_STM__TRIGGER__TRIGGER_IN__MODE__AS_HW_TRIGGER
Trigger input is used as hardware trigger:

Disturbance and length controlled just by hardware trigger.

G_CAN_STM__TRIGGER__PROPERTY_ID__TRIGGER_IN__POLARITY
0 : trigger input polarity is low active

1 : trigger input polarity is high active

G_CAN_STM__TRIGGER__PROPERTY_ID__TRIGGER_OUT__ENABLE
0 : trigger output is not used

1 : trigger output is used

G_CAN_STM__TRIGGER__PROPERTY_ID__TRIGGER_OUT__SOURCE
The following values are possible:

G_CAN_STM__TRIGGER__TRIGGER_OUT__SOURCE__DISTURBANCE
Source for trigger output: disturbance

G_CAN_STM__TRIGGER__TRIGGER_OUT__SOURCE__END_OF_ID
Source for trigger output: CAN identifier received

G_CAN_STM__TRIGGER__TRIGGER_OUT__SOURCE__CRC_DELIMITER
Source for trigger output: CAN CRC delimiter

G_CAN_STM__TRIGGER__TRIGGER_OUT__SOURCE__ESI
Source for trigger output: CAN ESI bit (CAN Error State Indicator, CAN FD frames only)

G_CAN_STM__TRIGGER__TRIGGER_OUT__SOURCE__STUFF_BIT_COUNTER
Source for trigger output: CAN stuff bit counter (CAN FD frames only)

G_CAN_STM__TRIGGER__TRIGGER_OUT__SOURCE__ERROR_FRAME
Source for trigger output: CAN error frame

G_CAN_STM__TRIGGER__TRIGGER_OUT__SOURCE__START_OF_FRAME
Source for trigger output: start of CAN frame

G_CAN_STM__TRIGGER__PROPERTY_ID__ID_TRIGGER_MODE
The following values are possible:

G_CAN_STM__TRIGGER__ID_TRIGGER_MODE__EVERY_ID
Disturbance is triggered by every ID

G_CAN_STM__TRIGGER__ID_TRIGGER_MODE__ID_IN_ID_LIST
Disturbance is triggered by IDs in ID list

(see G_CanStm_Trigger_IdList_Write and G_CanStm_Trigger_IdList_Read)

G_CAN_STM__TRIGGER__ID_TRIGGER_MODE__ID_NOT_IN_ID_LIST
Disturbance is triggered by IDs that or not in ID list

(see G_CanStm_Trigger_IdList_Write and G_CanStm_Trigger_IdList_Read)

`G_CAN_STM__TRIGGER__PROPERTY_ID__ID_LIST__AUTO_FILL__ENABLE`
0 : disable automatic filling of ID list

1 : enable automatic filling of ID list

`G_CAN_STM__TRIGGER__PROPERTY_ID__ID_LIST__AUTO_FILL__ERROR`
This property is read only

Error while automatic filling of ID list

The following values are possible:

`G_CAN_STM__TRIGGER__ID_LIST__AUTO_FILL__ERROR__NO_ERROR`
No error

`G_CAN_STM__TRIGGER__ID_LIST__AUTO_FILL__ERROR__ID_LIST_FULL`
An ID couldn't be stored because list was already full

`G_CAN_STM__TRIGGER__PROPERTY_ID__ID_LIST__AUTO_FILL__NUMBER_OF_IDS`
This property is read only

Automatic filling of ID list: number of CAN identifiers stored in ID list

`G_CAN_STM__TRIGGER__PROPERTY_ID__ID_LIST__MAX_NUMBER_OF_IDS`
This property is read only

Maximum number of IDs in ID list

`Value`
Property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_CanStm_Trigger_Properties_GetById

G_CanStm_Trigger_Properties_GetById — Get property values

Description

Use this command to query trigger property values by ID.

Definition

```
G_Error_t
G_CanStm_Trigger_Properties_GetById(
 const G_PortHandle_t  portHandle,
 const G_CanStm_Trigger_Properties_SetById_Cmd_t * const cmd,
 G_CanStm_Trigger_Properties_GetById_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`NumberOfProperties`
Number of properties to be queried

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Ids`
Pointer to array with property IDs

The following values are possible:

`G_CAN_STM__TRIGGER__PROPERTY_ID__TRIGGER_ENABLE`
0 : trigger is disabled

1 : trigger is enabled

`G_CAN_STM__TRIGGER__PROPERTY_ID__TRIGGER_IN__ENABLE`
0 : trigger input is not used

1 : trigger input is used

`G_CAN_STM__TRIGGER__PROPERTY_ID__TRIGGER_IN__MODE`
The following values are possible:

`G_CAN_STM__TRIGGER__TRIGGER_IN__MODE__AS_TRIGGER_ENABLE`
Trigger input is used as trigger enable:

If property G_CAN_STM__TRIGGER__PROPERTY_ID__TRIGGER_IN__ENABLE is **1** and the trigger input level is equal to property G_CAN_STM__TRIGGER__PROPERTY_ID__TRIGGER_IN__POLARITY the trigger is enabled.

G_CAN_STM__TRIGGER__TRIGGER_IN__MODE__AS_HW_TRIGGER
Trigger input is used as hardware trigger:

Disturbance and length controlled just by hardware trigger.

G_CAN_STM__TRIGGER__PROPERTY_ID__TRIGGER_IN__POLARITY
0 : trigger input polarity is low active

1 : trigger input polarity is high active

G_CAN_STM__TRIGGER__PROPERTY_ID__TRIGGER_OUT__ENABLE
0 : trigger output is not used

1 : trigger output is used

G_CAN_STM__TRIGGER__PROPERTY_ID__TRIGGER_OUT__SOURCE
The following values are possible:

G_CAN_STM__TRIGGER__TRIGGER_OUT__SOURCE__DISTURBANCE
Source for trigger output: disturbance

G_CAN_STM__TRIGGER__TRIGGER_OUT__SOURCE__END_OF_ID
Source for trigger output: CAN identifier received

G_CAN_STM__TRIGGER__TRIGGER_OUT__SOURCE__CRC_DELIMITER
Source for trigger output: CAN CRC delimiter

G_CAN_STM__TRIGGER__TRIGGER_OUT__SOURCE__ESI
Source for trigger output: CAN ESI bit (CAN Error State Indicator, CAN FD frames only)

G_CAN_STM__TRIGGER__TRIGGER_OUT__SOURCE__STUFF_BIT_COUNTER
Source for trigger output: CAN stuff bit counter (CAN FD frames only)

G_CAN_STM__TRIGGER__TRIGGER_OUT__SOURCE__ERROR_FRAME
Source for trigger output: CAN error frame

G_CAN_STM__TRIGGER__TRIGGER_OUT__SOURCE__START_OF_FRAME
Source for trigger output: start of CAN frame

G_CAN_STM__TRIGGER__PROPERTY_ID__ID_TRIGGER_MODE
The following values are possible:

G_CAN_STM__TRIGGER__ID_TRIGGER_MODE__EVERY_ID
Disturbance is triggered by every ID

G_CAN_STM__TRIGGER__ID_TRIGGER_MODE__ID_IN_ID_LIST
Disturbance is triggered by IDs in ID list

(see G_CanStm_Trigger_IdList_Write and G_CanStm_Trigger_IdList_Read)

G_CAN_STM__TRIGGER__ID_TRIGGER_MODE__ID_NOT_IN_ID_LIST
Disturbance is triggered by IDs that or not in ID list

(see G_CanStm_Trigger_IdList_Write and G_CanStm_Trigger_IdList_Read)

G_CAN_STM__TRIGGER__PROPERTY_ID__ID_LIST__AUTO_FILL__ENABLE
0 : disable automatic filling of ID list

1 : enable automatic filling of ID list

`G_CAN_STM__TRIGGER__PROPERTY_ID__ID_LIST__AUTO_FILL__ERROR`
This property is read only

Error while automatic filling of ID list

The following values are possible:

`G_CAN_STM__TRIGGER__ID_LIST__AUTO_FILL__ERROR__NO_ERROR`
No error

`G_CAN_STM__TRIGGER__ID_LIST__AUTO_FILL__ERROR__ID_LIST_FULL`
An ID couldn't be stored because list was already full

`G_CAN_STM__TRIGGER__PROPERTY_ID__ID_LIST__AUTO_FILL__NUMBER_OF_IDS`
This property is read only

Automatic filling of ID list: number of CAN identifiers stored in ID list

`G_CAN_STM__TRIGGER__PROPERTY_ID__ID_LIST__MAX_NUMBER_OF_IDS`
This property is read only

Maximum number of IDs in ID list

`rsp`
Returns response parameters

The structure contains the following elements:

`NumberOfProperties`
Number of properties

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Values`
Pointer to buffer for with returned values

The pointer needs to be initialized before function call.

Be sure to provide a buffer big enough for all returned property values!

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_CanStm_Trigger_IdList_Write

G_CanStm_Trigger_IdList_Write — Write ID list with triggered / disregarded IDs

Description

Use this command to write an ID list with triggered / disregarded IDs, depending on property `G_CAN_STM__TRIGGER__PROPERTY_ID__ID_TRIGGER_MODE`.

Definition

```
G_Error_t
G_CanStm_Trigger_IdList_Write(
 const G_PortHandle_t  portHandle,
 const G_CanStm_Trigger_IdList_Write_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_CAN_STM__TRIGGER__ID_LIST__WRITE__CMD_FLAG__NONE`
No flag is set

`G_CAN_STM__TRIGGER__ID_LIST__WRITE__CMD_FLAG__RESET`
The ID list will be reset

`IdListOffset`
Offset for writing IDs

This can be useful if auto-fill is enabled (see property `G_CAN_STM__TRIGGER__PROPERTY_ID__ID_LIST__AUTO_FILL__ENABLE`) and you want to add additional IDs to the list.

`NumberOfIds`
Number of IDs to be written

`Ids`
Pointer to array with IDs to be written

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_CanStm_Trigger_IdList_Read

G_CanStm_Trigger_IdList_Read — Read ID list with triggered / disregarded IDs

Description

Use this command to read an ID list with triggered / disregarded IDs, depending on property `G_CAN_STM__TRIGGER__PROPERTY_ID__ID_TRIGGER_MODE`.

Definition

```
G_Error_t
G_CanStm_Trigger_IdList_Read(
 const G_PortHandle_t  portHandle,
 const G_CanStm_Trigger_IdList_Read_Cmd_t * const cmd,
 G_CanStm_Trigger_IdList_Read_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_CAN_STM__TRIGGER__ID_LIST__READ__CMD_FLAG__NONE`
No flag is set

`IdListOffset`
Offset for reading IDs

`NumberOfIds`
Number of IDs to be read

Be sure to provide a buffer big enough for `NumberOfIds` via `Ids` in resp

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_CAN_STM__TRIGGER__ID_LIST__READ__RSP_FLAG__NONE`
No flag is set

`IdListOffset`
Returns ID list reading offset

`NumberOfIds`
Number of IDs that are returned

`Ids`
Returns IDs

Be sure to provide a buffer big enough for `NumberOfIds`!

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Diagnostic Functions

These **G-API** commands provide the diagnostics functionality of the hardware. If the return value of type **G_Error_t** does not correspond to `G_NO_ERROR`, then an error has occurred. The error description can be called by the command G_GetLastErrorDescription .

G_Diag_Start

G_Diag_Start — Start the diagnostics

Description

This command starts the diagnostics. After terminating the starting operation only this function will be quitted.

Definition

```
G_Error_t
G_Diag_Start(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Diag_RequestMode_t mode,
 const u32_t timeout,
 const u8_t * const requestData,
 const u32_t requestLength,
 G_Error_t * const lastErrorCode,
 u8_t * const responseData,
 u32_t * const responseLength
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`mode`
Variable of type **G_Diag_RequestMode_t** indicating the request mode

The following return values are available:

- `G_DIAG__REQUEST_MODE__PHYSICAL_WITH_RESPONSE`

  Physical request with response
- `G_DIAG__REQUEST_MODE__FUNCTIONAL_WITH_RESPONSE`

  Functional request with response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_PHYSICAL_WITH_RESPONSE`

  Stop after physical request with response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_FUNCTIONAL_WITH_RESPONSE`

  Stop after functional request with response
- `G_DIAG__REQUEST_MODE__PHYSICAL_WITHOUT_RESPONSE`

  Physical request without response
- `G_DIAG__REQUEST_MODE__FUNCTIONAL_WITHOUT_RESPONSE`

  Functional request without response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_PHYSICAL_WITHOUT_RESPONSE`

  Stop after physical request without response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_FUNCTIONAL_WITHOUT_RESPONSE`

Stop after functional request without response

`timeout`
Receive timeout (in milliseconds)

`requestData`
Pointer to the request data buffer

`requestLength`
Size of request data buffer

`lastErrorCode`
Pointer to a variable of type **G_Error_t**. It returns the error code of the last error that has occurred in the firmware (or `G_NO_ERROR` = no error). For getting an error description, you can call G_Common_GetFirmwareErrorDescription.

`responseData`
Pointer to the response data buffer

`responseLength`
Pointer to the variable `reponseLength`

Content when invoked: Size of response data buffer

Returned content: Size of response data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Start the diagnostics by **G_Diag_Start**

```
u8_t channel;
G_Diag_RequestMode_t mode;
u32_t timeout;
u8_t requestData[2];
u32_t requestLength;
G_Error_t lastErrorCode;
u8_t responseData[1024];
u32_t responseLength;

…

channel = 0;            //Multisession channel 0
mode = G_DIAG__REQUEST_MODE__PHYSICAL_WITH_RESPONSE;
timeout = 1000;         //1000 milliseconds
requestData[0] = 0x11;
requestData[1] = 0x22;
requestLength = 2;
responseLength = 1024;

rc =
  G_Diag_Start(
    portHandle,
    channel,
    mode,
    timeout,
    requestData,
    requestLength,
    &lastErrorCode,
    responseData,
    &responseLength
  );

if (rc == G_NO_ERROR) {

…

}
```

In this example the required variables are declared first. With `requestData[2]` you can set a request buffer of 2 bytes and with `responseData[1024]` a response buffer of 1024 bytes. The sizes of both buffers are subsequently entered into the variables `requestLength` or `responseLength`. Furthermore the required values and parameters are input into the corresponding variables. Additionally to the port handle they are subsequently transferred to the function **G_Diag_Start** . In case of certain variables it is required to transfer their addresses or the pointers to the related buffers. This is necessary because the corresponding return values are saved in several variables. As for `responseLength` for example, it contains on function call the size of the response buffer and on function return the size of the response data. The command is executed completely after receiving a response or an error or after processing the timeout. If the return code `rc` with the value `G_NO_ERROR` indicates an error-free execution, the contents of the variables can be evaluated.

G_Diag_InitiateStart

G_Diag_InitiateStart — Initiate the diagnostics start

Description

This command initiates the diagnostics start.

Please consider that on function return it cannot be ensured that the diagnostics has been started. The starting request is only transferred to the hardware. If you want to make sure that the diagnostics has been started on function return, it is necessary to use G_Diag_Start.

Definition

```
G_Error_t
G_Diag_InitiateStart(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Diag_RequestMode_t mode,
 const u8_t * const requestData,
 const u32_t requestLength
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`mode`
Variable of type **G_Diag_RequestMode_t** indicating the request mode

The following return values are available:

- `G_DIAG__REQUEST_MODE__PHYSICAL_WITH_RESPONSE`

  Physical request with response
- `G_DIAG__REQUEST_MODE__FUNCTIONAL_WITH_RESPONSE`

  Functional request with response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_PHYSICAL_WITH_RESPONSE`

  Stop after physical request with response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_FUNCTIONAL_WITH_RESPONSE`

  Stop after functional request with response
- `G_DIAG__REQUEST_MODE__PHYSICAL_WITHOUT_RESPONSE`

  Physical request without response
- `G_DIAG__REQUEST_MODE__FUNCTIONAL_WITHOUT_RESPONSE`

  Functional request without response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_PHYSICAL_WITHOUT_RESPONSE`

  Stop after physical request without response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_FUNCTIONAL_WITHOUT_RESPONSE`

Stop after functional request without response

`requestData`
Pointer to the request data buffer

`requestLength`
Size of request data buffer

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Start the diagnostics by **G_Diag_InitiateStart**

```
u8_t channel;
G_Diag_RequestMode_t mode;
u8_t requestData[2];
u32_t requestLength;

…

channel = 0;            //Multisession channel 0
mode = G_DIAG__REQUEST_MODE__PHYSICAL_WITH_RESPONSE;
requestData[0] = 0x11;
requestData[1] = 0x22;
requestLength = 2;

rc =
  G_Diag_InitiateStart(
    portHandle,
    channel,
    mode,
    requestData,
    requestLength
  );
```

In this example the required variables are declared first. With `requestData[2]` you can set a request buffer of 2 bytes, whose size is subsequently entered into the variable `requestLength`. Furthermore the required values and parameters are input into the related variables. Additionally to the port handle they are subsequently transferred to the function **G_Diag_InitiateStart** . `requestData` is the pointer to the request buffer.

G_Diag_Stop

G_Diag_Stop — Stop diagnostics

Description

This command stops the diagnostics. After terminating the stop operation only this function is quitted.

Definition

```
G_Error_t
G_Diag_Stop(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Diag_RequestMode_t mode,
 const u32_t timeout,
 const u8_t * const requestData,
 const u32_t requestLength,
 G_Error_t * const lastErrorCode,
 u8_t * const responseData,
 u32_t * const responseLength
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`mode`
Variable of type **G_Diag_RequestMode_t** indicating the request mode

The following return values are available:

- `G_DIAG__REQUEST_MODE__PHYSICAL_WITH_RESPONSE`

  Physical request with response
- `G_DIAG__REQUEST_MODE__FUNCTIONAL_WITH_RESPONSE`

  Functional request with response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_PHYSICAL_WITH_RESPONSE`

  Stop after physical request with response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_FUNCTIONAL_WITH_RESPONSE`

  Stop after functional request with response
- `G_DIAG__REQUEST_MODE__PHYSICAL_WITHOUT_RESPONSE`

  Physical request without response
- `G_DIAG__REQUEST_MODE__FUNCTIONAL_WITHOUT_RESPONSE`

  Functional request without response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_PHYSICAL_WITHOUT_RESPONSE`

  Stop after physical request without response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_FUNCTIONAL_WITHOUT_RESPONSE`

Stop after functional request without response

`timeout`
Receive timeout (in milliseconds)

`requestData`
Pointer to the request data buffer

`requestLength`
Size of request data buffer

`lastErrorCode`
Pointer to a variable of type **G_Error_t**. In this variable, the error code of the last error that has occurred in the firmware (or `G_NO_ERROR` = no error) is returned. For getting an error description you can call G_Common_GetFirmwareErrorDescription.

`responseData`
Pointer to the response data buffer

`responseLength`
Pointer to the variable `reponseLength`

Content when invoked: Size of response data buffer

Returned content: Size of response data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Stop the diagnostics with **G_Diag_Stop**

```
u8_t channel;
G_Diag_RequestMode_t mode;
u32_t timeout;
u8_t requestData[2];
u32_t requestLength;
G_Error_t lastErrorCode;
u8_t responseData[1024];
u32_t responseLength;

…

channel = 0;            //Multisession channel 0
mode = G_DIAG__REQUEST_MODE__PHYSICAL_WITH_RESPONSE;
timeout = 1000;         //1000 milliseconds
requestData[0] = 0x11;
requestData[1] = 0x22;
requestLength = 2;
responseLength = 1024;

rc =
  G_Diag_Stop(
    portHandle,
    channel,
    mode,
    timeout,
    requestData,
    requestLength,
    &lastErrorCode,
    responseData,
    &responseLength
  );

if (rc == G_NO_ERROR) {

…

}
```

In this example the required variables are declared first. With `requestData[2]` you can set a request buffer of 2 bytes and with `responseData[1024]` a response buffer of 1024 bytes. The sizes of both buffers are subsequently entered into the variables `requestLength` or `responseLength`. Furthermore the required values and parameters are input into the related variables. Additionally to the port handle they are subsequently transferred to the function **G_Diag_Stop** . In case of certain variables it is necessary to transfer their addresses or the pointers to the corresponding buffers. This is necessary because the corresponding return values are saved in several variables. As for `responseLength` for example, it contains on function call the size of the response buffer and on function return the size of the response data. The command is executed completely after receiving a response or an error or after processing the timeout. If the return code `rc` with the value `G_NO_ERROR` indicates an error-free execution, the contents of the variables can be evaluated.

G_Diag_InitiateStop

G_Diag_InitiateStop — Initiate the diagnostics stop

Description

This command initiates the diagnostics stop.

Please consider that on function return it cannot be ensured that the diagnostics has been stopped. The stopping request is only transferred to the hardware. If you want to make sure that the diagnostics has been stopped, it is required to use G_Diag_Stop.

Definition

```
G_Error_t
G_Diag_InitiateStop(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Diag_RequestMode_t mode,
 const u8_t * const requestData,
 const u32_t requestLength
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`mode`
Variable of type **G_Diag_RequestMode_t** indicating the request mode

The following return values are available:

- `G_DIAG__REQUEST_MODE__PHYSICAL_WITH_RESPONSE`

  Physical request with response
- `G_DIAG__REQUEST_MODE__FUNCTIONAL_WITH_RESPONSE`

  Functional request with response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_PHYSICAL_WITH_RESPONSE`

  Stop after physical request with response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_FUNCTIONAL_WITH_RESPONSE`

  Stop after functional request with response
- `G_DIAG__REQUEST_MODE__PHYSICAL_WITHOUT_RESPONSE`

  Physical request without response
- `G_DIAG__REQUEST_MODE__FUNCTIONAL_WITHOUT_RESPONSE`

  Functional request without response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_PHYSICAL_WITHOUT_RESPONSE`

  Stop after physical request without response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_FUNCTIONAL_WITHOUT_RESPONSE`

Stop after functional request without response

`requestData`
Pointer to the request data buffer

`requestLength`
Size of request data buffer

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Stop the diagnostics with **G_Diag_InitiateStop**

```
u8_t channel;
G_Diag_RequestMode_t mode;
u8_t requestData[2];
u32_t requestLength;

…

channel = 0;            //Multisession channel 0
mode = G_DIAG__REQUEST_MODE__PHYSICAL_WITH_RESPONSE;
requestData[0] = 0x11;
requestData[1] = 0x22;
requestLength = 2;

rc =
  G_Diag_InitiateStop(
    portHandle,
    channel,
    mode,
    requestData,
    requestLength
);
```

In this example the required variables are declared first. With `requestData[2]` a request buffer of 2 bytes is set. Its size is subsequently entered into the variable `requestLength`. Furthermore the required values and parameters are input into the related variables. Additionally to the port handle they are subsequently transferred to the function **G_Diag_InitiateStop** . `requestData` is the pointer to the request buffer.

G_Diag_GetState

G_Diag_GetState — Get the diagnostics state

Description

The actual diagnostics state can be queried by this command.

Definition

```
G_Error_t
G_Diag_GetState(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Diag_GetState_CmdFlags_t cmdFlags,
 G_Error_t * const lastErrorCode,
 G_Diag_State_t * const state,
 G_Diag_GetState_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`cmdFlags`
Variable of type **G_Diag_GetState_CmdFlags_t**

The following flags are available:

- `G_DIAG__GET_STATE__CMD_FLAG__NONE`

  No flag, if no other has to be transferred
- `G_DIAG__GET_STATE__CMD_FLAG__RESET_LAST_ERROR`

  Reset the error code of last error that has occurred in the firmware

`lastErrorCode`
Pointer to a variable of type **G_Error_t**. In this variable, the error code of the last error that has occurred in the firmware (or `G_NO_ERROR` = no error) is returned. For getting an error description, you can call G_Common_GetFirmwareErrorDescription.

`state`
Pointer to a variable of type **G_Diag_State_t**. The current diagnostics status is returned in this variable.

The following return values are available:

- `G_DIAG__STATE__NOT_INITIALIZED`

  not initialized
- `G_DIAG__STATE__DISCONNECTED`

  disconnected
- `G_DIAG__STATE__CONNECT_IN_PROGRESS`

The connecting is in progress.

- `G_DIAG__STATE__CONNECTED`

  connected

- `G_DIAG__STATE__DISCONNECT_IN_PROGRESS`

  The disconnection is in progress

`rspFlags`
Pointer to a variable of type **G_Diag_GetState_RspFlags_t**

The following return flags are available:

- `G_DIAG__GET_STATE__RSP_FLAG__NONE`

  No flag was set
- `G_DIAG__GET_STATE__RSP_FLAG__BUSY`

  A request has not been responded yet/ successfully sent
- `G_DIAG__GET_STATE__RSP_FLAG__SYNC_RX_BUFFER_NOT_EMPTY`

  Synchronous receiving buffer is not empty
- `G_DIAG__GET_STATE__RSP_FLAG__ASYNC_RX_BUFFER_NOT_EMPTY`

  Asynchronous receiving buffer is not empty
- `G_DIAG__GET_STATE__RSP_FLAG__UUDT_RX_BUFFER_NOT_EMPTY`

  UUDT receive buffer is not empty (UUDT = **U** nacknowledged **U** nsegmented **D** ata **T** ransfer)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Query of the diagnostics state with **G_Diag_GetState**

```
u8_t channel;
G_Diag_GetState_CmdFlags_t cmdFlags;
G_Error_t lastErrorCode;
G_Diag_State_t state;
G_Diag_GetState_RspFlags_t rspFlags;

…

channel = 0;            //Multisession channel 0
cmdFlags = G_DIAG__GET_STATE__CMD_FLAG__RESET_LAST_ERROR;

rc =
  G_Diag_GetState(
    portHandle,
    channel,
    cmdFlags,
    &lastErrorCode,
    &state,
    &rspFlags
  );

if (rc == G_NO_ERROR) {

…

}
```

In this example, first the required variables are declared and then the required values are entered. These are subsequently transferred to the function **G_Diag_GetState** additionally to the port handle. In

case of certain variables it is required to transfer their addresses too. This is necessary, because the related return values are saved in these variables. As for `lastErrorCode` for example, it contains on function return the error code of the last error that has occurred in the firmware. If the return code `rc` with the value `G_NO_ERROR` indicates an error-free execution, the contents of the variables can be evaluated.

G_Diag_GetState_Async

G_Diag_GetState_Async — Query the diagnostics status (by asynchronous query)

Description

This command starts the asynchronous query of the diagnostics status. As soon as the response is available, the callback function set with G_Diag_GetState_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Diag_GetState_Async(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Diag_GetState_CmdFlags_t cmdFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`cmdFlags`
Variable of type **G_Diag_GetState_CmdFlags_t**

The following flags are available:

- `G_DIAG__GET_STATE__CMD_FLAG__NONE`

  No flag, if no other has to be transferred
- `G_DIAG__GET_STATE__CMD_FLAG__RESET_LAST_ERROR`

  Reset the error code of the last error that has occurred in the firmware

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Diag_GetState_Async_Callback

G_Diag_GetState_Async_Callback — Callback function for the asynchronous query of the diagnostics status

Description

This function will be automatically called if a response for the asynchronous query of the diagnostics status with G_Diag_GetState_Async is available. It has to be entered by the user and to be set by the function G_Diag_GetState_Async_AddCallback.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Diag_GetState_Async_Callback(
 const G_PortHandle_t  portHandle,
 const u8_t channel,
 const G_Error_t  lastErrorCode,
 const G_Diag_State_t  state,
 const G_Diag_GetState_RspFlags_t  rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`lastErrorCode`
Error code of the last error that has occurred in the firmeware. For getting an error description you can call G_Common_GetFirmwareErrorDescription.

`state`
Current diagnostics status

The following return values are available:

- `G_DIAG__STATE__NOT_INITIALIZED`

  Not initialized
- `G_DIAG__STATE__DISCONNECTED`

  Disconnected
- `G_DIAG__STATE__CONNECT_IN_PROGRESS`

  The connecting is in progress.
- `G_DIAG__STATE__CONNECTED`

  Connected
- `G_DIAG__STATE__DISCONNECT_IN_PROGRESS`

  The disconnection is in progress.

`rspFlags`
Return flags

The following return flags are available:

- `G_DIAG__GET_STATE__RSP_FLAG__NONE`

  No flag was set
- `G_DIAG__GET_STATE__RSP_FLAG__BUSY`

  A request has not been responded yet/ successfully sent
- `G_DIAG__GET_STATE__RSP_FLAG__SYNC_RX_BUFFER_NOT_EMPTY`

  Synchronous receiving buffer is not empty
- `G_DIAG__GET_STATE__RSP_FLAG__ASYNC_RX_BUFFER_NOT_EMPTY`

  Asynchronous receiving buffer is not empty
- `G_DIAG__GET_STATE__RSP_FLAG__UUDT_RX_BUFFER_NOT_EMPTY`

  UUDT receiving buffer is not empty (UUDT = **U**nacknowledged **U**nsegmented **D**ata **T**ransfer)

G_Diag_GetState_Async_AddCallback

G_Diag_GetState_Async_AddCallback — Set the callback function for the asynchronous query of the diagnostics status

Description

This command defines a callback function for the asynchronous query of the diagnostics status with G_Diag_GetState_Async.

Definition

```
G_Error_t
G_Diag_GetState_Async_AddCallback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Diag_GetState_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`callback`
Function pointer of the callback function (see G_Diag_GetState_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Diag_GetState_Async_RemoveCallback

G_Diag_GetState_Async_RemoveCallback — Remove the callback function for G_Diag_GetState_Async

Description

This command removes the callback function for the asynchronous query of the diagnostics status.

Definition

```
G_Error_t
G_Diag_GetState_Async_RemoveCallback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Diag_Request

G_Diag_Request — Send a diagnostics request

Description

This command sends a diagnostics request.

Definition

```
G_Error_t
G_Diag_Request(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Diag_RequestMode_t mode,
 const u32_t timeout,
 const u8_t * const requestData,
 const u32_t requestLength,
 G_Error_t * const lastErrorCode,
 u8_t * const responseData,
 u32_t * const responseLength
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`mode`
Variable of type **G_Diag_RequestMode_t** indicating the request mode

The following return values are available:

- `G_DIAG__REQUEST_MODE__PHYSICAL_WITH_RESPONSE`

  Physical request with response
- `G_DIAG__REQUEST_MODE__FUNCTIONAL_WITH_RESPONSE`

  Functional request with response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_PHYSICAL_WITH_RESPONSE`

  Stop after physical request with response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_FUNCTIONAL_WITH_RESPONSE`

  Stop after functional request with response
- `G_DIAG__REQUEST_MODE__PHYSICAL_WITHOUT_RESPONSE`

  Physical request without response
- `G_DIAG__REQUEST_MODE__FUNCTIONAL_WITHOUT_RESPONSE`

  Functional request without response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_PHYSICAL_WITHOUT_RESPONSE`

  Stop after physical request without response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_FUNCTIONAL_WITHOUT_RESPONSE`

Stop after functional request without response

`timeout`
Receive timeout (in milliseconds)

`requestData`
Pointer to the request data buffer

`requestLength`
Size of request data buffer

`lastErrorCode`
Pointer to a variable of type **G_Error_t**. The error code of the last error that has occurred in the firmware (or `G_NO_ERROR` = no error) is returned in this variable. For getting an error description, you can call G_Common_GetFirmwareErrorDescription.

`responseData`
Pointer to the response data buffer

`responseLength`
Pointer to the variable `reponseLength`

Content when invoked: Size of response data buffer

Returned content: Size of response data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Send the diagnostics request with **G_Diag_Request**

```
u8_t channel;
G_Diag_RequestMode_t mode;
u32_t timeout;
u8_t requestData[2];
u32_t requestLength;
G_Error_t lastErrorCode;
u8_t responseData[1024];
u32_t responseLength;

…

channel = 0;            //Multisession channel 0
mode = G_DIAG__REQUEST_MODE__PHYSICAL_WITH_RESPONSE;
timeout = 1000;         //1000 milliseconds
requestData[0] = 0x11;
requestData[1] = 0x22;
requestLength = 2;
responseLength = 1024;

rc =
  G_Diag_Request(
    portHandle,
    channel,
    mode,
    timeout,
    requestData,
    requestLength,
    &lastErrorCode,
    responseData,
    &responseLength
  );

if (rc == G_NO_ERROR) {

  …

}
```

In this example the required variables are declared first. With `requestData[2]` you can set a request buffer of 2 bytes and with `responseData[1024]` a response buffer of 1024 bytes, whose sizes are subsequently entered into the variables `requestLength` or `responseLength`. Additionally the required values and parameters are input into the related variables. These are subsequently transferred to the function **G_Diag_Request** additionally to the port handle. In case of certain variables it is required that their addresses or pointers to the related buffers are transferred too. This is necessary, because the related return values are saved in certain variables. As for `responseLength` for example it contains on function call the size of the response buffer and on function return the size of the response data. The command is executed completely after receiving a response or an error or after processing the timeout. If the return code `rc` with the value `G_NO_ERROR` indicates an error-free execution, the contents of the variables can be evaluated.

G_Diag_QueueRequest

G_Diag_QueueRequest — Send the diagnostics request

Description

This command enqueues a diagnostics request into the sending operation of the defined hardware.

Please consider that on function return it cannot be ensured that the corresponding request has been transmitted. The request is only transferred to the hardware. If you want to ensure that this request has been sent, please use G_Diag_Request.

Definition

```
G_Error_t
G_Diag_QueueRequest(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Diag_RequestMode_t mode,
 const u8_t * const requestData,
 const u32_t requestLength
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`mode`
Variable of type **G_Diag_RequestMode_t** indicating the request mode

The following return values are available:

- `G_DIAG__REQUEST_MODE__PHYSICAL_WITH_RESPONSE`

  Physical request with response
- `G_DIAG__REQUEST_MODE__FUNCTIONAL_WITH_RESPONSE`

  Functional request with response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_PHYSICAL_WITH_RESPONSE`

  Stop after physical request with response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_FUNCTIONAL_WITH_RESPONSE`

  Stop after functional request with response
- `G_DIAG__REQUEST_MODE__PHYSICAL_WITHOUT_RESPONSE`

  Physical request without response
- `G_DIAG__REQUEST_MODE__FUNCTIONAL_WITHOUT_RESPONSE`

  Functional request without response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_PHYSICAL_WITHOUT_RESPONSE`

  Stop after physical request without response
- `G_DIAG__REQUEST_MODE__STOP_AFTER_FUNCTIONAL_WITHOUT_RESPONSE`

Stop after functional request without response

`requestData`
Pointer to the request data buffer

`requestLength`
Size of request data buffer

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Send the diagnostics request with **G_Diag_QueueRequest**

```
u8_t channel;
G_Diag_RequestMode_t mode;
u8_t requestData[2];
u32_t requestLength;

…

channel = 0;            //Multisession channel 0
mode = G_DIAG__REQUEST_MODE__PHYSICAL_WITH_RESPONSE;
requestData[0] = 0x11;
requestData[1] = 0x22;
requestLength = 2;

rc =
  G_Diag_QueueRequest(
    portHandle,
    channel,
    mode,
    requestData,
    requestLength
);
```

In this example the required variables are declared first. With `requestData[2]` a request buffer of 2 bytes is set, whose size is subsequently input into the variable `requestLength`. Additionally the required values and parameters are entered into the related variables. These are subsequently transferred to the function **G_Diag_QueueRequest** additionally to the port handle. `requestData` is the pointer to the request buffer.

G_Diag_GetResponse

G_Diag_GetResponse — Get the diagnostics response

Description

By means of this command, you get a diagnostics response from the specified hardware.

Definition

```
G_Error_t
G_Diag_GetResponse(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const u32_t timeout,
 G_Diag_State_t * const state,
 G_Error_t * const lastErrorCode,
 u8_t * const responseData,
 u32_t * const responseLength
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`timeout`
Receive timeout (in milliseconds)

`state`
Pointer to a variable of type **G_Diag_State_t**. The current diagnostics status is returned in this variable.

As for the return values, the following contents are available:

- `G_DIAG__STATE__NOT_INITIALIZED`

  Not initialized
- `G_DIAG__STATE__DISCONNECTED`

  Disconnected
- `G_DIAG__STATE__CONNECT_IN_PROGRESS`

  The connecting is in progress
- `G_DIAG__STATE__CONNECTED`

  Connected
- `G_DIAG__STATE__DISCONNECT_IN_PROGRESS`

  The disconnection is in progress

`lastErrorCode`
Pointer to a variabale of type **G_Error_t**. In this variable, the error code of the last error that has occurred in the firmware (or `G_NO_ERROR` = no error) is returned. For getting an error description you can call G_Common_GetFirmwareErrorDescription.

`responseData`
Pointer to the response data buffer

`responseLength`
Pointer to the variable `reponseLength`

Content when invoked: Size of response data buffer

Returned content: Size of response data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Get the diagnostics response with **G_Diag_GetResponse**

```
u8_t channel;
u32_t timeout;
G_Diag_State_t state;
u8_t lastErrorCode;
u8_t responseData[1024];
u32_t responseLength;

…

channel = 0;            //Multisession channel 0
timeout = 1000;         //1000 milliseconds
responseLength = 1024;

rc =
  G_Diag_GetResponse(
    portHandle,
    channel,
    timeout,
    &state,
    &lastErrorCode,
    responseData,
    &responseLength
  );

if (rc == G_NO_ERROR) {

  …

}
```

In this example the required variables are declared first. With `responseData[1024]` a response buffer of 1024 bytes is set, whose size is subsequently input into the variable `responseLength`. Additionally the required values are entered into the related variables. These are subsequently transferred to the function **G_Diag_GetResponse** additionally to the port handle. In case of certain variables it is required to transfer their addresses or the pointers to the related buffers too. This is necessary, because the related return values are saved in these variables. As for example `responseLength` it contains on function call the size of the response buffer and on function return the size of the response data. The command is executed completely after receiving a response or an error or after pro-

cessing the timeout. If the return code `rc` with the value `G_NO_ERROR` indicates an error-free execution, the contents of the variables can be evaluated.

G_Diag_GetResponse_Async_Enable

G_Diag_GetResponse_Async_Enable — Activate the asynchronous query of diagnostics responses

Description

This command activates the asynchronous query of diagnostics responses. As soon as a response is available, the callback function set with G_Diag_GetResponse_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Diag_GetResponse_Async_Enable(
 const G_PortHandle_t portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Diag_GetResponse_Async_Disable

G_Diag_GetResponse_Async_Disable — Deactivate the asynchronous query of diagnostics responses

Description

This command deactivates the asynchronous query of diagnostics responses.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Diag_GetResponse_Async_Disable(
 const G_PortHandle_t portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Diag_GetResponse_Async_Callback

G_Diag_GetResponse_Async_Callback — Callback function for the asynchronous query of a diagnostics response

Description

This function will be called as soon as a response for the asynchronous query of a diagnostics response is available. It has to be entered by the user and to be set with G_Diag_GetResponse_Async_AddCallback.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Diag_GetResponse_Async_Callback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Diag_State_t  state,
 const G_Error_t  lastErrorCode,
 const u8_t * const responseData,
 const u32_t  responseLength
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`state`
Actual diagnostics state

The following return values are available:

- `G_DIAG__STATE__NOT_INITIALIZED`

  Not initialized
- `G_DIAG__STATE__DISCONNECTED`

  Disconnected
- `G_DIAG__STATE__CONNECT_IN_PROGRESS`

  The connecting is in progress
- `G_DIAG__STATE__CONNECTED`

  Connected
- `G_DIAG__STATE__DISCONNECT_IN_PROGRESS`

  The disconnection is in progress

`lastErrorCode`
Error code of the last error that has occurred in the firmware. For getting an error description, you can call G_Common_GetFirmwareErrorDescription

`responseData`
Pointer to the response data buffer

`responseLength`
Response data size in byte

G_Diag_GetResponse_Async_AddCallback

G_Diag_GetResponse_Async_AddCallback — Set the callback function for the asynchronous query of a diagnostics response

Description

This command defines a callback function for the asynchronous query of a diagnostics response with G_Diag_GetResponse_Async_Enable

Definition

```
G_Error_t
G_Diag_GetResponse_Async_AddCallback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Diag_GetResponse_Async_Callback_t  callback
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

`callback`
: Function pointer of the callback function (see G_Diag_GetResponse_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Diag_GetResponse_Async_RemoveCallback

G_Diag_GetResponse_Async_RemoveCallback — Remove the callback function for G_Diag_GetResponse_Async_Enable

Description

This command removes the callback function for the asynchronous query of a diagnostics response.

Definition

```
G_Error_t
G_Diag_GetResponse_Async_RemoveCallback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Diag_Flag_SetById

G_Diag_Flag_SetById — Set diagnostics flag values by ID

Description

Use this command to set diagnostics flag values by ID.

Definition

```
G_Error_t
G_Diag_Flag_SetById(
 const G_PortHandle_t portHandle,
 const u8_t  channel,
 const G_Diag_Flag_SetById_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`cmd`
Command parameters

The structure contains the following elements:

`Id`
Flag ID

The following values are possible:

`G_DIAG__FLAG_ID__DISABLE_21_HANDLING`
The negative response **BusyRepeatRequest** is not treated.

`G_DIAG__FLAG_ID__DISABLE_23_HANDLING`
The negative response **RoutineNotComplete** is not treated.

`G_DIAG__FLAG_ID__DISABLE_78_HANDLING`
The negative response **RequestCorrectlyReceivedResponsePending** is not treated.

`G_DIAG__FLAG_ID__ASYNC_RESPONSE_AS_SYNC`
Unexpectedly received diagnostics responses are written to the normal diagnostics receiving buffer.

`G_DIAG__FLAG_ID__AUTOMATIC_EMPTY__SYNC`
Enable automatic reading of the normal diagnostics receiving buffer.

`G_DIAG__FLAG_ID__AUTOMATIC_EMPTY__ASYNC`
Enable automatic reading of the receiving buffer for unexpected diagnostics responses.

`G_DIAG__FLAG_ID__AUTOMATIC_EMPTY__UUDT`
Enable automatic reading of the receiving buffer for UUDT responses.

`Value`
Flag value to be set

0 = false, **1** = true

`reserved`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Diag_Flag_GetById

G_Diag_Flag_GetById — Get diagnostics flag values by ID

Description

Use this command to query diagnostics flag values by ID.

Definition

```
G_Error_t
G_Diag_Flag_GetById(
 const G_PortHandle_t portHandle,
 const u8_t  channel,
 const G_Diag_Flag_GetById_Cmd_t * const cmd,
 G_Diag_Flag_GetById_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`cmd`
Command parameters

The structure contains the following elements:

`Id`
Flag ID

The following values are possible:

`G_DIAG__FLAG_ID__DISABLE_21_HANDLING`
The negative response **BusyRepeatRequest** is not treated.

`G_DIAG__FLAG_ID__DISABLE_23_HANDLING`
The negative response **RoutineNotComplete** is not treated.

`G_DIAG__FLAG_ID__DISABLE_78_HANDLING`
The negative response **RequestCorrectlyReceivedResponsePending** is not treated.

`G_DIAG__FLAG_ID__ASYNC_RESPONSE_AS_SYNC`
Unexpectedly received diagnostics responses are written to the normal diagnostics receiving buffer.

`G_DIAG__FLAG_ID__AUTOMATIC_EMPTY__SYNC`
Enable automatic reading of the normal diagnostics receiving buffer.

`G_DIAG__FLAG_ID__AUTOMATIC_EMPTY__ASYNC`
Enable automatic reading of the receiving buffer for unexpected diagnostics responses.

`G_DIAG__FLAG_ID__AUTOMATIC_EMPTY__UUDT`
Enable automatic reading of the receiving buffer for UUDT responses.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`Id`
Flag ID

The following values are possible:

`G_DIAG__FLAG_ID__DISABLE_21_HANDLING`
The negative response **BusyRepeatRequest** is not treated.

`G_DIAG__FLAG_ID__DISABLE_23_HANDLING`
The negative response **RoutineNotComplete** is not treated.

`G_DIAG__FLAG_ID__DISABLE_78_HANDLING`
The negative response **RequestCorrectlyReceivedResponsePending** is not treated.

`G_DIAG__FLAG_ID__ASYNC_RESPONSE_AS_SYNC`
Unexpectedly received diagnostics responses are written to the normal diagnostics receiving buffer.

`G_DIAG__FLAG_ID__AUTOMATIC_EMPTY__SYNC`
Enable automatic reading of the normal diagnostics receiving buffer.

`G_DIAG__FLAG_ID__AUTOMATIC_EMPTY__ASYNC`
Enable automatic reading of the receiving buffer for unexpected diagnostics responses.

`G_DIAG__FLAG_ID__AUTOMATIC_EMPTY__UUDT`
Enable automatic reading of the receiving buffer for UUDT responses.

`Value`
Returns flag value

0 = false, **1** = true

`reserved`
reserved parameter

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Diag_GetLastError

G_Diag_GetLastError — Get last diagnostics error

Description

Use this command to query the last diagnostics error.

Definition

```
G_Error_t
G_Diag_GetLastError(
 const G_PortHandle_t portHandle,
 const u8_t  channel,
 G_Error_t * const lastErrorCode
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`lastErrorCode`
Returns last diagnostics error code

The error description for this error code can be queried with G_GetErrorDescription .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Diag_Property_SetById

G_Diag_Property_SetById — Set diagnostics property value

Description

Use this command to set a diagnostics property value.

Definition

```
G_Error_t
G_Diag_Property_SetById(
 const G_PortHandle_t portHandle,
 const u8_t  channel,
 const G_Diag_Property_SetById_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`cmd`
Command parameters

The structure contains the following elements:

`Id`
Diagnostics property ID

The following values are possible:

`G_DIAG__PROPERTY_ID__REQUEST_DELAY__21__MS`
Diagnostics request delay after reception of an awaited diagnostics response with negative response code 0x21 (busy repeat request)

in ms, e.g. 10 (default = 0)

`G_DIAG__PROPERTY_ID__REQUEST_DELAY__23__MS`
Minimum time to wait after successful transmission of a functional addressed request message with no response required

in ms, e.g. 0 (default = 0)

`G_DIAG__PROPERTY_ID__REQUEST_DELAY__PHYSICAL__MS`
Minimum time to wait after successful transmission of a physically addressed request message with no response required

in ms, e.g. 0 (default = 0)

`G_DIAG__PROPERTY_ID__REQUEST_DELAY__FUNCTIONAL__MS`
Minimum time to wait after successful transmission of a functional addressed request message with no response required.

in ms, e.g. 0 (default = 0)

`G_DIAG__PROPERTY_ID__TX_BUFFER_SIZE`
Size of the transmit buffer, in bytes

`G_DIAG__PROPERTY_ID__RX_BUFFER_SIZE__SYNC`
Size of the receive buffer for expected responses, in bytes

`G_DIAG__PROPERTY_ID__RX_BUFFER_SIZE__ASYNC`
Size of the receive buffer for unexpected responses, in bytes

`G_DIAG__PROPERTY_ID__RX_TIMEOUT_OPT_RESPONSE__MS`
Diagnostics response timeout for an optional response

in ms, e.g. 200

On diagnostics initialization this timeout is set to the normal response timeout (**P2Max**)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Value`
Value of the property to be set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Diag_Property_GetById

G_Diag_Property_GetById — Get diagnostics property value

Description

Use this command to query a diagnostics property value.

Definition

```
G_Error_t
G_Diag_Property_GetById(
 const G_PortHandle_t portHandle,
 const u8_t  channel,
 const G_Diag_Property_GetById_Cmd_t * const cmd,
 G_Diag_Property_GetById_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`cmd`
Command parameters

The structure contains the following elements:

`Id`
Diagnostics property ID

The following values are possible:

`G_DIAG__PROPERTY_ID__REQUEST_DELAY__21__MS`
Diagnostics request delay after reception of an awaited diagnostics response with negative response code 0x21 (busy repeat request)

in ms, e.g. 10 (default = 0)

`G_DIAG__PROPERTY_ID__REQUEST_DELAY__23__MS`
Minimum time to wait after successful transmission of a functional addressed request message with no response required

in ms, e.g. 0 (default = 0)

`G_DIAG__PROPERTY_ID__REQUEST_DELAY__PHYSICAL__MS`
Minimum time to wait after successful transmission of a physically addressed request message with no response required

in ms, e.g. 0 (default = 0)

`G_DIAG__PROPERTY_ID__REQUEST_DELAY__FUNCTIONAL__MS`
Minimum time to wait after successful transmission of a functional addressed request message with no response required.

in ms, e.g. 0 (default = 0)

G_DIAG__PROPERTY_ID__TX_BUFFER_SIZE
Size of the transmit buffer, in bytes

G_DIAG__PROPERTY_ID__RX_BUFFER_SIZE__SYNC
Size of the receive buffer for expected responses, in bytes

G_DIAG__PROPERTY_ID__RX_BUFFER_SIZE__ASYNC
Size of the receive buffer for unexpected responses, in bytes

G_DIAG__PROPERTY_ID__RX_TIMEOUT_OPT_RESPONSE__MS
Diagnostics response timeout for an optional response

in ms, e.g. 200

On diagnostics initialization this timeout is set to the normal response timeout (**P2Max**)

reserved1
reserved parameter (must be initialized with **0**)

reserved2
reserved parameter (must be initialized with **0**)

rsp
Returns response parameters

The structure contains the following elements:

Id
Diagnostics property ID

The following values are possible:

G_DIAG__PROPERTY_ID__REQUEST_DELAY__21__MS
Diagnostics request delay after reception of an awaited diagnostics response with negative response code 0x21 (busy repeat request)

in ms, e.g. 10 (default = 0)

G_DIAG__PROPERTY_ID__REQUEST_DELAY__23__MS
Minimum time to wait after successful transmission of a functional addressed request message with no response required

in ms, e.g. 0 (default = 0)

G_DIAG__PROPERTY_ID__REQUEST_DELAY__PHYSICAL__MS
Minimum time to wait after successful transmission of a physically addressed request message with no response required

in ms, e.g. 0 (default = 0)

G_DIAG__PROPERTY_ID__REQUEST_DELAY__FUNCTIONAL__MS
Minimum time to wait after successful transmission of a functional addressed request message with no response required.

in ms, e.g. 0 (default = 0)

G_DIAG__PROPERTY_ID__TX_BUFFER_SIZE
Size of the transmit buffer, in bytes

G_DIAG__PROPERTY_ID__RX_BUFFER_SIZE__SYNC
Size of the receive buffer for expected responses, in bytes

`G_DIAG__PROPERTY_ID__RX_BUFFER_SIZE__ASYNC`
Size of the receive buffer for unexpected responses, in bytes

`G_DIAG__PROPERTY_ID__RX_TIMEOUT_OPT_RESPONSE__MS`
Diagnostics response timeout for an optional response

in ms, e.g. 200

On diagnostics initialization this timeout is set to the normal response timeout (**P2Max**)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Value`
Returns property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Diag_Session_Reset

G_Diag_Session_Reset — Reset diag session

Description

Use this command to reset the diag session.

The diag configuration parameters will not be lost, but all state variables are reset to their default values. Used TP connections are also reset immediately.

Definition

```
G_Error_t
G_Diag_Session_Reset(
 const G_PortHandle_t portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Diag_Reset

G_Diag_Reset — Reset the whole diag channel

Description

Use this command to reset the whole diag channel.

All diag channel parameters get lost and all diag channel resources are freed (timers, TP channel usage, buffers, ...).

Definition

```
G_Error_t
G_Diag_Reset(
 const G_PortHandle_t portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Ethernet Functions

These **G-API** functions provide control over the Ethernet interface functionality of your hardware, including transport and diagnostics protocols.

1 Diag

These commands are used to control the Ethernet diagnostics.

G_Ethernet_Diag_Config_Off

G_Ethernet_Diag_Config_Off — Switch off diagnostic

Description

Use this command to switch off diagnostic.

Definition

```
G_Error_t
G_Ethernet_Diag_Config_Off(
 const G_PortHandle_t portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Diag multisession channel

A multisession channel is required if multiple applications are working with the diagnostics on the same interface. For each channel, the diagnostics can be configured with different parameters.

Usually, it is not neccessary to use multisession channels. It is recommended to use channel **`0`** in this case.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Diag_Config_Uds

G_Ethernet_Diag_Config_Uds — Configure Ethernet UDS diagnostics

Description

Use this command to configure UDS diagnostics.

Definition

```
G_Error_t
G_Ethernet_Diag_Config_Uds(
 const G_PortHandle_t portHandle,
 const u8_t  channel,
 const u32_t  globalTimeout,
 const G_Ethernet_Diag_Config_Uds_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Diag multisession channel

A multisession channel is required if multiple applications are working with the diagnostics on the same interface. For each channel, the diagnostics can be configured with different parameters.

Usually, it is not neccessary to use multisession channels. It is recommended to use channel **0** in this case.

`globalTimeout`
Global diag timeout in milliseconds

`cmd`
Structure with command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__DIAG__CONFIG__UDS__FLAG__NONE`
No flag is set

`G_ETHERNET__DIAG__CONFIG__UDS__FLAG__ENABLE_TESTER_PRESENT`
Enable Tester Present signal

`ResponseTimeout`
Normal response timeout in milliseconds, e.g. **200 milliseconds**

`ResponseTimeout_78`
Extended response timeout in milliseconds, e.g. **5100 milliseconds** (after reception of negative response code **0x78** - request correctly received - response pending)

`RequestRepetitions`
Number of request repetitions after response timeout, e.g. **2**

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`TesterPresent`
Structure with Tester Present parameters

These parameters are only required when commmand flag `G_ETHERNET__DIAG__CONFIG__UDS__FLAG__ENABLE_TESTER_PRESENT` is set.

The structure contains the following elements:

`Cycle`
Cycle time in milliseconds

`RequestMode`
Request mode (see G_Diag_RequestMode_t)

`Length`
Tester Present data length in bytes (max. **8 bytes**)

`Data`
Tester Present data bytes (max. **8 bytes**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

2 MII Multiplexer

MII Multiplexer functionality

MII stands for **M**edia **I**ndependent **I**nterface.

With the MII Multiplexer you can multiplex connections between different communication sources and targets directly on the device.

G_Ethernet_MiiMux_ResetOwnSelections

G_Ethernet_MiiMux_ResetOwnSelections — Reset MII Multiplexer selections on selected interface

Description

Use this command to reset the MII Multiplexer selections for the interface specified by `portHandle`.

In contrast to command G_Ethernet_MiiMux_ResetAllSelections , only the connections of the selected interface will be reset. If all connections of all interfaces of the device shall be reset, use command G_Ethernet_MiiMux_ResetAllSelections instead!

Definition

```
G_Error_t
G_Ethernet_MiiMux_ResetOwnSelections(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_MiiMux_ResetAllSelections

G_Ethernet_MiiMux_ResetAllSelections — Reset all MII Multiplexer connections

Description

Use this command to reset all MII Multiplexer connections on every ETHERNET interface of the selected device.

If you want to reset only the MII Multiplexer connections of a specific interface, use command G_Ethernet_MiiMux_ResetOwnSelections !

Definition

```
G_Error_t
G_Ethernet_MiiMux_ResetAllSelections(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_MiiMux_Selection_Set

G_Ethernet_MiiMux_Selection_Set — Set MII Multiplexer selection

Description

Use this command to set a MII Multiplexer connection.

Definition

```
G_Error_t
G_Ethernet_MiiMux_Selection_Set(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_MiiMux_Selection_Set_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__MII_MUX__SELECTION__SET__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__MII_MUX__SELECTION__SET__CMD_FLAG__OWN_TARGET`
0 : use `TargetInstanceId` as target instance ID (targets from other ETHERNET interfaces can be used)

1 : use own instance ID as target instance ID (the target of the current ETHERNET interface is used)

`G_ETHERNET__MII_MUX__SELECTION__SET__CMD_FLAG__OWN_SOURCE`
0 : use `SourceInstanceId` as source instance ID (sources from other ETHERNET interfaces can be used)

1 : use own instance ID as source instance ID (the source of the current ETHERNET interface is used)

`TargetType`
Target type

The following values are possible:

`G_ETHERNET__MII_MUX__TARGET_TYPE__MAC_RX`
The target is the ethernet media access controller.

`G_ETHERNET__MII_MUX__TARGET_TYPE__PHY_TX`
The target is channel 0 of the switch merging the data before the sending side of the phy.

`G_ETHERNET__MII_MUX__TARGET_TYPE__PHY_TX_1`
The target is channel 1 of the switch merging the data before the sending side of the phy.

`G_ETHERNET__MII_MUX__TARGET_TYPE__PHY_TX_2`
The target is channel 2 of the switch merging the data before the sending side of the phy.

`G_ETHERNET__MII_MUX__TARGET_TYPE__PKT_FILTER`
The target is the packet filter.

`TargetInstanceId`
Instance ID of the target specified by `TargetType`, beginning with **0**

`SourceType`
Source type

The following values are possible:

`G_ETHERNET__MII_MUX__SOURCE_TYPE__PHY_RX`
The source is receiving side of the phy.

`G_ETHERNET__MII_MUX__SOURCE_TYPE__MAC_TX`
The source is the output of the ethernet media access controller.

`G_ETHERNET__MII_MUX__SOURCE_TYPE__PKT_GEN`
The source is the hardware packet generator.

`G_ETHERNET__MII_MUX__SOURCE_TYPE__NOT_USED`
The source is not used.

`G_ETHERNET__MII_MUX__SOURCE_TYPE__PKT_FILTER`
The source is the output of the packet filter.

`SourceInstanceId`
Instance ID of the source specified by `SourceType`, beginning with **0**

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_MiiMux_Selection_Get

G_Ethernet_MiiMux_Selection_Get — Get MII Multiplexer selection

Description

Use this command to query a MII Multiplexer selection.

Definition

```
G_Error_t
G_Ethernet_MiiMux_Selection_Get(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_MiiMux_Selection_Get_Cmd_t * const cmd,
 G_Ethernet_MiiMux_Selection_Get_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__MII_MUX__SELECTION__GET__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__MII_MUX__SELECTION__GET__CMD_FLAG__OWN_TARGET`
0 : use `TargetInstanceId` as target instance ID (targets from other ETHERNET interfaces can be used)

1 : use own instance ID as target instance ID (the target of the current ETHERNET interface is used)

`TargetType`
Target type

The following values are possible:

`G_ETHERNET__MII_MUX__TARGET_TYPE__MAC_RX`
The target is the ethernet media access controller.

`G_ETHERNET__MII_MUX__TARGET_TYPE__PHY_TX`
The target is channel 0 of the switch merging the data before the sending side of the phy.

`G_ETHERNET__MII_MUX__TARGET_TYPE__PHY_TX_1`
The target is channel 1 of the switch merging the data before the sending side of the phy.

`G_ETHERNET__MII_MUX__TARGET_TYPE__PHY_TX_2`
The target is channel 2 of the switch merging the data before the sending side of the phy.

G_ETHERNET__MII_MUX__TARGET_TYPE__PKT_FILTER
The target is the packet filter.

TargetInstanceId
Instance ID of the target specified by TargetType, beginning with 0

reserved1
reserved parameter (must be initialized with 0)

reserved2
reserved parameter (must be initialized with 0)

rsp
Returns response parameters

The structure contains the following elements:

Flags
Response flags

The following values are possible and can be combined:

G_ETHERNET__MII_MUX__SELECTION__GET__RSP_FLAG__NONE
No flag is set

G_ETHERNET__MII_MUX__SELECTION__GET__RSP_FLAG__OWN_TARGET
0 : use TargetInstanceId as target instance ID (target is from another ETHERNET interface)

1 : own instance ID as target instance ID (the target on the current ETHERNET interface is used)

G_ETHERNET__MII_MUX__SELECTION__SET__CMD_FLAG__OWN_SOURCE
0 : use SourceInstanceId as source instance ID (source is from another ETHERNET interface)

1 : own instance ID as source instance ID (the source on the current ETHERNET interface is used)

TargetType
Target type

The following values are possible:

G_ETHERNET__MII_MUX__TARGET_TYPE__MAC_RX
The target is the ethernet media access controller.

G_ETHERNET__MII_MUX__TARGET_TYPE__PHY_TX
The target is channel 0 of the switch merging the data before the sending side of the phy.

G_ETHERNET__MII_MUX__TARGET_TYPE__PHY_TX_1
The target is channel 1 of the switch merging the data before the sending side of the phy.

G_ETHERNET__MII_MUX__TARGET_TYPE__PHY_TX_2
The target is channel 2 of the switch merging the data before the sending side of the phy.

G_ETHERNET__MII_MUX__TARGET_TYPE__PKT_FILTER
The target is the packet filter.

TargetInstanceId
Instance ID of the target specified by TargetType, beginning with 0

`SourceType`
Source type

The following values are possible:

`G_ETHERNET__MII_MUX__SOURCE_TYPE__PHY_RX`
The source is receiving side of the phy.

`G_ETHERNET__MII_MUX__SOURCE_TYPE__MAC_TX`
The source is the output of the ethernet media access controller.

`G_ETHERNET__MII_MUX__SOURCE_TYPE__PKT_GEN`
The source is the hardware packet generator.

`G_ETHERNET__MII_MUX__SOURCE_TYPE__NOT_USED`
The source is not used.

`G_ETHERNET__MII_MUX__SOURCE_TYPE__PKT_FILTER`
The source is the output of the packet filter.

`SourceInstanceId`
Instance ID of the source specified by `SourceType`, beginning with **0**

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

3 Monitor

Monitor functions of the Ethernet interface

G_Ethernet_Monitor_Config

G_Ethernet_Monitor_Config — Configuration of Ethernet Monitor

Description

Use this command for configuration of the Ethernet Monitor.

Definition

```
G_Error_t
G_Ethernet_Monitor_Config(
 const G_PortHandle_t portHandle,
 const G_Ethernet_Monitor_Config_Flags_t  cmdFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__MONITOR__CONFIG__FLAG__NONE`
No flag is set

`G_ETHERNET__MONITOR__CONFIG__FLAG__CLEAR_BUFFER_ON_START`
0 : Don't delete monitor items on monitor start

1 : Delete monitor items on monitor start

`G_ETHERNET__MONITOR__CONFIG__FLAG__CLEAR_BUFFER_ON_STOP`
0 : Don't delete monitor items on monitor stop

1 : Delete monitor items on monitor stop

`G_ETHERNET__MONITOR__CONFIG__FLAG__SORT_CHRONOLOGICAL`
0 : Monitor items are not sorted: they are enqueued in the order they are processed

1 : Monitor items are sorted in chronological order

`G_ETHERNET__MONITOR__CONFIG__FLAG__USE_API_BUFFER`
0 : Monitor items are buffered in the device internal memory

1 : Monitor items are bufferes in API memory, allowing a greater buffer size

Flag `G_ETHERNET__MONITOR__CONFIG__FLAG__SORT_CHRONOLOGICAL` is disregarded since sorting items is not possible in API layer.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Monitor_Start

G_Ethernet_Monitor_Start — Start Ethernet Monitor

Description

Use this command to start the Ethernet Monitor.

Definition

```
G_Error_t
G_Ethernet_Monitor_Start(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Monitor_Stop

G_Ethernet_Monitor_Stop — Stop Ethernet Monitor

Description

Use this command to stop the Ethernet Monitor.

The monitor configuration will stay valid and the monitor can be re-started with G_Ethernet_Monitor_Start .

To reset monitor configuration, use command G_Ethernet_Monitor_Reset .

Definition

```
G_Error_t
G_Ethernet_Monitor_Stop(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Monitor_Reset

G_Ethernet_Monitor_Reset — Reset Ethernet Monitor

Description

Use this command to reset the Ethernet Monitor configuration.

Definition

```
G_Error_t
G_Ethernet_Monitor_Reset(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Monitor_GetData

G_Ethernet_Monitor_GetData — Query Ethernet Monitor data

Description

Use this command to query Ethernet Monitor data.

Definition

```
G_Error_t
G_Ethernet_Monitor_GetData(
 const G_PortHandle_t  portHandle,
 u16_t * const numberOfItems,
 u32_t * const length,
 u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfItems`
Returns number of items in buffer `data`

`length`
in : size of buffer `data`, in bytes

out : size of monitor data returned in buffer `data`, in bytes

`data`
Returns stream with items

An item of the stream contains the following elements:

`ItemHeader`
Header of monitor item

The structure contains the following elements:

`ItemLength`
Length of the whole item, in bytes (has to be a multiple of 4, because items are 32 bit aligned)

`ItemType`
Item type

The following values are possible:

`G_ETHERNET__MONITOR__ITEM_TYPE__BUFFER_OVERRUN`
A buffer overrun occurred

This item type has no further item specific information.

`G_ETHERNET__MONITOR__ITEM_TYPE__FRAME`
Ethernet frame

Uses data union field `Frame`.

`G_ETHERNET__MONITOR__ITEM_TYPE__USER_EVENT`
User event

Uses data union field `UserEvent`.

A user event can be written with G_Ethernet_Monitor_WriteUserEvent .

`TimeStamp`
Time stamp of the monitor item, in nanoseconds

u
Monitor item data union

Contains monitor item data depending on the value of `ItemType`.

The union contains the following elements:

`Frame`
Contains monitor item data for item type `G_ETHERNET__MONITOR__ITEM_TYPE__FRAME`.

The structure contains the following elements:

`Flags`
Frame flags

The following values are possible and can be combined:

`G_ETHERNET__MONITOR__ITEM__FRAME__FLAG__NONE`
No flag is set

`G_ETHERNET__MONITOR__ITEM__FRAME__FLAG__TX`
0 : Item has been received

1 : Item has been transmitted

`reserved`
reserved parameter (must be initialized with **0**)

`FrameDataLength`
Length of frame data, in bytes

`FrameData`
Frame data bytes

`UserEvent`
Contains monitor item data for item type `G_ETHERNET__MONITOR__ITEM_TYPE__USER_EVENT`.

The structure contains the following elements:

`UserEvent`
User event ID

`reserved1`
reserved parameter (must be initialized with **0**)

`Length`
Data length, in bytes

`reserved2`
reserved parameter (must be initialized with **0**)

`Data`
User event data bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Monitor_GetTime

G_Ethernet_Monitor_GetTime — Get monitor start time and current monitor time

Description

Use this command to query the monitor start time and the current monitor time.

Definition

```
G_Error_t
G_Ethernet_Monitor_GetTime(
 const G_PortHandle_t  portHandle,
 G_Ethernet_Monitor_GetTime_RspFlags_t * const rspFlags,
 u64_t * const monitorStartTime,
 u64_t * const currentMonitorTime
);
```

Parameters

`portHandle`
Handle to the communication port

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_ETHERNET__MONITOR__GET_TIME__RSP_FLAG__NONE`
No flag is set

`G_ETHERNET__MONITOR__GET_TIME__RSP_FLAG__MONITOR_STARTED`
The monitor has been started

`monitorStartTime`
Returns monitor start time, in nanoseconds

`currentMonitorTime`
Returns current monitor time, in nanoseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Monitor_WriteUserEvent

G_Ethernet_Monitor_WriteUserEvent — Write user event

Description

Use this command to write a user event to the Ethernet Monitor.

Definition

```
G_Error_t
G_Ethernet_Monitor_WriteUserEvent(
 const G_PortHandle_t  portHandle,
 const u8_t  userEvent,
 const u8_t  length,
 const u8_t * const data
);
```

Parameters

`portHandle`
: Handle to the communication port

`userEvent`
: User event ID

`length`
: Length of user event data, in bytes

`data`
: User event data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Monitor_Property_SetById

G_Ethernet_Monitor_Property_SetById — Set monitor properties

Description

Use this command to set Ethernet Monitor properties.

Definition

```
G_Error_t
G_Ethernet_Monitor_Property_SetById(
 const G_PortHandle_t  portHandle,
 const u32_t  id,
 const u32_t  value
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Property ID

The following values are possible:

`G_ETHERNET__MONITOR__PROPERTY_ID__MONITOR_SORTING__HOLD_TIME__NS`
Defines the minimum time that all monitor entries are hold in the monitor sorting buffer before the application can access them.

`G_ETHERNET__MONITOR__PROPERTY_ID__MONITOR_START__DELAY_TIME__US`
Defines the delay after which the monitor starts recording valid entries

`G_ETHERNET__MONITOR__PROPERTY_ID__MONITOR_STOP__DELAY_TIME__US`
Defines the delay after which the monitor stops processing entries, all valid entries with time stamps older than the monitor stop time will be dropped, time stamps earlier than the monitor stop time are still added to the monitor sorting buffer and can be accessed by the application.

`value`
Property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Monitor_Property_GetById

G_Ethernet_Monitor_Property_GetById — Query monitor properties

Description

Use this command to query monitor properties.

Definition

```
G_Error_t
G_Ethernet_Monitor_Property_GetById(
 const G_PortHandle_t  portHandle,
 const u32_t  id,
 u32_t * const value
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Property ID

The following values are possible:

`G_ETHERNET__MONITOR__PROPERTY_ID__MONITOR_SORTING__HOLD_TIME__NS`
Defines the minimum time that all monitor entries are hold in the monitor sorting buffer before the application can access them.

`G_ETHERNET__MONITOR__PROPERTY_ID__MONITOR_START__DELAY_TIME__US`
Defines the delay after which the monitor starts recording valid entries

`G_ETHERNET__MONITOR__PROPERTY_ID__MONITOR_STOP__DELAY_TIME__US`
Defines the delay after which the monitor stops processing entries, all valid entries with time stamps older than the monitor stop time will be dropped, time stamps earlier than the monitor stop time are still added to the monitor sorting buffer and can be accessed by the application.

`value`
Returns property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

4 Node

Node functions of the Ethernet interface

A code example for the configuration of ETHERNET nodes can be found in the samples directory under *Ethernet_TxFifo_Tar*.

G_Ethernet_Node_LocalIp4Address_Add

G_Ethernet_Node_LocalIp4Address_Add — Add local IPv4 address

Description

Use this function to add a local IPv4 Address to the Ethernet interface. This address can then be used for communication with other devices.

Definition

```
G_Error_t
G_Ethernet_Node_LocalIp4Address_Add(
 const G_PortHandle_t portHandle,
 const G_Ethernet_Node_LocalIp4Address_Add_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__NODE__LOCAL_IP4_ADDRESS__ADD__CMD_FLAG__NONE`
No flag is set

`IpAddress`
IPv4 address to be added (in network byte order)

`NetMask`
Net mask corresponding to IPv4 address (in network byte order)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Node_LocalIp4Address_Remove

G_Ethernet_Node_LocalIp4Address_Remove — Remove local IPv4 address

Description

Use this function to remove a local IPv4 Address from the Ethernet interface.

Definition

```
G_Error_t
G_Ethernet_Node_LocalIp4Address_Remove(
 const G_PortHandle_t portHandle,
 const G_Ethernet_Node_LocalIp4Address_Remove_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__NODE__LOCAL_IP4_ADDRESS__REMOVE__CMD_FLAG__NONE`
No flag is set

`IpAddress`
IPv4 address to be removed (in network byte order)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Node_LocalIp4Address_GetAll

G_Ethernet_Node_LocalIp4Address_GetAll — Query all local IPv4 addresses

Description

Use this command to query all configured local IPv4 addresses of the Ethernet interface.

Definition

```
G_Error_t
G_Ethernet_Node_LocalIp4Address_GetAll(
 const G_PortHandle_t portHandle,
 const G_Ethernet_Node_LocalIp4Address_GetAll_Cmd_t * const cmd,
 G_Ethernet_Node_LocalIp4Address_GetAll_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__NODE__LOCAL_IP4_ADDRESS__GET_ALL__CMD_FLAG__NONE`
No flag is set

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_ETHERNET__NODE__LOCAL_IP4_ADDRESS__GET_ALL__RSP_FLAG__NONE`
No flag is set

`NumberOfAddresses`
Number of returned IPv4 addresses in buffer `Addresses`

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Addresses`
Array of returned IPv4 addresses

The array elements have the following structure:

`IpAddress`
IPv4 address (in network byte order)

`NetMask`
Net mask corresponding to IPv4 address (in network byte order)

`reserved`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Node_LocalIp6Address_Add

G_Ethernet_Node_LocalIp6Address_Add — Add local IPv6 address

Description

Use this function to add a local IPv6 Address to the Ethernet interface. This address can then be used for communication with other devices.

Definition

```
G_Error_t
G_Ethernet_Node_LocalIp6Address_Add(
 const G_PortHandle_t portHandle,
 const G_Ethernet_Node_LocalIp6Address_Add_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__NODE__LOCAL_IP4_ADDRESS__ADD__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__NODE__LOCAL_IP6_ADDRESS__ADD__CMD_FLAG__USE_VLAN_ID`
Use VLAN ID defined by `VlanId`

A **V**irtual **L**ocal **A**rea **N**etwork can be configured with G_Ethernet_Node_Vlan_Config .

`IpAddress`
IPv6 address to be added (in network byte order)

`PrefixLength`
Length of prefix in bits

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

VlanId
VLAN ID

Id for a **V**irtual **L**ocal **A**rea **N**etwork

Used if flag `G_ETHERNET__NODE__LOCAL_IP6_ADDRESS__ADD__CMD_FLAG__USE_VLAN_ID` is set.

A VLAN has to be configured with G_Ethernet_Node_Vlan_Config first.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Node_LocalIp6Address_Remove

G_Ethernet_Node_LocalIp6Address_Remove — Remove local IPv6 address

Description

Use this function to remove a local IPv6 Address from the Ethernet interface.

Definition

```
G_Error_t
G_Ethernet_Node_LocalIp6Address_Remove(
 const G_PortHandle_t portHandle,
 const G_Ethernet_Node_LocalIp6Address_Remove_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__NODE__LOCAL_IP6_ADDRESS__REMOVE__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__NODE__LOCAL_IP6_ADDRESS__REMOVE__CMD_FLAG__USE_VLAN_ID`
Use VLAN ID defined by `VlanId`

`IpAddress`
IPv6 address to be removed (in network byte order)

VlanId
VLAN ID

Id for a **V**irtual **L**ocal **A**rea **N**etwork

Used if flag `G_ETHERNET__NODE__LOCAL_IP6_ADDRESS__REMOVE__CMD_FLAG__USE_VLAN_ID` is set.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Node_LocalIp6Address_GetAll

G_Ethernet_Node_LocalIp6Address_GetAll — Query all local IPv6 addresses

Description

Use this command to query all configured local IPv6 addresses of the Ethernet interface.

Definition

```
G_Error_t
G_Ethernet_Node_LocalIp6Address_GetAll(
 const G_PortHandle_t portHandle,
 const G_Ethernet_Node_LocalIp6Address_GetAll_Cmd_t * const cmd,
 G_Ethernet_Node_LocalIp6Address_GetAll_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__NODE__LOCAL_IP6_ADDRESS__GET_ALL__CMD_FLAG__NONE`
No flag is set

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_ETHERNET__NODE__LOCAL_IP6_ADDRESS__GET_ALL__RSP_FLAG__NONE`
No flag is set

`NumberOfAddresses`
Number of returned IPv4 addresses in buffer `Addresses`

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Addresses`
Array of returned IPv4 addresses

The array elements have the following structure:

`IpAddress`
IPv4 address (in network byte order)

`NetMask`
Net mask corresponding to IPv4 address (in network byte order)

`reserved`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Node_Vlan_Config

G_Ethernet_Node_Vlan_Config — Configuration of a Virtual Local Area Network

Description

Use this command to configure a Virtual Local Area Network.

Definition

```
G_Error_t
G_Ethernet_Node_Vlan_Config(
 const G_PortHandle_t portHandle,
 const G_Ethernet_Node_Vlan_Config_Cmd_t * const cmd
);
```

Parameters

`portHandle`
: Handle to the communication port

cmd
: Command parameters

  The structure contains the following elements:

  Flags
  : Command flags

    The following values are possible and can be combined:

    `G_ETHERNET__NODE__VLAN__CONFIG__CMD_FLAG__NONE`
    : No flag is set

  VlanId
  : ID of the Virtual Local Area Network

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Node_Vlan_Delete

G_Ethernet_Node_Vlan_Delete — Delete a configured Virtual Local Area Network

Description

Use this command to delete a configured Virtual Local Area Network.

Definition

```
G_Error_t
G_Ethernet_Node_Vlan_Delete(
 const G_PortHandle_t portHandle,
 const G_Ethernet_Node_Vlan_Delete_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

cmd
Command parameters

The structure contains the following elements:

Flags
Command flags

The following values are possible and can be combined:

`G_ETHERNET__NODE__VLAN__DELETE__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__NODE__VLAN__DELETE__CMD_FLAG__DELETE_ALL`
Delete all configured VLANs

VlanId
ID of VLAN to be deleted (disregarded if flag `G_ETHERNET__NODE__VLAN__DELETE__CMD_FLAG__DELETE_ALL` is set)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Node_MacAddress_Get

G_Ethernet_Node_MacAddress_Get — Query the MAC-Address

Description

Use this command to query the MAC-Address of an interface.

Definition

```
G_Error_t
G_Ethernet_Node_MacAddress_Get(
  const G_PortHandle_t portHandle,
  const G_Ethernet_Node_MacAddress_Get_Cmd_t * const cmd,
  G_Ethernet_Node_MacAddress_Get_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

cmd
Command parameters

The structure contains the following elements:

Flags
Command flags

The following values are possible and can be combined:

`G_ETHERNET__NODE__MAC_ADDRESS__GET__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__NODE__MAC_ADDRESS__GET__CMD_FLAG__USE_VLAN_ID`
Uses VLAN ID defined by `VlanId` to get the mac address of the corresponding VLAN interface. Requires the VLAN interface to exist.

VlanId
Id for a **V**irtual **L**ocal **A**rea **N**etwork

Used if flag `G_ETHERNET__NODE__MAC_ADDRESS__GET__CMD_FLAG__USE_VLAN_ID` is set.

`rsp`
Returns response parameters

The structure contains the following elements:

`MacAddress`
MAC-Address struct which contains the 6 Byte MAC

Flags
Command flags

The following values are possible and can be combined:

`G_ETHERNET__NODE__MAC_ADDRESS__GET__RSP_FLAG__NONE`
No flag is set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Node_MacAddress_Set

G_Ethernet_Node_MacAddress_Set — Set the MAC-Address

Description

Use this command to Set the MAC-Address of an interface.

Definition

```
G_Error_t
G_Ethernet_Node_MacAddress_Set(
  const G_PortHandle_t portHandle,
  const G_Ethernet_Node_MacAddress_Set_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

cmd
Command parameters

The structure contains the following elements:

Flags
Command flags

The following values are possible and can be combined:

`G_ETHERNET__NODE__MAC_ADDRESS__SET__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__NODE__MAC_ADDRESS__SET__CMD_FLAG__USE_VLAN_ID`
Uses VLAN ID defined by `VlanId` to change the mac address of the corresponding VLAN interface. Requires the VLAN interface to exist.

`MacAddress`
MAC-Address to be set

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

VlanId
Id for a **V**irtual **L**ocal **A**rea **N**etwork

Used if flag `G_ETHERNET__NODE__MAC_ADDRESS__SET__CMD_FLAG__USE_VLAN_ID` is set.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Node_DataRate_Get

G_Ethernet_Node_DataRate_Get — Query the node data rate

Description

Use this command to query the node data rate.

Definition

```
G_Error_t
G_Ethernet_Node_DataRate_Get(
 const G_PortHandle_t portHandle,
 u32_t * const dataRate
);
```

Parameters

`portHandle`
: Handle to the communication port

`dataRate`
: Returns node data rate, in Mbps (e.g. 10, 100, 1000, …)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Node_DataRate_Set

G_Ethernet_Node_DataRate_Set — Set the node data rate

Description

Use this command to set the node data rate.

Definition

```
G_Error_t
G_Ethernet_Node_DataRate_Set(
 const G_PortHandle_t portHandle,
 const u32_t  dataRate
);
```

Parameters

`portHandle`
: Handle to the communication port

`dataRate`
: Node data rate to be set, in Mbps (e.g. 10, 100, 1000, …)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

5 PHY

Functions for Ethernet PHY configuration

G_Ethernet_Phy_LinkState_Get

G_Ethernet_Phy_LinkState_Get — Query PHY link state

Description

Use this command to query the link state of the PHY.

Definition

```
G_Error_t
G_Ethernet_Phy_LinkState_Get(
 const G_PortHandle_t  portHandle,
 u8_t * const linkState  // use type "G_Ethernet_Phy_LinkState_t"
);
```

Parameters

`portHandle`
Handle to the communication port

`linkState`
Returns link state

The following values are possible:

`G_ETHERNET__PHY__LINK_STATE__UNKNOWN`
Unknown link state

`G_ETHERNET__PHY__LINK_STATE__NOT_LINKED`
Not linked

`G_ETHERNET__PHY__LINK_STATE__LINKED`
Linked

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Phy_TransceiverType_Get

G_Ethernet_Phy_TransceiverType_Get — Query transceiver type

Description

Use this command to query the transceiver type.

Definition

```
G_Error_t
G_Ethernet_Phy_TransceiverType_Get(
 const G_PortHandle_t  portHandle,
 u8_t * const transceiverType  // use type
 "G_Ethernet_Phy_TransceiverType_t"
);
```

Parameters

`portHandle`
Handle to the communication port

`transceiverType`
Returns transceiver type

The following values are possible:

`G_ETHERNET__PHY__TRANSCEIVER_TYPE__UNKNOWN`
Unknown transceiver type

`G_ETHERNET__PHY__TRANSCEIVER_TYPE__1000_BASE_T`
Transceiver type **1000 BASE T**

`G_ETHERNET__PHY__TRANSCEIVER_TYPE__BROAD_R_REACH__TJA1100`
Transceiver type **BroadR-Reach TJA1100**

`G_ETHERNET__PHY__TRANSCEIVER_TYPE__AETH_88Q2112`
Transceiver type **AETH 88Q2112**

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Phy_LinkQuality_Get

G_Ethernet_Phy_LinkQuality_Get — Query link quality

Description

Use this command to query the link quality.

Definition

```
G_Error_t
G_Ethernet_Phy_LinkQuality_Get(
  const G_PortHandle_t  portHandle,
  u32_t * const linkQuality
);
```

Parameters

`portHandle`
: Handle to the communication port

`linkQuality`
: Returns the link quality (0..100) in percent, 0xffffffff if link quality could not be detected.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Phy_Master_Slave_Mode_Get

G_Ethernet_Phy_Master_Slave_Mode_Get — Query the operation mode

Description

Use this command to query the operation mode.

Definition

```
G_Error_t
G_Ethernet_Phy_Master_Slave_Mode_Get(
  const G_PortHandle_t  portHandle,
  u8_t * const operationMode
);
```

Parameters

`portHandle`
Handle to the communication port

`operationMode`
Returns the operation mode.

The following values are possible:

`G_ETHERNET__PHY__MASTER_SLAVE_MODE__MASTER`
The operation mode is set to **master**.

`G_ETHERNET__PHY__MASTER_SLAVE_MODE__SLAVE`
The operation mode is set to **slave**.

`G_ETHERNET__PHY__MASTER_SLAVE_MODE__UNKNOWN`
The operation mode is set to an unknown state.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Phy_Master_Slave_Mode_Set

G_Ethernet_Phy_Master_Slave_Mode_Set — Set the operation mode

Description

Use this command to set the operation mode.

Definition

```
G_Error_t
G_Ethernet_Phy_Master_Slave_Mode_Set(
  const G_PortHandle_t  portHandle,
  const u8_t  operationMode // use type G_Ethernet_Phy_OperationMode_t
);
```

Parameters

`portHandle`
Handle to the communication port

`operationMode`
Operation mode to be set.

The following values are possible:

`G_ETHERNET__PHY__MASTER_SLAVE_MODE__USE_HW_SWITCH`
Sets the operation mode of the transceiver according to the hardware switch on the module.

`G_ETHERNET__PHY__MASTER_SLAVE_MODE__MASTER`
Sets the operation mode of the transceiver to **master**.

`G_ETHERNET__PHY__MASTER_SLAVE_MODE__SLAVE`
Sets the operation mode of the transceiver to **slave**.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Phy_TxAmplitude_Get

G_Ethernet_Phy_TxAmplitude_Get — Query the tx amplitude

Description

Use this command to query the tx apmlitude.

Definition

```
G_Error_t
G_Ethernet_Phy_TxAmplitude_Get(
  const G_PortHandle_t  portHandle,
  u32_t *  const txAmplidude
);
```

Parameters

`portHandle`
: Handle to the communication port

`txAmplidude`
: Returns the tx amplidude in mV.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Phy_TxAmplitude_Set

G_Ethernet_Phy_TxAmplitude_Set — Set the Tx Amplitude

Description

Use this command to set the operation mode.

Definition

```
G_Error_t
G_Ethernet_Phy_TxAmplitude_Set(
  const G_PortHandle_t  portHandle,
  const u32_t  txAmplidude
);
```

Parameters

`portHandle`
: Handle to the communication port

`txAmplidude`
: tx amplidude in mV to be set.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Phy_DataRate_Get

G_Ethernet_Phy_DataRate_Get — Query the data rate.

Description

Use this command to query the data rate.

Definition

```
G_Error_t
G_Ethernet_Phy_DataRate_Get(
  const G_PortHandle_t  portHandle,
  u32_t * const dataRate
);
```

Parameters

`portHandle`
Handle to the communication port

`dataRate`
Returns the data rate in Mbps (for example 10, 100, 1000).

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Phy_DataRate_Set

G_Ethernet_Phy_DataRate_Set — Set the data rate

Description

Use this command to set the data rate.

Definition

```
G_Error_t
G_Ethernet_Phy_DataRate_Set(
  const G_PortHandle_t  portHandle,
  const u32_t  dataRate
);
```

Parameters

`portHandle`
: Handle to the communication port

`dataRate`
: data rate in Mbps (for example 10, 100, 1000) to be set.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Phy_Temperature_Get

G_Ethernet_Phy_Temperature_Get — Query the temperature

Description

Use this command to get the temperature of the phy in celsius.

Definition

```
G_Error_t
G_Ethernet_Phy_Temperature_Get(
 const G_PortHandle_t  portHandle,
 u32_t * const Temperature
);
```

Parameters

`portHandle`
    Handle to the communication port

`temperature`
    temperature of the phy in celsius

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Phy_LegacyMode_Get

G_Ethernet_Phy_LegacyMode_Get — Query the legacy mode

Description

Use this command to get the current legacy mode of the phy, if this feature is supported by the phy.

Definition

```
G_Error_t
G_Ethernet_Phy_LegacyMode_Get(
 const G_PortHandle_t  portHandle,
 u32_t * const LegacyMode
);
```

Parameters

`portHandle`
: Handle to the communication port

`LegacyMode`
: Legacy mode of the phy. The response depends on the used phy.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Phy_LegacyMode_Set

G_Ethernet_Phy_LegacyMode_Set — Set the legacy mode

Description

Use this command to set the legacy mode of the phy, if this feature is supported by the phy.

Definition

```
G_Error_t
G_Ethernet_Phy_LegacyMode_Set(
 const G_PortHandle_t  portHandle,
 const u32_t LegacyMode
);
```

Parameters

`portHandle`
: Handle to the communication port

`LegacyMode`
: Legacy mode of the phy. This value depends on the used phy.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Phy_HwRevision_Get

G_Ethernet_Phy_HwRevision_Get — Query the hardware revision

Description

Use this command to get the 4 Bytes of the Hardware Revision of the Ethernet Transceiver.

Definition

```
G_Error_t
G_API
G_Ethernet_Phy_HwRevision_Get(
 const G_PortHandle_t  portHandle,
 u32_t * const HwRevision
)
```

Parameters

`portHandle`
: Handle to the communication port

`HwRevision`
: 4 Bytes of the Hardware Revision of the Ethernet Transceiver

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

6 Ping

Ping functions of the Ethernet interface

G_Ethernet_Ping_Start

G_Ethernet_Ping_Start — Send PING command

Description

Use this command to send a PING command.

Definition

```
G_Error_t
G_Ethernet_Ping_Start(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_Ping_Start_Cmd_t * const cmd,
 G_Ethernet_Ping_Start_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__PING__START__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__PING__START__CMD_FLAG__USE_VLAN_ID`
Use parameter `VlanId` for specifying a VLAN.

A VLAN needs to be configured with G_Ethernet_Node_Vlan_Config .

`G_ETHERNET__PING__START__CMD_FLAG__DO_NOT_WAIT`
Do not wait for a response to the PING command.

If this flag is set, the command call will return after the PING has been sent. The PING state can be queried with G_Ethernet_Ping_GetState .

`Protocol`
Ethernet protocol

The following values are possible:

`G_ETHERNET__PING__PROTOCOL__ICMP4`
ICMP 4 protocol

`G_ETHERNET__PING__PROTOCOL__ICMP6`
ICMP 6 protocol

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`PayloadLength`
Length of payload for PING command, in bytes

`Timeout`
Timeout for PING command, in bytes

`VlanId`
VLAN ID

Used if flag `G_ETHERNET__PING__START__CMD_FLAG__USE_VLAN_ID` is set.

A VLAN needs to be configured with G_Ethernet_Node_Vlan_Config .

`RemoteAddress`
Remote address, in network byte order

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible:

`G_ETHERNET__PING__START__RSP_FLAG__NONE`
No flag is set

`State`
PING state

The following values are possible: G_Ethernet_Ping_State_t;

`reserved1`
reserved parameter

`reserved2`
reserved parameter

`reserved3`
reserved parameter

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Ping_GetState

G_Ethernet_Ping_GetState — Query state of PING command

Description

Use this command to query the state of the PING command.

Definition

```
G_Error_t
G_Ethernet_Ping_GetState(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_Ping_GetState_Cmd_t * const cmd,
 G_Ethernet_Ping_GetState_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_ETHERNET__PING__GET_STATE__CMD_FLAG__NONE`
No flag is set

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible:

`G_ETHERNET__PING__GET_STATE__RSP_FLAG__NONE`
No flag is set

`State`
PING state

The following values are possible: G_Ethernet_Ping_State_t;

`reserved1`
reserved parameter

`reserved2`
reserved parameter

`reserved3`
reserved parameter

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

7 Packet Generator

The Packet Generator can be used to generate Ethernet Frames and simulate errors in the transmission path.

See Section 2, " MII Multiplexer " for commands used to route the frames to various targets.

G_Ethernet_PktGen_Reset

G_Ethernet_PktGen_Reset — Reset Packet Generator configuration

Description

Use this command to reset the Packet Generator configuration.

Definition

```
G_Error_t
G_Ethernet_PktGen_Reset(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_PktGen_Config

G_Ethernet_PktGen_Config — Packet Generator configuration

Description

Use this command to configure the Packet Generator.

Definition

```
G_Error_t
G_Ethernet_PktGen_Config(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_PktGen_Config_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__PKT_GEN__CONFIG__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__PKT_GEN__CONFIG__CMD_FLAG__ADD_PREAMBLE_AND_SFD`
0 : do not insert preamble and start frame delimiter

1 : insert preamble and start frame delimiter before data

Preamble and start frame delimiter are 8 octets in front of an ethernet packet.

`G_ETHERNET__PKT_GEN__CONFIG__CMD_FLAG__APPEND_FCS`
0 : do not append frame check sequence

1 : calculate and append frame check sequence

FCS is the frame check sequence (4 octets) at the end of an ethernet packet.

`G_ETHERNET__PKT_GEN__CONFIG__CMD_FLAG__START`
0 : don't start packet generator (configuration only)

1 : start packet generator (configuration and start)

`G_ETHERNET__PKT_GEN__CONFIG__CMD_FLAG__WRONG_FCS`
0 : generate correct frame check sequence

1 : generate wrong frame check sequence

`G_ETHERNET__PKT_GEN__CONFIG__CMD_FLAG__CONTINUOUS_RUN`
0 : generate `NumberOfPackets` packets

1 : continuously generate packets (`NumberOfPackets` is ignored)

`Mode`
Packet Generator mode

The following values are possible:

`G_ETHERNET__PKT_GEN__MODE__1_GBPS`
Gigabit Ethernet

`G_ETHERNET__PKT_GEN__MODE__10_MBPS`
10 Mbps

`G_ETHERNET__PKT_GEN__MODE__100_MBPS`
100 Mbps

`reserved`
reserved parameter (must be initialized with **0**)

`Length`
Frame length

An Ethernet frame (exclusive preamble and start frame delimiter, inclusive frame check sequence) has a minimum length of 64 bytes.

`Pause`
Time between ethernet packets, in nano seconds

`NumberOfPackets`
Number of packets to be sent

`Stream`
Contains frame data ("`u8_t Data[Length]`")

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_PktGen_GetState

G_Ethernet_PktGen_GetState — Query packet Generator state

Description

Use this command to query the state of the Packet Generator.

Definition

```
G_Error_t
G_Ethernet_PktGen_GetState(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_PktGen_GetState_Cmd_t * const cmd,
 G_Ethernet_PktGen_GetState_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_ETHERNET__PKT_GEN__GET_STATE__CMD_FLAG__NONE`
No flag is set

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_ETHERNET__PKT_GEN__GET_STATE__RSP_FLAG__NONE`
No flag is set

`G_ETHERNET__PKT_GEN__GET_STATE__RSP_FLAG__CONFIGURED`
0 : Packet Generator is not configured

1 : Packet Generator is configured

`G_ETHERNET__PKT_GEN__GET_STATE__RSP_FLAG__RUNNING`
0 : Packet Generator is not running

1 : Packet Generator is running

`G_ETHERNET__PKT_GEN__GET_STATE__RSP_FLAG__FINISHED`
0 : packet generation is not finished

1 : packet generation is finished

`G_ETHERNET__PKT_GEN__GET_STATE__RSP_FLAG__ADD_PREAMBLE_AND_SFD`
0 : do not insert preamble and start frame delimiter

1 : insert preamble and start frame delimiter before data

Preamble and start frame delimiter are 8 octets in front of an ethernet packet.

`G_ETHERNET__PKT_GEN__GET_STATE__RSP_FLAG__APPEND_FCS`
0 : do not append frame check sequence

1 : calculate and append frame check sequence

FCS is the frame check sequence (4 octets) at the end of an ethernet packet.

`G_ETHERNET__PKT_GEN__GET_STATE__RSP_FLAG__WRONG_FCS`
0 : generate correct frame check sequence

1 : generate wrong frame check sequence

`G_ETHERNET__PKT_GEN__GET_STATE__RSP_FLAG__CONTINUOUS_RUN`
0 : continuous run disabled

1 : continuous run enabled

`Mode`
Packet Generator mode

The following values are possible:

`G_ETHERNET__PKT_GEN__MODE__1_GBPS`
Gigabit Ethernet

`G_ETHERNET__PKT_GEN__MODE__10_MBPS`
10 Mbps

`G_ETHERNET__PKT_GEN__MODE__100_MBPS`
100 Mbps

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Pause`
Time between ethernet packets, in nano seconds

`PacketLength`
Packet length in bytes

`NumberOfPackets`
Number of packets to be sent

`PacketCounter`
Packet counter (0..`NumberOfPackets`)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

8 Packet Filter

The Packet Filter is a hardware filter, that filters incoming packets before they arrive at the MAC-Controller. It is used to reduce load on incoming traffic. This feature is useful if just certain packets should be monitored or certain packages should not be received.

G_Ethernet_PktFilter_Set

G_Ethernet_PktFilter_Set — Set Packet Filter

Description

Use this command to set the Packet Filter.

Definition

```
G_Error_t
G_Ethernet_PktFilter_Set(
 const G_PortHandle_t  portHandle,
  const G_Ethernet_PktFilter_Filter_Set_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__PKT_FLT__FLT__SET__CMD__FLAG__NONE`
No flag is set

`G_ETHERNET__PKT_FLT__FLT__SET__CMD__FLAG__NEGATIVE_FILTER`
Filter is inverted: The filter does block all packets that suit the given criteria instead of letting them pass

`G_ETHERNET__PKT_FLT__FLT__SET__CMD__FLAG__FLT_BY_VLAN_ID`
Use this flag to filter packets by VLAN-Tag. Uses the given `VlanId`.

`G_ETHERNET__PKT_FLT__FLT__SET__CMD__FLAG__FLT_BY_ETHER_TYPE`
Use this flag to filter packets by EtherType. Uses the given `EtherType`.

`G_ETHERNET__PKT_FLT__FLT__SET__CMD__FLAG__FLT_BY_NEXT_HEADER`
Use this flag to filter packets by NextHeader-Field. Uses the given `NextHeader`.

`G_ETHERNET__PKT_FLT__FLT__SET__CMD__FLAG__FLT_BY_SRC_IP`
Use this flag to filter packets by Source-Address. Uses the given `SourceIpAddresses`.

`G_ETHERNET__PKT_FLT__FLT__SET__CMD__FLAG__FLT_BY_DST_IP`
Use this flag to filter packets by Destination-Address. Uses the given `DestinationIpAddresses`.

`G_ETHERNET__PKT_FLT__FLT__SET__CMD__FLAG__FLT_BY_SRC_PORT`
Use this flag to filter packets by Source-Port. Uses the given `SourcePort`.

`G_ETHERNET__PKT_FLT__FLT__SET__CMD__FLAG__FLT_BY_DST_PORT`
Use this flag to filter packets by Destination-Port. Uses the given `DestinationPort`.

G_ETHERNET__PKT_FLT__FLT__SET__CMD__FLAG__IPV6_ADDRESS
Required if the source and/or destination ip address are IPv6 addresses

G_ETHERNET__PKT_FLT__FLT__SET__CMD__FLAG__LINK_CONDITIONS_BY_AND
Links the filter conditions of this filter instance by logical "and" instead of "or"

`FilterInstanceId`
Id of the filter instance.

The following values are possible: 0..9

`reserved1`
reserved parameter (must be initialized with **0**)

`VlanId`
value for the VLAN tag filter

`EtherType`
value for the ethertype-filter

`NextHeader`
value for the next header filter

`reserved2`
reserved parameter (must be initialized with **0**)

`SourceIpAddresses`
contains the source ip address filter ("`u8_t Data[16]`")

`DestinationIpAddresses`
contains the destination ip address filter ("`u8_t Data[16]`")

`SourcePort`
value for the source port filter

`DestinationPort`
value for the destination port filter

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_PktFilter_Reset

G_Ethernet_PktFilter_Reset — Reset the Packet Filter

Description

Use this command to reset the Packet Filter.

Definition

```
G_Error_t
G_Ethernet_PktFilter_Reset(
  const G_PortHandle_t  portHandle,
  const G_Ethernet_PktFilter_Filter_Reset_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__PKT_FLT__FLT__RESET__CMD__FLAG_NONE`
No flag is set

`G_ETHERNET__PKT_FLT__FLT__RESET__CMD__FLAG__RESET_ALL`
Resets all packet filter instances on this interface.

`FilterInstanceId`
Id of the filter instance.

The following values are possible: 0..9

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

9 RxFifo

These commands are used to control and configure the **RX FIFO** functionality.

A code example for using the RX FIFO can be found in the samples directory under *Ethernet_TxFifo_RxFifo*.

G_Ethernet_RxFifo_Config

G_Ethernet_RxFifo_Config — RX Fifo configuration

Description

Use this command to configure the RX Fifo.

A code example for using the RX FIFO can be found in the samples directory under *Ethernet_TxFifo_RxFifo*.

Definition

```
G_Error_t
G_Ethernet_RxFifo_Config(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_RxFifo_Config_Cmd_t * const cmd,
 G_Ethernet_RxFifo_Config_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__RX_FIFO__CONFIG__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__RX_FIFO__CONFIG__CMD_FLAG__GENERATE_RX_FIFO_ID`
An RX Fifo ID is generated automatically, parameter `RxFifoId` is disregarded.

`RxFifoId`
ID of the RX Fifo to be configured

Disregarded if flag `G_ETHERNET__RX_FIFO__CONFIG__CMD_FLAG__GENERATE_RX_FIFO_ID` is set.

`CorrespondingTxFifoId`
ID of the corresponding TX Fifo

A RX Fifo needs a configured TX Fifo for Ethernet communication.

The TX Fifo can be configured with `G_Ethernet_TxFifo_Config_...` commands.

`FifoDepth`
Number of entries the FIFO can hold (starting from **1**)

`FifoDataLength`
Data length of a TX FIFO entry (starting from **1**)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`RxFifoId`
Returns RX Fifo ID

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_RxFifo_Delete

G_Ethernet_RxFifo_Delete — Delete RX Fifo

Description

Use this command to delete a RX FIFO and free all its resources.

A code example for using the RX FIFO can be found in the samples directory under *Ethernet_TxFifo_RxFifo*.

Definition

```
G_Error_t
G_Ethernet_RxFifo_Delete(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_RxFifo_Delete_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__RX_FIFO__DELETE__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__RX_FIFO__DELETE__CMD_FLAG__DELETE_ALL`
Delete all RX Fifos

`RxFifoId`
ID of the RX Fifo to be deleted (disregarded if flag `G_ETHERNET__RX_FIFO__DELETE__CMD_FLAG__DELETE_ALL` is set)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_RxFifo_Read

G_Ethernet_RxFifo_Read — Read RX Fifo entries

Description

Use this command to read RX Fifo entries.

Definition

```
G_Error_t
G_Ethernet_RxFifo_Read(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_RxFifo_Read_CmdFlags_t  cmdFlags,
 const u32_t  rxFifoId,
 u32_t * const numberOfEntries,
 u32_t * const numberOfBytes,
 u32_t * const numberOfRemainingEntries,
 u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible:

`G_ETHERNET__RX_FIFO__READ__CMD_FLAG__NONE`
No flag is set

`rxFifoId`
ID of the RX Fifo

`numberOfEntries`
in : maximum number of entries to be read (**0** = read all available entries)

out : number of read entries

`numberOfBytes`
in : maximum number of bytes to be read

out : number of bytes read

`numberOfRemainingEntries`
Number of remaining entries

`data`
Returns entry data

Use type `G_Ethernet_RxFifo_Entry_t`

The entries are aligned to multiple of 4 bytes.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

10 Tar

These commands are used to control and configure the **TAR** functionality.

With **T**ransmit **A**fter **R**eception Ethernet frames are sent automatically triggered by the reception of specific Ethernet frames.

A code example for using the TAR functionality can be found in the samples directory under *Ethernet_TxFifo_Tar*.

G_Ethernet_Tar_Config_Udp4

G_Ethernet_Tar_Config_Udp4 — Configuration of TAR for UDPv4 protocol

Description

Use this command for configuration of TAR for UDPv4 protocol.

Definition

```
G_Error_t
G_Ethernet_Tar_Config_Udp4(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_Tar_Config_Udp4_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__TAR__CONFIG__UDP4__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__TAR__CONFIG__UDP4__CMD_FLAG__SET_REMOTE_ADDRESS`
0 : use first address and port you communicate with as remote address

1 : set remote address from parameter `RemoteAddress`

`G_ETHERNET__TAR__CONFIG__UDP4__CMD_FLAG__SET_LOCAL_ADDRESS`
0 : use default address and any port

1 : set local address from parameter `LocalAddress`, no broadcast-reception possible!

`Channel`
TAR multisession channel

A multisession channel is required if with multiple ip address configurations on the same interface. For each channel, the TAR can be configured with different parameters.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`RemoteAddress`
Remote address, in network byte order

`LocalAddress`
Local address, in network byte order

`RemotePort`
Remote port

`LocalPort`
Local port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Tar_Config_Tcp4

G_Ethernet_Tar_Config_Tcp4 — Configuration of TAR for TCPv4 protocol

Description

Use this command for configuration of TAR for TCPv4 protocol.

Definition

```
G_Error_t
G_Ethernet_Tar_Config_Tcp4(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_Tar_Config_Tcp4_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__TAR__CONFIG__TCP4__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__TAR__CONFIG__TCP4__CMD_FLAG__SET_REMOTE_ADDRESS`
0 : use first address and port you communicate with as remote address

1 : set remote address from parameter `RemoteAddress`

`G_ETHERNET__TAR__CONFIG__TCP4__CMD_FLAG__SET_LOCAL_ADDRESS`
0 : use default address and any port

1 : set local address from parameter `LocalAddress`, no broadcast-reception possible!

`G_ETHERNET__TAR__CONFIG__TCP4__CMD_FLAG__IS_SERVER`
0 : act as TCP client

1 : act as TCP server

`Channel`
TAR multisession channel

A multisession channel is required if with multiple ip address configurations on the same interface. For each channel, the TAR can be configured with different parameters.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`RemoteAddress`
Remote address, in network byte order

`LocalAddress`
Local address, in network byte order

`RemotePort`
Remote port

`LocalPort`
Local port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Tar_Config_Udp6

G_Ethernet_Tar_Config_Udp6 — Configuration of TAR for UDPv6 protocol

Description

Use this command for configuration of TAR for UDPv6 protocol.

Definition

```
G_Error_t
G_Ethernet_Tar_Config_Udp6(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_Tar_Config_Udp6_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__TAR__CONFIG__UDP6__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__TAR__CONFIG__UDP6__CMD_FLAG__SET_REMOTE_ADDRESS`
0 : use first address and port you communicate with as remote address

1 : set remote address from parameter `RemoteAddress`

`G_ETHERNET__TAR__CONFIG__UDP6__CMD_FLAG__SET_LOCAL_ADDRESS`
0 : use default address and any port

1 : set local address from parameter `LocalAddress`, no broadcast-reception possible!

`G_ETHERNET__TAR__CONFIG__UDP6__CMD_FLAG__USE_VLAN_ID`
Use VLAN ID defined by `VlanId`

A **V**irtual **L**ocal **A**rea **N**etwork can be configured with G_Ethernet_Node_Vlan_Config .

`Channel`
TAR multisession channel

A multisession channel is required if with multiple ip address configurations on the same interface. For each channel, the TAR can be configured with different parameters.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`RemoteAddress`
Remote address, in network byte order

`LocalAddress`
Local address, in network byte order

`RemotePort`
Remote port

`LocalPort`
Local port

VlanId
VLAN ID

Id for a **V**irtual **L**ocal **A**rea **N**etwork

Used if flag `G_ETHERNET__TAR__CONFIG__UDP6__CMD_FLAG__USE_VLAN_ID` is set.

A VLAN has to be configured with G_Ethernet_Node_Vlan_Config first.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Tar_Config_Tcp6

G_Ethernet_Tar_Config_Tcp6 — Configuration of TAR for TCPv6 protocol

Description

Use this command for configuration of TAR for TCPv6 protocol.

Definition

```
G_Error_t
G_Ethernet_Tar_Config_Tcp6(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_Tar_Config_Tcp6_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__TAR__CONFIG__TCP6__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__TAR__CONFIG__TCP6__CMD_FLAG__SET_REMOTE_ADDRESS`
0 : use first address and port you communicate with as remote address

1 : set remote address from parameter `RemoteAddress`

`G_ETHERNET__TAR__CONFIG__TCP6__CMD_FLAG__SET_LOCAL_ADDRESS`
0 : use default address and any port

1 : set local address from parameter `LocalAddress`, no broadcast-reception possible!

`G_ETHERNET__TAR__CONFIG__TCP6__CMD_FLAG__IS_SERVER`
0 : act as TCP client

1 : act as TCP server

`G_ETHERNET__TAR__CONFIG__TCP6__CMD_FLAG__USE_VLAN_ID`
Use VLAN ID defined by `VlanId`

A **V**irtual **L**ocal **A**rea **N**etwork can be configured with G_Ethernet_Node_Vlan_Config .

`Channel`
TAR multisession channel

A multisession channel is required if with multiple ip address configurations on the same interface. For each channel, the TAR can be configured with different parameters.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`RemoteAddress`
Remote address, in network byte order

`LocalAddress`
Local address, in network byte order

`RemotePort`
Remote port

`LocalPort`
Local port

VlanId
VLAN ID

Id for a **V**irtual **L**ocal **A**rea **N**etwork

Used if flag `G_ETHERNET__TAR__CONFIG__TCP6__CMD_FLAG__USE_VLAN_ID` is set.

A VLAN has to be configured with G_Ethernet_Node_Vlan_Config first.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Tar_Config_Off

G_Ethernet_Tar_Config_Off — Reset TAR configuration

Description

Use this command to reset the TAR configuration.

Definition

```
G_Error_t
G_Ethernet_Tar_Config_Off(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_Tar_Config_Off_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following flags are possible and can be combined:

`G_ETHERNET__TAR__CONFIG__OFF__CMD_FLAG__NONE`
No flag is set

`Channel`
TAR multisession channel

A multisession channel is required if with multiple ip address configurations on the same interface. For each channel, the TAR can be configured with different parameters.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Tar_AddItems

G_Ethernet_Tar_AddItems — Add TAR items

Description

Use this command to add TAR items.

Definition

```
G_Error_t
G_Ethernet_Tar_AddItems(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const u32_t  numberOfItems,
 const G_Ethernet_Tar_Item_t * const items
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
TAR multisession channel

A multisession channel is required if with multiple ip address configurations on the same interface. For each channel, the TAR can be configured with different parameters.

`numberOfItems`
Number of TAR items to be added

`items`
TAR items to be added

A TAR item contains the following elements:

`TxFlags`
TX item flags

The following values are possible and can be combined:

`G_ETHERNET__TAR__TX_ITEM__FLAG__NONE`
No flag is set

`G_ETHERNET__TAR__TX_ITEM__FLAG__SEND_NEXT`
Send next item immediately without waiting for trigger message

`G_ETHERNET__TAR__TX_ITEM__FLAG__RESTART`
After this item has been sent, the queue restarts from the 1st item

`G_ETHERNET__TAR__TX_ITEM__FLAG__BREAK`
The trigger condition is ignored `Delay` times (nothing is sent)

`RxFlags`
RX item flags

The following flags are possible and can be combined:

`G_ETHERNET__TAR__RX_ITEM__FLAG__NONE`
No flag is set

`G_ETHERNET__TAR__RX_ITEM__FLAG__CHECK_RX_DATA`
Only send TAR item if received data matches `RxData`

`Delay`
Delay for sending TAR item, in milliseconds

If flag `G_ETHERNET__TAR__TX_ITEM__FLAG__BREAK` is set, this parameters defines the number of times the transmission is suspended.

`TxLength`
Length of TX data `TxData`, in bytes

`RxLength`
Length of RX data `RxData`, in bytes

Only used if flag `G_ETHERNET__TAR__RX_ITEM__FLAG__CHECK_RX_DATA` is set.

When using UDP and flag `G_ETHERNET__TAR__RX_ITEM__FLAG__CHECK_RX_DATA` is not set, `RxLength` needs to be **`0`**.

`TxData`
Pointer to TX Data of TAR item that will be sent if the trigger condition is met

`RxData`
Pointer to RX Data of item that triggers the transmission

Only used if flag `G_ETHERNET__TAR__RX_ITEM__FLAG__CHECK_RX_DATA` is set.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Tar_Start

G_Ethernet_Tar_Start — Start TAR

Description

Use this command to start TAR.

Definition

```
G_Error_t
G_Ethernet_Tar_Start(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_Tar_Start_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Channel`
TAR multisession channel

A multisession channel is required if with multiple ip address configurations on the same interface. For each channel, the TAR can be configured with different parameters.

`Mode`
TAR start mode

The following values are possible:

`G_ETHERNET__TAR__START__MODE__RESTART`
Restart TAR

TAR always starts with item number **1**.

`G_ETHERNET__TAR__START__MODE__RESUME`
Resume TAR

TAR resumes operation after it had been stopped

If TAR has not been running before, it starts with item number **1**.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Tar_Stop

G_Ethernet_Tar_Stop — Stop TAR

Description

Use this command to stop TAR.

Definition

```
G_Error_t
G_Ethernet_Tar_Stop(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_Tar_Stop_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Channel`
TAR multisession channel

A multisession channel is required if with multiple ip address configurations on the same interface. For each channel, the TAR can be configured with different parameters.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_Tar_GetState

G_Ethernet_Tar_GetState — Query TAR state

Description

Use this command to query the TAR state.

Definition

```
G_Error_t
G_Ethernet_Tar_GetState(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_Tar_GetState_Cmd_t * const cmd,
 G_Ethernet_Tar_GetState_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Channel`
TAR multisession channel

A multisession channel is required if with multiple ip address configurations on the same interface. For each channel, the TAR can be configured with different parameters.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`Channel`
TAR multisession channel

A multisession channel is required if with multiple ip address configurations on the same interface. For each channel, the TAR can be configured with different parameters.

`Flags`
Response flags

The following values are possible and can be combined:

G_ETHERNET__TAR__GET_STATE__RSP_FLAG__NONE
No flag is set

G_ETHERNET__TAR__GET_STATE__RSP_FLAG__STARTED
TAR has been started

Type
TAR type

The following values are possible:

G_ETHERNET__TAR__TYPE__OFF
TAR is not configured

G_ETHERNET__TAR__TYPE__UDP4
TAR is configured for UDPv4 operation

G_ETHERNET__TAR__TYPE__TCP4
TAR is configured for TCPv4 operation

G_ETHERNET__TAR__TYPE__UDP6
TAR is configured for UDPv6 operation

G_ETHERNET__TAR__TYPE__TCP6
TAR is configured for TCPv6 operation

reserved
reserved parameter (must be initialized with **0**)

NumberOfTxItems
Number of TX items that are configured

CurrentTxItem
Index of TX item that will be sent next (starting with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

11 TP - Transport Protocol

Transport protocol functions of the Ethernet interface

G_Ethernet_Tp_Init_Iso13400

G_Ethernet_Tp_Init_Iso13400 — Initialize ISO 13400 transport protocol

Description

Use this command to initialize the ISO 13400 transport protocol on the Ethernet interface.

Definition

```
G_Error_t
G_Ethernet_Tp_Init_Iso13400(
 const G_PortHandle_t portHandle,
 const u8_t  channel,
 const G_Ethernet_Tp_Init_Iso13400_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Tp multisession channel

A multisession channel is required if multiple applications are working with the transport protocol on the same interface. For each channel, the transport protocol can be configured with different parameters.

Usually, it is not neccessary to use multisession channels. It is recommended to use channel `0` in this case.

A multisession channel can be obtained by G_Tp_Channel_Get .

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__TP__INIT__ISO13400__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__TP__INIT__ISO13400__CMD_FLAG__DISABLE_TX_SIDE__PHYSICAL`
not set : enable physical sender side

set : disable physical sender side

`G_ETHERNET__TP__INIT__ISO13400__CMD_FLAG__DISABLE_RX_SIDE__PHYSICAL`
not set : enable physical receiver side

set : disable physical receiver side

G_ETHERNET__TP__INIT__ISO13400__CMD_FLAG__DISABLE_TX_SIDE__FUNCTIONAL
not set : enable functional sender side

set : disable functional sender side

G_ETHERNET__TP__INIT__ISO13400__CMD_FLAG__DISABLE_RX_SIDE__FUNCTIONAL
not set : enable functional receiver side

set : disable functional receiver side

G_ETHERNET__TP__INIT__ISO13400__CMD_FLAG__DETECT_PROTOCOL_VERSION
not set : do not detect protocol version

set : detect protocol version

G_ETHERNET__TP__INIT__ISO13400__CMD_FLAG__USE_ROUTING_ACTIVATION_REQ_OEM
not set : send routing activation request without OEM specific data

set : send routing activation request with OEM specific data `RoutingActivationReqOem`

G_ETHERNET__TP__INIT__ISO13400__CMD_FLAG__DIAG_WITH_DOIP_ENTITY_ITSELF
not set : physical diagnosis requests are sent to `RemoteAddress_Physical`

set : physical diagnosis requests are sent to the remote **DoIP** entity itself, parameter `RemoteAddress_Physical` is ignored

G_ETHERNET__TP__INIT__ISO13400__CMD_FLAG__CHECK_DOIP_ENTITY_ADDRESS
not set : the remote logical **DoIP** entity address is determined by a successfully received **DoIP** "vehicle identification response" or "routing activation response"

set : compare received **DoIP** entity address with configured remote **DoIP** entity address `RemoteAddress_DoIpEntity`

G_ETHERNET__TP__INIT__ISO13400__CMD_FLAG__CHECK_EID
not set : the received **EID** is not checked

set : compare received **EID** with configured `Eid`

G_ETHERNET__TP__INIT__ISO13400__CMD_FLAG__CHECK_VIN
not set : the received **VIN** is not checked

set : compare received **VIN** with configured `Vin`

G_ETHERNET__TP__INIT__ISO13400__CMD_FLAG__FIRST_UDP_RESPONSE_ONLY
not set : collect **UDP** responses on an **UDP** broadcast request (wait until `UdpRxTimeout` elapses on an **UDP** broadcast request or a matching response was received)

set : the first received **UDP** response on an **UDP** broadcast request is used as response (waiting for further **UDP** responses is stopped). Setting this flag speeds up connecting with remote **DoIP** entity if **DoIP** protocol version detection is enabled and the remote IP address is not set.

`ProtocolVersion`
Protocol version

1 = ISO/DIS 13400-2:2010

2 = ISO 13400-2:2012

`VehIdentReqMode`
Vehicle Identification Request Mode

The following values are possible:

`G_ETHERNET__TP__ISO13400__VEH_IDENT_REQ_MODE__NORMAL`
Vehicle identification request message (**DoIP** payload type = **0x0001**)

`G_ETHERNET__TP__ISO13400__VEH_IDENT_REQ_MODE__WITH_EID`
Vehicle identification request message with **EID** (with entity identification, **DoIP** payload type = **0x0002**)

`G_ETHERNET__TP__ISO13400__VEH_IDENT_REQ_MODE__WITH_VIN`
Vehicle identification request message with **VIN** (vehicle identification number, **DoIP** payload type = **0x0003**)

`G_ETHERNET__TP__INIT__ISO13400__CMD_FLAG__USE_VLAN_ID`
Use VLAN ID

A **V**irtual **L**ocal **A**rea **N**etwork can be configured with G_Ethernet_Node_Vlan_Config .

`reserved1`
reserved parameter (must be initialized with **0**)

`LocalAddress`
Our logical address (external test equipment)

`RemoteAddress_DoIpEntity`
Logical address of remote **DoIP** entity, used if flag `G_ETHERNET__TP__INIT__ISO13400__CMD_FLAG__CHECK_DOIP_ENTITY_ADDRESS` is set

`RemoteAddress_Physical`
Remote address for diagnostics messages, physical addressing

`RemoteAddress_Functional`
Remote address for diagnostics messages, functional addressing

`TimeoutSendVehIdentReq`
Timeout for sending a "Vehicle Identification Request" message [microseconds]

`TimeoutWaitVehIdentRes`
Timeout for waiting on a "Vehicle Identification Response" message [microseconds]

`TimeoutTcpConnect`
Timeout for TCP connect [microseconds]

`TimeoutSendRoutingActivationReq`
Timeout for sending a "Routing Activation Request" message [microseconds]

`TimeoutWaitRoutingActivationRes`
Timeout for waiting on a "Routing Activation Response" message [microseconds]

`TimeoutSendDiagMsg`
Timeout for sending a "diagnostics messsage" [microseconds]

`TimeoutWaitDiagMsgAck`
Timeout for waiting on a "Diagnostic Message Positive Acknowledgement" or "Diagnostic Message Negative Acknowledgement" message. [microseconds]

`DiagMsgRepetitions`
Number of repetitions of sending a "Diagnostic Message" after `TimeoutWaitDiagMsgAck` has been elapsed

`VehIdentReqRepetitions`
Number of repetitions of sending a "Vehicle Identification Request" after `TimeoutSendVehIdentReq` has been elapsed

`LocalIpAddressType`
Type of local IP Address

The following values are possible:

`G_ETHERNET__IP_ADDRESS_TYPE__ANY_IP_ADDRESS`
Any IP address

`G_ETHERNET__IP_ADDRESS_TYPE__IP4_ADDRESS`
IPv4 address

`G_ETHERNET__IP_ADDRESS_TYPE__IP6_ADDRESS`
IPv6 address

`G_ETHERNET__IP_ADDRESS_TYPE__ANY_IP4_ADDRESS`
Any IPv4 address

`G_ETHERNET__IP_ADDRESS_TYPE__ANY_IP6_ADDRESS`
Any IPv6 address

`G_ETHERNET__IP_ADDRESS_TYPE__ANY_LINK_LOCAL_IP4_ADDRESS`
Any link local IPv4 address

`LocalIpAddress`
Local IP address (in network byte order)

`RemoteIpAddress`
Remote IP address (in network byte order)

`RoutingActivationReqOem`
Array with 4 bytes for OEM specific data for routing activation request. Used if flag `G_ETHERNET__TP__INIT__ISO13400__CMD_FLAG__USE_ROUTING_ACTIVATION_REQ_OEM` is set.

`Eid`
Array with 6 bytes for entity identification, used if flag `G_ETHERNET__TP__INIT__ISO13400__CMD_FLAG__CHECK_EID` is set or `VehIdentReqMode` is set to `G_ETHERNET__TP__ISO13400__VEH_IDENT_REQ_MODE__WITH_EID`

`reserved3`
reserved parameter (must be initialized with **0**)

`reserved4`
reserved parameter (must be initialized with **0**)

`Vin`
Array with 17 bytes for vehicle identification number (**VIN**) as specified in **ISO 3779**, used if flag `G_ETHERNET__TP__INIT__ISO13400__CMD_FLAG__CHECK_VIN` is set or `VehIdentReqMode` is set to `G_ETHERNET__TP__ISO13400__VEH_IDENT_REQ_MODE__WITH_VIN`

`reserved5`
reserved parameter (must be initialized with **0**)

`reserved6`
reserved parameter (must be initialized with **0**)

`reserved7`
reserved parameter (must be initialized with **0**)

`VlanId`
VLAN ID

Only used if flag `G_ETHERNET__TP__INIT__ISO13400__CMD_FLAG__USE_VLAN_ID` is set.

A **V**irtual **L**ocal **A**rea **N**etwork can be configured with G_Ethernet_Node_Vlan_Config.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

12 TxFifo

These commands are used to control and configure the **TX FIFO** functionality.

A code example for using the TX FIFO can be found in the samples directory under *Ethernet_TxFifo_Tar*.

G_Ethernet_TxFifo_Config_Raw

G_Ethernet_TxFifo_Config_Raw — Configuration of raw TX FIFO

Description

Use this command for configuration of a TX FIFO for sending raw ethernet frames.

Definition

```
G_Error_t
G_Ethernet_TxFifo_Config_Raw(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_TxFifo_Config_Raw_Cmd_t * const cmd,
 G_Ethernet_TxFifo_Config_Raw_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__TX_FIFO__CONFIG__RAW__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__TX_FIFO__CONFIG__RAW__CMD_FLAG__GENERATE_TX_FIFO_ID`
0 : Use TX FIFO ID defined by `TxFifoId` (**0x00000000** .. **0x7FFFFFFF**)

1 : Generate a TX FIFO ID automatically

`TxFifoId`
ID for TX FIFO

Disregarded if flag
`G_ETHERNET__TX_FIFO__CONFIG__RAW__CMD_FLAG__GENERATE_TX_FIFO_ID` is set.

`FifoDepth`
Number of entries the FIFO can hold (starting from **1**)

`FifoDataLength`
Data length of a TX FIFO entry (starting from **14**)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`TxFifoId`
TX FIFO ID

`FifoDataLength`
Data length of a TX FIFO entry (starting from **1**)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_TxFifo_Config_Icmp6

G_Ethernet_TxFifo_Config_Icmp6 — Configuration of ICMPv6 TX FIFO

Description

Use this command for configuration of a raw TX FIFO for sending ICMPv6 ethernet frames.

Definition

```
G_Error_t
G_Ethernet_TxFifo_Config_Icmp6(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_TxFifo_Config_Icmp6_Cmd_t * const cmd,
 G_Ethernet_TxFifo_Config_Icmp6_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__TX_FIFO__CONFIG__ICMP6__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__TX_FIFO__CONFIG__ICMP6__CMD_FLAG__GENERATE_TX_FIFO_ID`
0 : Use TX FIFO ID defined by `TxFifoId` (**0x00000000** .. **0x7FFFFFFF**)

1 : Generate a TX FIFO ID automatically

`G_ETHERNET__TX_FIFO__CONFIG__ICMP6__CMD_FLAG__SET_LOCAL_ADDRESS`
0 : use default address and any port

1 = set local address from parameter `LocalAddress`

`TxFifoId`
ID for TX FIFO

Disregarded if flag `G_ETHERNET__TX_FIFO__CONFIG__ICMP6__CMD_FLAG__GENERATE_TX_FIFO_ID` is set.

`FifoDepth`
Number of entries the FIFO can hold (starting from **1**)

`FifoDataLength`
Data length of a TX FIFO entry (starting from **8**)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`RemoteAddress`
Address of remote node, in network byte order

`LocalAddress`
Address of local node, in network byte order

Used if flag `G_ETHERNET__TX_FIFO__CONFIG__ICMP6__CMD_FLAG__SET_LOCAL_ADDRESS` is set.

`RemotePort`
Port of remote node

`LocalPort`
Port of local node

Used if flag `G_ETHERNET__TX_FIFO__CONFIG__ICMP6__CMD_FLAG__SET_LOCAL_ADDRESS` is set.

`rsp`
Returns response parameters

The structure contains the following elements:

`TxFifoId`
TX FIFO ID

`FifoDataLength`
Data length of a TX FIFO entry (starting from **1**)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_TxFifo_Config_Udp6

G_Ethernet_TxFifo_Config_Udp6 — Configuration of ICMPv6 TX FIFO

Description

Use this command for configuration of a raw TX FIFO for sending UDPv6 ethernet frames.

Definition

```
G_Error_t
G_Ethernet_TxFifo_Config_Udp6(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_TxFifo_Config_Udp6_Cmd_t * const cmd,
 G_Ethernet_TxFifo_Config_Udp6_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__TX_FIFO__CONFIG__UDP6__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__TX_FIFO__CONFIG__UDP6__CMD_FLAG__GENERATE_TX_FIFO_ID`
0 : Use TX FIFO ID defined by `TxFifoId` (**0x00000000** .. **0x7FFFFFFF**)

1 : Generate a TX FIFO ID automatically

`G_ETHERNET__TX_FIFO__CONFIG__UDP6__CMD_FLAG__SET_LOCAL_ADDRESS`
0 : use default address and any port

1 = set local address from parameter `LocalAddress`

`G_ETHERNET__TX_FIFO__CONFIG__UDP6__CMD_FLAG__USE_VLAN_ID`
Use VLAN ID defined by `VlanId`.

`G_ETHERNET__TX_FIFO__CONFIG__UDP6__CMD_FLAG__SET_REMOTE_ADDR_FROM_RX`
Sets the remote address from the packet received by the corresponding Rx-Fifo. When this flag is set the value in `RemoteAddress` is ignored.

`G_ETHERNET__TX_FIFO__CONFIG__UDP6__CMD_FLAG__SET_MCAST_HOPS`
Sets the multicast hops field in outgoing packets to the value given in `MulticastHops`.

`G_ETHERNET__TX_FIFO__CONFIG__UDP6__CMD_FLAG__USE_MCAST_ADDR_FLT`
When the `RemoteAddress` is set to a multicast address, the address given in `MulticastAddressFilter` determines from which source address those multicast packets will be received. Packets from other sources will be discarded.

`TxFifoId`
ID for TX FIFO

Disregarded if flag `G_ETHERNET__TX_FIFO__CONFIG__UDP6__CMD_FLAG__GENERATE_TX_FIFO_ID` is set.

`FifoDepth`
Number of entries the FIFO can hold (starting from **1**)

`FifoDataLength`
Data length of a TX FIFO entry (starting from **1**)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`RemoteAddress`
Address of remote node, in network byte order

`LocalAddress`
Address of local node, in network byte order

Used if flag `G_ETHERNET__TX_FIFO__CONFIG__UDP6__CMD_FLAG__SET_LOCAL_ADDRESS` is set.

`RemotePort`
Port of remote node

`LocalPort`
Port of local node

Used if flag `G_ETHERNET__TX_FIFO__CONFIG__ICMP6__CMD_FLAG__SET_LOCAL_ADDRESS` is set.

`VlanId`
Identifier of the Vlan.

Used if flag `G_ETHERNET__TX_FIFO__CONFIG__UDP6__CMD_FLAG__USE_VLAN_ID` is set.

`MulticastHops`
Number of hops a multicast packet is allowed to go. If not set the default is 1.

Used if flag `G_ETHERNET__TX_FIFO__CONFIG__UDP6__CMD_FLAG__SET_MCAST_HOPS` is set.

`reserved3`
reserved parameter (must be initialized with **0**)

`MulticastPortFilter`
Filters the source port of an incoming multicast-packet. Set to **0** for any.

Used if flag `G_ETHERNET__TX_FIFO__CONFIG__UDP6__CMD_FLAG__USE_MCAST_ADDR_FLT` is set.

`MulticastAddressFilter`
Filters the source ip address of an incoming multicast-packet. Set to all **0** for any.

Used if flag `G_ETHERNET__TX_FIFO__CONFIG__UDP6__CMD_FLAG__USE_MCAST_ADDR_FLT` is set.

`rsp`
Returns response parameters

The structure contains the following elements:

`TxFifoId`
TX FIFO ID

`FifoDataLength`
Data length of a TX FIFO entry (starting from **1**)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_TxFifo_Config_Tcp6

G_Ethernet_TxFifo_Config_Tcp6 — Configuration of ICMPv6 TX FIFO

Description

Use this command for configuration of a raw TX FIFO for sending TCPv6 ethernet frames.

Before working with a TX FIFO for TCP, you need to connect the FIFO with command G_Ethernet_TxFifo_Connect .

Definition

```
G_Error_t
G_Ethernet_TxFifo_Config_Tcp6(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_TxFifo_Config_Tcp6_Cmd_t * const cmd,
 G_Ethernet_TxFifo_Config_Tcp6_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__TX_FIFO__CONFIG__TCP6__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__TX_FIFO__CONFIG__TCP6__CMD_FLAG__GENERATE_TX_FIFO_ID`
0 : Use TX FIFO ID defined by `TxFifoId` (**0x00000000** .. **0x7FFFFFFF**)

1 : Generate a TX FIFO ID automatically

`G_ETHERNET__TX_FIFO__CONFIG__TCP6__CMD_FLAG__SET_LOCAL_ADDRESS`
0 : use default address and any port

1 = set local address from parameter `LocalAddress`

`G_ETHERNET__TX_FIFO__CONFIG__TCP6__CMD_FLAG__USE_VLAN_ID`
Use VLAN ID defined by `VlanId`.

`G_ETHERNET__TX_FIFO__CONFIG__TCP6__CMD_FLAG__ACT_AS_SERVER`
The Tx-Fifo acts as a server and listens to incoming connections. It is not sending connect requests to clients. Start listening as soon as **connect** is called.

`TxFifoId`
ID for TX FIFO

Disregarded if flag
`G_ETHERNET__TX_FIFO__CONFIG__TCP6__CMD_FLAG__GENERATE_TX_FIFO_ID` is set.

`FifoDepth`
Number of entries the FIFO can hold (starting from **1**)

`FifoDataLength`
Data length of a TX FIFO entry (starting from **1**)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`RemoteAddress`
Address of remote node, in network byte order

`LocalAddress`
Address of local node, in network byte order

Used if flag `G_ETHERNET__TX_FIFO__CONFIG__TCP6__CMD_FLAG__SET_LOCAL_ADDRESS` is set.

`RemotePort`
Port of remote node

`LocalPort`
Port of local node

Used if flag `G_ETHERNET__TX_FIFO__CONFIG__ICMP6__CMD_FLAG__SET_LOCAL_ADDRESS` is set.

`rsp`
Returns response parameters

The structure contains the following elements:

`TxFifoId`
TX FIFO ID

`FifoDataLength`
Data length of a TX FIFO entry (starting from **1**)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_TxFifo_Config_Udp4

G_Ethernet_TxFifo_Config_Udp4 — Configuration of ICMPv6 TX FIFO

Description

Use this command for configuration of a raw TX FIFO for sending UDPv4 ethernet frames.

Definition

```
G_Error_t
G_Ethernet_TxFifo_Config_Udp4(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_TxFifo_Config_Udp4_Cmd_t * const cmd,
 G_Ethernet_TxFifo_Config_Udp4_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__TX_FIFO__CONFIG__UDP4__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__TX_FIFO__CONFIG__UDP4__CMD_FLAG__GENERATE_TX_FIFO_ID`
0 : Use TX FIFO ID defined by `TxFifoId` (**0x00000000** .. **0x7FFFFFFF**)

1 : Generate a TX FIFO ID automatically

`G_ETHERNET__TX_FIFO__CONFIG__UDP4__CMD_FLAG__SET_LOCAL_ADDRESS`
0 : use default address and any port

1 = set local address from parameter `LocalAddress`

`G_ETHERNET__TX_FIFO__CONFIG__UDP4__CMD_FLAG__SET_REMOTE_ADDR_FROM_RX`
0 : use remote address from parameter "RemoteAddress"

1 = set remote address from reception fifo `LocalAddress`

`G_ETHERNET__TX_FIFO__CONFIG__UDP4__CMD_FLAG__MCAST_LOOP_DISABLE`
Disables the reception of multicast packets in the corresponding Rx-Fifo, that are sent by this Tx-Fifo.

`G_ETHERNET__TX_FIFO__CONFIG__UDP4__CMD_FLAG__SET_MCAST_HOPS`
Sets the multicast hops field in outgoing packets to the value given in `MulticastHops`.

`TxFifoId`
ID for TX FIFO

Disregarded if flag G_ETHERNET__TX_FIFO__CONFIG__UDP4__CMD_FLAG__GENERATE_TX_FIFO_ID is set.

FifoDepth
Number of entries the FIFO can hold (starting from 1)

FifoDataLength
Data length of a TX FIFO entry (starting from 1)

reserved1
reserved parameter (must be initialized with 0)

reserved2
reserved parameter (must be initialized with 0)

RemoteAddress
Address of remote node, in network byte order

LocalAddress
Address of local node, in network byte order

Used if flag G_ETHERNET__TX_FIFO__CONFIG__UDP4__CMD_FLAG__SET_LOCAL_ADDRESS is set.

RemotePort
Port of remote node

LocalPort
Port of local node

Used if flag G_ETHERNET__TX_FIFO__CONFIG__ICMP6__CMD_FLAG__SET_LOCAL_ADDRESS is set.

MulticastHops
Number of hops a multicast packet is allowed to go. If not set the default is 1.

Used if flag G_ETHERNET__TX_FIFO__CONFIG__UDP6__CMD_FLAG__SET_MCAST_HOPS is set.

reserved3
reserved parameter (must be initialized with 0)

rsp
Returns response parameters

The structure contains the following elements:

TxFifoId
TX FIFO ID

FifoDataLength
Data length of a TX FIFO entry (starting from 1)

reserved1
reserved parameter (must be initialized with 0)

reserved2
reserved parameter (must be initialized with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_TxFifo_Config_Tcp4

G_Ethernet_TxFifo_Config_Tcp4 — Configuration of ICMPv6 TX FIFO

Description

Use this command for configuration of a raw TX FIFO for sending TCPv4 ethernet frames.

Before working with a TX FIFO for TCP, you need to connect the FIFO with command G_Ethernet_TxFifo_Connect .

Definition

```
G_Error_t
G_Ethernet_TxFifo_Config_Tcp4(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_TxFifo_Config_Tcp4_Cmd_t * const cmd,
 G_Ethernet_TxFifo_Config_Tcp4_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__TX_FIFO__CONFIG__TCP4__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__TX_FIFO__CONFIG__TCP4__CMD_FLAG__GENERATE_TX_FIFO_ID`
0 : Use TX FIFO ID defined by `TxFifoId` (**0x00000000** .. **0x7FFFFFFF**)

1 : Generate a TX FIFO ID automatically

`G_ETHERNET__TX_FIFO__CONFIG__TCP4__CMD_FLAG__SET_LOCAL_ADDRESS`
0 : use default address and any port

1 = set local address from parameter `LocalAddress`

`G_ETHERNET__TX_FIFO__CONFIG__TCP4__CMD_FLAG__ACT_AS_SERVER`
The Tx-Fifo acts as a server and listens to incoming connections. It is not sending connect requests to clients. Start listening as soon as **connect** is called.

`TxFifoId`
ID for TX FIFO

Disregarded if flag
`G_ETHERNET__TX_FIFO__CONFIG__TCP4__CMD_FLAG__GENERATE_TX_FIFO_ID` is set.

`FifoDepth`
Number of entries the FIFO can hold (starting from **1**)

`FifoDataLength`
Data length of a TX FIFO entry (starting from **1**)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`RemoteAddress`
Address of remote node, in network byte order

`LocalAddress`
Address of local node, in network byte order

Used if flag `G_ETHERNET__TX_FIFO__CONFIG__TCP4__CMD_FLAG__SET_LOCAL_ADDRESS` is set.

`RemotePort`
Port of remote node

`LocalPort`
Port of local node

Used if flag `G_ETHERNET__TX_FIFO__CONFIG__ICMP6__CMD_FLAG__SET_LOCAL_ADDRESS` is set.

`rsp`
Returns response parameters

The structure contains the following elements:

`TxFifoId`
TX FIFO ID

`FifoDataLength`
Data length of a TX FIFO entry (starting from **1**)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_TxFifo_Delete

G_Ethernet_TxFifo_Delete — Delete TX FIFO

Description

Use this command to delete a TX FIFO.

Definition

```
G_Error_t
G_Ethernet_TxFifo_Delete(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_TxFifo_Delete_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__TX_FIFO__DELETE__CMD_FLAG__NONE`
No flag is set

`G_ETHERNET__TX_FIFO__DELETE__CMD_FLAG__DELETE_ALL`
All TX FIFOs are deleted, parameter `TxFifoId` is disregarded

`TxFifoId`
ID of the TX FIFO to be deleted

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_TxFifo_Write

G_Ethernet_TxFifo_Write — Write TX FIFO entries

Description

Use this command to write TX FIFO entries.

Definition

```
G_Error_t
G_Ethernet_TxFifo_Write(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_TxFifo_Write_CmdFlags_t  cmdFlags,
 const u32_t  txFifoId,
 const u32_t  numberOfEntries,
 const G_Ethernet_TxFifo_Write_Entry_t * const entries
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__TX_FIFO__WRITE__CMD_FLAG__NONE`
No flag is set

`txFifoId`
ID of the TX FIFO

`numberOfEntries`
Number of entries to be written

`Entries to be written`
An entry contains the following elements:

`Length`
Data length, in bytes

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`reserved4`
reserved parameter (must be initialized with **0**)

`reserved5`
reserved parameter (must be initialized with **0**)

`reserved6`
reserved parameter (must be initialized with **0**)

`Data`
Pointer to data bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_TxFifo_Connect

G_Ethernet_TxFifo_Connect — Establish a TCP connection

Description

Use this command to establish a TCP connection. This is mandatory before working with a TX FIFO that is configured for TCP.

To reset the connection call G_Ethernet_TxFifo_Disconnect .

Definition

```
G_Error_t
G_Ethernet_TxFifo_Connect(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_TxFifo_Connect_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__TX_FIFO__CONNECT__CMD_FLAG__NONE`
No flag is set

`TxFifoId`
ID of the TX FIFO

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_TxFifo_Disconnect

G_Ethernet_TxFifo_Disconnect — Reset a TCP TX FIFO connection

Description

Use this command to Reset a TCP TX FIFO connection:

Definition

```
G_Error_t
G_Ethernet_TxFifo_Disconnect(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_TxFifo_Disconnect_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__TX_FIFO__DISCONNECT__CMD_FLAG__NONE`
No flag is set

`TxFifoId`
ID of the TX FIFO

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Ethernet_TxFifo_GetState

G_Ethernet_TxFifo_GetState — Query TX FIFO state

Description

Use this command to query the TX FIFO state.

Definition

```
G_Error_t
G_Ethernet_TxFifo_GetState(
 const G_PortHandle_t  portHandle,
 const G_Ethernet_TxFifo_GetState_Cmd_t * const cmd,
 G_Ethernet_TxFifo_GetState_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_ETHERNET__TX_FIFO__GET_STATE__CMD_FLAG__NONE`
No flag is set

`TxFifoId`
ID of the TX FIFO

`rsp`
Returns response parameters

The structure contains the following elements:

`ConnectionState`
Connection state of the TX FIFO

The following values are possible:

`G_ETHERNET__TX_FIFO__CONNECTION_STATE__UNKNOWN`
The connection state is unknown

`G_ETHERNET__TX_FIFO__CONNECTION_STATE__DISCONNECTED`
The TX FIFO is not connected

`G_ETHERNET__TX_FIFO__CONNECTION_STATE__CONNECT_IN_PROGRESS`
A connection is in progress

`G_ETHERNET__TX_FIFO__CONNECTION_STATE__CONNECTED`
The TX FIFO is connected

`G_ETHERNET__TX_FIFO__CONNECTION_STATE__DISCONNECT_IN_PROGRESS`
A disconnection is in progress

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

FlexRay Functions

These **G-API** functions provide access to the FlexRay-Functionality of your Hardware. If the return value of type **G_Error_t** does not correspond to `G_NO_ERROR`, an error has occurred. A description of this error can be retrieved by calling G_GetLastErrorDescription .

1 Overview

1.1 Command Sequence

After power on, the following command sequence is recommended to set up FlexRay communication:

G_FlexRay_Init

G_FlexRay_Config or G_FlexRay_ConfigWithXmlFile

G_FlexRay_Startup_Config [1]

G_FlexRay_Message_Ramp_Define [1]

G_FlexRay_Message_Checksum_Crc_Define [1]

G_FlexRay_Message_Checksum_Crc_Assign [1]

G_FlexRay_Message_Tx_Define

G_FlexRay_Message_Rx_Define

G_FlexRay_Buffers_Config

G_FlexRay_Communication_Start

G_FlexRay_Message_Transmit [1]

G_FlexRay_Message_Rx_Read [1]

G_FlexRay_Communication_Stop

[1] this command is optional

2 Buffers

These commands are used to configure the FlexRay message buffers.

G_FlexRay_Buffers_Config

G_FlexRay_Buffers_Config — Allocate message buffers

Description

Allocate the message buffers for all defined messages, the receive shadow buffers and the receive FIFO buffers for message monitoring.

Definition

```
G_Error_t
G_FlexRay_Buffers_Config(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

3 Common

Common FlexRay commands

G_FlexRay_Init

G_FlexRay_Init — Init FlexRay controller

Description

This command initializes the FlexRay controller.

Definition

```
G_Error_t
G_FlexRay_Init(
 const G_PortHandle_t portHandle,
 const G_FlexRay_Init_Flags_t  flags,
 const G_FlexRay_ControllerType_t  controllerType,
 const G_FlexRay_TransceiverType_t  transceiverType
);
```

Parameters

`portHandle`
Handle to the communication port

`flags`
Init flags

The following values are possible and can be combined:

`G_FLEXRAY__INIT__FLAG__NONE`
No flag is set

`G_FLEXRAY__INIT__FLAG__SINGLE_CHANNEL`
Enable single channel mode

`controllerType`
Type of FlexRay interface controller

The following values are possible:

`G_FLEXRAY__CONTROLLER_TYPE__MFR4300`
Controller type **Freescale MFR4300**

`G_FLEXRAY__CONTROLLER_TYPE__MFR4310`
Controller type **Freescale MFR4310**

`transceiverType`
Type of FlexRay interface transceiver

The following values are possible:

`G_FLEXRAY__TRANSCEIVER_TYPE__TJA1080`
Transceiver type **NXP TJA1080**

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Config

G_FlexRay_Config — Configure FlexRay cluster

Description

This command configures the FlexRay cluster.

For configuring a FlexRay cluster with an **XML File**, use function G_FlexRay_ConfigWithXmlFile.

Definition

```
G_Error_t
G_FlexRay_Config(
 const G_PortHandle_t portHandle,
 const G_FlexRay_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Structure with FlexRay cluster parameters

Please refer to the "Configuration Constraints" section in the "FlexRay Communications System Protocol Specification document, version 2.1, revision A" for more detailed information on individual parameters.

The structure contains the following elements:

`gColdStartAttempts`
Defines the maximum number of times that a node in this cluster is permitted to attempt to start the cluster by initiating schedule synchronization.

Range: **2** - **31**

`gListenNoise`
Defines the upper limit for the start up and wake up listen timeout in the presence of noise. Expressed as a multiple of the cluster constant `pdListenTimeout`.

Range: **2** - **16**

`gMacroPerCycle`
Defines the duration of a **FlexRay Communication cycle** in macroticks (**MT**).

Range: **10** - **16000 MT**

`gNumberOfMinislots`
Defines the number of minislots in the **dynamic segment** .

Range: **0** - **7986**

`gNumberOfStaticSlots`
Defines the number of Static slots in the **static segment** .

gOffsetCorrectionStart
Defines the start of the offset correction phase within the **NIT** (**Network Idle Time**), expressed as the number of macroticks from the start of the cycle.

Range: **9** - **15999 MT**

gMaxWithoutClockCorrectionFatal
Defines the threshold used for testing the vClockCorrectionFailed counter.

Defines the number of consecutive **even / odd cycle pairs** with missing clock correction terms that will cause the protocol to transition from the **POC: normal active** or **POC: normal passive** state into the **POC: halt** state.

POC: Protocol Operation Control

Range: gMaxWithoutClockCorrectionPassive - **even / odd cycle pairs**

gMaxWithoutClockCorrectionPassive
Defines the threshold used for testing the **vClockCorrectionFailed** counter.

Defines the number of consecutive **even / odd cycle pairs** with missing clock correction terms that will cause the protocol to transition from the **POC: normal active** to the **POC: normal passive** state.

If gMaxWithoutClockCorrectionPassive is set to a value greater than or equal to gMaxWithoutClockCorrectionFatal, then the **CC** enters the **POC:halt state** directly (i.e., without first entering the **POC:normal passive state**).

POC: Protocol Operation Control

Range: **1** - **15 even / odd cycle pairs**

gNetworkManagementVectorLength
Defines the length of the Network Management vector in a cluster.

Range: **0** - **12 bytes**

gPayloadLengthStatic
Defines the payload length of a static frame in **two-byte-words** (All static frames in a cluster have the same payload length)

Range: **0** - **cPayloadLengthMax**

gSyncNodeMax
Defines the maximum number of nodes that may send frames with the sync frame indicator bit set.

Range: **2** - **cSyncNodeMax**

gdActionPointOffset
Defines the number of macroticks the action point is offset from the beginning of a **Static slot** or **Symbol window .**

Range: **1** - **63 MT**

gdBit
Defines the nominal bit time **(= 1/fx: SPEED) .**

Range: **cSamplesPerBit * gdSampleClockPeriod** (in microseconds)

gdCasRxLowMax
Defines the upper limit of the **CAS** acceptance window.

gdCASRxLowMax[gdBit] = ceil(2 * (gdTSSTransmitter[gdBit] + cdCAS[gdBit]) * (1 + 2 * cClockDeviationMax))

CAS: Collision Avoidance Symbol

Range: **67** - **99 gdBit**

`gdDynamicSlotIdlePhase`
Defines the duration of the idle phase within a dynamic slot.

Range: **0** - **2 MiniSlot**

`gdMinislotActionPointOffset`
Defines the number of macroticks the minislot action point is offset from the beginning of a minislot.

Range: **1** - **31 MT**

`gdMinislot`
Defines the duration of a minislot.

Range: **2** - **63 MT**

`gdStaticSlot`
Defines the duration of a **Static slot** in the **Static segment .**

Range: **4** - **661 MT**

`gdSymbolWindow`
Defines the duration of the **Symbol window .**

Range: **0** - **142 MT**

`gdTSSTransmitter`
Defines the number of bits in the **Transmission Start Sequence .**

Range: **3** - **15 gdBit**

`gdWakeupSymbolRxIdle`
Defines the number of bits used by the node to test the duration of the 'idle' portion of a received wakeup symbol.

The duration is equal to (**gdWakeupSymbolTxIdle - gdWakeupSymbolTxLow**)/ **2** minus a safe part (collisions, clock differences, and other effects can deform the TX-wakeup pattern).

Range: **14** - **59 gdBit**

`gdWakeupSymbolRxLow`
Defines the number of bits used by the node to test the LOW portion of a received wakeup symbol (this lower limit of zero bits has to be received to detect the LOW portion by the receiver).

The duration is equal to **gdWakeupSymbolTxLow** minus a safe part (active stars, clock differences, and other effects can deform the TX-wakeup pattern).

Range: **11** - **59 gdBit**

`gdWakeupSymbolTxIdle`
Defines the number of bits used by the node to transmit the “idle” part of a wakeup symbol.

The duration is equal to **cdWakeupSymbolTxIdle .**

Range: **45** - **180 gdBit**

`gdWakeupSymbolTxLow`
Defines the Number of bits used by the node to transmit the LOW part of a wakeup symbol.

The duration is equal to **cdWakeupSymbolTxLow** .

Range: **15** - **60 gdBit**

`gdWakeupSymbolRxWindow`
Defines the size of the window used to detect wakeups.

Detection of a wakeup requires a low and idle period (from one **WUS**) and a low period (from another **WUS**) to be detected entirely within a window of this size.

The duration is equal to **gdWakeupSymbolTxIdle + 2 * gdWakeupSymbolTxLow** plus a safe part (clock differences and other effects can deform the TX-wakeup pattern).

WUS: Wakeup symbol

Range: **76** - **301 gdBit**

`pAllowHaltDueToClock`
Defines the Boolean flag that controls the transition to the **POC: halt** state due to a clock synchronization error.

If set to **true** , the communication controller is allowed to transition to **POC: halt** .

If set to **false** , the communication controller will not transition to the **POC: halt** state but will enter or remain in the **POC: normal passive** state (self healing would still be possible).

POC: Protocol Operation Control

Range: **0** - **31 even / odd cycle pairs**

`pAllowPassiveToActive`
Defines the number of **consecutive even / odd cycle pairs** that must have valid clock correction terms before the communication controller will be allowed to transition from the **POC: normal passive** state to **POC: normal active** state.

If set to **0** , the communication controller is not allowed to transition from **POC: normal passive** to **POC: normal active** state.

Range: **0 31 even/ odd cycle pairs**

`pClusterDriftDamping`
Defines the Local Cluster **drift damping factor** used for rate correction.

Range: **0** - **20 microticks**

`pDecodingCorrection`
Defines the value used by the receiver to calculate the difference between primary time reference point and secondary time reference point.

Range: **14** - **143 microticks**

`pDelayCompensationA`
Defines the value used to compensate for reception delays on FlexRay **Channel A** .

This covers assumed propagation delay up to **cPropagationDelayMax** for microticks in the range of **0.0125 microseconds** to **0.05 microseconds** .

In practice, the minimum of the propagation delays of all sync nodes should be applied.

Range: **0** - **200 microticks**

`pDelayCompensationB`
Defines the value used to compensate for reception delays on FlexRay **Channel B**.

This covers assumed propagation delay up to **cPropagationDelayMax** for microticks in the range of **0.0125 microseconds** to **0.05 microseconds**.

In practice, the minimum of the propagation delays of all sync nodes should be applied.

Range: **0** - **200 microticks**

`pExternOffsetCorrection`
Defines the number of microticks added or subtracted to the **NIT (Network Idle Time)**, to carry out a host-requested external offset correction.

Range: **0** - **7 microticks**

`pExternRateCorrection`
Defines the number of microticks added or subtracted to the cycle to carry out a host-requested external rate correction.

Range: **0** - **7 microticks**

`pKeySlotId`
Defines the identifier of the slot used to transmit the startup frame, sync frame, or designated single slot frame.

Range: **1** - **cStaticSlotIdMax**

`pKeySlotUsedForStartup`
Defines the flag indicating the Key Slot is used to transmit a startup frame.

If **pKeySlotUsedForStartup** is set to **true**, then **pKeySlotUsedForSync** must also be set to **true**.

Range: **Boolean**

`pKeySlotUsedForSync`
Defines the flag indicating the Key Slot is used to transmit a sync frame.

If **pKeySlotUsedForStartup** is set to **true**, then **pKeySlotUsedForSync** must also be set to **true**.

Range: **Boolean**

`pLatestTx`
Defines the number of the last minislot in which a frame transmission can start in the dynamic segment.

If **pKeySlotUsedForStartup** is set to **true**, then **pKeySlotUsedForSync** must also be set to **true**.

Range: **0** - **7980 Minislots**

`pChannels`
Indicates channels to which the node is connected.

Range:

0 : FlexRay **Channel A**

1 : FlexRay **Channel B**

2 : FlexRay **Channel A** and **Channel B**

`pSingleSlotEnabled`
Defines the flag indicating whether or not the node shall enter single slot mode following startup.

Range: **Boolean**

`pMicroPerCycle`
Defines the nominal number of microticks in the communication cycle of the local node.

If nodes have different microtick durations this number will differ from node to node.

Range: **640** - **640000 microticks**

`pMicroPerMacroNom`
Defines the number of microticks per nominal macrotick that all implementations must support.

If nodes have different microtick durations this number will differ from node to node.

Range: **cMicroPerMacroNomMin** - **cMicroPerMacroNomMax**

`pPayloadLengthDynMax`
Defines the maximum payload length of a dynamic frame in **two-byte-words** .

Range: **0** - **cPayloadLengthMax**

`pMacroInitialOffsetA`
Defines the integer number of macroticks between the **Static slot** boundary and the following macrotick boundary of the secondary time reference point based on the nominal macrotick duration.

Range: **2** - **68 MT**

`pMacroInitialOffsetB`
Defines the integer number of macroticks between the **Static slot** boundary and the following macrotick boundary of the secondary time reference point based on the nominal macrotick duration.

Range: **2** - **68 MT**

`pMicroInitialOffsetA`
Defines the Number of microticks between the closest macrotick boundary described by **pMacroInitialOffsetA** and the secondary time reference point.

The parameter depends on **pDelayCompensationA** .

Range: **0** - **239 MT**

`pMicroInitialOffsetB`
Defines the Number of microticks between the closest macrotick boundary described by **pMacroInitialOffsetB** and the secondary time reference point.

The parameter depends on **pDelayCompensationB** .

Range: **0** - **239 MT**

`pOffsetCorrectionOut`
Defines the magnitude of the maximum permissible offset correction value.

Range: **13** - **15567 MT**

`pRateCorrectionOut`
Defines the magnitude of the maximum permissible rate correction value.

Range: **2 - 1923 MT**

`pWakeupChannel`
Defines the FlexRay channel used by the node to send a wakeup pattern.

Range:

0 : FlexRay **Channel A**

1 : FlexRay **Channel B**

`pWakeupPattern`
Defines the number of repetitions of the wakeup symbol that are combined to form a wakeup pattern when the node enters the **POC: wakeup send** state.

Range: **2 - 63 MT**

`pdAcceptedStartupRange`
Defines the expanded range of measured clock deviation allowed for startup frames during integration.

Range: **0 - 1875 MT**

`pdMaxDrift`
Defines the maximum drift offset between two nodes that operate with unsynchronized clocks over one communication cycle.

Range: **2 - 1923 MT**

`pdListenTimeout`
Defines the value for the startup listen timeout and wakeup listen timeout.

Although this is a node local parameter, the real time equivalent of this value should be the same for all nodes in the cluster.

Range: **1284 - 1283846 MT**

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_ConfigWithXmlFile

G_FlexRay_ConfigWithXmlFile — Configure FlexRay cluster parameters via XML file

Description

This command configures a FlexRay cluster with parameters stored in an XML file.

Definition

```
G_Error_t
G_FlexRay_ConfigWithXmlFile(
 const G_PortHandle_t portHandle,
 const char * const path
);
```

Parameters

`portHandle`
: Handle to the communication port

`path`
: Path to XML file

An example of an XML configuration file can be found in the **G-API** -FlexRay-samples.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_ConfigFromFibex

G_FlexRay_ConfigFromFibex — Configure FlexRay interface from **FIBEX** database

Description

This command configures a FlexRay interface with parameters specified in an **ASAM MCD-2 NET database (FIBEX - Field Bus Data Exchange Format)**. FIBEX describes an XML exchange format for data exchange between tools that deal with bus communication systems.

The following FIBEX versions are currently supported:

- v1.2.0
- v2.0.0
- v2.0.1
- v3.0.0
- v3.1.0

Definition

```
G_Error_t
G_FlexRay_ConfigFromFibex(
 const G_PortHandle_t portHandle,
 const char * const path,
 const char * const clusterShortName,
 const char * const controllerShortName
);
```

Parameters

`portHandle`
Handle to the communication port

`path`
Path to the **FIBEX database file**

`clusterShortName`
FIBEX short name of flexray cluster

This name identifies the FlexRay cluster the FlexRay interface should be integrated into. If **NULL** is passed instead of **char*** the first FlexRay cluster is selected.

`controllerShortName`
FIBEX short name of flexray controller

Identifies the FlexRay controller that the FlexRay interface should be integrated into.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_GetConfigData_FromXmlFile

G_FlexRay_GetConfigData_FromXmlFile — Get FlexRay cluster parameters from XML file

Description

This command returns FlexRay cluster parameters that are stored in an XML file. These parameters can be used to configure a FlexRay cluster with G_FlexRay_Config.

Definition

```
G_Error_t
G_FlexRay_GetConfigData_FromXmlFile(
 const G_PortHandle_t portHandle,
 const char * const path,
 G_FlexRay_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
: Handle to the communication port

`path`
: Path to XML file

`parameters`
: Returns structure with FlexRay cluster configuration parameters (see G_FlexRay_Config_Parameters_t)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_GetConfigData_FromFibex

G_FlexRay_GetConfigData_FromFibex — Get FlexRay cluster parameters from **FIBEX** database

Description

This command returns a structure with FlexRay cluster parameters specified in an **ASAM MCD-2 NET database (FIBEX - Field Bus Data Exchange Format)** . FIBEX describes an XML exchange format for data exchange between tools that deal with bus communication systems.

The following FIBEX versions are currently supported:

- v1.2.0
- v2.0.0
- v2.0.1
- v3.0.0
- v3.1.0

Definition

```
G_Error_t
G_FlexRay_GetConfigData_FromFibex(
 const G_PortHandle_t portHandle,
 const char * const path,
 const char * const clusterShortName,
 const char * const controllerShortName,
 G_FlexRay_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`path`
Path to the **FIBEX database file**

`clusterShortName`
FIBEX short name of flexray cluster

This name identifies the FlexRay cluster the FlexRay interface should be integrated into. If **NULL** is passed instead of **char *** the first FlexRay cluster is selected.

`controllerShortName`
FIBEX short name of flexray controller

Identifies the FlexRay controller that the FlexRay interface should be integrated into.

`parameters`
Returns structure with FlexRay cluster configuration parameters (see G_FlexRay_Config_Parameters_t)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Property_SetById

G_FlexRay_Property_SetById — Set the value of a FlexRay property.

Description

Use this command to set the value of a FlexRay property.

Definition

```
G_Error_t
G_FlexRay_Property_SetById(
 const G_PortHandle_t portHandle,
 const u32_t  id,
 const u32_t  value
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
FlexRay Property ID

The following values are possible:

`G_FLEXRAY__PROPERTY_ID__NUMBER_OF_MSG_BUFFERS`
This property is read only

Total number of used message buffers in FlexRay communication controller for transmission and reception.

`G_FLEXRAY__PROPERTY_ID__NUMBER_OF_MSG_BUFFERS__TX_STATIC`
This property is read only

Number of used message buffers in FlexRay communication controller for transmission in FlexRay static segment.

`G_FLEXRAY__PROPERTY_ID__NUMBER_OF_MSG_BUFFERS__TX_DYNAMIC`
This property is read only

Number of used message buffers in FlexRay communication controller for transmission in FlexRay dynamic segment.

`G_FLEXRAY__PROPERTY_ID__NUMBER_OF_MSG_BUFFERS__RX_FIFO_A`
This property is read only

Number of used message buffers in FlexRay communication controller for reception on channel A.

`G_FLEXRAY__PROPERTY_ID__NUMBER_OF_MSG_BUFFERS__RX_FIFO_B`
This property is read only

Number of used message buffers in FlexRay communication controller for reception on channel B.

`G_FLEXRAY__PROPERTY_ID__COUNTER__JOB_LIST__SYNCHRONIZE`
This property is read only

Counter for job list synchronizations.

`G_FLEXRAY__PROPERTY_ID__COUNTER__TX_REQUEST__ERROR__STATIC`
This property is read only

Counter for failing transmission requests within static segment.

`G_FLEXRAY__PROPERTY_ID__COUNTER__TX_REQUEST__ERROR__DYNAMIC`
This property is read only

Counter for failing transmission requests within dynamic segment.

`G_FLEXRAY__PROPERTY_ID__COUNTER__TX_REQUEST__SUCCESS__STATIC`
This property is read only

Counter for successful transmission requests within static segment.

`G_FLEXRAY__PROPERTY_ID__COUNTER__TX_REQUEST__SUCCESS__DYNAMIC`
This property is read only

Counter for successful transmission requests within dynamic segment.

`G_FLEXRAY__PROPERTY_ID__JOB_LIST__DECOUPLED_TX__OFFSET__US`
This property is read only

Time between scheduled start of job "decoupled transmission" and start of transmission in microseconds.

`G_FLEXRAY__PROPERTY_ID__JOB_LIST__DECOUPLED_TX__OFFSET__MT`
This property is read only

Time between scheduled start of job "decoupled transmission" and start of transmission in macro ticks.

`G_FLEXRAY__PROPERTY_ID__COUNTER__DISABLE_TX_MSG_BUFFER__ERROR`
This property is read only

The counter is incremented if disabling of a TX buffer fails.

`G_FLEXRAY__PROPERTY_ID__COUNTER__DISABLE_TX_MSG_BUFFER__SUCCESS`
This property is read only

The counter is incremented if disabling of a TX buffer was successful.

`G_FLEXRAY__PROPERTY_ID__COUNTER__TX_NULL_FRAME__ERROR`
This property is read only

The counter is incremented if transmission request of a null frame fails.

`G_FLEXRAY__PROPERTY_ID__COUNTER__TX_NULL_FRAME__SUCCESS`
This property is read only

The counter is incremented if transmission request of a null frame was successful.

`G_FLEXRAY__PROPERTY_ID__UNKNOWN`
This is a place holder

`value`
Property value to be set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Property_GetById

G_FlexRay_Property_GetById — Query the value of a FlexRay property.

Description

Use this command to query the value of a FlexRay property.

Definition

```
G_Error_t
G_FlexRay_Property_GetById(
 const G_PortHandle_t portHandle,
 const u32_t  id,
 u32_t * const value
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
FlexRay Property ID

The following values are possible:

`G_FLEXRAY__PROPERTY_ID__NUMBER_OF_MSG_BUFFERS`
This property is read only

Total number of used message buffers in FlexRay communication controller for transmission and reception.

`G_FLEXRAY__PROPERTY_ID__NUMBER_OF_MSG_BUFFERS__TX_STATIC`
This property is read only

Number of used message buffers in FlexRay communication controller for transmission in FlexRay static segment.

`G_FLEXRAY__PROPERTY_ID__NUMBER_OF_MSG_BUFFERS__TX_DYNAMIC`
This property is read only

Number of used message buffers in FlexRay communication controller for transmission in FlexRay dynamic segment.

`G_FLEXRAY__PROPERTY_ID__NUMBER_OF_MSG_BUFFERS__RX_FIFO_A`
This property is read only

Number of used message buffers in FlexRay communication controller for reception on channel A.

`G_FLEXRAY__PROPERTY_ID__NUMBER_OF_MSG_BUFFERS__RX_FIFO_B`
This property is read only

Number of used message buffers in FlexRay communication controller for reception on channel B.

`G_FLEXRAY__PROPERTY_ID__COUNTER__JOB_LIST__SYNCHRONIZE`
This property is read only

Counter for job list synchronizations.

`G_FLEXRAY__PROPERTY_ID__COUNTER__TX_REQUEST__ERROR__STATIC`
This property is read only

Counter for failing transmission requests within static segment.

`G_FLEXRAY__PROPERTY_ID__COUNTER__TX_REQUEST__ERROR__DYNAMIC`
This property is read only

Counter for failing transmission requests within dynamic segment.

`G_FLEXRAY__PROPERTY_ID__COUNTER__TX_REQUEST__SUCCESS__STATIC`
This property is read only

Counter for successful transmission requests within static segment.

`G_FLEXRAY__PROPERTY_ID__COUNTER__TX_REQUEST__SUCCESS__DYNAMIC`
This property is read only

Counter for successful transmission requests within dynamic segment.

`G_FLEXRAY__PROPERTY_ID__JOB_LIST__DECOUPLED_TX__OFFSET__US`
This property is read only

Time between scheduled start of job "decoupled transmission" and start of transmission in microseconds.

`G_FLEXRAY__PROPERTY_ID__JOB_LIST__DECOUPLED_TX__OFFSET__MT`
This property is read only

Time between scheduled start of job "decoupled transmission" and start of transmission in macro ticks.

`G_FLEXRAY__PROPERTY_ID__COUNTER__DISABLE_TX_MSG_BUFFER__ERROR`
This property is read only

The counter is incremented if disabling of a TX buffer fails.

`G_FLEXRAY__PROPERTY_ID__COUNTER__DISABLE_TX_MSG_BUFFER__SUCCESS`
This property is read only

The counter is incremented if disabling of a TX buffer was successful.

`G_FLEXRAY__PROPERTY_ID__COUNTER__TX_NULL_FRAME__ERROR`
This property is read only

The counter is incremented if transmission request of a null frame fails.

`G_FLEXRAY__PROPERTY_ID__COUNTER__TX_NULL_FRAME__SUCCESS`
This property is read only

The counter is incremented if transmission request of a null frame was successful.

`G_FLEXRAY__PROPERTY_ID__UNKNOWN`
This is a place holder

`value`
Returns property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

4 Communication

These commands are used to start and stop the general FlexRay communication.

G_FlexRay_Communication_Start

G_FlexRay_Communication_Start — Initiate start of the FlexRay communication

Description

Initiate the start of the FlexRay communication.

Definition

```
G_Error_t
G_FlexRay_Communication_Start(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Communication_Stop

G_FlexRay_Communication_Stop — Stop FlexRay communication

Description

Stop the FlexRay communication.

Definition

```
G_Error_t
G_FlexRay_Communication_Stop(
 const G_PortHandle_t portHandle,
 const G_FlexRay_Communication_Stop_Mode_t  mode
);
```

Parameters

`portHandle`
: Handle to the communication port

`mode`
: Stop mode

  The following values are possible:

  `G_FLEXRAY__COMMUNICATION__STOP__MODE__STOP`
  : Stops the communication of the FlexRay communication controller at the end of current FlexRay cycle

  `G_FLEXRAY__COMMUNICATION__STOP__MODE__HALT`
  : Halts the FlexRay communication at the end of current FlexRay cycle

  `G_FLEXRAY__COMMUNICATION__STOP__MODE__ABORT`
  : Immediately aborts all FlexRay communication

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

5 ConfigMode

These commands are used to control the FlexRay config mode.

G_FlexRay_ConfigMode_Enter

G_FlexRay_ConfigMode_Enter — Enter config mode

Description

Initiate a transition from **POC: ready** state into **POC: config** state.

Definition

```
G_Error_t
G_FlexRay_ConfigMode_Enter(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_ConfigMode_Leave

G_FlexRay_ConfigMode_Leave — Leave config mode

Description

Initiate a transition from **POC: config** state into **POC: ready** state.

Definition

```
G_Error_t
G_FlexRay_ConfigMode_Leave(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

6 Diag

These commands are used to control the FlexRay diagnostics.

G_FlexRay_Diag_Config_Uds

G_FlexRay_Diag_Config_Uds — Configure FlexRay UDS diagnostics

Description

Use this command to configure UDS diagnostics.

Definition

```
G_Error_t
G_FlexRay_Diag_Config_Uds(
 const G_PortHandle_t portHandle,
 const u8_t  channel,
 const u32_t  globalTimeout,
 const G_FlexRay_Diag_Config_Uds_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Diag multisession channel

A multisession channel is required if multiple applications are working with the diagnostics on the same interface. For each channel, the diagnostics can be configured with different parameters.

Usually, it is not neccessary to use multisession channels. It is recommended to use channel **0** in these cases.

`globalTimeout`
Global diag timeout in milliseconds

`parameters`
Structure with diag configuration parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_FLEXRAY__DIAG__CONFIG__UDS__FLAG__NONE`
No flag is set

`G_FLEXRAY__DIAG__CONFIG__UDS__FLAG__ENABLE_TESTER_PRESENT`
Enable Tester Present signal

`ResponseTimeout`
Normal response timeout in milliseconds, e.g. **200 milliseconds**

`ResponseTimeout_78`
Extended response timeout in milliseconds, e.g. **5100 milliseconds** (after reception of negative response code **0x78** - request correctly received - response pending)

`RequestRepetitions`
Number of request repetitions after response timeout, e.g. **2**

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`TesterPresent`
Structure with Tester Present parameters

These parameters are only required when commmand flag `G_FLEXRAY__DIAG__CONFIG__UDS__FLAG__ENABLE_TESTER_PRESENT` is set.

The structure contains the following elements:

`Cycle`
Cycle time in milliseconds

`RequestMode`
Request mode (see G_Diag_RequestMode_t)

`Length`
Tester Present data length in bytes (max. **8 bytes**)

`Data`
Tester Present data bytes (max. **8 bytes**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Diag_Config_Off

G_FlexRay_Diag_Config_Off — Deactivate diagnostics

Description

Use this command to deactivate diagnostics.

Definition

```
G_Error_t
G_FlexRay_Diag_Config_Off(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Diag multisession channel

A multisession channel is required if multiple applications are working with the diagnostics on the same interface. For each channel, the diagnostics can be configured with different parameters.

Usually, it is not neccessary to use multisession channels. It is recommended to use channel **`0`** in these cases.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

7 FrameTrigger

These commands are used to configure and delete FlexRay frame triggers.

G_FlexRay_FrameTrigger_Config

G_FlexRay_FrameTrigger_Config — Frame trigger configuration

Description

Use this command to configure a FlexRay frame trigger.

Definition

```
G_Error_t
G_FlexRay_FrameTrigger_Config(
 const G_PortHandle_t portHandle,
 const G_FlexRay_FrameTrigger_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Frame trigger parameters

The structure contains the following elements:

`Flags`
Command Flags

The following values are possible and can be combined:

`G_FLEXRAY__FRAME_TRIGGER__CONFIG__FLAG__NONE`
No flag is set

`G_FLEXRAY__FRAME_TRIGGER__CONFIG__FLAG__PAYLOAD_PREAMBLE`
Use payload preamble

`G_FLEXRAY__FRAME_TRIGGER__CONFIG__FLAG__ALLOW_DYNAMIC_LENGTH`
Allow dynamic length

`G_FLEXRAY__FRAME_TRIGGER__CONFIG__FLAG__ALWAYS_TRANSMIT`
Always transmit

`G_FLEXRAY__FRAME_TRIGGER__CONFIG__FLAG__USE_IMMEDIATE_PDU`
Use immediate PDU defined by `LPduId_ImmediatePduId`

`LPduId_ImmediatePduId`
If flag `G_FLEXRAY__FRAME_TRIGGER__CONFIG__FLAG__USE_IMMEDIATE_PDU` is set, this parameter contains the **Immediate PDU** ID.

If flag `G_FLEXRAY__FRAME_TRIGGER__CONFIG__FLAG__USE_IMMEDIATE_PDU` is not set, this parameter contains the **L-PDU** ID.

`SlotId`
Slot ID

`BaseCycle`
Base cycle

`CycleRepetition`
Cycle repetition

`Channel`
FlexRay channel

The following values are possible:

`G_FLEXRAY__CHANNEL__A`
Channel A

`G_FLEXRAY__CHANNEL__B`
Channel B

`G_FLEXRAY__CHANNEL__A_B`
Channel A and B

`G_FLEXRAY__CHANNEL__NO_CHANNEL`
No channel selected

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_FrameTrigger_Delete

G_FlexRay_FrameTrigger_Delete — Delete a frame trigger

Description

Use this command to delete a frame trigger.

Definition

```
G_Error_t
G_FlexRay_FrameTrigger_Delete(
 const G_PortHandle_t portHandle,
 const G_FlexRay_FrameTrigger_Delete_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command Flags

The following values are possible and can be combined:

`G_FLEXRAY__FRAME_TRIGGER__DELETE__FLAG__NONE`
No flag is set

`SlotId`
Slot ID

`BaseCycle`
Base cycle

`Channel`
FlexRay channel

The following values are possible:

`G_FLEXRAY__CHANNEL__A`
Channel A

`G_FLEXRAY__CHANNEL__B`
Channel B

`G_FLEXRAY__CHANNEL__A_B`
Channel A and B

`G_FLEXRAY__CHANNEL__NO_CHANNEL`
No channel selected

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

8 Message

These commands are used to control FlexRay messages.

G_FlexRay_Message_Tx_Define

G_FlexRay_Message_Tx_Define — Define FlexRay transmit message

Description

Define a FlexRay transmit message.

Definition

```
G_Error_t
G_FlexRay_Message_Tx_Define(
 const G_PortHandle_t portHandle,
 const G_FlexRay_Message_Tx_Define_Parameters_t * const parameters,
 G_FlexRay_Message_Handle_t * const messageHandle
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Structure with message parameters

The structure contains the following elements:

`SlotId`
Defines the FlexRay slot Identifier (1 - 2047)

`BaseCycle`
Defines the first communication cycle where the frame is sent (0 - 63)

`CycleRepetition`
Defines the number of communication cycles (after the first cycle) whenever the frame is sent again (valid numbers are: **1**, **2**, **4**, **8**, **16**, **32**, **64**)

`Channel`
FlexRay channel

The following values are possible:

`G_FLEXRAY__CHANNEL__A`
Channel A

`G_FLEXRAY__CHANNEL__B`
Channel B

`G_FLEXRAY__CHANNEL__A_B`
Channel A and B

`G_FLEXRAY__CHANNEL__NO_CHANNEL`
No channel selected

`PayloadLength`
Defines the length of the payload data segment in **two-byte-words** (0 - 127)

`CycleTime`
Defines the time between two transmission cycles of this message (in milliseconds) (0 - 63)

If the cycle time has elapsed, a message will be transmitted at the next slot defined by `SlotId`, `BaseCycle` and `CycleRepetition`

`Flags`
Message Flags

The following values are possible and can be combined:

`G_FLEXRAY__MESSAGE__TX__UPDATE__FLAG__UNKNOWN_BIT`
Unknown bit (1st bit) (see FlexRay-Specification)

`G_FLEXRAY__MESSAGE__TX__UPDATE__FLAG__NONE`
No flag is set

`G_FLEXRAY__MESSAGE__TX__UPDATE__FLAG__PAYLOAD_PREAMBLE`
Indicates the existence of vector information in the payload segment of the data frame

For a static frame, it indicates **Network Management Vector** and for a dynamic frame, it indicates **Message Identifier**

`G_FLEXRAY__MESSAGE__TX__UPDATE__FLAG__NULL_FRAME`
This flag is deprecated

`G_FLEXRAY__MESSAGE__TX__UPDATE__FLAG__SYNC_FRAME`
This flag is deprecated

`G_FLEXRAY__MESSAGE__TX__UPDATE__FLAG__STARTUP_FRAME`
This flag is deprecated

`G_FLEXRAY__MESSAGE__TX__UPDATE__FLAG__AUTOSTART_TRANSMISSION`
Indicates that transmission starts automatically

`G_FLEXRAY__MESSAGE__TX__UPDATE__FLAG__REPEAT_TRANSMISSION`
Indicates that the Frame is transmitted repeatedly

(default: Frame is transmitted upon change)

`Payload`
Payload **Data byte 1..max. Data byte 254** (0..127 two-byte-words)

`messageHandle`
Returns the Handle to the FlexRay message

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Message_Tx_Update

G_FlexRay_Message_Tx_Update — Update FlexRay message parameters of an already defined FlexRay message

Description

Update the FlexRay message parameters of an already defined FlexRay message.

Definition

```
G_Error_t
G_FlexRay_Message_Tx_Update(
 const G_PortHandle_t portHandle,
 const G_FlexRay_Message_Handle_t  messageHandle,
 const u8_t  payloadLength,
 const u16_t  cycleTime,
 const G_FlexRay_Message_Tx_Update_Flags_t  flags,
 const u8_t * const payload
);
```

Parameters

`portHandle`
Handle to the communication port

`messageHandle`
Handle to a FlexRay message (can be retrieved with G_FlexRay_Message_Tx_Define)

`payloadLength`
Defines the length of the payload data segment in **two-byte-words** (**0** - **127**)

`cycleTime`
Defines the time between two transmission cycles of this message (in milliseconds) (0 - 63)

If the cycle time has elapsed, a message will be transmitted at the next slot defined by `SlotId`, `BaseCycle` and `CycleRepetition`

`flags`
Message Flags

The following values are possible and can be combined:

`G_FLEXRAY__MESSAGE__TX__UPDATE__FLAG__NONE`
No flag is set

`G_FLEXRAY__MESSAGE__TX__UPDATE__FLAG__UNKNOWN_BIT`
Unknown bit (1st bit) (see FlexRay-Specification)

`G_FLEXRAY__MESSAGE__TX__UPDATE__FLAG__PAYLOAD_PREAMBLE`
Indicates the existence of vector information in the payload segment of the data frame

For a static frame, it indicates **Network Management Vector** and for a dynamic frame, it indicates **Message Identifier**

`G_FLEXRAY__MESSAGE__TX__UPDATE__FLAG__NULL_FRAME`
Indicates that the data frame in the payload segment is NULL (**Static segment** only)

`G_FLEXRAY__MESSAGE__TX__UPDATE__FLAG__SYNC_FRAME`
Indicates that this frame is a synchronization frame

`G_FLEXRAY__MESSAGE__TX__UPDATE__FLAG__STARTUP_FRAME`
Indicates that this frame is a startup frame

`G_FLEXRAY__MESSAGE__TX__UPDATE__FLAG__AUTOSTART_TRANSMISSION`
Indicates that transmission starts automatically

`G_FLEXRAY__MESSAGE__TX__UPDATE__FLAG__REPEAT_TRANSMISSION`
Indicates that the Frame is transmitted repeatedly

(default: Frame is transmitted upon change)

`payload`
Payload **Data byte 1..max. Data byte 254** (0 to 127 two-byte-words)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Message_Tx_ChangeData

G_FlexRay_Message_Tx_ChangeData — Change data of a TX message

Description

Use this command to change the data of a FlexRay transmit message.

Definition

```
G_Error_t
G_FlexRay_Message_Tx_ChangeData(
 const G_PortHandle_t portHandle,
 const G_FlexRay_Message_Handle_t  messageHandle,
 const u16_t  numberOfMessagesBeforeRestore,
 const u8_t  payloadLength,
 const u8_t * const mask,
 const u8_t * const data
)
```

Parameters

`portHandle`
Handle to the communication port

`messageHandle`
Handle to a FlexRay message (can be retrieved with G_FlexRay_Message_Tx_Define)

`numberOfMessagesBeforeRestore`
Defines the number of messages that will be sent before the original message data is restored (**0** = do not restore original message data)

`payloadLength`
Defines the length of the payload data segment in **two-byte-words** (0..127)

`mask`
Data mask bytes (0..254) (0..127 two-byte-words)

For each set bit of the data mask, the corresponding data bit is changed to the bit-value specified by `data`

`data`
Data bytes (**0..254**) (**0..127 two-byte-words**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Message_Rx_Define

G_FlexRay_Message_Rx_Define — Define FlexRay receive message

Description

Define a FlexRay receive message.

Definition

```
G_Error_t
G_FlexRay_Message_Rx_Define(
 const G_PortHandle_t portHandle,
 const u16_t  slotId,
 const u8_t  baseCycle,
 const u8_t  cycleRepetition,
 const G_FlexRay_Channel_t  channel,
 G_FlexRay_Message_Handle_t * const messageHandle
);
```

Parameters

`portHandle`
Handle to the communication port

`slotId`
Defines the FlexRay slot identifier (**1** - **2047**)

`baseCycle`
Defines the first communication cycle where the frame is sent (0 - 63)

`cycleRepetition`
Defines the number of communication cycles (after the first cycle) whenever the frame is sent again (valid numbers are: **1**, **2**, **4**, **8**, **16**, **32**, **64**)

`channel`
FlexRay channel

The following values are possible:

`G_FLEXRAY__CHANNEL__A`
Channel A

`G_FLEXRAY__CHANNEL__B`
Channel B

`G_FLEXRAY__CHANNEL__A_B`
Channel A and B

`G_FLEXRAY__CHANNEL__NO_CHANNEL`
No channel selected

`messageHandle`
Returns the Handle to the FlexRay message

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Message_Rx_Read

G_FlexRay_Message_Rx_Read — Read last received FlexRay message

Description

Read the last received FlexRay message.

Definition

```
G_Error_t
G_FlexRay_Message_Rx_Read(
 const G_PortHandle_t portHandle,
 const G_FlexRay_Message_Handle_t  messageHandle,
 u16_t * const slotId,
 u8_t * const cycle,
 G_FlexRay_Channel_t * const channel,
 G_FlexRay_Message_Rx_Read_RspFlags_t * const rspFlags,
 u8_t * const payloadLength,
 u8_t * const payload
);
```

Parameters

`portHandle`
Handle to the communication port

`messageHandle`
Handle to a FlexRay message

`slotId`
Returns the FlexRay slot identifier of the received frame (**81** - **2047**)

`cycle`
Returns the communication cycle where the frame has been received (0 - 63)

`channel`
FlexRay channel

The following values are possible:

`G_FLEXRAY__CHANNEL__A`
Channel A

`G_FLEXRAY__CHANNEL__B`
Channel B

`G_FLEXRAY__CHANNEL__A_B`
Channel A and B

`G_FLEXRAY__CHANNEL__NO_CHANNEL`
No channel selected

`rspFlags`
Returns the flags of the received frame

The following values are possible and can be combined:

`G_FLEXRAY__MESSAGE__RX__READ__FLAG__NONE`
No flag is set

`G_FLEXRAY__MESSAGE__RX__READ__FLAG__UNKNOWN_BIT`
Unknown bit (1st bit) (see FlexRay-Specification)

`G_FLEXRAY__MESSAGE__RX__READ__FLAG__PAYLOAD_PREAMBLE`
Indicates the existence of vector information in the payload segment of the data frame

For a static frame, it indicates **Network Management Vector** and for a dynamic frame, it indicates **Message Identifier**

`G_FLEXRAY__MESSAGE__RX__READ__FLAG__NULL_FRAME`
Indicates that the data frame in the payload segment is NULL (**Static segment** only)

`G_FLEXRAY__MESSAGE__RX__READ__FLAG__SYNC_FRAME`
Indicates that this frame is a synchronization frame

`G_FLEXRAY__MESSAGE__RX__READ__FLAG__STARTUP_FRAME`
Indicates that this frame is a startup frame

`G_FLEXRAY__MESSAGE__RX__READ__FLAG__PAYLOAD_UPDATED`
Indicates that the payload of this frame has been updated

`payloadLength`
Returns the length of the payload data segment of the frame received in **two-byte-words** (**0** - **127**)

`payload`
Payload (Data byte 1..max. Data byte 254) (0..127 two-byte-words)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Message_Transmit

G_FlexRay_Message_Transmit — Update and transmit FlexRay message

Description

Update and transmit a FlexRay message.

Definition

```
G_Error_t
G_FlexRay_Message_Transmit(
 const G_PortHandle_t portHandle,
 const G_FlexRay_Message_Handle_t  messageHandle,
 const u8_t  payloadLength,
 const G_FlexRay_Message_Transmit_Flags_t  flags,
 const u8_t * const payload
);
```

Parameters

`portHandle`
Handle to the communication port

`messageHandle`
Handle to a FlexRay message

`payloadLength`
Defines the length of the payload data segment in **two-byte-words** (**0 - 127**)

`flags`
Message flags

The following values are possible and can be combined:

`G_FLEXRAY__MESSAGE__TRANSMIT__FLAG__NONE`
No flag set

`G_FLEXRAY__MESSAGE__RX__READ__FLAG__PAYLOAD_PREAMBLE`
Indicates the existence of vector information in the payload segment of the data frame

For a static frame, it indicates **Network Management Vector** and for a dynamic frame, it indicates **Message Identifier**

`payload`
Payload **Data byte 1..max. Data byte 254** (**0..127 two-byte-words**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Message_Delete

G_FlexRay_Message_Delete — Delete FlexRay message

Description

Delete a FlexRay message.

Definition

```
G_Error_t
G_FlexRay_Message_Delete(
 const G_PortHandle_t portHandle,
 const G_FlexRay_Message_Handle_t  messageHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

`messageHandle`
: Handle to a FlexRay message

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Message_GetHandle

G_FlexRay_Message_GetHandle — Query handle to a FlexRay message

Description

Query a handle to a FlexRay message.

Definition

```
G_Error_t
G_FlexRay_Message_GetHandle(
 const G_PortHandle_t portHandle,
 const u16_t  slotId,
 const u8_t  baseCycle,
 const u8_t  cycleRepetition,
 const G_FlexRay_Channel_t  channel,
 G_FlexRay_Message_Handle_t * const messageHandle
);
```

Parameters

`portHandle`
Handle to the communication port

`slotId`
Defines the FlexRay slot Identifier (**1** - **2047**)

`baseCycle`
Defines the first communication cycle where the frame is sent (0 - 63)

`cycleRepetition`
Defines the number of communication cycles (after the first cycle) whenever the frame is sent again (valid numbers are: **1**, **2**, **4**, **8**, **16**, **32**, **64**)

`channel`
FlexRay channel

The following values are possible:

`G_FLEXRAY__CHANNEL__A`
Channel A

`G_FLEXRAY__CHANNEL__B`
Channel B

`G_FLEXRAY__CHANNEL__A_B`
Channel A and B

`G_FLEXRAY__CHANNEL__NO_CHANNEL`
No channel selected

`messageHandle`
Returns a handle to the FlexRay message

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Message_Checksum_Crc_Define

G_FlexRay_Message_Checksum_Crc_Define — Define CRC checksum

Description

Define the **Cyclic Redundancy Check** (**CRC**) calculation routine to guarantee signal data integrity.

To achieve fast CRC calculation, the algorithm is based on a lookup table without augmented zero bytes.

The algorithm is only usable with polynomial orders of **8**, **16**, **24** or **32**.

Definition

```
G_Error_t
G_FlexRay_Message_Checksum_Crc_Define(
 const G_PortHandle_t portHandle,
 const u32_t  polynomial,
 const u32_t  initialValue,
 const u32_t  finalXorValue,
 const u8_t  order,
 const G_FlexRay_Message_Checksum_Crc_Flags_t  flags,
 G_FlexRay_Message_Checksum_Handle_t * const crcHandle
);
```

Parameters

`portHandle`
Handle to the communication port

`polynomial`
Defines the **Cyclic Redundancy Check** (**CRC**) **polynomial**

`initialValue`
Defines the initial **CRC** value belonging to that algorithm

`finalXorValue`
Defines the bit pattern for the final XOR operator

`order`
Defines the order of **CRC polynomial** (**8**, **16**, **24** or **32**)

`flags`
CRC flags

The following values are possible and can be combined:

`G_FLEXRAY__MESSAGE__CHECKSUM__CRC__FLAG__NONE`
No flag is set

`G_FLEXRAY__MESSAGE__CHECKSUM__CRC__FLAG__NONE_DIRECT`
Algorithm with augmented zero bits

Default: no augmented zero bits

`G_FLEXRAY__MESSAGE__CHECKSUM__CRC__FLAG__REFLECT_DATA_BYTES`
The data byte is reflected (reversed) before processing

`G_FLEXRAY__MESSAGE__CHECKSUM__CRC__FLAG__REFLECT_RESULT_BEFORE_XOR`
The **CRC** will be reflected (reversed) before the final XOR is applied

`crcHandle`
Returns the handle to the **CRC** calculation routine

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Message_Checksum_Crc_Assign

G_FlexRay_Message_Checksum_Crc_Assign — Assign CRC checksum to FlexRay message

Description

Add a **Cyclic Redundancy Check** (**CRC**) calculation routine to the FlexRay transmit message identified by the **messageHandle** parameter.

Definition

```
G_Error_t
G_FlexRay_Message_Checksum_Crc_Assign(
 const G_PortHandle_t portHandle,
 const G_FlexRay_Message_Handle_t  messageHandle,
 const G_FlexRay_Message_Checksum_Handle_t  crcHandle,
 const u8_t  calculationStartByte,
 const u8_t  numberOfBytes,
 const u8_t  crcStartByte,
 G_FlexRay_Message_Checksum_Handle_t * const assignedCrcHandle
);
```

Parameters

`portHandle`
Handle to the communication port

`messageHandle`
Handle to the FlexRay message

`crcHandle`
Handle to the **CRC** calculation routine

(Retrieved with G_FlexRay_Message_Checksum_Crc_Define)

`calculationStartByte`
Defines the payload data byte from that on **CRC** data is taken in ascending order (**0** - **253**)

`numberOfBytes`
Defines the number of payload data bytes from that **CRC** is calculated (1 - 254)

`crcStartByte`
Defines the payload data byte to that the **CRC** result is written

`assignedCrcHandle`
Returns the Handle to the assigned **CRC**

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Message_Ramp_Define

G_FlexRay_Message_Ramp_Define — Define and assign a ramp to a FlexRay message

Description

Defines and adds a **Ramp Counter** (alive counter) calculation routine to the FlexRay transmit message identified by the **messageHandle** parameter.

Definition

```
G_Error_t
G_FlexRay_Message_Ramp_Define(
 const G_PortHandle_t portHandle,
 const G_FlexRay_Message_Handle_t  messageHandle,
 const G_FlexRay_Message_Ramp_Define_Parameters_t * const parameters,
 G_FlexRay_Message_Ramp_Handle_t * const rampHandle
);
```

Parameters

`portHandle`
Handle to the communication port

`messageHandle`
Handle to the FlexRay message

`parameters`
Structure with ramp parameters

The structure contains the following elements:

`Signal`
Signal definition for the ramp (see G_Common_Signal_t)

`Mode`
Defines the ramp mode

The following parameters are possible:

`G_FLEXRAY__MESSAGE__RAMP__MODE__TRIANGULAR`
Triangular ramp

`G_FLEXRAY__MESSAGE__RAMP__MODE__SAWTOOTH`
Sawtooth ramp

`reserved`
reserved parameter (must be initialized with **0**)

`NumberOfFlanks`
Defines the number of flanks the ramp signal is generated for

`CycleTime`
Defines the cyclic time interval (in milliseconds) after that the ramp is recalculated

`Prescaler`
Defines the factor that prescales the ramp update rate based on the cyclic repetition of the FlexRay message that hosted the ramp

`InitialValue`
Defines the initial value of the ramp signal

`MinValue`
Defines the minimum value of the ramp signal

`MaxValue`
Defines the maximum value of the ramp signal

`Increment`
Defines the increment of the ramp signal as a floating point number

`Flags`
Defines the ramp flags

The following values are possible and can be combined:

`G_FLEXRAY__MESSAGE__RAMP__FLAG__NONE`
No flag is set

`G_FLEXRAY__MESSAGE__RAMP__FLAG__DIRECTION_UP`
Indicates that the ramp begins in up-counting direction

`G_FLEXRAY__MESSAGE__RAMP__FLAG__SYNCHRONOUS`
Indicates that the ramp signal updates synchronous to the FlexRay message repetition

`G_FLEXRAY__MESSAGE__RAMP__FLAG__AUTOSTART`
Indicates that the ramp signal generation starts automatically with the transmission of the associated FlexRay message

`rampHandle`
Returns the handle to the ramp definition

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Message_Ramp_Control

G_FlexRay_Message_Ramp_Control — Control a ramp

Description

Control a **Ramp Counter** (alive counter) calculation routine.

Definition

```
G_Error_t
G_FlexRay_Message_Ramp_Control(
 const G_PortHandle_t portHandle,
 const G_FlexRay_Message_Ramp_Handle_t  rampHandle,
 const G_FlexRay_Message_Ramp_Control_Mode_t  mode
);
```

Parameters

`portHandle`
Handle to the communication port

`rampHandle`
Handle to a Ramp definition

`mode`
Ramp control mode

The following values are possible:

`G_FLEXRAY__MESSAGE__RAMP__CONTROL__MODE__STOP`
Stops the generation of the signal ramp

`G_FLEXRAY__MESSAGE__RAMP__CONTROL__MODE__START`
Starts / continues the generation of the signal ramp

`G_FLEXRAY__MESSAGE__RAMP__CONTROL__MODE__RESTART`
Restarts the generation of the signal ramp

`G_FLEXRAY__MESSAGE__RAMP__CONTROL__MODE__REVERSE`
Reverses (reflects) the direction of the generated signal ramp

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

9 Module

These commands are used to control the FlexRay module parameters.

G_FlexRay_Module_GetInfo

G_FlexRay_Module_GetInfo — Returns the module information of the FlexRay controller

Description

This command returns the module information of the FlexRay controller.

Definition

```
G_Error_t
G_FlexRay_Module_GetInfo(
 const G_PortHandle_t  portHandle,
 u8_t * const protocolEngineVersion,
 u8_t * const hostInterfaceVersion
);
```

Parameters

`portHandle`
: Handle to the communication port

`protocolEngineVersion`
: Returns the version number of the protocol engine

`hostInterfaceVersion`
: Returns the version number of the controller host interface

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

10 Monitor

These commands are used to control the FlexRay monitor.

G_FlexRay_Monitor_Config

G_FlexRay_Monitor_Config — Configures the monitoring settings

Description

This command configures the monitoring settings.

Definition

```
G_Error_t
G_FlexRay_Monitor_Config(
 const G_PortHandle_t portHandle,
 const G_FlexRay_Monitor_Config_Flags_t  cmdFlags,
 const u32_t  apiBufferSize
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_FLEXRAY__MONITOR__CONFIG__FLAG__NONE`
No flag is set

`G_FLEXRAY__MONITOR__CONFIG__FLAG__MONITOR_NULL_FRAMES`
NULL frames received at individual message buffers will be added to the monitor data

`G_FLEXRAY__MONITOR__CONFIG__FLAG__MONITOR_RX_FIFO`
All frames received by the reception FIFO buffers will be added to the monitor data

`G_FLEXRAY__MONITOR__CONFIG__FLAG__MONITOR_RX_MESSAGES`
All frames received by individual message buffers will be added to the monitor data

`G_FLEXRAY__MONITOR__CONFIG__FLAG__MONITOR_TX_MESSAGES`
All frames transmitted by individual message buffers will be added to the monitor data

`G_FLEXRAY__MONITOR__CONFIG__FLAG__MONITOR_EVENTS`
Internal events of the FlexRay communication will be added to the monitor data

`G_FLEXRAY__MONITOR__CONFIG__FLAG__MONITOR_CHANNEL_A`
Enable monitoring for **FlexRay Channel A**

`G_FLEXRAY__MONITOR__CONFIG__FLAG__MONITOR_CHANNEL_B`
Enable monitoring for **FlexRay Channel B**

`G_FLEXRAY__MONITOR__CONFIG__FLAG__CLEAR_BUFFER_ON_START`
The monitor buffer is cleared when the monitor is started

`G_FLEXRAY__MONITOR__CONFIG__FLAG__CLEAR_BUFFER_ON_STOP`
The monitor buffer is cleared when the monitor is stopped

`G_FLEXRAY__MONITOR__CONFIG__FLAG__SORT_CHRONOLOGICAL`
The monitor entries will be sorted chronologically

`G_FLEXRAY__MONITOR__CONFIG__FLAG__ENABLE_START_EVENT`
Event **Monitor Start** is written to monitor buffer when monitor is started.

`G_FLEXRAY__MONITOR__CONFIG__FLAG__ENABLE_STOP_EVENT`
Event **Monitor Stop** is written to monitor buffer when monitor is stopped.

`apiBufferSize`
Defines the size of the api-internal monitor buffer in bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Monitor_Start

G_FlexRay_Monitor_Start — Start Monitor

Description

This command starts the monitoring.

Before the monitor can be started, it has to be configured with G_FlexRay_Monitor_Config.

Definition

```
G_Error_t
G_FlexRay_Monitor_Start(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Monitor_Stop

G_FlexRay_Monitor_Stop — Stop Monitor

Description

This command stops the monitoring.

After the monitor is stopped, the monitor configuration stays valid. It can be restarted with G_FlexRay_Monitor_Start without being configured again.

For releasing all monitor resources, command G_FlexRay_Monitor_Config_Off needs to be called.

Definition

```
G_Error_t
G_FlexRay_Monitor_Stop(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Monitor_GetData

G_FlexRay_Monitor_GetData — Get monitor data

Description

This command returns the received monitor data.

Definition

```
G_Error_t
G_FlexRay_Monitor_GetData(
 const G_PortHandle_t  portHandle,
 u8_t * const bufferOverrun,
 u32_t * const length,
 u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`bufferOverrun`
Returns whether a buffer overrun has occurred (`bufferOverrun` = 1) or not (`bufferOverrun` = 0)

`length`
On command call: length of data buffer in byte

On return: length of received monitor data

`data`
Returns monitor data

All received monitor items are returned in one block of data. Due to the dynamic data length of every entry, the user has to extract the monitor items manually from the data buffer.

Each monitor entry has the following structure:

Byte	Indication	Description
0..1	Length	Returns the length of the current FlexRay monitor entry
2..3	Type	Defines the type of the current FlexRay monitor entry The following values are possible: 0 - Unknown entry type 1 - RX frame 2 - TX frame 3 - Cycle start event 4 - POC state changed event 5 - User Event 6 - Wake Up signal received

Byte	Indication	Description
		7 - Syntax Error **8** - Content Error **9** - Boundary Violation **10** - Transmission Conflict **11** - Monitor Start **12** - Monitor Stop
4..11	TimeStamp	Returns the timestamp of the current FlexRay monitor entry (in nanoseconds)
12.. (Length-1)	Data	Data corresponding to the monitor entry type

Table 1 Structure of a FlexRay monitor entry

`Data` has to be interpreted dependent to the value of `Type` as explained in the following tables.

`Type = 1` (**RX frame**)

Byte	Indication	Description
12	Channel	Returns the FlexRay channel where the frame has been received The following values are possible: **0 - Channel A** **1 - Channel B** **2 - Channel A and Channel B**
13	FrameHeaderFlags	**Bit 0** - reserved **Bit 1** - PayloadPreambleIndicator Indicates the existence of vector information in the payload segment of the data frame For a static frame, it indicates **Network Management Vector** and for a dynamic frame, it indicates **Message Identifier** **Bit 2** - NullFrameIndicator Indicates that the data frame in the payload segment is NULL (**Static segment** only) **Bit 3** - SyncFrameIndicator Indicates that this frame is a synchronization frame **Bit 4** - StartUpFrameIndicator Indicates that this frame is a startup frame
14..15	SlotId	FlexRay slot Identifier of the received frame (1 - 2047)
16	Cycle	Communication cycle where the frame has been received (0 - 63)

Byte	Indication	Description
17	PayloadLength	length of the payload data segment of the frame received in **two-byte-words** (0 - 127)
18..19	HeaderCrc	Header **CRC** (**Cyclic Redundancy Check**)
20..273	Data **1..max. Data 254**	Payload **Data byte 1..max. Data byte 254** (0 to 127 two-byte-words)

Table 2 Data part of a FlexRay RX frame monitor entry

The FlexRay payload segment contains **0** to **254** bytes (**0** to **127 two-byte-words**) of data. As the **PayloadLength** parameter contains the number of **two-byte-words** , the payload **Data** contains an even number of bytes.

Type = **2** (**TX frame**)
(see Rx Frame)

Type = **3** (**Cycle Start Event**)

Byte	Indication	Description
12	Cycle	FlexRay cycle

Table 3 Data part of a FlexRay Cycle Start Event monitor entry

Type = **4** (**POC State Changed Event**)

Byte	Indication	Description
12	POC state	New POC state The following values are possible: **0** : POC state **config** **1** : POC state **default config** **2** : POC state **default halt** **3** : POC state **normal active** **4** : POC state **normal passive** **5** : POC state **ready** **6** : POC state **startup** **7** : POC state **wakeup** **8** : POC state **unknown**

Table 4 Data part of a FlexRay POC State Changed Event monitor entry

Type = **5** (**User Event**)

Byte	Indication	Description
12	UserEvent	User Event ID
13	reserved0	Reserved byte
14	Length	Length of user event data, in bytes
15	reserved1	Reserved byte
16...	Data	User event data bytes

Table 5 Data part of a FlexRay User Event Event monitor entry

Type = **6** (**Wake Up signal received**)

Byte	Indication	Description
12	Channel	FlexRay channel

Table 6 Data part of a FlexRay Wake Up signal received Event monitor entry

Type = **7, 8, 9, 10** (**Syntax Error, Content Error, Boundary Violation, Transmission Conflict**)

Byte	Indication	Description
12	SlotId	Slot ID
14	Cycle	Cycle
15	Channel	FlexRay channel

Table 7 Data part of a FlexRay Syntax Error, Content Error, Boundary Violation, Transmission Conflict Event monitor entry

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Monitor_Config_Off

G_FlexRay_Monitor_Config_Off — Release Monitor resources

Description

This command releases the monitor resources.

Definition

```
G_Error_t
G_FlexRay_Monitor_Config_Off(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Monitor_EnableEvent

G_FlexRay_Monitor_EnableEvent — Enable monitor event

Description

This command enables the monitor to signal the event specified by `eventHandle` as soon as monitor data is received.

For general notes on **G-API** event handling, see Section 5, "Events".

Definition

```
G_Error_t
G_FlexRay_Monitor_EnableEvent(
 const G_PortHandle_t portHandle,
 const G_Common_EventHandle_t * const eventHandle
);
```

Parameters

`portHandle`
Handle to the communication port

`eventHandle`
Handle to the event

Each time monitor data is received, the specified handle is signaled.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Monitor_DisableEvent

G_FlexRay_Monitor_DisableEvent — Disable monitor event

Description

This command disables a monitor event.

After this function has been executed, no event will be signaled when new monitor data is received.

For general notes on **G-API** event handling, see Section 5, "Events".

Definition

```
G_Error_t
G_FlexRay_Monitor_DisableEvent(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Monitor_WriteUserEvent

G_FlexRay_Monitor_WriteUserEvent — Write user event into monitor buffer

Description

Use this command to write a user event into the monitor buffer.

A user event consists of an event identifier and event data bytes.

Definition

```
G_Error_t
G_FlexRay_Monitor_WriteUserEvent(
 const G_PortHandle_t portHandle,
 const u8_t  userEvent,
 const u8_t  length,
 const u8_t * const data
);
```

Parameters

`portHandle`
: Handle to the communication port

`userEvent`
: User Event Identifier

`length`
: Length of User Event Data in bytes

`data`
: User Event Data bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Monitor_GetTime

G_FlexRay_Monitor_GetTime — Return current monitor time and monitor start time

Description

Use this command to query the current monitor time and the monitor start time.

Definition

```
G_Error_t
G_FlexRay_Monitor_GetTime(
 const G_PortHandle_t portHandle,
 G_FlexRay_Monitor_GetTime_RspFlags_t * const rspFlags,
 u64_t * const monitorStartTime,
 u64_t * const currentMonitorTime
);
```

Parameters

`portHandle`
Handle to the communication port

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_FLEXRAY__MONITOR__GET_TIME__RSP_FLAG__NONE`
No flag is set

`G_FLEXRAY__MONITOR__GET_TIME__RSP_FLAG__MONITOR_STARTED`
The monitor has been started

`monitorStartTime`
Returns the start time of the monitor in nanoseconds

`currentMonitorTime`
Returns the current monitor time in nanoseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Monitor_DefineFilter

G_FlexRay_Monitor_DefineFilter — Define monitor filter

Description

Use this command to define a monitor filter

Definition

```
G_Error_t
G_FlexRay_Monitor_DefineFilter(
 const G_PortHandle_t portHandle,
 const G_FlexRay_Monitor_DefineFilter_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_FLEXRAY__MONITOR__DEFINE_FILTER__CMD_FLAG__NONE`
No flag is set

`Mode`
Filter mode

The following values are possible:

`G_FLEXRAY__MONITOR__DEFINE_FILTER__MODE__ACCEPT_ALL`
All available data is monitored

`G_FLEXRAY__MONITOR__DEFINE_FILTER__MODE__REJECT_ALL`
No data is monitored

`G_FLEXRAY__MONITOR__DEFINE_FILTER__MODE__ACCEPT_SLOT_ID`
The slot ID defined by `SlotId_Begin` is monitored

`G_FLEXRAY__MONITOR__DEFINE_FILTER__MODE__REJECT_SLOT_ID`
The slot ID defined by `SlotId_Begin` is not monitored

`G_FLEXRAY__MONITOR__DEFINE_FILTER__MODE__ACCEPT_SLOT_ID_RANGE`
The slot ID range defined by `SlotId_Begin` and `SlotId_End` is monitored

`G_FLEXRAY__MONITOR__DEFINE_FILTER__MODE__REJECT_SLOT_ID_RANGE`
The slot ID range defined by `SlotId_Begin` and `SlotId_End` is not monitored

`Channel`
FlexRay channel

The following values are possible:

`G_FLEXRAY__CHANNEL__A`
Channel A

`G_FLEXRAY__CHANNEL__B`
Channel B

`G_FLEXRAY__CHANNEL__A_B`
Channel A and B

`G_FLEXRAY__CHANNEL__NO_CHANNEL`
No channel selected

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`SlotId_Begin`
Defines a slot ID for monitoring a specific slot ID, or the start of a slot ID range for monitoring a slot ID range

`SlotId_End`
Defines the end of a slot ID range for monitoring a slot ID range

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Monitor_Property_SetById

G_FlexRay_Monitor_Property_SetById — Set the value of a FlexRay monitor property

Description

Use this command to set the value of a FlexRay monitor property.

Definition

```
G_Error_t
G_FlexRay_Monitor_Property_SetById(
 const G_PortHandle_t portHandle,
 const u32_t  id,
 const u32_t  value
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
FlexRay Monitor Property ID

The following values are possible:

`G_FLEXRAY__MONITOR__PROPERTY_ID__MONITOR_SORTING__HOLD_TIME__NS`
This parameter defines the minimum time that all monitor entries are hold in the monitor sorting buffer before the application can access them (in nanoseconds)

`G_FLEXRAY__MONITOR__PROPERTY_ID__MONITOR_START__DELAY_TIME__US`
This parameter defines the delay after which the monitor starts recording valid entries (in microseconds)

`G_FLEXRAY__MONITOR__PROPERTY_ID__MONITOR_STOP__DELAY_TIME__US`
This parameter defines the delay after which the monitor stops processing entries, all valid entries with time stamps older the monitor stop time will be dropped, time stamps earlier than the monitor stop time are still added to the monitor sorting buffer and can be accessed by the application (in microseconds)

`G_FLEXRAY__MONITOR__PROPERTY_ID__UNKNOWN`
This is a place holder

`value`
Property value to be set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Monitor_Property_GetById

G_FlexRay_Monitor_Property_GetById — Query the value of a FlexRay monitor property

Description

Use this command to query the value of a FlexRay monitor property.

Definition

```
G_Error_t
G_FlexRay_Monitor_Property_GetById(
 const G_PortHandle_t portHandle,
 const u32_t  id,
 u32_t * const value
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
FlexRay Monitor Property ID

The following values are possible:

`G_FLEXRAY__MONITOR__PROPERTY_ID__MONITOR_SORTING__HOLD_TIME__NS`
This parameter defines the minimum time that all monitor entries are hold in the monitor sorting buffer before the application can access them (in nanoseconds)

`G_FLEXRAY__MONITOR__PROPERTY_ID__MONITOR_START__DELAY_TIME__US`
This parameter defines the delay after which the monitor starts recording valid entries (in microseconds)

`G_FLEXRAY__MONITOR__PROPERTY_ID__MONITOR_STOP__DELAY_TIME__US`
This parameter defines the delay after which the monitor stops processing entries, all valid entries with time stamps older the monitor stop time will be dropped, time stamps earlier than the monitor stop time are still added to the monitor sorting buffer and can be accessed by the application (in microseconds)

`G_FLEXRAY__MONITOR__PROPERTY_ID__UNKNOWN`
This is a place holder

`value`
Returns property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

11 L-PDU

These commands are used to configure and delete FlexRay L-PDUs. L-PDU stands for **L**ogical **P**rotocol **D**ata **U**nit.

G_FlexRay_LPdu_Config

G_FlexRay_LPdu_Config — L-PDU configuration

Description

Use this command to configure a FlexRay L-PDU (**L** ogical **P** rotocol **D** ata **U** nit).

Definition

```
G_Error_t
G_FlexRay_LPdu_Config(
 const G_PortHandle_t portHandle,
 const G_FlexRay_LPdu_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
L-PDU parameters

The structure contains the following elements:

`Flags`
Command Flags

The following values are possible and can be combined:

`G_FLEXRAY__L_PDU__CONFIG__FLAG__NONE`
No flag is set

`G_FLEXRAY__L_PDU__CONFIG__FLAG__SET_UNUSED_BITS`
Set unused bits to the value defined by `UnusedBitValue`

`LPduId`
ID of the L-PDU

This unique ID is used to identify the L-PDU for further actions.

`ByteLengthMax`
Maximum byte length of the L-PDU

`UnusedBitValue`
If flags `G_FLEXRAY__L_PDU__CONFIG__FLAG__SET_UNUSED_BITS` is set, unused bits in this L-PDU will be set to the value defined by this parameter.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_LPdu_Delete

G_FlexRay_LPdu_Delete — Delete a L-PDU

Description

Use this command to delete a L-PDU (**L** ogical **P** rotocol **D** ata **U** nit).

Definition

```
G_Error_t
G_FlexRay_LPdu_Delete(
 const G_PortHandle_t portHandle,
 const G_FlexRay_LPdu_Delete_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command Flags

The following values are possible and can be combined:

`G_FLEXRAY__L_PDU__DELETE__FLAG__NONE`
No flag is set

`LPduId`
ID of the L-PDU (as configured with G_FlexRay_LPdu_Config)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

12 MTS

These commands are used to control the FlexRay **M**edia **T**est **S**ymbol (**MTS**).

G_FlexRay_MTS_Send

G_FlexRay_MTS_Send — Send FlexRay media test symbol

Description

Send a **Media Test Symbol** (**MTS**) in the next available **Symbol window**

Definition

```
G_Error_t
G_FlexRay_MTS_Send(
 const G_PortHandle_t portHandle,
 const G_FlexRay_Channel_t  channel,
 const u8_t  cycleCounterValue,
 const u8_t  cycleCounterMask
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
FlexRay channel

The following values are possible:

`G_FLEXRAY__CHANNEL__A`
Channel A

`G_FLEXRAY__CHANNEL__B`
Channel B

`G_FLEXRAY__CHANNEL__A_B`
Channel A and B

`G_FLEXRAY__CHANNEL__NO_CHANNEL`
No channel selected

`cycleCounterValue`
Defines the filter value for the **MTS** cycle counter filter

`cycleCounterMask`
Defines the filter mask for the **MTS** cycle counter filter

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_MTS_GetState

G_FlexRay_MTS_GetState — Query FlexRay media test symbol state

Description

Query the status of the **Media Test Symbol** (**MTS**) reception in the **Symbol window** of a **FlexRay Communication cycle.**

Definition

```
G_Error_t
G_FlexRay_MTS_GetState(
 const G_PortHandle_t portHandle,
 const G_FlexRay_Channel_t  channel,
 G_FlexRay_MTS_State_t * const mtsState
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
FlexRay channel

The following values are possible:

`G_FLEXRAY__CHANNEL__A`
Channel A

`G_FLEXRAY__CHANNEL__B`
Channel B

`G_FLEXRAY__CHANNEL__A_B`
Channel A and B

`G_FLEXRAY__CHANNEL__NO_CHANNEL`
No channel selected

`mtsState`
Returns the status of the **Media Test Symbol** (**MTS**) reception in the **Symbol window** of a **FlexRay Communication cycle**

The following values are possible:

`G_FLEXRAY__MTS__STATE__RECEIVED`
Valid **MTS** has been received

`G_FLEXRAY__MTS__STATE__RECEIVED__SYNTAX_ERROR`
Valid **MTS** has been received but a syntax error was detected

`G_FLEXRAY__MTS__STATE__RECEIVED__BOUNDARY_VIOLATION`
Valid **MTS** has been received but a boundary violation was detected

`G_FLEXRAY__MTS__STATE__RECEIVED__SYNTAX_ERROR__BOUNDARY_VIOLATION`
Valid **MTS** has been received but a syntax error and a boundary violation were detected

`G_FLEXRAY__MTS__STATE__NOT_RECEIVED`
No valid **MTS** has been received

`G_FLEXRAY__MTS__STATE__NOT_RECEIVED__SYNTAX_ERROR`
No valid **MTS** has been received and a syntax error was detected

`G_FLEXRAY__MTS__STATE__NOT_RECEIVED__BOUNDARY_VIOLATION`
No valid **MTS** has been received and a boundary violation was detected

`G_FLEXRAY__MTS__STATE__NOT_RECEIVED__SYNTAX_ERROR__BOUNDARY_VIOLATION`
No valid MTS has been received and a syntax error and a boundary violation were detected

`G_FLEXRAY__MTS__STATE__UNKNOWN`
An error has occurred

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

13 Node

These commands are used to control general properties of the FlexRay node.

G_FlexRay_Node_GetState

G_FlexRay_Node_GetState — Query state of FlexRay interface node

Description

Query the state of a FlexRay interface node.

Definition

```
G_Error_t
G_FlexRay_Node_GetState(
 const G_PortHandle_t portHandle,
 G_FlexRay_Node_PocState_t * const pocState,
 G_FlexRay_Node_SyncState_t * const syncState,
 G_FlexRay_Node_WakeupState_t * const wakeupState,
 G_FlexRay_Node_TransceiverFlags_t * const transceiverFlagsA,
 G_FlexRay_Node_TransceiverFlags_t * const transceiverFlagsB
);
```

Parameters

`portHandle`
Handle to the communication port

`pocState`
Returns the **Protocol Operation Control** (**Protocol Operation Control**) state

The following values are possible:

`G_FLEXRAY__NODE__POC_STATE__CONFIG`
Configuration state

`G_FLEXRAY__NODE__POC_STATE__DEFAULT_CONFIG`
State prior to config, only left with an explicit configuration request

`G_FLEXRAY__NODE__POC_STATE__HALT`
Error state, can only be left by reinitialization

`G_FLEXRAY__NODE__POC_STATE__NORMAL_ACTIVE`
Normal operation

`G_FLEXRAY__NODE__POC_STATE__NORMAL_PASSIVE`
Errors detected, no transmission of data, but attempting to return to normal operation

`G_FLEXRAY__NODE__POC_STATE__READY`
State reached from `G_FLEXRAY__NODE__POC_STATE__CONFIG` after concluding the configuration

`G_FLEXRAY__NODE__POC_STATE__STARTUP`
The FlexRay module transmits only startup frames

`G_FLEXRAY__NODE__POC_STATE__WAKEUP`
The FlexRay module sends a wakeup pattern if it could not find anything on the bus

`G_FLEXRAY__NODE__POC_STATE__UNKNOWN`
Unknown state due to error

`syncState`
Returns the FlexRay **Communication Controller Synchronization state**

The following values are possible:

G_FLEXRAY__NODE__SYNC_STATE__ASYNC
The FlexRay communication controller is NOT synchronous to the FlexRay global time

G_FLEXRAY__NODE__SYNC_STATE__SYNC
The FlexRay communication controller is synchronous to the FlexRay global time

wakeupState
Returns the **Wakeup state**

The following values are possible:

G_FLEXRAY__NODE__WAKEUP_STATE__UNDEFINED
WAKEUPSTATE_UNDEFINED

G_FLEXRAY__NODE__WAKEUP_STATE__RECEIVED_HEADER
WAKEUPSTATE_RECEIVED_HEADER

G_FLEXRAY__NODE__WAKEUP_STATE__RECEIVED_WUP
WAKEUPSTATE_RECEIVED_WUP

G_FLEXRAY__NODE__WAKEUP_STATE__COLLISSION_HEADER
WAKEUPSTATE_COLLISION_HEADER

G_FLEXRAY__NODE__WAKEUP_STATE__COLLISION_WUP
WAKEUPSTATE_COLLISION_WUP

G_FLEXRAY__NODE__WAKEUP_STATE__COLLISION_UNKNOWN
WAKEUPSTATE_COLLISION_UNKNOWN

G_FLEXRAY__NODE__WAKEUP_STATE__TRANSMITTED
WAKEUPSTATE_TRANSMITTED

transceiverFlagsA
Returns the FlexRay Transceiver status flags (TJA1080) of the FlexRay Transceiver on **Channel A**

The following values are possible and can be combined:

G_FLEXRAY__NODE__TRANSCEIVER_FLAG__NONE
No flag is set

G_FLEXRAY__NODE__TRANSCEIVER_FLAG__LOCAL_WAKEUP
Local wake-up source flag

G_FLEXRAY__NODE__TRANSCEIVER_FLAG__REMOTE_WAKEUP
Remote wake-up source flag

G_FLEXRAY__NODE__TRANSCEIVER_FLAG__NODE_CONFIG
Node configuration flag

G_FLEXRAY__NODE__TRANSCEIVER_FLAG__POWER_ON
Power on status flag

G_FLEXRAY__NODE__TRANSCEIVER_FLAG__BUS_ERROR
Bus error status flag

G_FLEXRAY__NODE__TRANSCEIVER_FLAG__TEMP_HIGH
Temperature high status flag

G_FLEXRAY__NODE__TRANSCEIVER_FLAG__TEMP_MEDIUM
Temperature medium status flag

`G_FLEXRAY__NODE__TRANSCEIVER_FLAG__TXEN_BGE_CLAMPED`
TXEN_BGE clpd status flag

`G_FLEXRAY__NODE__TRANSCEIVER_FLAG__UVVBAT`
UVVBAT status flag

`G_FLEXRAY__NODE__TRANSCEIVER_FLAG__UVVCC`
UVVCC status flag

`G_FLEXRAY__NODE__TRANSCEIVER_FLAG__UVVIO`
UVVIO status flag

`G_FLEXRAY__NODE__TRANSCEIVER_FLAG__STAR_LOCKED`
Star-locked mode entered

`G_FLEXRAY__NODE__TRANSCEIVER_FLAG__TRXD_COLLISION`
TRXD collision detected

`transceiverFlagsB`
Returns the FlexRay Transceiver status flags (TJA1080) of the FlexRay Transceiver on **Channel B**

See FlexRay Transceiver Flags for a list of possible values

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Node_BusTermination_Enable

G_FlexRay_Node_BusTermination_Enable — Enable internal bus termination

Description

Use this command to enable the internal bus termination.

The internal bus termination is enabled by default. Use G_FlexRay_Node_BusTermination_Disable to disable the internal bus termination.

Internal bus termination is only available on dedicated devices.

If not supported by the device, the command will return with a firmware error code.

Definition

```
G_Error_t
G_FlexRay_Node_BusTermination_Enable(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Node_BusTermination_Disable

G_FlexRay_Node_BusTermination_Disable — Disable internal bus termination

Description

Use this command to disable the internal bus termination.

For re-enabling the internal bus termination, use G_FlexRay_Node_BusTermination_Enable.

Internal bus termination is only available on dedicated devices.

If not supported by the device, the command will return with a firmware error code.

Definition

```
G_Error_t
G_FlexRay_Node_BusTermination_Disable(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Node_SendWakeUp

G_FlexRay_Node_SendWakeUp — Send Wake Up Pattern

Description

Use this command to send a wake up pattern.

A wake up pattern is used to wake a cluster that is in sleep mode. It consists of a defined number of a wake up symbol that is composed of **n** bits transmitted at **LOW** level, followed by **m** bits transmitted at **IDLE** level.

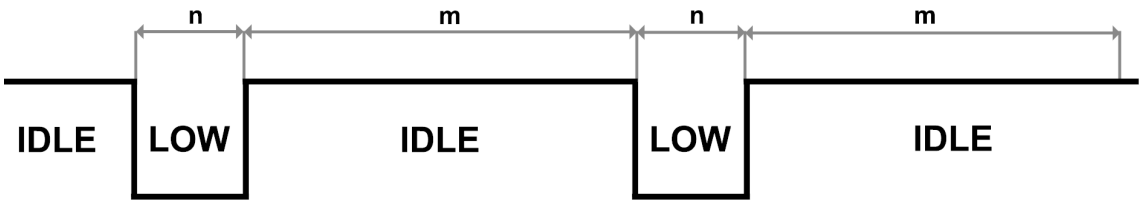

Figure 1 FlexRay Wake Up Pattern consisting of 2 Wake Up Symbols

The values for **n** and **m** and the number of wake up symbols are defined at configuration time (see Section 3, "Common").

Definition

```
G_Error_t
G_FlexRay_Node_SendWakeUp(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

14 N-PDU Pool

These commands are used to add N-PDus to different PDU Pools, used by the transport protocol.

N-PDU stands for **Network Layer Protocol Data Unit** . These PDUs are used by the transport protocol. Multiple N-PDUs can be assembled into one L-PDU (**Data Link Layer Protocol Data Unit**).

For the transport protocol it is neccessary to divide N-PDUs into several PDU Pools, used for different tasks (transmission and reception of physical and functional data).

G_FlexRay_NPduPool_Tx_AddNPdu

G_FlexRay_NPduPool_Tx_AddNPdu — Add a PDU to an N-PDU Pool

Description

Use this command to add a TX N-PDU to an N-PDU Pool that is used by the transport protocol.

At the moment only 1 to 1 mapping of **N-PDU** to **L-PDU** is supported, so the following has to be true: "`MaxLength_NPdu == MaxLength_LPdu`" and "`Offset == 0`".

Definition

```
G_Error_t
G_FlexRay_NPduPool_Tx_AddNPdu(
 const G_PortHandle_t portHandle,
 const u32_t  txNPduPoolId,
 const G_FlexRay_NPduPool_Tx_AddNPdu_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`txNPduPoolId`
ID of TX N-PDU Pool (starting with **0**)

`parameters`
Structure with PDU parameters

The structure contains the following parameters:

`Flags`
PDU flags

The following values are possible and can be combined:

`G_FLEXRAY__N_PDU_POOL__TX__ADD_N_PDU__FLAG__NONE`
No flag is set

`G_FLEXRAY__N_PDU_POOL__TX__ADD_N_PDU__FLAG__UPDATE_BIT_EXISTS`
The PDU contains an update bit

`SlotId`
FlexRay slot Identifier (1..2047)

`BaseCycle`
First communication cycle where the **L-PDU** is sent (0..63)

`CycleRepetition`
Number of communication cycles (after the first cycle) whenever the L-PDU is sent again (**1**, **2**, **4**, **8**, **16**, **32**, **64**)

`Channel`
FlexRay channel

The following values are possible:

`G_FLEXRAY__CHANNEL__A`
Channel A

`G_FLEXRAY__CHANNEL__B`
Channel B

`G_FLEXRAY__CHANNEL__A_B`
Channel A and B

`G_FLEXRAY__CHANNEL__NO_CHANNEL`
No channel selected

`MaxLength_LPdu`
Maximum number of data bytes for **L-PDU**

`MaxLength_NPdu`
Maximum number of data bytes for **N-PDU**

`Offset`
Offset of **N-PDU** within **L-PDU**

`PduUpdateBitPosition`
Position of update bit

Relevant if flag `G_FLEXRAY__N_PDU_POOL__TX__ADD_N_PDU__FLAG__UPDATE_BIT_EXISTS` is set. Range: **0..2031** and **0..((MaxLength_LPdu * 8) - 1)**

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_NPduPool_Tx_AddPdu

G_FlexRay_NPduPool_Tx_AddPdu — Add Pdu to TX NPdu Pool

Description

Use this command to add a TX PDU to an N-PDU Pool that is used by the transport protocol.

Definition

```
G_Error_t
G_FlexRay_NPduPool_Tx_AddPdu(
 const G_PortHandle_t portHandle,
 const u32_t  txNPduPoolId,
 const G_FlexRay_NPduPool_Tx_AddPdu_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`txNPduPoolId`
ID of TX N-PDU Pool (starting with **0**)

`parameters`
Structure with PDU parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_FLEXRAY__N_PDU_POOL__TX__ADD_PDU__FLAG__NONE`
No flag is set

`PduId`
ID of the PDU that has been configured with G_FlexRay_Pdu_Config

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_NPduPool_Tx_DeletePool

G_FlexRay_NPduPool_Tx_DeletePool — Delete TX N-PDU Pool

Description

Use this command to delete a TX N-PDU Pool that is used by the transport protocol.

Definition

```
G_Error_t
G_FlexRay_NPduPool_Tx_DeletePool(
 const G_PortHandle_t portHandle,
 const u32_t  txNPduPoolId
);
```

Parameters

`portHandle`
: Handle to the communication port

`txNPduPoolId`
: ID of TX N-PDU Pool (starting with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_NPduPool_Rx_AddNPdu

G_FlexRay_NPduPool_Rx_AddNPdu — Add a PDU to an N-PDU Pool

Description

Use this command to add an RX N-PDU to an N-PDU Pool that is used by the transport protocol.

Definition

```
G_Error_t
G_FlexRay_NPduPool_Rx_AddNPdu(
 const G_PortHandle_t portHandle,
 const u32_t  rxNPduPoolId,
 const G_FlexRay_NPduPool_Rx_AddNPdu_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`rxNPduPoolId`
ID of RX N-PDU Pool (starting with **0**)

`parameters`
Structure with PDU parameters

The structure contains the following parameters:

`Flags`
PDU flags

The following values are possible and can be combined:

`G_FLEXRAY__N_PDU_POOL__RX__ADD_N_PDU__FLAG__NONE`
No flag is set

`G_FLEXRAY__N_PDU_POOL__RX__ADD_N_PDU__FLAG__PDU_UPDATE_BIT_EXISTS`
Use parameter `PduUpdateBitPosition`

`SlotId`
FlexRay slot Identifier (1..2047)

`BaseCycle`
First communication cycle where the **L-PDU** is sent (0..63)

`CycleRepetition`
Number of communication cycles (after the first cycle) whenever the L-PDU is sent again (**1**, **2**, **4**, **8**, **16**, **32**, **64**)

`Channel`
FlexRay channel

The following values are possible:

`G_FLEXRAY__CHANNEL__A`
Channel A

`G_FLEXRAY__CHANNEL__B`
Channel B

`G_FLEXRAY__CHANNEL__A_B`
Channel A and B

`G_FLEXRAY__CHANNEL__NO_CHANNEL`
No channel selected

`Offset`
Offset of **N-PDU** within **L-PDU**

`MaxLength_LPdu`
Maximum number of data bytes for L-PDU

`MaxLength_NPdu`
Maximum number of data bytes for N-PDU

`PduUpdateBitPosition`
Relevant if flag `G_FLEXRAY__N_PDU_POOL__RX__ADD_N_PDU__FLAG__PDU_UPDATE_BIT_EXISTS` is set.

Range: **0..2031** and **0..((MaxLength_LPdu * 8) - 1)**

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`MaxLength_LPdu` and `MaxLength_NPdu` are allowed to be zero for backward compatibility. In this case all three parameters `Offset`, `MaxLength_LPdu` and `MaxLength_NPdu` have to be zero.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_NPduPool_Rx_AddPdu

G_FlexRay_NPduPool_Rx_AddPdu — Add Pdu to RX NPdu Pool

Description

Use this command to add an RX PDU to an N-PDU Pool that is used by the transport protocol.

Definition

```
G_Error_t
G_FlexRay_NPduPool_Rx_AddPdu(
 const G_PortHandle_t portHandle,
 const u32_t  rxNPduPoolId,
 const G_FlexRay_NPduPool_Rx_AddPdu_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`rxNPduPoolId`
ID of RX N-PDU Pool (starting with **0**)

`parameters`
Structure with PDU parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_FLEXRAY__N_PDU_POOL__RX__ADD_PDU__FLAG__NONE`
No flag is set

`PduId`
ID of the PDU that has been configured with G_FlexRay_Pdu_Config

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_NPduPool_Rx_DeletePool

G_FlexRay_NPduPool_Rx_DeletePool — Delete RX N-PDU Pool

Description

Use this command to delete an RX N-PDU Pool that is used by the transport protocol.

Definition

```
G_Error_t
G_FlexRay_NPduPool_Rx_DeletePool(
 const G_PortHandle_t portHandle,
 const u32_t  rxNPduPoolId
);
```

Parameters

`portHandle`
: Handle to the communication port

`rxNPduPoolId`
: ID of RX N-PDU Pool (starting with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

15 PDU

These commands are used to configure and delete FlexRay PDUs. PDU stands for **P**rotocol **D**ata **U**nit.

L-PDU stands for Data **L**ink Layer **PDU**.

N-PDU stands for **N**etwork Layer **PDU**.

I-PDU stands for **I**nteraction Layer **PDU**.

G_FlexRay_Pdu_Config

G_FlexRay_Pdu_Config — PDU configuration

Description

Use this command to configure a FlexRay PDU (**P** rotocol **D** ata **U** nit).

Definition

```
G_Error_t
G_FlexRay_Pdu_Config(
 const G_PortHandle_t portHandle,
 const G_FlexRay_Pdu_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
PDU parameters

The structure contains the following elements:

`Flags`
Command Flags

The following values are possible and can be combined:

`G_FLEXRAY__PDU__CONFIG__FLAG__NONE`
No flag is set

`PduId`
ID of the PDU

This unique ID is used to identify the PDU for further actions.

`ByteLengthMax`
Maximum byte length of the PDU

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Pdu_Delete

G_FlexRay_Pdu_Delete — Delete a PDU

Description

Use this command to delete a PDU (**P** rotocol **D** ata **U** nit).

Definition

```
G_Error_t
G_FlexRay_Pdu_Delete(
 const G_PortHandle_t portHandle,
 const G_FlexRay_Pdu_Delete_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command Flags

The following values are possible and can be combined:

`G_FLEXRAY__PDU__DELETE__FLAG__NONE`
No flag is set

`PduId`
ID of the PDU (as configured with G_FlexRay_Pdu_Config)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Pdu_Search

G_FlexRay_Pdu_Search — Query PDU information

Description

Use this command to query information about a specific PDU.

Definition

```
G_Error_t
G_FlexRay_Pdu_Search(
 const G_PortHandle_t portHandle,
 const G_FlexRay_Pdu_Search_Cmd_t * const cmd,
 G_FlexRay_Pdu_Search_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
The following values are possible and can be combined:

`G_FLEXRAY__PDU__SEARCH__CMD_FLAG__NONE`
No flag is set

`SlotId`
Slot ID of the PDU

`Channel`
FlexRay channel

The following values are possible:

`G_FLEXRAY__CHANNEL__A`
Channel A

`G_FLEXRAY__CHANNEL__B`
Channel B

`G_FLEXRAY__CHANNEL__A_B`
Channel A and B

`G_FLEXRAY__CHANNEL__NO_CHANNEL`
No channel selected

`Cycle`
PDU cycle

`PduBytePosition`
Start byte position of PDU within frame payload

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`rsp`
Returns response parameters

The stucture contains the following elements:

`FrameTrigger_Flags`
Frame Trigger flags

The following values are possible and can be combined:

`G_FLEXRAY__FRAME_TRIGGER__FLAG__NONE`
No flag is set

`G_FLEXRAY__FRAME_TRIGGER__FLAG__PAYLOAD_PREAMBLE`
0 : payload preamble bit is cleared

1 : payload preamble bit is set

`G_FLEXRAY__FRAME_TRIGGER__FLAG__ALLOW_DYNAMIC_LENGTH`
This flag is relevant for dynamic slots only

0 : LPDU / immediate PDU is transmitted with full LPDU / immediate PDU length

1 : LPDU / immediate PDU is transmitted with a dynamic LPDU / immediate PDU length

`G_FLEXRAY__FRAME_TRIGGER__FLAG__ALWAYS_TRANSMIT`
This flag should not be set

`G_FLEXRAY__FRAME_TRIGGER__FLAG__USE_IMMEDIATE_PDU`
0 : Use LPDU

1 : Use immediate PDU

`G_FLEXRAY__FRAME_TRIGGER__FLAG__SEND_NULL_FRAMES`
This flag should not be set

0 : Null frames are not sent

1 : If no LPDU / immediate PDU is ready for transmit, a null frame is transmitted (in static segment only)

`FrameTrigger_SlotId`
Slot ID of Frame Trigger

`FrameTrigger_BaseCycle`
Base cycle of Frame Trigger

`FrameTrigger_CycleRepetition`
Cycle repetition of Frame Trigger

`FrameTrigger_Channel`
FlexRay channel

The following values are possible:

G_FLEXRAY__CHANNEL__A
Channel A

G_FLEXRAY__CHANNEL__B
Channel B

G_FLEXRAY__CHANNEL__A_B
Channel A and B

G_FLEXRAY__CHANNEL__NO_CHANNEL
No channel selected

FrameTrigger_FillByteValue
Fill byte value of Frame Trigger

FrameTrigger_reserved1
reserved parameter (must be initialized with **0**)

FrameTrigger_reserved2
reserved parameter (must be initialized with **0**)

LPdu_Flags
L-PDU flags

The following values are possible and can be combined:

G_FLEXRAY__LPDU__FLAGS__NONE
No flag is set

G_FLEXRAY__LPDU__FLAG__SET_UNUSED_BYTES
If this flag is set, unused bytes are set to UnusedByteValue

G_FLEXRAY__LPDU__FLAG__SLOT_BASED_LPDU_ID
0 : "Normal" L-PDU ID

1 : Slot based L-PDU ID (for backward compatibility)

LPdu_Id
L-PDU ID

LPdu_ByteLengthMax
Maximum byte length of L-PDU

LPdu_UnusedByteValue
Value of unused bytes of L-PDU

LPdu_NumberOfPdus
Number of PDUs in L-PDU

LPdu_reserved
reserved parameter (must be initialized with **0**)

PduInLPdu_Flags
PDU in L-PDU flags

The following values are possible and can be combined:

G_FLEXRAY__PDU_IN_LPDU__FLAGS__NONE
No flag is set

G_FLEXRAY__PDU_IN_LPDU__FLAG__USE_UPDATE_BIT
Use update bit

`PduInLPdu_UpdateBitPosition`
Update bit position of PDU in L-PDU

`PduInLPdu_BytePosition`
Byte position of PDU in L-PDU

`PduInLPdu_reserved`
reserved parameter (must be initialized with **0**)

`Pdu_Flags`
PDU flags

The following values are possible and can be combined:

`G_FLEXRAY__PDU__FLAGS__NONE`
No flag is set

`G_FLEXRAY__PDU__FLAG__SLOT_BASED_PDU_ID`
0 : "Normal" PDU ID

1 : Slot based PDU ID (for backward compatibility)

`Pdu_Id`
PDU ID

`Pdu_ByteLengthMax`
Maximum byte length of PDU

`Pdu_reserved1`
reserved parameter (must be initialized with **0**)

`Pdu_reserved2`
reserved parameter (must be initialized with **0**)

`Pdu_reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

16 PDU in L-PDU

These commands are used to configure and delete PDUs of L-PDUs.

G_FlexRay_PduInLPdu_Config

G_FlexRay_PduInLPdu_Config — Add a PDU to an L-PDU

Description

Use this command to add a PDU to an L-PDU.

The PDU has to be configured with G_FlexRay_Pdu_Config before it can be added.

Definition

```
G_Error_t
G_FlexRay_PduInLPdu_Config(
 const G_PortHandle_t portHandle,
 const G_FlexRay_PduInLPdu_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command Flags

The following values are possible and can be combined:

`G_FLEXRAY__PDU_IN_LPDU__CONFIG__FLAG__NONE`
No flag is set

`G_FLEXRAY__PDU_IN_LPDU__CONFIG__FLAG__USE_UPDATE_BIT`
Use update bit defined by `UpdateBitPosition`

`PduId`
ID of the PDU that is added to the L-PDU

The PDU has to be configured with G_FlexRay_Pdu_Config before it can be added.

`LPduId`
ID of the L-PDU

The L-PDU has to be configured with G_FlexRay_LPdu_Config first.

`UpdateBitPosition`
Bit postition of the update bit inside the PDU if flag `G_FLEXRAY__PDU_IN_LPDU__CONFIG__FLAG__USE_UPDATE_BIT` is set

`BytePosition`
Byte position of the PDU within the L-PDU

`reserved`
reserved parameter (must be initialized with `0`)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_PduInLPdu_Delete

G_FlexRay_PduInLPdu_Delete — Remove PDU from L-PDU

Description

Use this command to remove a PDU from an L-PDU.

Definition

```
G_Error_t
G_FlexRay_PduInLPdu_Delete(
 const G_PortHandle_t portHandle,
 const G_FlexRay_PduInLPdu_Delete_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command Flags

The following values are possible and can be combined:

`G_FLEXRAY__PDU_IN_L_PDU__DELETE__FLAG__NONE`
No flag is set

`PduId`
ID of the PDU that is removed from the L-PDU (as configured with G_FlexRay_Pdu_Config)

`LPduId`
ID of the L-PDU (as configured with G_FlexRay_LPdu_Config)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

17 Startup

These commands are used to control the FlexRay startup behavior.

G_FlexRay_Startup_Config

G_FlexRay_Startup_Config — Configure startup-behavior of FlexRay interface

Description

This command configures the startup behavior of a FlexRay interface.

Definition

```
G_Error_t
G_FlexRay_Startup_Config(
 const G_PortHandle_t portHandle,
 const G_FlexRay_Startup_Config_Flags_t  cmdFlags,
 const u16_t  startupCycles,
 const u16_t  timeout,
 const u8_t  coldStartAttempts
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_FLEXRAY__STARTUP__CONFIG__FLAG__NONE`
No flag is set

`G_FLEXRAY__STARTUP__CONFIG__FLAG__ALLOW_COLD_START`
This node is permitted to initiate the cold start of the FlexRay cluster (**leading cold starter**)

`G_FLEXRAY__STARTUP__CONFIG__FLAG__ALLOW_AUTO_RESTART`
This node is permitted to automatically reattempt the startup of the FlexRay cluster after synchronization has been lost

If the node is configured not to allow communication to be halted due to severe clock calculation errors (**pAllowHaltDueToClock** is false), then this node will remain in the **POC:normal passive** state until schedule synchronization has been established. Therefore it does not reattempt to startup the cluster, even when `G_FLEXRAY__STARTUP__CONFIG__FLAG__ALLOW_AUTO_RESTART` flag is set.

`startupCycles`
Defines the maximum number of times that a node in this cluster is permitted to repeat the whole startup procedure after the configured number of allowed cold start attempts has been elapsed and none of these attempts was successful

`timeout`
Defines the timeout after the configured number of allowed cold start attempts has been elapsed before reentering the next startup cycle (in microseconds)

`coldStartAttempts`
Defines the maximum number of times that a node in this cluster is permitted to attempt to start the cluster by initiating schedule synchronization

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Startup_GetInfo

G_FlexRay_Startup_GetInfo — Get information about the FlexRay module startup

Description

This command returns information about the FlexRay module startup.

Definition

```
G_Error_t
G_FlexRay_Startup_GetInfo(
 const G_PortHandle_t portHandle,
 u16_t * const startupCycle,
 u8_t * const coldStartAttempt,
 G_FlexRay_Startup_GetInfo_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`startupCycle`
Returns the number of the start sequence cycle in that the FlexRay module has reached the **POC: normal active** state

POC: Protocol Operation Control

`coldStartAttempt`
Returns the number of the cold start attempt at which end the FlexRay module has reached the **POC: normal active** state

POC: Protocol Operation Control

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_FLEXRAY__STARTUP__GET_INFO__RSP_FLAG__NONE`
No flag is set

`G_FLEXRAY__STARTUP__GET_INFO__RSP_FLAG__LEADING_COLD_START_PATH`
The FlexRay module has reached the **POC: normal active** state via the leading cold start path

This indicates that this node has started the network

`G_FLEXRAY__STARTUP__GET_INFO__RSP_FLAG__LEADING_COLD_START_PATH_NOISE`
The FlexRay module has reached the **POC: normal active** state via the leading cold start path under noise conditions

This indicates there was some activity on the FlexRay bus while the FlexRay module was starting up the cluster

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

18 Tp

These commands are used to initialize **FlexRay Transport Protocol** functionality.

After the transport protocol has been initialized, it can be used by diagnostics protocols and functions of **g_api_tp.dll .**

The **Flexray Transport Protocol** uses **C-PDUs** which stands for **Communication-PDUs .**

G_FlexRay_Tp_Init_Iso10681

G_FlexRay_Tp_Init_Iso10681 — Initialize ISO10681 transport protocol.

Description

Use this command to initialize the **ISO10681** transport protocol

Definition

```
G_Error_t
G_FlexRay_Tp_Init_Iso10681(
 const G_PortHandle_t portHandle,
 const u8_t  channel,
 const G_FlexRay_Tp_Init_Iso10681_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Tp multisession channel

A multisession channel is required if multiple applications are working with the transport protocol on the same interface. For each channel, the transport protocol can be configured with different parameters.

Usually, it is not neccessary to use multisession channels. It is recommended to use channel **0** in these cases.

A multisession channel for the transport protocol can be obtained by G_Tp_Channel_Get .

`parameters`
Structure with transport protocol parameters

The structure contains the following elements:

`Flags`
Transport protocol flags

The following values are possible and can be combined:

`G_FLEXRAY__TP__INIT__ISO_10681__FLAG__NONE`
No flag is set

`G_FLEXRAY__TP__INIT__ISO_10681__FLAG__DISABLE_TX_SIDE__PHYSICAL`
Disable physical (unicast) transmissions (enabled by default)

`G_FLEXRAY__TP__INIT__ISO_10681__FLAG__DISABLE_RX_SIDE__PHYSICAL (enabled by default)`
Disable physical (unicast) reception (enabled by default)

`G_FLEXRAY__TP__INIT__ISO_10681__FLAG__DISABLE_TX_SIDE__FUNCTIONAL`
Disable functional (multicast) transmission (enabled by default)

`G_FLEXRAY__TP__INIT__ISO_10681__FLAG__ENABLE_RX_SIDE__FUNCTIONAL`
Enable functional (multicast) reception (disabled by default)

G_FLEXRAY__TP__INIT__ISO_10681__FLAG__ENABLE_TX_PHYSICAL_ACK
Enable acknowledge for physical transmission (disabled by default)

The receiver has to send a flow control to acknowledge the successful reception.

G_FLEXRAY__TP__INIT__ISO_10681__FLAG__USE_LOCAL_ADDRESS_PHY_RX_SIDE
Use parameter LocalAddress_Physical_RxSide.

G_FLEXRAY__TP__INIT__ISO_10681__FLAG__USE_REMOTE_ADDRESS_PHY_RX_SIDE
Use parameter RemoteAddress_Physical_RxSide.

G_FLEXRAY__TP__INIT__ISO_10681__FLAG__USE_LOCAL_ADDRESS_FUNC_RX_SIDE
Use parameter LocalAddress_Functional_RxSide.

G_FLEXRAY__TP__INIT__ISO_10681__FLAG__USE_REMOTE_ADDRESS_FUNC_RX_SIDE
Use parameter RemoteAddress_Functional_RxSide.

SourceAddress_Physical
Physical source address (for unicast communication)

TargetAddress_Physical
Physical target address (for unicast communication)

SourceAddress_Functional
Functional source address (for multicast communication)

TargetAddress_Functional
Functional target address (for multicast communication)

Timeout_As
Timeout for transmission of any **C-PDU** on sender side in microseconds (e.g. **100000**)

Timeout_Ar
Timeout for transmission of any **C-PDU** on receiver side in microseconds (e.g. **100000**)

Timeout_Bs
Timeout for reception of a **flow control C-PDU** on sender side in microseconds (e.g. **100000**)

Timeout_Cr
Timeout for reception of a **consecutive frame C-PDU** or **last frame C-PDU** on receiver side in microseconds (e.g. **100000**)

Time_Br
Time until transmission of a **flow control C-PDU** on receiver side in microseconds (e.g. **0**)

Time_Cs
Time until transmission of **consecutive frame C-PDU** or **last frame C-PDU** on sender side in microseconds (e.g. **0**)

BfS
BufferSize (ISO name = BfS) parameter transmitted in a **flow control CTS C-PDU** from receiver side (our side) to sender side (other side) (should be **0** , **0** signals the sender that no more **flow control frames** shall be sent during the transmission of the segmented message)

BC
Bandwidth Control (ISO name = BC) parameter transmitted in a **flow control CTS C-PDU** from receiver side (our side) to sender side (other side) (should be **0** , **0** = no bandwidth control)

`RetriesMax`
Retry counter (ISO name = C_RetriesMax) parameter

`MaxTxPduLength_Physical`
Maximum number of bytes used in a **C-PDU** for physical transmission (**0** = use maximum available N_PDU length, **else** : 8..254)

`MaxTxPduLength_Functional`
Maximum number of bytes used in a **C-PDU** for functional transmission (**0** = use maximum available N_PDU length, **else** : 8..254)

`TxPduPendingLimit`
0 : no limit of pending CF/LF Tx PDUs on physical Tx side, so transmission is started on each TriggerTransmit-call (if `TimeCs` or the received MNPC is 0 (no bandwidth control))

1 ≤ **N** ≤ **255** : number of pending CF/LF Tx PDUs on physical Tx side is limited (**N** = max number of transmissions started with a TriggerTransmit-call but not confirmed by a TxConfirmation-call)

`reserved2`
reserved parameter (must be initialized with **0**)

`TxNPduPoolId_Physical`
Pool identifier for N_PDUs that are used as **C_PDUs** for physical transmission

`TxNPduPoolId_Functional`
Pool identifier for N_PDUs that are used as **C_PDUs** for functional transmission

`RxNPduPoolId_Physical`
Pool identifier for N_PDUs that are used as **C_PDUs** for physical reception

`RxNPduPoolId_Functional`
Pool identifier for N_PDUs that are used as **C_PDUs** for functional reception

`LocalAddress_Physical_RxSide`
If flag `G_FLEXRAY__TP__INIT__ISO_10681__FLAG__USE_LOCAL_ADDRESS_PHY_RX_SIDE` is set, this parameter is the local address for physical communication on our RX side (parameter `SourceAddress_Physical` is the local address for physical communication on our TX side).

If flag `G_FLEXRAY__TP__INIT__ISO_10681__FLAG__USE_LOCAL_ADDRESS_PHY_RX_SIDE` is not set, this parameter is ignored (parameter `SourceAddress_Physical` is the local address for physical communication on our TX side and on our RX side).

`RemoteAddress_Physical_RxSide`
If flag `G_FLEXRAY__TP__INIT__ISO_10681__FLAG__USE_REMOTE_ADDRESS_PHY_RX_SIDE` is set, this parameter is the remote address for physical communication on our RX side (parameter `TargetAddress_Physical` is the remote address for physical communication on our TX side).

If flag `G_FLEXRAY__TP__INIT__ISO_10681__FLAG__USE_REMOTE_ADDRESS_PHY_RX_SIDE` is not set, this parameter is ignored (parameter `TargetAddress_Physical` is the remote address for physical communication on our TX side and on our RX side).

`LocalAddress_Functional_RxSide`
If flag `G_FLEXRAY__TP__INIT__ISO_10681__FLAG__USE_LOCAL_ADDRESS_FUNC_RX_SIDE` is set, this parameter is the local address for functional communication on our RX side (parameter `SourceAddress_Functional` is the local address for functional communication on our TX side).

If flag G_FLEXRAY__TP__INIT__ISO_10681__FLAG__USE_LOCAL_ADDRESS_FUNC_RX_SIDE is not set, this parameter is ignored (parameter SourceAddress_Functional is the local address for functional communication on our TX side and on our RX side).

RemoteAddress_Functional_RxSide

If flag G_FLEXRAY__TP__INIT__ISO_10681__FLAG__USE_REMOTE_ADDRESS_FUNC_RX_SIDE is set, this parameter is the remote address for functional communication on our RX side (parameter TargetAddress_Functional is the remote address for functional communication on our TX side).

If flag G_FLEXRAY__TP__INIT__ISO_10681__FLAG__USE_REMOTE_ADDRESS_FUNC_RX_SIDE is not set, this parameter is ignored (parameter TargetAddress_Functional is the remote address for functional communication on our TX side and on our RX side).

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_Tp_Init_Autosar

G_FlexRay_Tp_Init_Autosar — Initialize AUTOSAR transport protocol

Description

Use this command to initialize the **AUTOSAR** transport protocol

Definition

```
G_Error_t
G_FlexRay_Tp_Init_Autosar(
 const G_PortHandle_t portHandle,
 const u8_t  channel,
 const G_FlexRay_Tp_Init_Autosar_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Tp multisession channel

A multisession channel is required if multiple applications are working with the transport protocol on the same interface. For each channel, the transport protocol can be configured with different parameters.

Usually, it is not neccessary to use multisession channels. It is recommended to use channel **0** in these cases.

A multisession channel for the transport protocol can be obtained by G_Tp_Channel_Get .

`parameters`
Structure with transport protocol parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_FLEXRAY__TP__INIT__AUTOSAR__FLAG__NONE`
No flag is set

`G_FLEXRAY__TP__INIT__AUTOSAR__FLAG__DISABLE_TX_SIDE__PHYSICAL`
Disable physical sender side

`G_FLEXRAY__TP__INIT__AUTOSAR__FLAG__DISABLE_RX_SIDE__PHYSICAL`
Disable physical receiver side

`G_FLEXRAY__TP__INIT__AUTOSAR__FLAG__DISABLE_TX_SIDE__FUNCTIONAL`
Disable functional sender side

`G_FLEXRAY__TP__INIT__AUTOSAR__FLAG__ENABLE_RX_SIDE__FUNCTIONAL`
Enable functional receiver side

G_FLEXRAY__TP__INIT__AUTOSAR__FLAG__ALLOW_SEGMENTATION__FUNCTIONAL
Allow segmentation within a 1:n connection, see AUTOSAR type **FrTpGrpSeg**

G_FLEXRAY__TP__INIT__AUTOSAR__FLAG__ALLOW_CANCELLATION_FC_FRAMES
Allow flow control frames with flow state **CNLDO** (cancellation - data outdated), **CNLNB** (cancellation - no buffer) and **CNLOR** (cancellation - other reason) see AUTOSAR **Flow Control (FC)** (Specification of FlexRay Transport Layer V2.3.0 or older only)

G_FLEXRAY__TP__INIT__AUTOSAR__FLAG__USE_TIME_CS__PHYSICAL
If not set: use received `StMin` from remote node

If set: use parameter `TimeCs_Physical` as time between sending two consecutive frames on physical sender side

`AckType`
Acknowledge type (see AUTOSAR type **FrTpAckType**)

The following parameters are possible:

G_FLEXRAY__TP__AUTOSAR__ACK_TYPE__ACK_WITHOUT_RETRY
Acknowledge without retry

G_FLEXRAY__TP__AUTOSAR__ACK_TYPE__ACK_WITH_RETRY
Acknowledge with retry

G_FLEXRAY__TP__AUTOSAR__ACK_TYPE__NO_ACK
No acknowledge

G_FLEXRAY__TP__AUTOSAR__ACK_TYPE__UNKNOWN
This is just a placeholder

`AddressingType`
Addressing type (see AUTOSAR type **FrTpAdrType**)

The following values are possible:

G_FLEXRAY__TP__AUTOSAR__ADDRESSING_TYPE__ONE_BYTE
One byte

G_FLEXRAY__TP__AUTOSAR__ADDRESSING_TYPE__TWO_BYTES
Two bytes

G_FLEXRAY__TP__AUTOSAR__ADDRESSING_TYPE__UNKNOWN
This is just a placeholder

`reserved`
reserved parameter (must be initialized with **0**)

`LocalAddress_Physical`
See AUTOSAR type **FrTpLa**

`RemoteAddress_Physical`
See AUTOSAR type **FrTpRa**

`LocalAddress_Functional`
See AUTOSAR type **FrTpLa**

`RemoteAddress_Functional`
See AUTOSAR type **FrTpRa**

`TimeoutAs`
Timeout in microseconds between the PDU transmit request for the first PDU of the group used in the current connection of the Transport Layer to the FlexRay Interface and the corre-

sponding confirmation of the FlexRay Interface (when having sent the last PDU of the group used in this connection) on the sender side (**SF-x** , **FF-x** , **CF** or **FC** (in case of Transmit Cancellation)), see AUTOSAR type **FrTpTimeoutAs**

`TimeoutAr`
Timeout in microseconds between the PDU transmit request of the Transport Layer to the FlexRay Interface and the corresponding confirmation of the FlexRay Interface on the receiver side (for **FC** or **AF**), see AUTOSAR type **FrTpTimeoutAr**

`TimeoutBs`
Timeout in microseconds for waiting for an **FC** or **AF** on the sender side, see AUTOSAR type **FrTpTimeoutBs**

`TimeoutCr`
Timeout in microseconds for waiting for a **CF** or **FF-x** (in case of retry) after receiving the last **CF** or after sending a **FC** or **AF** on the receiver side, see AUTOSAR type **FrTpTimeoutCr**

`TimeBr_Physical`
Time in microseconds between receiving the last **CF** of a block or an **FF-x** (or **SF-x**) and sending out an **FC** or **AF** , see AUTOSAR type **FrTpTimeBr**

`TimeCs_Physical`
Time in microseconds between the sending of two consecutive **CFs** or between a **CF** and a **FC** (for Transmit Cancellation) or between reception of a **FC** or **AF** and sending of the next **CF** or a **FC** (for Transmit Cancellation), see AUTOSAR type **FrTpTimeCs**

This parameter is only relevant if flag `G_FLEXRAY__TP__INIT__AUTOSAR__FLAG__USE_TIME_CS__PHYSICAL` is set else the received value `STmin` from remote node is used.

`TimeCs_Functional`
Time in microseconds between the sending of two consecutive **CFs** or between **FF** and **CF** , see AUTOSAR type **FrTpTimeCs**

`MaxRetriesAr`
Maximum number of times of trying to send a frame when a **TIMEOUT AR** occurs (depending on whether retry is configured), see AUTOSAR type **FrTpMaxAr**

`MaxRetriesAs`
Maximum number of times of trying to send a frame when a **TIMEOUT AS** occurs (depending on whether retry is configured), see AUTOSAR type **FrTpMaxAs**

`MaxRetries_Physical`
Maximum number of retries (if retry is configured for the particular channel), see AUTOSAR type **FrTpMaxRn**

`MaxWaits_Physical`
Upper limit of allowed waits (flow control frame or acknowledge frame with flow status = wait), see AUTOSAR type **FRTP_MAX_BUFREQ**

`BlockSize_Physical`
0x00..0xFF when retry is deactived, **1..16** when retry is actived, see AUTOSAR parameter **FRTP_BS**

`StMin_Physical`
Minimum amount of time between two succeeding **CFs** in milliseconds (0x00..0x7F = 0..127ms or in microseconds (0xA1..0xA9 = 100..900us), our node is receiver and this value is sent to remote node, which has to delay the succeeding **CFs** , see AUTOSAR type **FRTP_STMIN**

`MaxTxPduLength_Physical`
Maximum number of bytes used in a PDU for physical transmission (**0** = use maximum available N_PDU length, **else** : 10..254)

This is not an AUTOSAR parameter, it is for testing purposes only and should be set to **0**.

`MaxTxPduLength_Functional`
Maximum number of bytes used in a PDU for functional transmission (**0** = use maximum available N_PDU length, **else** : 10..254)

This is not an AUTOSAR parameter, it is for testing purposes only and should be set to **0**.

`TxNPduPoolId_Physical`
Pool identifier for N_PDUs that are used as PDUs for physical transmission (see command G_FlexRay_NPduPool_Tx_AddNPdu)

`TxNPduPoolId_Functional`
Pool identifier for N_PDUs that are used as PDUs for functional transmission (see command G_FlexRay_NPduPool_Tx_AddNPdu)

`RxNPduPoolId_Physical`
Pool identifier for N_PDUs that are used as PDUs for physical reception (see command G_FlexRay_NPduPool_Rx_AddNPdu)

`RxNPduPoolId_Functional`
Pool identifier for N_PDUs that are used as PDUs for functional reception (see command G_FlexRay_NPduPool_Rx_AddNPdu)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

19 TX Fifo

The FlexRay TX Fifo is used to initiate the transmission of multiple PDUs.

G_FlexRay_TxFifo_Config_Pdu

G_FlexRay_TxFifo_Config_Pdu — Configure FlexRay TX Fifo

Description

Use this command to configure the FlexRay TX Fifo for sending standard PDUs.

See G_FlexRay_Pdu_Config for PDU configuration.

Definition

```
G_Error_t
G_FlexRay_TxFifo_Config_Pdu(
 const G_PortHandle_t portHandle,
 const G_FlexRay_TxFifo_Config_Pdu_Cmd_t * const cmd,
 G_FlexRay_TxFifo_Config_Pdu_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_FLEXRAY__TX_FIFO__CONFIG__PDU__FLAG__NONE`
No flag is set

`G_FLEXRAY__TX_FIFO__CONFIG__PDU__FLAG__GENERATE_TX_FIFO_ID`
The ID of the TX Fifo is generated automatically and returned in response parameter `TxFifoId`

`TxFifoId`
ID of the TX Fifo that is created

This parameter is disregarded if flag `G_FLEXRAY__TX_FIFO__CONFIG__PDU__FLAG__GENERATE_TX_FIFO_ID` is set.

Possible range for user defined IDs: **0x00000000..0x7FFFFFFF**

`FifoDepth`
Specifies how many PDUs will fit into the TX Fifo

`TxPduPendingLimit`
Limit for pending TX PDUs

0 : no limit of pending TX PDUs, so transmission is started on each TriggerTransmit-Call

1 <= **N** <= **255** : number of pending TX PDUs is limited (**N** = max number of transmissions started with a TriggerTransmit-Call but not confirmed by a TxConfirmation-Call)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`PduId`
ID of TX PDUs

`reserved4`
reserved parameter (must be initialized with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`TxFifoId`
ID of the created TX Fifo

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_TxFifo_Config_NPduPool

G_FlexRay_TxFifo_Config_NPduPool — Configure FlexRay TX Fifo

Description

Use this command to configure the FlexRay TX Fifo for sending N PDU Pool PDUs.

See G_FlexRay_TxFifo_Config_NPduPool for N PDU Pool configuration.

Definition

```
G_Error_t
G_FlexRay_TxFifo_Config_NPduPool(
 const G_PortHandle_t portHandle,
 const G_FlexRay_TxFifo_Config_NPduPool_Cmd_t * const cmd,
 G_FlexRay_TxFifo_Config_NPduPool_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_FLEXRAY__TX_FIFO__CONFIG__N_PDU_POOL__FLAG__NONE`
No flag is set

`G_FLEXRAY__TX_FIFO__CONFIG__N_PDU_POOL__FLAG__GENERATE_TX_FIFO_ID`
The ID of the TX Fifo is generated automatically and returned in response parameter `TxFifoId`

`TxFifoId`
ID of the TX Fifo that is created

This parameter is disregarded if flag `G_FLEXRAY__TX_FIFO__CONFIG__N_PDU_POOL__FLAG__GENERATE_TX_FIFO_ID` is set.

Possible range for user defined IDs: **0x00000000..0x7FFFFFFF**

`FifoDepth`
Specifies how many PDUs will fit into the TX Fifo

`TxPduPendingLimit`
Limit for pending TX PDUs

0 : no limit of pending TX PDUs, so transmission is started on each TriggerTransmit-Call

1 <= **N** <= **255** : number of pending TX PDUs is limited (**N** = max number of transmissions started with a TriggerTransmit-Call but not confirmed by a TxConfirmation-Call)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`TxNPduPoolId`
ID of the TX N PDU Pool that shall be used for transmission

`reserved4`
reserved parameter (must be initialized with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`TxFifoId`
ID of the created TX Fifo

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_TxFifo_Delete

G_FlexRay_TxFifo_Delete — Delete FlexRay TX Fifo

Description

Use this command to delete a FlexRay TX Fifo.

Definition

```
G_Error_t
G_FlexRay_TxFifo_Delete(
 const G_PortHandle_t portHandle,
 const G_FlexRay_TxFifo_Delete_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_FLEXRAY__TX_FIFO__DELETE__FLAG__NONE`
No flag is set

`G_FLEXRAY__TX_FIFO__DELETE__FLAG__DELETE_ALL`
Parameter `TxFifoId` is disregarded and all existing TX Fifos are deleted

`TxFifoId`
ID of the TX Fifo that shall be deleted

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FlexRay_TxFifo_Write

G_FlexRay_TxFifo_Write — Write FlexRay TX Fifo entries

Description

Use this command to write FlexRay TX Fifo entries.

Definition

```
G_Error_t
G_FlexRay_TxFifo_Write(
 const G_PortHandle_t portHandle,
 const u32_t  txFifoId,
 const u32_t  numberOfEntries,
 const G_FlexRay_TxFifo_Write_Entry_t * const entries
);
```

Parameters

`portHandle`
Handle to the communication port

`txFifoId`
ID of the FlexRay TX Fifo, obtained by G_FlexRay_TxFifo_Config_Pdu or G_FlexRay_TxFifo_Config_NPduPool

`numberOfEntries`
Number of FIFO entries that shall be written

`entries`
Array of FIFO entries that shall be written

The structure contains the following elements:

`Length`
Number of data bytes of the FIFO entry

The maximum number of data bytes is **254**.

`reserved`
reserved parameter (must be initialized with **0**)

`Data`
Data bytes of FIFO entry

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

File System Functions

These **G-API** functions provide access to the file system of your device.

1 Common

Common functions for the File System Interface

G_Fs_Rename

G_Fs_Rename — Rename file or directory

Description

Use this command to rename a file or a directory on the device.

Definition

```
G_Error_t
G_FS_Rename(
 const G_PortHandle_t  portHandle,
 const G_FS_Rename_CmdFlags_t  cmdFlags,
 const u8_t  rootDirectoryType,
 const char * const sourceName,
 const char * const destinationName,
 const u32_t  timeout
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_FS__RENAME__CMD_FLAG__NONE`
No flag is set

`rootDirectoryType`
Root directory type

The following values are possible:

`G_FS__ROOT_DIRECTORY_TYPE__USER_DIRECTORY`
The directory for **user files** is the root directory

`G_FS__ROOT_DIRECTORY_TYPE__REPLAY_DIRECTORY`
The directory for **replay files** is the root directory

`sourceName`
Name of the source file or directory

`destinationName`
Name of the destination file or directory

`timeout`
Timeout for the operation in milliseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

2 File

File operation functions

G_FS_File_Save

G_FS_File_Save — Save file

Description

Use this command to save a file on the device.

Definition

```
G_Error_t
G_FS_File_Save(
 const G_PortHandle_t  portHandle,
 const G_FS_File_Save_CmdFlags_t  cmdFlags,
 const u8_t  rootDirectoryType,
 const char * const sourceFilePath,
 const char * const destinationFilePath,
 const u32_t  timeout
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_FS__FILE__SAVE__CMD_FLAG__NONE`
No flag is set

`rootDirectoryType`
Root directory type

The following values are possible:

`G_FS__ROOT_DIRECTORY_TYPE__USER_DIRECTORY`
The directory for **user files** is the root directory

`G_FS__ROOT_DIRECTORY_TYPE__REPLAY_DIRECTORY`
The directory for **replay files** is the root directory

`sourceFilePath`
Path to the source file (on the host system)

`destinationFilePath`
Path to the destination file (on the device, including subdirectories)

timeout
Timeout for file operation in milliseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FS_File_Read

G_FS_File_Read — Read file

Description

Use this command to read a file from the device.

Definition

```
G_Error_t
G_FS_File_Read(
 const G_PortHandle_t  portHandle,
 const G_FS_File_Read_CmdFlags_t  cmdFlags,
 const u8_t  rootDirectoryType,
 const char * const sourceFilePath,
 const char * const destinationFilePath,
 const u32_t  timeout
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_FS__FILE__READ__CMD_FLAG__NONE`
No flag is set

`rootDirectoryType`
Root directory type

The following values are possible:

`G_FS__ROOT_DIRECTORY_TYPE__USER_DIRECTORY`
The directory for **user files** is the root directory

`G_FS__ROOT_DIRECTORY_TYPE__REPLAY_DIRECTORY`
The directory for **replay files** is the root directory

`sourceFilePath`
Path to the source file (on the device, including subdirectories)

`destinationFilePath`
Path to the destination file (on the host system)

timeout
Timeout for file operation in milliseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FS_File_Delete

G_FS_File_Delete — Delete file

Description

Use this command to delete a file from the device.

Definition

```
G_Error_t
G_FS_File_Delete(
 const G_PortHandle_t  portHandle,
 const G_FS_File_Delete_CmdFlags_t  cmdFlags,
 const u8_t  rootDirectoryType,
 const char * const fileName,
 const u32_t  timeout
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_FS__FILE__DELETE__CMD_FLAG__NONE`
No flag is set

`rootDirectoryType`
Root directory type

The following values are possible:

`G_FS__ROOT_DIRECTORY_TYPE__USER_DIRECTORY`
The directory for **user files** is the root directory

`G_FS__ROOT_DIRECTORY_TYPE__REPLAY_DIRECTORY`
The directory for **replay files** is the root directory

`fileName`
Path to the file (on the device, including subdirectories)

timeout
Timeout for file operation in milliseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FS_File_Copy

G_FS_File_Copy — Copy file

Description

Use this command to copy a file on the device.

Definition

```
G_Error_t
G_FS_File_Copy(
 const G_PortHandle_t  portHandle,
 const G_FS_File_Copy_CmdFlags_t  cmdFlags,
 const u8_t  rootDirectoryType,
 const char * const sourceFilePath,
 const char * const destinationFilePath,
 const u32_t  timeout
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_FS__FILE__COPY__CMD_FLAG__NONE`
No flag is set

`rootDirectoryType`
Root directory type

The following values are possible:

`G_FS__ROOT_DIRECTORY_TYPE__USER_DIRECTORY`
The directory for **user files** is the root directory

`G_FS__ROOT_DIRECTORY_TYPE__REPLAY_DIRECTORY`
The directory for **replay files** is the root directory

`sourceFilePath`
Path to the source file (on the device, including subdirectories)

`destinationFilePath`
Path to the destination file (on the device, including subdirectories)

`timeout`
Timeout for the operation (in milliseconds)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FS_File_Move

G_FS_File_Move — Move file

Description

Use this command to move a file on the device.

Definition

```
G_Error_t
G_FS_File_Move(
 const G_PortHandle_t  portHandle,
 const G_FS_File_Move_CmdFlags_t  cmdFlags,
 const u8_t  rootDirectoryType,
 const char * const sourceFilePath,
 const char * const destinationFilePath,
 const u32_t  timeout
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_FS__FILE__MOVE__CMD_FLAG__NONE`
No flag is set

`rootDirectoryType`
Root directory type

The following values are possible:

`G_FS__ROOT_DIRECTORY_TYPE__USER_DIRECTORY`
The directory for **user files** is the root directory

`G_FS__ROOT_DIRECTORY_TYPE__REPLAY_DIRECTORY`
The directory for **replay files** is the root directory

`sourceFilePath`
Path to the source file (on the device, including subdirectories)

`destinationFilePath`
Path to the destination file (on the device, including subdirectories)

`timeout`
Timeout for the operation (in milliseconds)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FS_File_GetNumberOfFiles

G_FS_File_GetNumberOfFiles — Get number of files

Description

Use this command to query the number of files of a directory on the device.

Definition

```
G_Error_t
G_FS_File_GetNumberOfFiles(
 const G_PortHandle_t  portHandle,
 const G_FS_File_GetNumberOfFiles_CmdFlags_t  cmdFlags,
 const u8_t  rootDirectoryType,
 const char * const directory,
 u32_t * const numberOfFiles
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_FS__FILE__GET_NUMBER_OF_FILES__CMD_FLAG__NONE`
No flag is set

`rootDirectoryType`
Root directory type

The following values are possible:

`G_FS__ROOT_DIRECTORY_TYPE__USER_DIRECTORY`
The directory for **user files** is the root directory

`G_FS__ROOT_DIRECTORY_TYPE__REPLAY_DIRECTORY`
The directory for **replay files** is the root directory

`directory`
Subdirectory within root directory

The directory name has to start with '/'. If no subdirectory is used, the value for `directory` has to be '/'.

`numberOfFiles`
Returns number of files

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FS_File_GetInfoByNumber

G_FS_File_GetInfoByNumber — Get file info

Description

Use this command to query information of a specific file on the device.

Definition

```
G_FS_File_GetInfoByNumber(
 const G_PortHandle_t  portHandle,
 const G_FS_File_GetInfoByNumber_CmdFlags_t  cmdFlags,
 const u8_t  rootDirectoryType,
 const char * const directory,
 const u32_t  fileNumber,
 u32_t * const fileSize,
 u32_t * const fileNameSize,
 char * const fileName,
 G_FS_File_GetInfoByNumber_Rsp_Times_t * const times
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_FS__FILE__GET_INFO_BY_NUMBER__CMD_FLAG__NONE`
No flag is set

`rootDirectoryType`
Root directory type

The following values are possible:

`G_FS__ROOT_DIRECTORY_TYPE__USER_DIRECTORY`
The directory for **user files** is the root directory

`G_FS__ROOT_DIRECTORY_TYPE__REPLAY_DIRECTORY`
The directory for **replay files** is the root directory

`directory`
Subdirectory within root directory

The directory name has to start with '/'. If no subdirectory is used, the value for `directory` has to be '/'.

`fileNumber`
Number of the file in the directory (starting with **0**)

`fileSize`
Returns size of the file in bytes

`fileNameSize`
in : Size of buffer `fileName` in bytes

out : Returns size of the file name in bytes

`fileName`
Returns file name as zero terminated string

`times`
Returns file access times

The structure contains the following elements:

`LastAccessTime`
Last access time

`LastModificationTime`
Last modification time

`LastStatusChangeTime`
Last status change time

Times in nano seconds since POSIX Epoch: 1970-01-01 00:00:00 +0000 (UTC)

Not every device has a realtime clock, therefore the time values may not be correct on some devices.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

3 Directory

Directory operation functions

G_FS_Directory_GetNumberOfDirectories

G_FS_Directory_GetNumberOfDirectories — Get number of directories

Description

Use this command to query the number of sub directories.

Definition

```
G_Error_t
G_FS_Directory_GetNumberOfDirectories(
 const G_PortHandle_t  portHandle,
 const G_FS_Directory_GetNumberOfDirectories_CmdFlags_t  cmdFlags,
 const u8_t  rootDirectoryType,
 const char * const parentDirectory,
 u32_t * const numberOfDirectories
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_FS__DIRECTORY__GET_NUMBER_OF_DIRECTORIES__CMD_FLAG__NONE`
No flag is set

`rootDirectoryType`
Root directory type

The following values are possible:

`G_FS__ROOT_DIRECTORY_TYPE__USER_DIRECTORY`
The directory for **user files** is the root directory

`G_FS__ROOT_DIRECTORY_TYPE__REPLAY_DIRECTORY`
The directory for **replay files** is the root directory

`parentDirectory`
Path to the parent directory

The path has to start with '/'. If no value for the parent directory is used, just use '/'.

`numberOfDirectories`
Returns the number of directories

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FS_Directory_GetNameByNumber

G_FS_Directory_GetNameByNumber — Get directory name by number

Description

Use this command to query the name of a specific directory.

Definition

```
G_Error_t
G_FS_Directory_GetNameByNumber(
 const G_PortHandle_t  portHandle,
 const G_FS_Directory_GetNameByNumber_CmdFlags_t  cmdFlags,
 const u8_t  rootDirectoryType,
 const char * const parentDirectory,
 const u32_t  directoryNumber,
 u32_t * const directoryNameSize,
 char * const directoryName
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_FS__DIRECTORY__GET_NAME_BY_NUMBER__CMD_FLAG__NONE`
No flag is set

`rootDirectoryType`
Root directory type

The following values are possible:

`G_FS__ROOT_DIRECTORY_TYPE__USER_DIRECTORY`
The directory for **user files** is the root directory

`G_FS__ROOT_DIRECTORY_TYPE__REPLAY_DIRECTORY`
The directory for **replay files** is the root directory

`parentDirectory`
Path to the parent directory

The path has to start with '/'. If no value for the parent directory is used, just use '/'.

`directoryNumber`
Number of the directory to be queried (starting with **0**)

`directoryNameSize`
in : Size of buffer `directoryName` in bytes

out : Returns the size of the directory name in bytes

`directoryName`
Returns the directory name

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FS_Directory_Create

G_FS_Directory_Create — Create directory

Description

Use this command to create a directory on the device

Definition

```
G_Error_t
G_FS_Directory_Create(
 const G_PortHandle_t  portHandle,
 const G_FS_Directory_Create_CmdFlags_t  cmdFlags,
 const u8_t  rootDirectoryType,
 const char * const directoryName,
 const u32_t  timeout
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_FS__DIRECTORY__CREATE__CMD_FLAG__NONE`
No flag is set

`rootDirectoryType`
Root directory type

The following values are possible:

`G_FS__ROOT_DIRECTORY_TYPE__USER_DIRECTORY`
The directory for **user files** is the root directory

`G_FS__ROOT_DIRECTORY_TYPE__REPLAY_DIRECTORY`
The directory for **replay files** is the root directory

`directoryName`
Name of the new directory

`timeout`
Timeout for the operation (in milliseconds)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_FS_Directory_Delete

G_FS_Directory_Delete — Delete directory

Description

Use this command to delete a directory on the device

Definition

```
G_Error_t
G_FS_Directory_Delete(
 const G_PortHandle_t  portHandle,
 const G_FS_Directory_Delete_CmdFlags_t  cmdFlags,
 const u8_t  rootDirectoryType,
 const char * const directoryName,
 const u32_t  timeout
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_FS__DIRECTORY__DELETE__CMD_FLAG__NONE`
No flag is set

`rootDirectoryType`
Root directory type

The following values are possible:

`G_FS__ROOT_DIRECTORY_TYPE__USER_DIRECTORY`
The directory for **user files** is the root directory

`G_FS__ROOT_DIRECTORY_TYPE__REPLAY_DIRECTORY`
The directory for **replay files** is the root directory

`directoryName`
Name of the directory

`timeout`
Timeout for the operation (in milliseconds)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Io Functions

These **G-API** functions provide access to the Io-Functionality of your hardware, including relays, inputs and outputs.

In addition to set the analog outputs of the device directly, it is possible to output an array of values with definable delays for creating signal curves.

After a power-on or software reset, all relays and outputs are reset.

1 Common

Common functions for the Io interface

G_Io_InitInterface

G_Io_InitInterface — Init Io interface

Description

This command resets the selected Io interface without software reset to the initial state. Additionally, further configuration possibilities are offered.

Definition

```
G_Error_t
G_Io_InitInterface(
 const G_PortHandle_t  portHandle,
 const G_Io_InitInterface_CmdFlags_t  cmdFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_IO__INIT_INTERFACE__CMD_FLAG__ZERO`
No flag is set

`G_IO__INIT_INTERFACE__CMD_FLAG__RESET_OUTPUTS`
Reset outputs

`G_IO__INIT_INTERFACE__CMD_FLAG__RESET_RELAYS`
Reset relays

`G_IO__INIT_INTERFACE__CMD_FLAG__RESET_TRIGGERS`
Reset triggers

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

2 BroadR-Reach Fault Injection

Commands for controlling the BroadR-Reach Fault Injection functionality.

BroadR-Reach Fault Injection is used for test cases that simulate faults like short-circuits or line level attenuation.

G_Io_BrrFault_Faults_Set

G_Io_BrrFault_Faults_Set — Set BroadR-Reach faults

Description

Use this command to configure BroadR-Reach faults.

Definition

```
G_Error_t
G_Io_BrrFault_Faults_Set(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Io_BrrFault_Faults_Set_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
BroadR-Reach channel, starting with `0` (for working with multiple BroadR-Reach channels on the same interface)

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_IO__BRR_FAULT__FAULTS__SET__CMD_FLAG__NONE`
No flag is set

`BrrPLineFault`
BroadR-Reach Plus-line-fault

The following values are possible:

`G_IO__BRR_FAULT__BRR_P_LINE_FAULT__UNKNOWN`
Unknown fault

`G_IO__BRR_FAULT__BRR_P_LINE_FAULT__NO_FAULT`
No fault

`G_IO__BRR_FAULT__BRR_P_LINE_FAULT__SHORT_CIRCUIT_TO_KL30`
Short-circuit to KL30

`G_IO__BRR_FAULT__BRR_P_LINE_FAULT__SHORT_CIRCUIT_TO_KL31`
Short-circuit to KL31

`G_IO__BRR_FAULT__BRR_P_LINE_FAULT__FLOATING`
Line is floating

`BrrMLineFault`
BroadR-Reach Minus-line-fault

The following values are possible:

`G_IO__BRR_FAULT__BRR_M_LINE_FAULT__UNKNOWN`
Unknown fault

`G_IO__BRR_FAULT__BRR_M_LINE_FAULT__NO_FAULT`
No fault

`G_IO__BRR_FAULT__BRR_M_LINE_FAULT__SHORT_CIRCUIT_TO_KL30`
Short-circuit to KL30

`G_IO__BRR_FAULT__BRR_M_LINE_FAULT__SHORT_CIRCUIT_TO_KL31`
Short-circuit to KL31

`G_IO__BRR_FAULT__BRR_M_LINE_FAULT__FLOATING`
Line is floating

`BrrCommonFault`
BroadR-Reach common fault

The following values are possible:

`G_IO__BRR_FAULT__COMMON_FAULT__UNKNOWN`
Unknown fault

`G_IO__BRR_FAULT__COMMON_FAULT__NO_FAULT`
No fault

`G_IO__BRR_FAULT__COMMON_FAULT__SHORT_CIRCUIT_BRR_P_BRR_M`
Short-circuit between Plus-line and Minus-line

`G_IO__BRR_FAULT__COMMON_FAULT__ATTENUATION_3_DB`
Level attenuation 3 dB

`G_IO__BRR_FAULT__COMMON_FAULT__ATTENUATION_10_DB`
Level attenuation 10 dB

`reserved`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_BrrFault_Faults_Get

G_Io_BrrFault_Faults_Get — Get BroadR-Reach fault configuration

Description

Use this command to query the current BroadR-Reach fault configuration.

Definition

```
G_Error_t
G_Io_BrrFault_Faults_Get(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Io_BrrFault_Faults_Get_Cmd_t * const cmd,
 G_Io_BrrFault_Faults_Get_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
BroadR-Reach channel, starting with `0` (for working with multiple BroadR-Reach channels on the same interface)

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_IO__BRR_FAULT__FAULTS__GET__CMD_FLAG__NONE`
No flag is set

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_IO__BRR_FAULT__FAULTS__GET__RSP_FLAG__NONE`
No flag is set

`G_IO__BRR_FAULT__FAULTS__GET__RSP_FLAG__BUSY`
The device is currently busy applying the fault configuration.

`BrrPLineFault`
BroadR-Reach Plus-line-fault

The following values are possible:

G_IO__BRR_FAULT__BRR_P_LINE_FAULT__UNKNOWN
Unknown fault

G_IO__BRR_FAULT__BRR_P_LINE_FAULT__NO_FAULT
No fault

G_IO__BRR_FAULT__BRR_P_LINE_FAULT__SHORT_CIRCUIT_TO_KL30
Short-circuit to KL30

G_IO__BRR_FAULT__BRR_P_LINE_FAULT__SHORT_CIRCUIT_TO_KL31
Short-circuit to KL31

G_IO__BRR_FAULT__BRR_P_LINE_FAULT__FLOATING
Line is floating

`BrrMLineFault`
BroadR-Reach Minus-line-fault

The following values are possible:

G_IO__BRR_FAULT__BRR_M_LINE_FAULT__UNKNOWN
Unknown fault

G_IO__BRR_FAULT__BRR_M_LINE_FAULT__NO_FAULT
No fault

G_IO__BRR_FAULT__BRR_M_LINE_FAULT__SHORT_CIRCUIT_TO_KL30
Short-circuit to KL30

G_IO__BRR_FAULT__BRR_M_LINE_FAULT__SHORT_CIRCUIT_TO_KL31
Short-circuit to KL31

G_IO__BRR_FAULT__BRR_M_LINE_FAULT__FLOATING
Line is floating

`BrrCommonFault`
BroadR-Reach common fault

The following values are possible:

G_IO__BRR_FAULT__COMMON_FAULT__UNKNOWN
Unknown fault

G_IO__BRR_FAULT__COMMON_FAULT__NO_FAULT
No fault

G_IO__BRR_FAULT__COMMON_FAULT__SHORT_CIRCUIT_BRR_P_BRR_M
Short-circuit between Plus-line and Minus-line

G_IO__BRR_FAULT__COMMON_FAULT__ATTENUATION_3_DB
Level attenuation 3 dB

G_IO__BRR_FAULT__COMMON_FAULT__ATTENUATION_10_DB
Level attenuation 10 dB

`reserved`
reserved parameter (must be initialized with **0**)

`Relays`
Relays state

bit0=Relay1, bit1=Relay2, bit2=Relay3, ...

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_BrrFault_Property_SetById

G_Io_BrrFault_Property_SetById — Set BroadR-Reach Fault Property value

Description

Use this command to set BroadR-Reach Fault Property values.

Definition

```
G_Error_t
G_Io_BrrFault_Property_SetById(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Io_BrrFault_PropertyId_t  propertyId,
 const u32_t  value
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
BroadR-Reach channel, starting with `0` (for working with multiple BroadR-Reach channels on the same interface)

`propertyId`
Property ID

The following values are possible:

`G_IO__BRR_FAULT__PROPERTY_ID__RELAYS`
Relays state

bit0=Relay1, bit1=Relay2, bit2=Relay3, ...

The property `G_IO__BRR_FAULT__PROPERTY_ID__DIRECT_RELAY_ACCESS_ENABLED` has to be set to `1` before the relays states can be altered.

`G_IO__BRR_FAULT__PROPERTY_ID__DIRECT_RELAY_ACCESS_ENABLED`
Enable direct access to relays

This property has to be set to `1` before the relays states can be altered with property `G_IO__BRR_FAULT__PROPERTY_ID__RELAYS`.

`value`
The property value to be set.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_BrrFault_Property_GetById

G_Io_BrrFault_Property_GetById — Get BroadR-Reach Fault Property value

Description

Use this command to query BroadR-Reach Fault Property values.

Definition

```
G_Error_t
G_Io_BrrFault_Property_GetById(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Io_BrrFault_PropertyId_t  propertyId,
 u32_t * const value
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
BroadR-Reach channel, starting with `0` (for working with multiple BroadR-Reach channels on the same interface)

`propertyId`
Property ID

The following values are possible:

`G_IO__BRR_FAULT__PROPERTY_ID__RELAYS`
Relays state

bit0=Relay1, bit1=Relay2, bit2=Relay3, ...

The property `G_IO__BRR_FAULT__PROPERTY_ID__DIRECT_RELAY_ACCESS_ENABLED` has to be set to `1` before the relays states can be altered.

`G_IO__BRR_FAULT__PROPERTY_ID__DIRECT_RELAY_ACCESS_ENABLED`
Enable direct access to relays

This property has to be set to `1` before the relays states can be altered with property `G_IO__BRR_FAULT__PROPERTY_ID__RELAYS`.

`value`
Returns the property value.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

3 Inputs

Functions for controlling the inputs of the Io interface

G_Io_Inputs_Analog_GetState

G_Io_Inputs_Analog_GetState — Query state of analog inputs

Description

Query the state of the analog inputs.

Definition

```
G_Error_t
G_Io_Inputs_Analog_GetState(
 const G_PortHandle_t  portHandle,
 const u16_t  numberOfInputs,
 const u8_t * const inputs,
 u32_t * const states
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfInputs`
Number of the analog inputs to read

`inputs`
Pointer to array with input numbers

Each array member represents the number of one input.

Input numbers start with **1**, therefore the first input is input number **1**.

`states`
Returns array with input values

Each array member represents the value of one input, corresponding to the input numbers in array `inputs`.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Inputs_Analog_GetNumberOfInputs

G_Io_Inputs_Analog_GetNumberOfInputs — Query number of available analog inputs

Description

Query the number of available analog inputs.

The number of available analog inputs depends on the device type.

Definition

```
G_Io_Inputs_Analog_GetNumberOfInputs(
 const G_PortHandle_t  portHandle,
 u16_t * const numberOfInputs
);
```

Parameters

`portHandle`
: Handle to the communication port

`numberOfInputs`
: Returns the number of available analog inputs for this device type

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Inputs_Digital_GetState

G_Io_Inputs_Digital_GetState — Query state of digital inputs

Description

Query the state of the digital inputs.

Definition

```
G_Error_t
G_Io_Inputs_Digital_GetState(
 const G_PortHandle_t  portHandle,
 const u16_t  numberOfInputs,
 const u8_t * const inputs,
 u8_t * const states
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfInputs`
Number of the digital inputs to read

`inputs`
Pointer to array with input numbers

Each array member represents the number of one input.

Input numbers start with **1**, therefore the first input is input number **1**.

`states`
Returns array with input values

Each array member represents the value of one input, corresponding to the input numbers in array `inputs`.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Inputs_Digital_GetNumberOfInputs

G_Io_Inputs_Digital_GetNumberOfInputs — Query number of available digital inputs

Description

Query the number of available digital inputs.

The number of available digital inputs depends on the device type.

Definition

```
G_Io_Inputs_Digital_GetNumberOfInputs(
 const G_PortHandle_t  portHandle,
 u16_t * const numberOfInputs
);
```

Parameters

`portHandle`
: Handle to the communication port

`numberOfInputs`
: Returns the number of available digital inputs for this device type

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

4 Outputs

Functions for controlling the outputs of the Io interface

G_Io_Outputs_Analog_GetNumberOfOutputs

G_Io_Outputs_Analog_GetNumberOfOutputs — Query number of analog outputs

Description

Use this function to query the number of analog outputs of the device.

Definition

```
G_Error_t
G_Io_Outputs_Analog_GetNumberOfOutputs(
 const G_PortHandle_t  portHandle,
 u16_t * const numberOfOutputs
);
```

Parameters

`portHandle`
: Handle to the communication port

`numberOfOutputs`
: Returns number of analog outputs

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Outputs_Analog_Set

G_Io_Outputs_Analog_Set — Set analog outputs

Description

Set analog outputs.

Definition

```
G_Error_t
G_Io_Outputs_Analog_Set(
 const G_PortHandle_t  portHandle,
 const u16_t  numberOfOutputs,
 const u16_t * const outputs,
 const u32_t * const values
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfOutputs`
Number of analog outputs to be set

`outputs`
Pointer to array with output numbers

Each array member represents the number of one output.

Output numbers start with **1**, therefore the first analog output is output number **1**.

`values`
Pointer to array with output values

Each array member represents the value of one output, corresponding to the output numbers in array `outputs`.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Outputs_Analog_Curve_Init

G_Io_Outputs_Analog_Curve_Init — Initialize a signal curve

Description

Use this function to initialize a **signal curve**.

A **signal curve** is a set of values with corresponding delays. It can be used to output user definable curves (e.g. a sine wave) at the analog outputs.

After a curve has been initialized, it can be started and stopped multiple times by calling G_Io_Outputs_Analog_Curve_Start and respectively G_Io_Outputs_Analog_Curve_Stop.

When the curve is not needed anymore, G_Io_Outputs_Analog_Curve_Reset needs to be called for freeing all allocated resources of this curve.

Definition

```
G_Error_t
G_Io_Outputs_Analog_Curve_Init(
 const G_PortHandle_t  portHandle,
 const G_Io_Outputs_Analog_Curve_Init_CmdFlags_t  cmdFlags,
 const u16_t  outputNumber,
 const u32_t  numberOfValues,
 const G_Io_Outputs_Analog_Curve_Values_t * const values,
 G_Io_Outputs_Analog_Curve_Handle_t * const curveHandle
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_IO__OUTPUTS__ANALOG__CURVE__INIT__CMD_FLAG__NONE`
No flag is set

`outputNumber`
Number of the analog output

Output numbers start with **1**, therefore the first analog output is output number **1**.

`numberOfValues`
Number of curve values in array `values`

`values`
Curve values

Each curve value consists of the following elements:

`Flags`
Curve flags

The following values are possible and can be combined:

`G_IO__OUTPUTS__ANALOG__CURVE__FLAG__NONE`
No flag is set

`reserved`
reserved parameter (must be initialized with **0**)

`Value`
Curve value

This value defines the **output voltage** of the analog output dependent on the **bit size** of the **Digital to Analog Converter (DAC)** of the device.

As an example, the value for the **highest possible output voltage** of a **10 Bit DAC** would be **1023**, for a **12 Bit DAC** it would be **4095**, etc...

`Delay`
Time until the next value is output, in milliseconds

`curveHandle`
Returns a handle to the allocated curve which can be used for all subsequent curve operations

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

In this example, analog output 1 is used to output a sine wave curve. The values representing the curve, lie between **0** to **1024** which is the value range of a **10 Bit DAC**.

```
#include <math.h>
#include <stdio.h>
#include <windows.h>
#include "g_api_common.h"
#include "g_api_io.h"


//----------------------------------------------------------------------------
// function:    ErrorHandler
// description: evaluates error code and closes application if an error
 occurred
// parameters:  portHandle - port handle
//              errorCode - error code
//----------------------------------------------------------------------------
static void
ErrorHandler(
 const G_PortHandle_t  portHandle,
 const G_Error_t  errorCode
)
{
 if (errorCode != G_NO_ERROR) {
  printf("Error: %s\n", G_GetLastErrorDescription());
  (void) G_Common_CloseInterface(portHandle);

  ExitProcess(1);
 }
}


//----------------------------------------------------------------------------
```

```
// function:    main
// description: main function
// parameters:  none
//---------------------------------------------------------------------------
void
main(
 void
)
{
 G_PortHandle_t  portHandle;
 G_Io_Outputs_Analog_Curve_Values_t  values[256];
 G_Io_Outputs_Analog_Curve_Values_t * value;
 const u32_t  numberOfValues = 256;
 G_Io_Outputs_Analog_Curve_Handle_t  curveHandle;
 u32_t  i;
 double  increment;
 G_Error_t  rc;

 // build sine wave curve
 increment = (2 * 3.14159) / numberOfValues;

 for (i = 0, value = values; i < numberOfValues; i++, value++) {
  value->Flags = G_IO__OUTPUTS__ANALOG__CURVE__FLAG__NONE;
  value->reserved = 0;
  value->Delay = 1;  // 1 milliseconds delay
  value->Value = (u32_t) ((512 * sin(i * increment)) + 512);
 }

 // open interface
 rc =
  G_Common_OpenInterface(
   "IO1",
   &portHandle
  );

 ErrorHandler(
  portHandle,
  rc
 );

 // initialize curve
 rc =
  G_Io_Outputs_Analog_Curve_Init(
   portHandle,
   G_IO__OUTPUTS__ANALOG__CURVE__INIT__CMD_FLAG__NONE,
   1,  // output number
   numberOfValues,
   values,
   &curveHandle
  );

 ErrorHandler(
  portHandle,
  rc
 );

 // start curve with unlimited repetitions
 rc =
```

```
  G_Io_Outputs_Analog_Curve_Start(
   portHandle,
   G_IO__OUTPUTS__ANALOG__CURVE__START__CMD_FLAG__NONE,
   0xFFFFFFFF,  // repeat unlimited
   curveHandle
  );

 ErrorHandler(
  portHandle,
  rc
 );

 Sleep(5000);  // wait for 5s

 // stop curve
 rc =
  G_Io_Outputs_Analog_Curve_Stop(
   portHandle,
   G_IO__OUTPUTS__ANALOG__CURVE__STOP__CMD_FLAG__NONE,
   curveHandle
  );

 ErrorHandler(
  portHandle,
  rc
 );

 // free all curve resources
 rc =
  G_Io_Outputs_Analog_Curve_Reset(
   portHandle,
   curveHandle
  );

 ErrorHandler(
  portHandle,
  rc
 );

 // close interface
 rc = G_Common_CloseInterface(portHandle);

 ErrorHandler(
  portHandle,
  rc
 );
}
```

G_Io_Outputs_Analog_Curve_Reset

G_Io_Outputs_Analog_Curve_Reset — Reset all curve related resources

Description

Use this command to reset all resources that were allocated when the curve was initialized.

For an example, see Curve Example.

Definition

```
G_Error_t
G_Io_Outputs_Analog_Curve_Reset(
 const G_PortHandle_t  portHandle,
 const G_Io_Outputs_Analog_Curve_Handle_t  curveHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

`curveHandle`
: Handle of the curve

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Outputs_Analog_Curve_Start

G_Io_Outputs_Analog_Curve_Start — Start a curve

Description

Use this command to start a curve that has been initialized with G_Io_Outputs_Analog_Curve_Init .

The curve can be stopped with G_Io_Outputs_Analog_Curve_Stop .

For an example, see Curve Example.

Definition

```
G_Error_t
G_Io_Outputs_Analog_Curve_Start(
 const G_PortHandle_t  portHandle,
 const G_Io_Outputs_Analog_Curve_Start_CmdFlags_t  cmdFlags,
 const u32_t  numberOfRepetitions,
 const G_Io_Outputs_Analog_Curve_Handle_t  curveHandle
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_IO__OUTPUTS__ANALOG__CURVE__START__CMD_FLAG__NONE`
No flag is set

`numberOfRepetitions`
Number of times the curve is repeated (0xFFFFFFFF = unlimited)

`curveHandle`
Curve handle, retrieved with G_Io_Outputs_Analog_Curve_Init

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Outputs_Analog_Curve_Stop

G_Io_Outputs_Analog_Curve_Stop — Stop a curve

Description

Use this command to stop a curve that has been started with G_Io_Outputs_Analog_Curve_Start .

For an example, see Curve Example.

Definition

```
G_Error_t
G_Io_Outputs_Analog_Curve_Stop(
 const G_PortHandle_t  portHandle,
 const G_Io_Outputs_Analog_Curve_Stop_CmdFlags_t  cmdFlags,
 const G_Io_Outputs_Analog_Curve_Handle_t  curveHandle
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_IO__OUTPUTS__ANALOG__CURVE__STOP__CMD_FLAG__NONE`
No flag is set

`curveHandle`
Curve handle, retrieved with G_Io_Outputs_Analog_Curve_Init

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Outputs_Digital_Set

G_Io_Outputs_Digital_Set — Set digital outputs

Description

Set digital outputs.

Definition

```
G_Error_t
G_Io_Outputs_Digital_Set(
 const G_PortHandle_t  portHandle,
 const G_Io_Outputs_Digital_Set_CmdFlags_t  cmdFlags,
 const u16_t  numberOfOutputs,
 const u8_t * const outputs
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_IO__OUTPUTS__DIGITAL__SET__CMD_FLAG__ZERO`
No flag is set

`G_IO__OUTPUTS__DIGITAL__SET__CMD_FLAG__SELECT_ALL`
All digital outputs will be set

`numberOfOutputs` and `outputs` will not be considered.

`numberOfOutputs`
Number of digital outputs to be set

`outputs`
Pointer to array with output numbers

Each array member represents the number of one output.

Output numbers start with **1**, therefore the first output is output number **1**.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Outputs_Digital_Reset

G_Io_Outputs_Digital_Reset — Reset digital outputs

Description

Reset digital outputs.

Definition

```
G_Error_t
G_Io_Outputs_Digital_Reset(
 const G_PortHandle_t  portHandle,
 const G_Io_Outputs_Digital_Reset_CmdFlags_t  cmdFlags,
 const u16_t  numberOfOutputs,
 const u8_t * const outputs
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_IO__OUTPUTS__DIGITAL__RESET__CMD_FLAG__ZERO`
No flag is set

`G_IO__OUTPUTS__DIGITAL__RESET__CMD_FLAG__SELECT_ALL`
All digital outputs will be reset.

`numberOfOutputs` and `outputs` will not be considered.

`numberOfOutputs`
Number of digital outputs to be reset

`outputs`
Pointer to array with output numbers

Each array member represents the number of one output.

Output numbers start with **1**, therefore the first output is output number **1**.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Outputs_Digital_SetDirect

G_Io_Outputs_Digital_SetDirect — Set or reset multiple digital outputs

Description

Set or reset multiple digital outputs.

Definition

```
G_Error_t
G_Io_Outputs_Digital_SetDirect(
 const G_PortHandle_t  portHandle,
 const u16_t  numberOfOutputBytes,
 const u8_t * const outputBytes
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfOutputBytes`
Number of bytes `outputBytes` points to

`outputBytes`
Array with output bytes

Each bit represents one output.
- byte 0 bit 0 = output 1
- byte 0 bit 1 = output 2
- ...

If a bit is set, the corresponding output will be set.

If a bit is not set, the corresponding output will be reset.

Output numbers start with **1**, therefore the first output is output number **1**.

Example

In the following example, 8 digital outputs will be set / reset in an alternating manner: output 1 = 0, output 2 = 1, output 3 = 0, output 4 = 1, output 5 = 0, output 6 = 1, output 7 = 0, output 8 = 1.

```
u16_t  numberOfOutputBytes = 1;
u8_t  outputBytes = 0xAA;

rc =
 G_Io_Outputs_Digital_SetDirect(
  portHandle,
  numberOfOutputBytes,
  &outputBytes
 );

if (rc != G_NO_ERROR) {
 return rc;
```

```
}
```

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Outputs_Digital_GetState

G_Io_Outputs_Digital_GetState — Query digital output state

Description

Query the state of the digital outputs.

Definition

```
G_Error_t
G_Io_Outputs_Digital_GetState(
 const G_PortHandle_t  portHandle,
 const u16_t  numberOfOutputs,
 const u8_t * const outputs,
 u8_t * const states
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfOutputs`
Number of digital outputs to be queried

`outputs`
Pointer to array with output numbers

Each array member represents the number of one output.

Output numbers start with **1**, therefore the first output is output number **1**.

`states`
Returns output states

Each array member represents the output state of the `outputs` array member with the same index.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Outputs_Digital_GetNumberOfOutputs

G_Io_Outputs_Digital_GetNumberOfOutputs — Query number of available digital outputs

Description

Query the number of available digital outputs.

The number of available digital outputs depends on the device type.

Definition

```
G_Io_Outputs_Digital_GetNumberOfOutputs(
 const G_PortHandle_t  portHandle,
 u16_t * const numberOfOutputs
);
```

Parameters

`portHandle`
: Handle to the communication port

`numberOfOutputs`
: Returns the number of available digital outputs for this device type

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

5 Relays

Functions for controlling the Relays of the device

On some **GOEPEL electronic** devices, the available relays are connected to the digital outputs. For accessing relays on these devices, **digital io** commands described in Section 4, "Outputs" have to be used. Please refer to the hardware manual of your device!

G_Io_Relays_Set

G_Io_Relays_Set — Set relays

Description

Set relays.

Definition

```
G_Error_t
G_Io_Relays_Set(
 const G_PortHandle_t  portHandle,
 const G_Io_Relays_Set_CmdFlags_t  cmdFlags,
 const u16_t  numberOfRelays,
 const u8_t * const relays
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command Flags

The following values are possible and can be combined:

`G_IO__RELAYS__SET__CMD_FLAG__ZERO`
No flag is set

`G_IO__RELAYS__SET__CMD_FLAG__SELECT_ALL`
All relays will be set

`numberOfRelays` and `relays` will not be considered.

`numberOfRelays`
Number of relays to be set

`relays`
Pointer to array with relay numbers

Each array member represents the number of one relay.

Relay numbers start with **1**, therefore the first relay is relay number **1**.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Relays_Reset

G_Io_Relays_Reset — Reset relays

Description

Reset relays.

Definition

```
G_Error_t
G_Io_Relays_Reset(
 const G_PortHandle_t  portHandle,
 const G_Io_Relays_Reset_CmdFlags_t  cmdFlags,
 const u16_t  numberOfRelays,
 const u8_t * const relays
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command Flags

The following values are possible and can be combined:

`G_IO__RELAYS__RESET__CMD_FLAG__ZERO`
No flag is reset

`G_IO__RELAYS__RESET__CMD_FLAG__SELECT_ALL`
All relays will be reset

`numberOfRelays` and `relays` will not be considered.

`numberOfRelays`
Number of relays to be reset

`relays`
Pointer to array with relay numbers

Each array member represents the number of one relay.

Relay numbers start with **1**, therefore the first relay is relay number **1**.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Relays_SetDirect

G_Io_Relays_SetDirect — Set or reset multiple relays

Description

Set or reset multiple relays.

Definition

```
G_Error_t
G_Io_Relays_SetDirect(
 const G_PortHandle_t  portHandle,
 const u16_t  numberOfRelayBytes,
 const u8_t * const relayBytes
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfRelayBytes`
Number of relays `relayBytes` points to

`relayBytes`
Array with relay bytes

Each bit represents one relay.
- byte 0 bit 0 = relay 1
- byte 0 bit 1 = relay 2
- ...

If a bit is set, the corresponding relay will be set.

If a bit is not set, the corresponding relay will be reset.

Relay numbers start with **1**, therefore the first relay is relay number **1**.

Example

In the following example, 8 relays will be set / reset in an alternating manner: relay 1 = 0, relay 2 = 1, relay 3 = 0, relay 4 = 1, relay 5 = 0, relay 6 = 1, relay 7 = 0, relay 8 = 1.

```
u16_t  numberOfRelayBytes = 1;
u8_t  relayBytes = 0xAA;

rc =
 G_Io_Relays_SetDirect(
  portHandle,
  numberOfRelayBytes,
  &relayBytes
 );

if (rc != G_NO_ERROR) {
 return rc;
```

```
}
```

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Relays_GetState

G_Io_Relays_GetState — Query relay states

Description

Query the state of the relays.

Definition

```
G_Error_t
G_Io_Relays_GetState(
 const G_PortHandle_t  portHandle,
 const u16_t  numberOfRelays,
 const u8_t * const relays,
 u8_t * const states
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfRelays`
Number of relays to be queried

`relays`
Pointer to array with relay numbers

Each array member represents the number of one relay.

Relay numbers start with **1**, therefore the first relay is relay number **1**.

`states`
Returns relay states

Each array member represents the relay state of the `relays` array member with the same index.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Relays_GetStateDirect

G_Io_Relays_GetStateDirect — Query relay states

Description

Query the state of the relays.

Definition

```
G_Io_Relays_GetStateDirect(
 const G_PortHandle_t  portHandle,
 const u16_t  numberOfRelayBytes,
 u8_t * const relayBytes
);
```

Parameters

`portHandle`
: Handle to the communication port

`numberOfRelayBytes`
: Number of relays `relayBytes` points to

`relayBytes`
: Returns Array with relay states

  Each bit represents one relay.
  - byte 0 bit 0 = relay 1
  - byte 0 bit 1 = relay 2
  - ...

  If a bit is set, the corresponding relay is ON.

  If a bit is not set, the corresponding relay is OFF.

Relay numbers start with **1**, therefore the first relay is relay number **1**.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Relays_GetNumberOfRelays

G_Io_Relays_GetNumberOfRelays — Query number of available relays

Description

Query the number of available relays.

The number of available relays depends on the device type.

Definition

```
G_Error_t
G_Io_Relays_GetNumberOfRelays(
 const G_PortHandle_t  portHandle,
 u16_t * const numberOfRelays
);
```

Parameters

`portHandle`
: Handle to the communication port

`numberOfRelays`
: Returns the number of available relays for this device type

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

6 SENT Transmitter

Commands for controlling the SENT transmitter functionality.

The **SAE J2716 SENT** (Single Edge Nibble Transmission) protocol is a point-to-point scheme for transmitting signal values from a sensor to a controller.

G_Io_Sent_Transmitter_Reset

G_Io_Sent_Transmitter_Reset — Reset SENT transmitter instance

Description

Use this command to reset a SENT transmitter instance.

Definition

```
G_Error_t
G_Io_Sent_Transmitter_Reset(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel, starting with **0** (for working with multiple SENT instances on the same interface)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Sent_Transmitter_Init

G_Io_Sent_Transmitter_Init — Initialize SENT Transmitter

Description

Use this command to initialize SENT Transmitter poperties.

Definition

```
G_Error_t
G_Io_Sent_Transmitter_Init(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Io_Sent_Transmitter_Init_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel, starting with **0** (for working with multiple SENT instances on the same interface)

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command Flags

The following values are possible and can be combined:

`G_IO__SENT__TRANSMITTER__INIT__CMD_FLAG__NONE`
No flag is set

`SerialMsgFormat`
Serial message format

The following values are possible:

`G_IO__SENT__TRANSMITTER__SERIAL_MSG_FORMAT__NO_SERIAL_MSG`
No serial message

`G_IO__SENT__TRANSMITTER__SERIAL_MSG_FORMAT__SHORT`
16 frames, 4 bit ID, 8 bit data, 4 bit CRC

`G_IO__SENT__TRANSMITTER__SERIAL_MSG_FORMAT__ENHANCED`
18 frames, 4 or 8 bit ID, 16 or 12 bit data, 6 bit CRC

`EnhancedSerialMsgConfig`
Enhanced serial message configuration

The following elements are possible:

`G_IO__SENT__TRANSMITTER__ENHANCED_SERIAL_MSG_CONFIG__12_DATA_8_ID`
12 bit data, 8 bit ID

G_IO__SENT__TRANSMITTER__ENHANCED_SERIAL_MSG_CONFIG__16_DATA_4_ID
16 bit data, 4 bit ID

CrcMode
CRC Mode

The following values are possible:

G_IO__SENT__TRANSMITTER__CRC_MODE__LEGACY
Legacy mode, as in **SAE J2016** revision **FEB2008** and older

G_IO__SENT__TRANSMITTER__CRC_MODE__RECOMMENDED
Recommended mode, as in **SAE J2716** revision **JAN2010**

TxMode
Transmission mode

The following values are possible:

G_IO__SENT__TRANSMITTER__TX_MODE__SINGLE_SHOT
Single shot - send only one frame

G_IO__SENT__TRANSMITTER__TX_MODE__CONTINUOUS
Continuous - continuous transmission of frames (cyclic transmission within fast channel)

MsgTxMode
Message transmission mode

The following values are possible:

G_IO__SENT__TRANSMITTER__MSG_TX_MODE__SINGLE_SHOT
Message transmission mode

The following values are possible:

G_IO__SENT__TRANSMITTER__MSG_TX_MODE__SINGLE_SHOT
Single shot - send only one serial message

G_IO__SENT__TRANSMITTER__MSG_TX_MODE__CONTINUOUS
Continuous - continuous transmission of serial messages (cyclic transmission within slow channel)

PulseMode
Pulse mode

The following values are possible:

G_IO__SENT__TRANSMITTER__PULSE_MODE__CONST_LOW_LENGTH
The low pulse has the constant length PulseLength

G_IO__SENT__TRANSMITTER__PULSE_MODE__CONST_HIGH_LENGTH
The high pulse has the constant length PulseLength

PauseMode
Pause mode

The following values are possible:

G_IO__SENT__TRANSMITTER__PAUSE_MODE__NO_PAUSE
In continuous TX mode no pause pulse is generated

In single shot TX mode an extra pulse (pause pulse) is generated to finish the frame

`G_IO__SENT__TRANSMITTER__PAUSE_MODE__CONST_PAUSE_LENGTH`
The length of the pause is constant, `PauseLength` is the length of the pause pulse

`G_IO__SENT__TRANSMITTER__PAUSE_MODE__CONST_CYCLE_LENGTH`
The length of the pause is dynamically, but the length of the cycle is constant

`PauseLength` is the length of the complete cycle

`NumberOfDataNibbles`
Number of data nibbles (1..6)

`ClockTickLength`
Clock tick length [microseconds] (3..90)

`PulseLength`
Pulse length in clock ticks (> 4), see `PulseMode`

`PauseLength`
Pause length in clock ticks, see `PauseMode`

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Sent_Transmitter_SendData

G_Io_Sent_Transmitter_SendData — Send SENT data

Description

Use this command to transmit SENT data.

Definition

```
G_Error_t
G_Io_Sent_Transmitter_SendData(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Io_Sent_Transmitter_SendData_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel, starting with **0** (for working with multiple SENT instances on the same interface)

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_IO__SENT__TRANSMITTER__SEND_DATA__CMD_FLAG__NONE`
No flag is set

`G_IO__SENT__TRANSMITTER__SEND_DATA__CMD_FLAG__START_TX`
Start the transmission

`NumberOfDataNibbles`
Number of data nibbles to be transmitted (1..6)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Data`
Data to be transmitted

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Sent_Transmitter_SendMsg

G_Io_Sent_Transmitter_SendMsg — Send SENT message in slow channel

Description

Use this command to send a SENT message in the slow channel.

Definition

```
G_Error_t
G_Io_Sent_Transmitter_SendMsg(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Io_Sent_Transmitter_SendMsg_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel, starting with **0** (for working with multiple SENT instances on the same interface)

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_IO__SENT__TRANSMITTER__SEND_MSG__CMD_FLAG__NONE`
No flag is set

`G_IO__SENT__TRANSMITTER__SEND_MSG__CMD_FLAG__START_TX`
Start the transmission

`Id`
Message ID

`reserved`
reserved parameter (must be initialized with **0**)

`Data`
Message data to be transmitted

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

7 SPI Analyzer

Commands for controlling the SPI Analyzer functionality.

G_Io_Spi_A_Reset

G_Io_Spi_A_Reset — Reset SPI Analyzer

Description

Use this functions to reset the SPI Analyzer.

Definition

```
G_Error_t
G_Io_Spi_A_Reset(
 const G_PortHandle_t  portHandle,
 const u8_t  nodeId
);
```

Parameters

`portHandle`
: Handle to the communication port

`nodeId`
: SPI Analyzer node ID

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Spi_A_Init

G_Io_Spi_A_Init — Initialize SPI Analyzer

Description

Use this command to initialize the SPI Analyzer.

Definition

```
G_Error_t
G_Io_Spi_A_Init(
 const G_PortHandle_t  portHandle,
 const u8_t  nodeId,
 const G_Io_Spi_A_Init_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`nodeId`
SPI Analyzer node ID

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_IO__SPI__A__INIT__CMD_FLAG__NONE`
No flag is set

`G_IO__SPI__A__INIT__CMD__FLAG__CPHA`
Clock phase

0 : data is sampled on first clock edge

1 : data is sampled on second clock edge

`G_IO__SPI__A__INIT__CMD__FLAG__BIG_ENDIAN`
Bit order of data bits

0 : little endian: least significant data bit is transmitted first

1 : big endian: most significant data bit is transmitted first

`SpiDataWidth`
Number of data bits per SPI data word (**1** .. **32**)

`SsUsage`
Slave select usage (one bit for each slave select line)

bit0=0 : slave select line **0** is not used

bit0=1 : slave select line **0** is used

bitN=0 : slave select line **N** is not used

bitN=1 : slave select line **N** is used

`SsPolarity`
Slave select polarity (one bit for each slave select line)

bit0=0 : slave select line **0** is active low

bit0=1 : slave select line **0** is active high

bitN=0 : slave select line **N** is active low

bitN=1 : slave select line **N** is active high

`reserved1`
reserved parameter (must be initialized with **0**)

`SsIdleTime`
Slave select idle time in [ns] (slave select inactive period to detect an end frame and complete the SPI transfer)

On 4121 devices: min/max [**10 .. 5109** ns] (**20ns** steps internal, nearest neighbour)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

8 SPI Analyzer Monitor

Commands for controlling the SPI Analyzer Monitor functionality.

G_Io_Spi_A_Monitor_Reset

G_Io_Spi_A_Monitor_Reset — Reset SPI Analyzer Monitor

Description

Use this command to reset the SPI Analyzer Monitor.

Definition

```
G_Error_t
G_API
G_Io_Spi_A_Monitor_Reset(
 const G_PortHandle_t  portHandle,
 const u8_t  nodeId
);
```

Parameters

`portHandle`
: Handle to the communication port

`nodeId`
: SPI Analyzer node ID

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Spi_A_Monitor_Init

G_Io_Spi_A_Monitor_Init — Initialize SPI Analyzer Monitor

Description

Use this command to initialize the SPI Analyzer Monitor.

Definition

```
G_Error_t
G_Io_Spi_A_Monitor_Init(
 const G_PortHandle_t  portHandle,
 const u8_t  nodeId,
 const G_Io_Spi_A_Monitor_Init_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`nodeId`
SPI Analyzer node ID

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_IO__SPI__A__MONITOR__INIT__CMD_FLAG__NONE`
No flag is set

`BufferSize`
Buffer size in number of monitor items, **0** = default size

`SsEnable`
Slave select enable (one bit for each slave select line)

bit0=0 : slave select line **0** is not monitored

bit0=1 : slave select line **0** is monitored

bitN=0 : slave select line **N** is not monitored

bitN=1 : slave select line **N** is monitored

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Spi_A_Monitor_Start

G_Io_Spi_A_Monitor_Start — Start SPI Analyzer Monitor

Description

Use this command to start the SPI Analyzer Monitor.

Definition

```
G_Error_t
G_Io_Spi_A_Monitor_Start(
 const G_PortHandle_t  portHandle,
 const u8_t  nodeId,
 const G_Io_Spi_A_Monitor_Start_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`nodeId`
SPI Analyzer node ID

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_IO__SPI__A__MONITOR__START__CMD_FLAG__NONE`
No flag is set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Spi_A_Monitor_Stop

G_Io_Spi_A_Monitor_Stop — Stop SPI Analyzer Monitor

Description

Use this command to stop the SPI Analyzer Monitor.

Definition

```
G_Error_t
G_Io_Spi_A_Monitor_Stop(
 const G_PortHandle_t  portHandle,
 const u8_t  nodeId,
 const G_Io_Spi_A_Monitor_Stop_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`nodeId`
SPI Analyzer node ID

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_IO__SPI__A__MONITOR__STOP__CMD_FLAG__NONE`
No flag is set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

9 TOG

Commands for controlling the TOG functionality.

These commands provide control over the TOG (Thermischer Ölniveaugeber) functionality of the device. Each command requires a parameter named `tog` that specifies the TOG instance. When working with different TOG signals on the same interface, the signals are identified by this value.

For assigning a TOG signal to an output, command G_Io_Trigger_Source_Set has to be called.

Currently, only 1 TOG instance per interface is supported. Therefore the value for parameter `tog` must be set to **`0`**.

The TOG functionality is only supported on Series61 devices and magicCAR 3.

G_Io_Tog_Mode_Set

G_Io_Tog_Mode_Set — Set TOG mode

Description

This command is used to set the TOG mode. Several modes are available, including **static low**, **static high**, **sequence** and **inverted sequence**.

For assigning a TOG signal to an output, command G_Io_Trigger_Source_Set has to be called.

magicCAR3 devices provide a dedicated pin for TOG signals on the **Central Connector for UUT**. To use this Pin for TOG signals, route the TOG to **Digital Out 7**.

When the TOG mode is changed, a currently running TOG sequence will be finished before the change is applied.

By performing a **TOG reset** (`mode == G_IO__TOG__MODE__RESET`), the TOG sequence is stopped immediately and the mode is set to **static low**.

On **magicCAR3** devices, the TOG signal is inverted.

Definition

```
G_Error_t
G_Io_Tog_Mode_Set(
 const G_PortHandle_t  portHandle,
 const u8_t  tog,
 const G_Io_Tog_Mode_t  mode
);
```

Parameters

`portHandle`
Handle to the communication port

`tog`
TOG instance

`mode`
TOG mode

The following values are possible:

`G_IO__TOG__MODE__STATIC__LOW`
The TOG output is set to a **static low** level

`G_IO__TOG__MODE__STATIC__HIGH`
The TOG output is set to a **static high** level

`G_IO__TOG__MODE__SEQUENCE__NORMAL`
A TOG sequence is started with the timing values defined by G_Io_Tog_Timings_Set

`G_IO__TOG__MODE__SEQUENCE__INVERTED`
A TOG sequence is started with the timing values defined by G_Io_Tog_Timings_Set and inverted output levels

`G_IO__TOG__MODE__RESET`
The TOG signal is stopped immediately and the mode is set to `G_IO__TOG__MODE__STATIC__LOW`

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

A TOG signal is defined and routed to the first digital output of the device.

```
G_Io_Tog_Timings_Set_Parameters_t  timings;
G_Error_t  rc;

// set TOG mode
rc =
 G_Io_Tog_Mode_Set(
  PortHandle,
  0,  // TOG instance 0
  G_IO__TOG__MODE__SEQUENCE__NORMAL  // TOG sequence
 );

if (rc != G_NO_ERROR) {
 return rc;
}

// set TOG timings
timings.HeatingTime = 10000;  // 10ms
timings.CoolingTime = 50000;  // 50ms
timings.TemperatureTime = 10000;  // 10ms
timings.ReferenceTime = 20000;  // 20ms

rc =
 G_Io_Tog_Timings_Set(
  PortHandle,
  0,
  &timings
 );

if (rc != G_NO_ERROR) {
 return rc;
}

// route TOG signal to first digital output
rc =
 G_Io_Trigger_Source_Set(
  PortHandle,
  G_IO__TRIGGER__OUTPUT_TYPE__DIGITAL_OUT,  // output type
  1,  // output index
  G_IO__TRIGGER__SOURCE_TYPE__TOG,  // source type
  0  // source index (= TOG instance)
 );

if (rc != G_NO_ERROR) {
 return rc;
}
```

G_Io_Tog_Timings_Set

G_Io_Tog_Timings_Set — Set TOG timings

Description

Use this command to set TOG timing values.

If a timing value exceeds the allowed boundaries, an error is returned.

Definition

```
G_Error_t
G_Io_Tog_Timings_Set(
 const G_PortHandle_t  portHandle,
 const u8_t  tog,
 const G_Io_Tog_Timings_Set_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`tog`
TOG instance

`parameters`
Structure with timing parameters

The structure contains the following elements:

`HeatingTime`
Heating time in microseconds (max. **1000000**)

`CoolingTime`
Cooling time in microseconds (min. **1** , max. **16000000**)

`TemperatureTime`
Temperature time in microseconds (max. **1000000**)

`ReferenceTime`
Reference time in microseconds (max. **1000000**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

10 Trigger

Commands for controlling the Trigger functionality.

With the help of the **Trigger Matrix**, a wide range of trigger applications can be implemented. It is used for assigning a trigger source to a trigger output as shown in Figure 1, "Trigger Matrix (excerpt)".

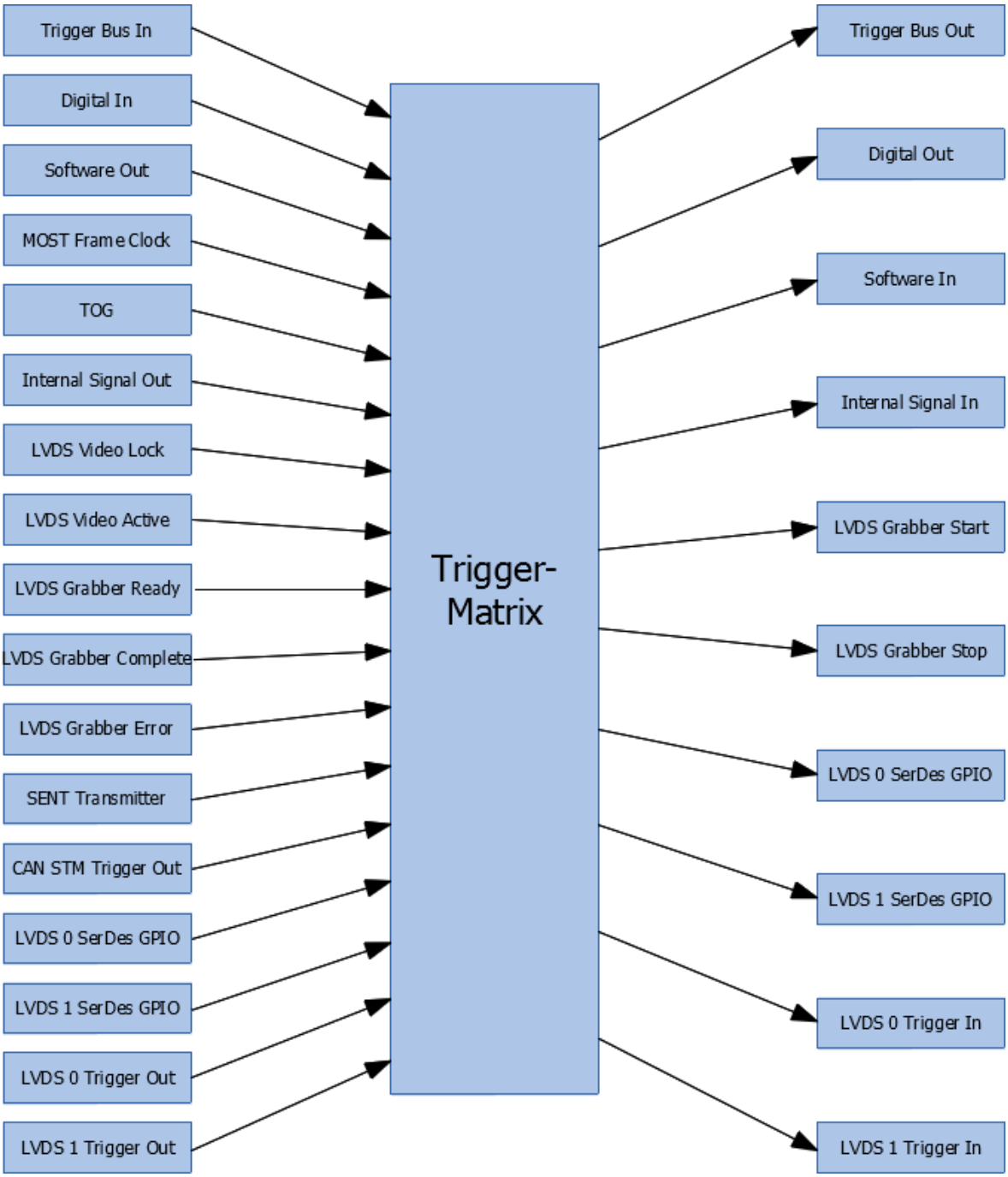

Figure 1 Trigger Matrix (excerpt)

Trigger signals can be generated internally or taken from an external input. Controlled by the **Trigger Matrix**, they can be used internally or routed to an output.

G_Io_Trigger_Source_Set

G_Io_Trigger_Source_Set — Trigger source to output routing

Description

Use this command to assign a **trigger signal source** to a **trigger signal output**.

Definition

```
G_Error_t
G_Io_Trigger_Source_Set(
 const G_PortHandle_t  portHandle,
 const G_Io_Trigger_OutputType_t  outputType,
 const u8_t  outputNumber,
 const G_Io_Trigger_SourceType_t  sourceType,
 const u8_t  sourceNumber
);
```

Parameters

`portHandle`
Handle to the communication port

`outputType`
Trigger signal destination

The following values are possible:

`G_IO__TRIGGER__OUTPUT_TYPE__UNKNOWN`
The trigger output is unknown

`G_IO__TRIGGER__OUTPUT_TYPE__TRIGGER_BUS_OUT`
Trigger bus out (not available on all devices)

If the trigger output is assigned to the trigger bus out, all trigger signals will be routed to the dedicated trigger bus line that feeds all devices of one PXI, PCI or **GOEPEL electronic** USB bus.

This mode is only available for PXI, PCI or rack-hosted USB devices.

`G_IO__TRIGGER__OUTPUT_TYPE__DIGITAL_OUT`
Digital output port

The trigger signals are routed to a digital output port.

`G_IO__TRIGGER__OUTPUT_TYPE__SOFTWARE_IN`
Software input (not available on all devices)

If the trigger output is assigned to a software input, the trigger signal can be used by communication interfaces of the device.

The communication interface has to be configured accordingly. This can be done with **G-API** commands like G_Can_Trigger_Input_Property_SetById or G_Lin_Trigger_Input_Property_SetById.

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_GRABBER__START`
LVDS Grabber start (not available on all devices)

The LVDS Grabber device will start the capture operation when the trigger source is active.

The external triggering of the capture operation needs to be activated with G_Lvds_FrameGrabber_ExternalTriggerMode_Start.

By default, the trigger source is set to digital input 1. This can be changed by using this command.

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_GRABBER__STOP`
LVDS Grabber stop (not available on all devices)

The LVDS Grabber device will stop the capture operation when the trigger source is active.

The external triggering of the capture operation needs to be activated with G_Lvds_FrameGrabber_ExternalTriggerMode_Start.

By default, the trigger source is set to digital input 2. This can be changed by using this command.

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_0_SER_DES_GPIO`
The trigger signals are routed to a Serializer / Deserializer GPIO of the first LVDS interface.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_1_SER_DES_GPIO`
The trigger signals are routed to a Serializer / Deserializer GPIO of the second LVDS interface.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_0_TRIGGER_IN`
The trigger signals are routed to the trigger input of the first LVDS interface.

The LVDS trigger input can be configured with G_Lvds_TriggerIn_SourceConfig_Set and G_Lvds_TriggerIn_TargetConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_1_TRIGGER_IN`
The trigger signals are routed to the trigger input of the second LVDS interface.

The LVDS trigger input can be configured with G_Lvds_TriggerIn_SourceConfig_Set and G_Lvds_TriggerIn_TargetConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__OUTPUT_TYPE__UART_RX`
The trigger signals are routed to the UART input.

`G_IO__TRIGGER__OUTPUT_TYPE__SPI_M_MISO`
The trigger signals are routed to SPI Master In Slave Out.

`G_IO__TRIGGER__OUTPUT_TYPE__SPI_S_MOSI`
The trigger signals are routed to SPI Master Out Slave In.

`G_IO__TRIGGER__OUTPUT_TYPE__SPI_S_SCLK`
The trigger signals are routed to SPI Slave Serial Clock.

`G_IO__TRIGGER__OUTPUT_TYPE__SPI_A_MISO`
The trigger signals are routed to SPI Analyzer Master In Slave Out.

`G_IO__TRIGGER__OUTPUT_TYPE__SPI_A_MOSI`
The trigger signals are routed to SPI Analyzer Master Out Slave In.

`G_IO__TRIGGER__OUTPUT_TYPE__SPI_A_SCLK`
The trigger signals are routed to SPI Analyzer Serial Clock.

`G_IO__TRIGGER__OUTPUT_TYPE__SPI_A_SS0`
The trigger signals are routed to SPI Slave Select 0.

`G_IO__TRIGGER__OUTPUT_TYPE__SPI_A_SS1`
The trigger signals are routed to SPI Slave Select 1.

`G_IO__TRIGGER__OUTPUT_TYPE__SPI_A_SS2`
The trigger signals are routed to SPI Slave Select 2.

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_2_SER_DES_GPIO`
The trigger signals are routed to a Serializer / Deserializer GPIO of the third LVDS interface.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_3_SER_DES_GPIO`
The trigger signals are routed to a Serializer / Deserializer GPIO of the fourth LVDS interface.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_2_TRIGGER_IN`
The trigger signals are routed to the trigger input of the third LVDS interface.

The LVDS trigger input can be configured with G_Lvds_TriggerIn_SourceConfig_Set and G_Lvds_TriggerIn_TargetConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_3_TRIGGER_IN`
The trigger signals are routed to the trigger input of the fourth LVDS interface.

The LVDS trigger input can be configured with G_Lvds_TriggerIn_SourceConfig_Set and G_Lvds_TriggerIn_TargetConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_MI_0_MFP`
The trigger signals are routed to the LVDS media interface 0 multi function pin.

`outputNumber`
Output number of the selected output type

Output and input numbers start with **1** .

If you prefer output and input numbering starting with **0** , use command G_Io_Trigger_Source_Set2 !

`sourceType`
Signal source selection

The following values are possible:

`G_IO__TRIGGER__SOURCE_TYPE__TRIGGER_BUS_IN`
Trigger bus input of the device (not available on all devices)

If the trigger source is assigned to the trigger bus in, trigger signals of the trigger bus line of a PXI, PCI or **GOEPEL electronic** USB Rack are used for triggering.

This mode is only available for PXI, PCI or rack-hosted USB devices.

`G_IO__TRIGGER__SOURCE_TYPE__DIGITAL_IN`
Digital input port

The trigger signal is taken from a digital input.

`G_IO__TRIGGER__SOURCE_TYPE__SOFTWARE_OUT`
Software output (not available on all devices)

If the trigger source is assigned to a software output, the trigger signal can be generated internally.

For a direct manipulation of the trigger signal, **G-API** function G_Io_Trigger_SoftwareOut_Write can be used.

`G_IO__TRIGGER__SOURCE_TYPE__MOST_NETWORK_FRAME_CLOCK`
MOST network frame clock signal

The MOST network frame clock signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__TOG`
TOG signal (Thermischer Ölniveaugeber)

The TOG signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_VIDEO_LOCK`
LVDS video lock signal

The LVDS video lock signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_VIDEO_ACTIVE`
LVDS video active signal

The LVDS video active signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_GRABBER_READY`
LVDS Frame Grabber ready signal

The LVDS Frame Grabber ready signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_GRABBER_COMPLETE`
LVDS Frame Grabber compelete signal

The LVDS Frame Grabber complete signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_GRABBER_ERROR`
LVDS Frame Grabber error signal

The LVDS Frame Grabber error signal is used as trigger source.

G_IO__TRIGGER__SOURCE_TYPE__SENT_TRANSMITTER
SENT Transmitter

The SENT Transmitter signal is used as trigger source.

G_IO__TRIGGER__SOURCE_TYPE__CAN_STM_TRIGGER_OUT
CAN STM Trigger Out

The CAN STM Trigger Out signal is used as a trigger source.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_0_SER_DES_GPIO
A GPIO signal of the Serializer / Deserializer board of the first LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_1_SER_DES_GPIO
A GPIO signal of the Serializer / Deserializer board of the second LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_0_TRIGGER_OUT
A trigger output signal of the first LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_1_TRIGGER_OUT
A trigger output signal of the second LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

G_IO__TRIGGER__SOURCE_TYPE__UART_TX
The UART output is used as the trigger source.

G_IO__TRIGGER__SOURCE_TYPE__INTERNAL_SOURCE
An internal source is used as the trigger source.

G_IO__TRIGGER__SOURCE_TYPE__SPI_M_MOSI
The SPI Master Master Out Slave In is used as the trigger source.

G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SCLK
The SPI Master Serial Clock is used as the trigger source.

G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SS0
The SPI Master Slave Select 0 is used as the trigger source.

G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SS1
The SPI Master Slave Select 1 is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SS2`
The SPI Master Slave Select 2 is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_S_MISO`
The SPI Slave Master In Slave Out is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_2_SER_DES_GPIO`
A GPIO signal of the Serializer / Deserializer board of the third LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_3_SER_DES_GPIO`
A GPIO signal of the Serializer / Deserializer board of the fourth LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_2_TRIGGER_OUT`
A trigger output signal of the third LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_3_TRIGGER_OUT`
A trigger output signal of the fourth LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_MI_0_MFP`
The LVDS media interface 0 multi function pin is used as a trigger source.

`SourceNumber`
source number of the selected source type

Output and input numbers start with **1** .

TOG instance numbers start with **0** .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

The first TOG signal instance is routed to the first digital output of the device.

```
G_Error_t  rc;

// route TOG signal to first digital output
```

```
rc =
 G_Io_Trigger_Source_Set(
  PortHandle,
  G_IO__TRIGGER__OUTPUT_TYPE__DIGITAL_OUT,  // output type
  1,  // output index
  G_IO__TRIGGER__SOURCE_TYPE__TOG,  // source type
  0  // source index (= TOG instance)
 );

if (rc != G_NO_ERROR) {
 return rc;
}
```

G_Io_Trigger_Source_Set2

G_Io_Trigger_Source_Set2 — Trigger source to output routing

Description

Use this command to assign a **trigger signal source** to a **trigger signal output .**

In contrast to command G_Io_Trigger_Source_Set `outputNumber` and `sourceNumber` are always starting with **0** to provide a more convenient numbering scheme, especially in cross-interface-applications.

Definition

```
G_Error_t
G_Io_Trigger_Source_Set2(
 const G_PortHandle_t  portHandle,
 const G_Io_Trigger_OutputType_t  outputType,
 const u8_t  outputNumber,
 const G_Io_Trigger_SourceType_t  sourceType,
 const u8_t  sourceNumber
);
```

Parameters

`portHandle`
Handle to the communication port

`outputType`
Trigger signal destination

The following values are possible:

`G_IO__TRIGGER__OUTPUT_TYPE__UNKNOWN`
The trigger output is unknown

`G_IO__TRIGGER__OUTPUT_TYPE__TRIGGER_BUS_OUT`
Trigger bus out (not available on all devices)

If the trigger output is assigned to the trigger bus out, all trigger signals will be routed to the dedicated trigger bus line that feeds all devices of one PXI, PCI or **GOEPEL electronic** USB bus.

This mode is only available for PXI, PCI or rack-hosted USB devices.

`G_IO__TRIGGER__OUTPUT_TYPE__DIGITAL_OUT`
Digital output port

The trigger signals are routed to a digital output port.

`G_IO__TRIGGER__OUTPUT_TYPE__SOFTWARE_IN`
Software input (not available on all devices)

If the trigger output is assigned to a software input, the trigger signal can be used by communication interfaces of the device.

The communication interface has to be configured accordingly. This can be done with **G-API** commands like G_Can_Trigger_Input_Property_SetById or G_Lin_Trigger_Input_Property_SetById .

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_GRABBER__START`
LVDS Grabber start (not available on all devices)

The LVDS Grabber device will start the capture operation when the trigger source is active.

The external triggering of the capture operation needs to be activated with G_Lvds_FrameGrabber_ExternalTriggerMode_Start.

By default, the trigger source is set to digital input 1. This can be changed by using this command.

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_GRABBER__STOP`
LVDS Grabber stop (not available on all devices)

The LVDS Grabber device will stop the capture operation when the trigger source is active.

The external triggering of the capture operation needs to be activated with G_Lvds_FrameGrabber_ExternalTriggerMode_Start.

By default, the trigger source is set to digital input 2. This can be changed by using this command.

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_0_SER_DES_GPIO`
The trigger signals are routed to a Serializer / Deserializer GPIO of the first LVDS interface.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_1_SER_DES_GPIO`
The trigger signals are routed to a Serializer / Deserializer GPIO of the second LVDS interface.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_0_TRIGGER_IN`
The trigger signals are routed to the trigger input of the first LVDS interface.

The LVDS trigger input can be configured with G_Lvds_TriggerIn_SourceConfig_Set and G_Lvds_TriggerIn_TargetConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_1_TRIGGER_IN`
The trigger signals are routed to the trigger input of the second LVDS interface.

The LVDS trigger input can be configured with G_Lvds_TriggerIn_SourceConfig_Set and G_Lvds_TriggerIn_TargetConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__OUTPUT_TYPE__UART_RX`
The trigger signals are routed to the UART input.

`G_IO__TRIGGER__OUTPUT_TYPE__SPI_M_MISO`
The trigger signals are routed to SPI Master In Slave Out.

`G_IO__TRIGGER__OUTPUT_TYPE__SPI_S_MOSI`
The trigger signals are routed to SPI Master Out Slave In.

`G_IO__TRIGGER__OUTPUT_TYPE__SPI_S_SCLK`
The trigger signals are routed to SPI Slave Serial Clock.

`G_IO__TRIGGER__OUTPUT_TYPE__SPI_A_MISO`
The trigger signals are routed to SPI Analyzer Master In Slave Out.

`G_IO__TRIGGER__OUTPUT_TYPE__SPI_A_MOSI`
The trigger signals are routed to SPI Analyzer Master Out Slave In.

`G_IO__TRIGGER__OUTPUT_TYPE__SPI_A_SCLK`
The trigger signals are routed to SPI Analyzer Serial Clock.

`G_IO__TRIGGER__OUTPUT_TYPE__SPI_A_SS0`
The trigger signals are routed to SPI Slave Select 0.

`G_IO__TRIGGER__OUTPUT_TYPE__SPI_A_SS1`
The trigger signals are routed to SPI Slave Select 1.

`G_IO__TRIGGER__OUTPUT_TYPE__SPI_A_SS2`
The trigger signals are routed to SPI Slave Select 2.

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_2_SER_DES_GPIO`
The trigger signals are routed to a Serializer / Deserializer GPIO of the third LVDS interface.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_3_SER_DES_GPIO`
The trigger signals are routed to a Serializer / Deserializer GPIO of the fourth LVDS interface.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_2_TRIGGER_IN`
The trigger signals are routed to the trigger input of the third LVDS interface.

The LVDS trigger input can be configured with G_Lvds_TriggerIn_SourceConfig_Set and G_Lvds_TriggerIn_TargetConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_3_TRIGGER_IN`
The trigger signals are routed to the trigger input of the fourth LVDS interface.

The LVDS trigger input can be configured with G_Lvds_TriggerIn_SourceConfig_Set and G_Lvds_TriggerIn_TargetConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__OUTPUT_TYPE__LVDS_MI_0_MFP`
The trigger signals are routed to the LVDS media interface 0 multi function pin.

`outputNumber`
Output number of the selected output type

Output and input numbers start with **0**.

`sourceType`
Signal source selection

The following values are possible:

`G_IO__TRIGGER__SOURCE_TYPE__TRIGGER_BUS_IN`
Trigger bus input of the device (not available on all devices)

If the trigger source is assigned to the trigger bus in, trigger signals of the trigger bus line of a PXI, PCI or **GOEPEL electronic** USB Rack are used for triggering.

This mode is only available for PXI, PCI or rack-hosted USB devices.

`G_IO__TRIGGER__SOURCE_TYPE__DIGITAL_IN`
Digital input port

The trigger signal is taken from a digital input.

`G_IO__TRIGGER__SOURCE_TYPE__SOFTWARE_OUT`
Software output (not available on all devices)

If the trigger source is assigned to a software output, the trigger signal can be generated internally.

For a direct manipulation of the trigger signal, **G-API** function G_Io_Trigger_SoftwareOut_Write can be used.

`G_IO__TRIGGER__SOURCE_TYPE__MOST_NETWORK_FRAME_CLOCK`
MOST network frame clock signal

The MOST network frame clock signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__TOG`
TOG signal (Thermischer Ölniveaugeber)

The TOG signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_VIDEO_LOCK`
LVDS video lock signal

The LVDS video lock signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_VIDEO_ACTIVE`
LVDS video active signal

The LVDS video active signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_GRABBER_READY`
LVDS Frame Grabber ready signal

The LVDS Frame Grabber ready signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_GRABBER_COMPLETE`
LVDS Frame Grabber compelete signal

The LVDS Frame Grabber complete signal is used as trigger source.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_GRABBER_ERROR
LVDS Frame Grabber error signal

The LVDS Frame Grabber error signal is used as trigger source.

G_IO__TRIGGER__SOURCE_TYPE__SENT_TRANSMITTER
SENT Transmitter

The SENT Transmitter signal is used as trigger source.

G_IO__TRIGGER__SOURCE_TYPE__CAN_STM_TRIGGER_OUT
CAN STM Trigger Out

The CAN STM Trigger Out signal is used as a trigger source.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_0_SER_DES_GPIO
A GPIO signal of the Serializer / Deserializer board of the first LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_1_SER_DES_GPIO
A GPIO signal of the Serializer / Deserializer board of the second LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_0_TRIGGER_OUT
A trigger output signal of the first LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_1_TRIGGER_OUT
A trigger output signal of the second LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

G_IO__TRIGGER__SOURCE_TYPE__UART_TX
The UART output is used as the trigger source.

G_IO__TRIGGER__SOURCE_TYPE__INTERNAL_SOURCE
An internal source is used as the trigger source.

G_IO__TRIGGER__SOURCE_TYPE__SPI_M_MOSI
The SPI Master Master Out Slave In is used as the trigger source.

G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SCLK
The SPI Master Serial Clock is used as the trigger source.

G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SS0
The SPI Master Slave Select 0 is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SS1`
The SPI Master Slave Select 1 is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SS2`
The SPI Master Slave Select 2 is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_S_MISO`
The SPI Slave Master In Slave Out is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_2_SER_DES_GPIO`
A GPIO signal of the Serializer / Deserializer board of the third LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_3_SER_DES_GPIO`
A GPIO signal of the Serializer / Deserializer board of the fourth LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_2_TRIGGER_OUT`
A trigger output signal of the third LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_3_TRIGGER_OUT`
A trigger output signal of the fourth LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_MI_0_MFP`
The LVDS media interface 0 multi function pin is used as a trigger source.

`SourceNumber`
source number of the selected source type

Output and input numbers start with **0** .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Trigger_Source_Get

G_Io_Trigger_Source_Get — Query trigger matrix routing

Description

Use this command to query the routing of the trigger matrix.

Definition

```
G_Error_t
G_Io_Trigger_Source_Get(
 const G_PortHandle_t  portHandle,
 const G_Io_Trigger_OutputType_t  outputType,
 const u8_t  outputNumber,
 G_Io_Trigger_SourceType_t * const sourceType,
 u8_t * const sourceNumber
);
```

Parameters

`portHandle`
Handle to the communication port

`outputType`
Returns the current trigger output

The following values are possible: G_Io_Trigger_OutputType_t;

`outputNumber`
Returns the instance number of the current trigger output

Output and input numbers start with **1**.

If you prefer output and input numbering starting with **0**, use command G_Io_Trigger_Source_Get2 !

`sourceType`
Returns the current trigger source

The following values are possible:

`G_IO__TRIGGER__SOURCE_TYPE__TRIGGER_BUS_IN`
Trigger bus input of the device (not available on all devices)

If the trigger source is assigned to the trigger bus in, trigger signals of the trigger bus line of a PXI, PCI or **GOEPEL electronic** USB Rack are used for triggering.

This mode is only available for PXI, PCI or rack-hosted USB devices.

`G_IO__TRIGGER__SOURCE_TYPE__DIGITAL_IN`
Digital input port

The trigger signal is taken from a digital input.

`G_IO__TRIGGER__SOURCE_TYPE__SOFTWARE_OUT`
Software output (not available on all devices)

If the trigger source is assigned to a software output, the trigger signal can be generated internally.

For a direct manipulation of the trigger signal, **G-API** function G_Io_Trigger_SoftwareOut_Write can be used.

`G_IO__TRIGGER__SOURCE_TYPE__MOST_NETWORK_FRAME_CLOCK`
MOST network frame clock signal

The MOST network frame clock signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__TOG`
TOG signal (Thermischer Ölniveaugeber)

The TOG signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_VIDEO_LOCK`
LVDS video lock signal

The LVDS video lock signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_VIDEO_ACTIVE`
LVDS video active signal

The LVDS video active signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_GRABBER_READY`
LVDS Frame Grabber ready signal

The LVDS Frame Grabber ready signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_GRABBER_COMPLETE`
LVDS Frame Grabber compelete signal

The LVDS Frame Grabber complete signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_GRABBER_ERROR`
LVDS Frame Grabber error signal

The LVDS Frame Grabber error signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SENT_TRANSMITTER`
SENT Transmitter

The SENT Transmitter signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__CAN_STM_TRIGGER_OUT`
CAN STM Trigger Out

The CAN STM Trigger Out signal is used as a trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_0_SER_DES_GPIO`
A GPIO signal of the Serializer / Deserializer board of the first LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_1_SER_DES_GPIO`
A GPIO signal of the Serializer / Deserializer board of the second LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_0_TRIGGER_OUT`
A trigger output signal of the first LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_1_TRIGGER_OUT`
A trigger output signal of the second LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__UART_TX`
The UART output is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__INTERNAL_SOURCE`
An internal source is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_M_MOSI`
The SPI Master Master Out Slave In is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SCLK`
The SPI Master Serial Clock is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SS0`
The SPI Master Slave Select 0 is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SS1`
The SPI Master Slave Select 1 is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SS2`
The SPI Master Slave Select 2 is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_S_MISO`
The SPI Slave Master In Slave Out is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_2_SER_DES_GPIO`
A GPIO signal of the Serializer / Deserializer board of the third LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_3_SER_DES_GPIO`
A GPIO signal of the Serializer / Deserializer board of the fourth LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_2_TRIGGER_OUT`
A trigger output signal of the third LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_3_TRIGGER_OUT`
A trigger output signal of the fourth LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_MI_0_MFP`
The LVDS media interface 0 multi function pin is used as a trigger source.

`sourceNumber`
Returns the instance number of the current trigger source

Output and input numbers start with **1**.

TOG instance numbers start with **0**.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Trigger_Source_Get2

G_Io_Trigger_Source_Get2 — Query trigger matrix routing

Description

Use this command to query the routing of the trigger matrix.

In contrast to command G_Io_Trigger_Source_Get `outputNumber` and `sourceNumber` are always starting with **0** to provide a more convenient numbering scheme, especially in cross-interface-applications.

Definition

```
G_Error_t
G_Io_Trigger_Source_Get2(
 const G_PortHandle_t  portHandle,
 const G_Io_Trigger_OutputType_t  outputType,
 const u8_t  outputNumber,
 G_Io_Trigger_SourceType_t * const sourceType,
 u8_t * const sourceNumber
);
```

Parameters

`portHandle`
Handle to the communication port

`outputType`
Returns the current trigger output

The following values are possible: G_Io_Trigger_OutputType_t;

`outputNumber`
Returns the instance number of the current trigger output

Output and input numbers start with **0**.

`sourceType`
Returns the current trigger source

The following values are possible:

`G_IO__TRIGGER__SOURCE_TYPE__TRIGGER_BUS_IN`
Trigger bus input of the device (not available on all devices)

If the trigger source is assigned to the trigger bus in, trigger signals of the trigger bus line of a PXI, PCI or **GOEPEL electronic** USB Rack are used for triggering.

This mode is only available for PXI, PCI or rack-hosted USB devices.

`G_IO__TRIGGER__SOURCE_TYPE__DIGITAL_IN`
Digital input port

The trigger signal is taken from a digital input.

`G_IO__TRIGGER__SOURCE_TYPE__SOFTWARE_OUT`
Software output (not available on all devices)

If the trigger source is assigned to a software output, the trigger signal can be generated internally.

For a direct manipulation of the trigger signal, **G-API** function G_Io_Trigger_SoftwareOut_Write can be used.

`G_IO__TRIGGER__SOURCE_TYPE__MOST_NETWORK_FRAME_CLOCK`
MOST network frame clock signal

The MOST network frame clock signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__TOG`
TOG signal (Thermischer Ölniveaugeber)

The TOG signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_VIDEO_LOCK`
LVDS video lock signal

The LVDS video lock signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_VIDEO_ACTIVE`
LVDS video active signal

The LVDS video active signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_GRABBER_READY`
LVDS Frame Grabber ready signal

The LVDS Frame Grabber ready signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_GRABBER_COMPLETE`
LVDS Frame Grabber compelete signal

The LVDS Frame Grabber complete signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_GRABBER_ERROR`
LVDS Frame Grabber error signal

The LVDS Frame Grabber error signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SENT_TRANSMITTER`
SENT Transmitter

The SENT Transmitter signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__CAN_STM_TRIGGER_OUT`
CAN STM Trigger Out

The CAN STM Trigger Out signal is used as a trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_0_SER_DES_GPIO`
A GPIO signal of the Serializer / Deserializer board of the first LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_1_SER_DES_GPIO
A GPIO signal of the Serializer / Deserializer board of the second LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_0_TRIGGER_OUT
A trigger output signal of the first LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_1_TRIGGER_OUT
A trigger output signal of the second LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

G_IO__TRIGGER__SOURCE_TYPE__UART_TX
The UART output is used as the trigger source.

G_IO__TRIGGER__SOURCE_TYPE__INTERNAL_SOURCE
An internal source is used as the trigger source.

G_IO__TRIGGER__SOURCE_TYPE__SPI_M_MOSI
The SPI Master Master Out Slave In is used as the trigger source.

G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SCLK
The SPI Master Serial Clock is used as the trigger source.

G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SS0
The SPI Master Slave Select 0 is used as the trigger source.

G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SS1
The SPI Master Slave Select 1 is used as the trigger source.

G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SS2
The SPI Master Slave Select 2 is used as the trigger source.

G_IO__TRIGGER__SOURCE_TYPE__SPI_S_MISO
The SPI Slave Master In Slave Out is used as the trigger source.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_2_SER_DES_GPIO
A GPIO signal of the Serializer / Deserializer board of the third LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_3_SER_DES_GPIO
A GPIO signal of the Serializer / Deserializer board of the fourth LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_2_TRIGGER_OUT`
A trigger output signal of the third LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_3_TRIGGER_OUT`
A trigger output signal of the fourth LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_MI_0_MFP`
The LVDS media interface 0 multi function pin is used as a trigger source.

`sourceNumber`
Returns the instance number of the current trigger source

Output and input numbers start with **0** .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Trigger_TimestampConfig_Set

G_Io_Trigger_TimestampConfig_Set — Set timestamp trigger configuration

Description

Use this command to set the timestamp trigger configuration.

Per default, an internal clock is used for generating timestamps of monitor items of the communication interfaces. The trigger unit can also be configured to use an external clock for generating timestamps or to provide its internal clock to an output, which then can be used by other devices. This has the advantage of precise and comparable timestamps across multiple devices.

In addition to this, a synchronization source can be configured to reset the timestamps. This can be used for a synchronized timestamp reset of multiple devices.

Definition

```
G_Error_t
G_Io_Trigger_TimestampConfig_Set(
 const G_PortHandle_t  portHandle,
 const G_Io_Trigger_TimestampConfig_Set_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`CmdFlags`
Command flags

The following values are possible and can be combined:

`G_IO__TRIGGER__TIMESTAMP_CONFIG__SET__CMD_FLAG__NONE`
No flag is set

`G_IO__TRIGGER__TIMESTAMP_CONFIG__SET__CMD_FLAG__SYNC_ENABLED`
Enable synchronization of the timestamps

When the trigger condition, defined by `SyncSourceType`, `SyncSourceNumber` and `SyncEdge` is met, the timestamps will be reset.

`G_IO__TRIGGER__TIMESTAMP_CONFIG__SET__CMD_FLAG__CLOCK_OUT_ENABLED`
Enable clock output

The internal clock, that is used for generating timestamps is provided at the clock bus line, enabling other devices to use this clock for generating comparable timestamps.

When this flag is set, `ClockSource` needs to be set to `G_IO__TRIGGER__TIMESTAMP_CLOCK_SOURCE__EXTERNAL`.

The clock output is only available for **GOEPEL electronic** PCI or USB Rack devices.

G_IO__TRIGGER__TIMESTAMP_CONFIG__SET__CMD_FLAG__NO_OFFSET
0 : SyncSourceNumber starting with **1**

1 : SyncSourceNumber starting with **0**

Historically the value for SyncSourceNumber is starting with **1**. For cross-interface applications it can be beneficial to set this flag and start from **0**.

SyncSourceType
Source for timestamp synchronization

A synchronization source can be configured to reset the timestamps. This can be used for a synchronized timestamp reset of multiple devices.

The following values are possible:

G_IO__TRIGGER__SOURCE_TYPE__TRIGGER_BUS_IN
Trigger bus input of the device (not available on all devices)

If the trigger source is assigned to the trigger bus in, trigger signals of the trigger bus line of a PXI, PCI or **GOEPEL electronic** USB Rack are used for triggering.

This mode is only available for PXI, PCI or rack-hosted USB devices.

G_IO__TRIGGER__SOURCE_TYPE__DIGITAL_IN
Digital input port

The trigger signal is taken from a digital input.

G_IO__TRIGGER__SOURCE_TYPE__SOFTWARE_OUT
Software output (not available on all devices)

If the trigger source is assigned to a software output, the trigger signal can be generated internally.

For a direct manipulation of the trigger signal, **G-API** function G_Io_Trigger_SoftwareOut_Write can be used.

G_IO__TRIGGER__SOURCE_TYPE__MOST_NETWORK_FRAME_CLOCK
MOST network frame clock signal

The MOST network frame clock signal is used as trigger source.

G_IO__TRIGGER__SOURCE_TYPE__TOG
TOG signal (Thermischer Ölniveaugeber)

The TOG signal is used as trigger source.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_VIDEO_LOCK
LVDS video lock signal

The LVDS video lock signal is used as trigger source.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_VIDEO_ACTIVE
LVDS video active signal

The LVDS video active signal is used as trigger source.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_GRABBER_READY
LVDS Frame Grabber ready signal

The LVDS Frame Grabber ready signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_GRABBER_COMPLETE`
LVDS Frame Grabber compelete signal

The LVDS Frame Grabber complete signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_GRABBER_ERROR`
LVDS Frame Grabber error signal

The LVDS Frame Grabber error signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SENT_TRANSMITTER`
SENT Transmitter

The SENT Transmitter signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__CAN_STM_TRIGGER_OUT`
CAN STM Trigger Out

The CAN STM Trigger Out signal is used as a trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_0_SER_DES_GPIO`
A GPIO signal of the Serializer / Deserializer board of the first LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_1_SER_DES_GPIO`
A GPIO signal of the Serializer / Deserializer board of the second LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_0_TRIGGER_OUT`
A trigger output signal of the first LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_1_TRIGGER_OUT`
A trigger output signal of the second LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__UART_TX`
The UART output is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__INTERNAL_SOURCE`
An internal source is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_M_MOSI`
The SPI Master Master Out Slave In is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SCLK`
The SPI Master Serial Clock is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SS0`
The SPI Master Slave Select 0 is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SS1`
The SPI Master Slave Select 1 is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SS2`
The SPI Master Slave Select 2 is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_S_MISO`
The SPI Slave Master In Slave Out is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_2_SER_DES_GPIO`
A GPIO signal of the Serializer / Deserializer board of the third LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_3_SER_DES_GPIO`
A GPIO signal of the Serializer / Deserializer board of the fourth LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_2_TRIGGER_OUT`
A trigger output signal of the third LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_3_TRIGGER_OUT`
A trigger output signal of the fourth LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_MI_0_MFP`
The LVDS media interface 0 multi function pin is used as a trigger source.

`SyncSourceNumber`
Instance number of the synchronization source

Output and input number offset is depending on flag `G_IO__TRIGGER__TIMESTAMP_CONFIG__SET__CMD_FLAG__NO_OFFSET`.

`SyncEdge`
Synchronization edge

Defines the edge of the synchronization signal that is used for synchronization.

The following values are possible:

`G_IO__TRIGGER__EDGE__UNKNOWN`
Edge is unknown

`G_IO__TRIGGER__EDGE__NONE`
No edge

`G_IO__TRIGGER__EDGE__FALLING`
Falling edge

`G_IO__TRIGGER__EDGE__RISING`
Rising edge

`G_IO__TRIGGER__EDGE__BOTH`
Rising and falling edge

`ClockSource`
Defines the clock signal that is used for timestamp generation

The following values are possible:

`G_IO__TRIGGER__TIMESTAMP_CLOCK_SOURCE__UNKNOWN`
The clock source is unknown

`G_IO__TRIGGER__TIMESTAMP_CLOCK_SOURCE__INTERNAL`
The internal clock is used for timestamp generation

`G_IO__TRIGGER__TIMESTAMP_CLOCK_SOURCE__EXTERNAL`
An external clock is used for timestamp generation

An external clock source is only available for PXI, PCI or **GOEPEL electronic** USB Rack hosted devices.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Trigger_TimestampConfig_Get

G_Io_Trigger_TimestampConfig_Get — Query timestamp trigger configuration

Description

Use this command to query the timestamp trigger configuration.

Definition

```
G_Error_t
G_Io_Trigger_TimestampConfig_Get(
 const G_PortHandle_t  portHandle,
 G_Io_Trigger_TimestampConfig_Get_Response_t * const response
);
```

Parameters

`portHandle`
Handle to the communication port

`response`
Returns response parameters

The structure contains the following elements:

`RspFlags`
Response flags

The following values are possible and can be combined:

`G_IO__TRIGGER__TIMESTAMP_CONFIG__GET__RSP_FLAG__NONE`
No flag is set

`G_IO__TRIGGER__TIMESTAMP_CONFIG__GET__RSP_FLAG__SYNC_ENABLED`
The timestamp synchronization is enabled

When the trigger condition, defined by `SyncSourceType`, `SyncSourceNumber` and `SyncEdge` is met, the timestamps will be reset.

`G_IO__TRIGGER__TIMESTAMP_CONFIG__GET__RSP_FLAG__CLOCK_OUT_ENABLED`
The clock output is enabled

The internal clock, that is used for generating timestamps is provided at the clock bus line, enabling other devices to use this clock for generating comparable timestamps.

`G_IO__TRIGGER__TIMESTAMP_CONFIG__GET__RSP_FLAG__EXT_CLOCK_LOCKED`
The timestamp generation is locked to an external clock

`G_IO__TRIGGER__TIMESTAMP_CONFIG__GET__RSP_FLAG__NO_OFFSET`
0 : `SyncSourceNumber` starting with **1**

1 : `SyncSourceNumber` starting with **0**

`SyncSourceType`
Source for timestamp synchronization

A synchronization source can be configured to reset the timestamps. This can be used for a synchronized timestamp reset of multiple devices.

The following values are possible:

`G_IO__TRIGGER__SOURCE_TYPE__TRIGGER_BUS_IN`
Trigger bus input of the device (not available on all devices)

If the trigger source is assigned to the trigger bus in, trigger signals of the trigger bus line of a PXI, PCI or **GOEPEL electronic** USB Rack are used for triggering.

This mode is only available for PXI, PCI or rack-hosted USB devices.

`G_IO__TRIGGER__SOURCE_TYPE__DIGITAL_IN`
Digital input port

The trigger signal is taken from a digital input.

`G_IO__TRIGGER__SOURCE_TYPE__SOFTWARE_OUT`
Software output (not available on all devices)

If the trigger source is assigned to a software output, the trigger signal can be generated internally.

For a direct manipulation of the trigger signal, **G-API** function G_Io_Trigger_SoftwareOut_Write can be used.

`G_IO__TRIGGER__SOURCE_TYPE__MOST_NETWORK_FRAME_CLOCK`
MOST network frame clock signal

The MOST network frame clock signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__TOG`
TOG signal (Thermischer Ölniveaugeber)

The TOG signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_VIDEO_LOCK`
LVDS video lock signal

The LVDS video lock signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_VIDEO_ACTIVE`
LVDS video active signal

The LVDS video active signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_GRABBER_READY`
LVDS Frame Grabber ready signal

The LVDS Frame Grabber ready signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_GRABBER_COMPLETE`
LVDS Frame Grabber compelete signal

The LVDS Frame Grabber complete signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_GRABBER_ERROR`
LVDS Frame Grabber error signal

The LVDS Frame Grabber error signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SENT_TRANSMITTER`
SENT Transmitter

The SENT Transmitter signal is used as trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__CAN_STM_TRIGGER_OUT`
CAN STM Trigger Out

The CAN STM Trigger Out signal is used as a trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_0_SER_DES_GPIO`
A GPIO signal of the Serializer / Deserializer board of the first LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_1_SER_DES_GPIO`
A GPIO signal of the Serializer / Deserializer board of the second LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_0_TRIGGER_OUT`
A trigger output signal of the first LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__LVDS_1_TRIGGER_OUT`
A trigger output signal of the second LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

`G_IO__TRIGGER__SOURCE_TYPE__UART_TX`
The UART output is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__INTERNAL_SOURCE`
An internal source is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_M_MOSI`
The SPI Master Master Out Slave In is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SCLK`
The SPI Master Serial Clock is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SS0`
The SPI Master Slave Select 0 is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SS1`
The SPI Master Slave Select 1 is used as the trigger source.

`G_IO__TRIGGER__SOURCE_TYPE__SPI_M_SS2`
The SPI Master Slave Select 2 is used as the trigger source.

G_IO__TRIGGER__SOURCE_TYPE__SPI_S_MISO
The SPI Slave Master In Slave Out is used as the trigger source.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_2_SER_DES_GPIO
A GPIO signal of the Serializer / Deserializer board of the third LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_3_SER_DES_GPIO
A GPIO signal of the Serializer / Deserializer board of the fourth LVDS interface is used as a trigger source.

The GPIO can be configured with G_Lvds_SerDesGpio_IocPinConfig_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_2_TRIGGER_OUT
A trigger output signal of the third LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_3_TRIGGER_OUT
A trigger output signal of the fourth LVDS interface is used as a trigger source.

The LVDS trigger output can be configured with G_Lvds_TriggerOut_Config_Set .

The instance ID of the interface referenced by the current port handle can be queried with command G_Common_GetInterfaceInfo.

G_IO__TRIGGER__SOURCE_TYPE__LVDS_MI_0_MFP
The LVDS media interface 0 multi function pin is used as a trigger source.

SyncSourceNumber
Instance number of the synchronization source

Output and input number offset is depending on flag G_IO__TRIGGER__TIMESTAMP_CONFIG__GET__CMD_FLAG__NO_OFFSET.

SyncEdge
Edge of the synchronization signal that is used for synchronization

The following values are possible:

G_IO__TRIGGER__EDGE__UNKNOWN
Edge is unknown

G_IO__TRIGGER__EDGE__NONE
No edge

G_IO__TRIGGER__EDGE__FALLING
Falling edge

G_IO__TRIGGER__EDGE__RISING
Rising edge

`G_IO__TRIGGER__EDGE__BOTH`
Rising and falling edge

`ClockSource`
Clock signal that is used for timestamp generation (see G_Io_Trigger_TimestampClockSource_t)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Trigger_Timestamp_Get

G_Io_Trigger_Timestamp_Get — Query current trigger timestamp timer value

Description

Use this command to query current trigger timestamp timer value.

Definition

```
G_Error_t
G_Io_Trigger_Timestamp_Get(
 const G_PortHandle_t  portHandle,
 u64_t * const timestamp
);
```

Parameters

`portHandle`
: Handle to the communication port

`timestamp`
: Returns current trigger timestamp timer value, in ns

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Trigger_TriggerBusDirection_Set

G_Io_Trigger_TriggerBusDirection_Set — Set bus direction of trigger channels

Description

Use this command to set the direction of trigger bus channels.

Trigger bus channels can be used to share signals across multiple devices over the PXI, PCI or USB (rack) bus. There are **8 trigger bus channels** available. Every trigger bus channel can only be driven by 1 device.

Trigger bus channels are not available on standalone devices like basicCAN or basicCAR.

Definition

```
G_Error_t
G_Io_Trigger_TriggerBusDirection_Set(
 const G_PortHandle_t  portHandle,
 const u8_t  numberOfChannels,
 const G_Io_Trigger_TriggerBusDirection_Set_Channel_t * const channels
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfChannels`
Number of channels whose direction is to be set

`channels`
Channel info

The structure contains the following elements:

`Channel`
Channel number (starting with **1**)

There are 8 trigger channels available.

If you prefer channel numbers starting with **0**, use command G_Io_Trigger_TriggerBusDirection_Set2 !

`TriggerBusDirection`
Direction of the trigger bus channel

The following values are possible:

`G_IO__TRIGGER__TRIGGER_BUS_DIRECTION__UNKNOWN`
The trigger bus direction is unknown

`G_IO__TRIGGER__TRIGGER_BUS_DIRECTION__IN`
The trigger bus channel is configured as input

`G_IO__TRIGGER__TRIGGER_BUS_DIRECTION__OUT`
The trigger bus channel is configured as output

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Trigger_TriggerBusDirection_Set2

G_Io_Trigger_TriggerBusDirection_Set2 — Set bus direction of trigger channels

Description

Use this command to set the direction of trigger bus channels.

Trigger bus channels can be used to share signals across multiple devices over the PXI, PCI or USB (rack) bus. There are **8 trigger bus channels** available. Every trigger bus channel can only be driven by 1 device.

In contrast to command G_Io_Trigger_TriggerBusDirection_Set channel numbers defined by `Channel` are starting with **0** to provide a more convenient numbering scheme, especially in cross-interface-applications.

Trigger bus channels are not available on standalone devices like basicCAN or basicCAR.

Definition

```
G_Error_t
G_Io_Trigger_TriggerBusDirection_Set2(
 const G_PortHandle_t  portHandle,
 const u8_t  numberOfChannels,
 const G_Io_Trigger_TriggerBusDirection_Set2_Channel_t * const channels
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfChannels`
Number of channels whose direction is to be set

`channels`
Channel info

The structure contains the following elements:

`Channel`
Channel number (starting with **0**)

There are 8 trigger channels available.

`TriggerBusDirection`
Direction of the trigger bus channel

The following values are possible:

`G_IO__TRIGGER__TRIGGER_BUS_DIRECTION__UNKNOWN`
The trigger bus direction is unknown

`G_IO__TRIGGER__TRIGGER_BUS_DIRECTION__IN`
The trigger bus channel is configured as input

`G_IO__TRIGGER__TRIGGER_BUS_DIRECTION__OUT`
The trigger bus channel is configured as output

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Trigger_TriggerBusDirection_Get

G_Io_Trigger_TriggerBusDirection_Get — Query bus direction of trigger channels

Description

Use this command to query the direction of trigger bus channels.

Trigger bus channels can be used to share signals across multiple devices over the PXI, PCI or USB (rack) bus. There are **8 trigger bus channels** available. Every trigger bus channel can only be driven by 1 device.

Trigger bus channels are not available on standalone devices like basicCAN or basicCAR.

Definition

```
G_Error_t
G_Io_Trigger_TriggerBusDirection_Get(
 const G_PortHandle_t  portHandle,
 u8_t * const numberOfChannels,
 G_Io_Trigger_TriggerBusDirection_Get_Channel_t * const channels
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfChannels`
In : Size of buffer `channels` (in number of channels)

Out : Number of returned channels

`channels`
Returns channel info

The structure contains the following elements:

`Channel`
Channel number (starting with **1**)

There are 8 trigger channels available.

If you prefer channel numbers starting with **0** , use command G_Io_Trigger_TriggerBusDirection_Get2 !

`TriggerBusDirection`
Direction of the trigger bus channel

The following values are possible:

`G_IO__TRIGGER__TRIGGER_BUS_DIRECTION__UNKNOWN`
The trigger bus direction is unknown

`G_IO__TRIGGER__TRIGGER_BUS_DIRECTION__IN`
The trigger bus channel is configured as input

`G_IO__TRIGGER__TRIGGER_BUS_DIRECTION__OUT`
The trigger bus channel is configured as output

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Trigger_TriggerBusDirection_Get2

G_Io_Trigger_TriggerBusDirection_Get2 — Query bus direction of trigger channels

Description

Use this command to query the direction of trigger bus channels.

Trigger bus channels can be used to share signals across multiple devices over the PXI, PCI or USB (rack) bus. There are **8 trigger bus channels** available. Every trigger bus channel can only be driven by 1 device.

In contrast to command G_Io_Trigger_TriggerBusDirection_Get channel numbers defined by `Channel` are starting with **0** to provide a more convenient numbering scheme, especially in cross-interface-applications.

Trigger bus channels are not available on standalone devices like basicCAN or basicCAR.

Definition

```
G_Error_t
G_Io_Trigger_TriggerBusDirection_Get2(
 const G_PortHandle_t  portHandle,
 u8_t * const numberOfChannels,
 G_Io_Trigger_TriggerBusDirection_Get2_Channel_t * const channels
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfChannels`
In : Size of buffer `channels` (in number of channels)

Out : Number of returned channels

`channels`
Returns channel info

The structure contains the following elements:

`Channel`
Channel number (starting with **0**)

There are 8 trigger channels available.

`TriggerBusDirection`
Direction of the trigger bus channel

The following values are possible:

`G_IO__TRIGGER__TRIGGER_BUS_DIRECTION__UNKNOWN`
The trigger bus direction is unknown

`G_IO__TRIGGER__TRIGGER_BUS_DIRECTION__IN`
The trigger bus channel is configured as input

`G_IO__TRIGGER__TRIGGER_BUS_DIRECTION__OUT`
The trigger bus channel is configured as output

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Trigger_SoftwareOut_Write

G_Io_Trigger_SoftwareOut_Write — Write to internal software output

Description

Use this command to write to an internal software output. This software output can be routed to different types of trigger outputs by using G_Io_Trigger_Source_Set .

Definition

```
G_Error_t
G_Io_Trigger_SoftwareOut_Write(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Io_Trigger_SoftwareOut_Write_Mode_t  mode,
 const u32_t  pulseDuration
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Software out channel (starting with **1**)

There are **16 software out** channels available.

If you prefer channel numbers starting with **0** , use command G_Io_Trigger_SoftwareOut_Write2 !

`mode`
Write mode

The following values are possible:

`G_IO__TRIGGER__SOFTWARE_OUT__WRITE__MODE__SET`
The software out is set

`G_IO__TRIGGER__SOFTWARE_OUT__WRITE__MODE__RESET`
The software out is reset

`G_IO__TRIGGER__SOFTWARE_OUT__WRITE__MODE__TOGGLE`
The software out is toggled

`pulseDuration`
Duration of the write action, in nanoseconds

0 : No pulse is generated. The software out is modified according to `mode` only.

not 0 : The software out is modified according to `mode` and after the time `pulseDuration`, the trigger line is toggled.

Resolution = 100 nanoseconds, Maximum = 25500 nanoseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Trigger_SoftwareOut_Write2

G_Io_Trigger_SoftwareOut_Write2 — Write to internal software output

Description

Use this command to write to an internal software output. This software output can be routed to different types of trigger outputs by using G_Io_Trigger_Source_Set .

In contrast to command G_Io_Trigger_SoftwareOut_Write the channel number defined by `channel` is starting with **0** to provide a more convenient numbering scheme, especially in cross-interface-applications.

Definition

```
G_Error_t
G_Io_Trigger_SoftwareOut_Write2(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Io_Trigger_SoftwareOut_Write_Mode_t  mode,
 const u32_t  pulseDuration
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Software out channel (starting with **0**)

There are **16 software out** channels available.

`mode`
Write mode

The following values are possible:

`G_IO__TRIGGER__SOFTWARE_OUT__WRITE__MODE__SET`
The software out is set

`G_IO__TRIGGER__SOFTWARE_OUT__WRITE__MODE__RESET`
The software out is reset

`G_IO__TRIGGER__SOFTWARE_OUT__WRITE__MODE__TOGGLE`
The software out is toggled

`pulseDuration`
Duration of the write action, in nanoseconds

0 : No pulse is generated. The software out is modified according to `mode` only.

not 0 : The software out is modified according to `mode` and after the time `pulseDuration`, the trigger line is toggled.

Resolution = 100 nanoseconds, Maximum = 25500 nanoseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Trigger_SoftwareOut_Read

G_Io_Trigger_SoftwareOut_Read — Query software output state

Description

Use this command to query the state of an internal software output.

Definition

```
G_Error_t
G_Io_Trigger_SoftwareOut_Read(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 G_Io_Trigger_SoftwareOut_Read_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Number of the software output channel that is to be queried (starting with **1**)

If you prefer channel numbers starting with **0**, use command G_Io_Trigger_SoftwareOut_Read2 !

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_IO__TRIGGER__SOFTWARE_OUT__READ__RSP_FLAG__NONE`
No flag is set

`G_IO__TRIGGER__SOFTWARE_OUT__READ__RSP_FLAG__IS_SET`
The software out channel is set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Trigger_SoftwareOut_Read2

G_Io_Trigger_SoftwareOut_Read2 — Query software output state

Description

Use this command to query the state of an internal software output.

In contrast to command G_Io_Trigger_SoftwareOut_Read the channel number defined by `channel` is starting with **0** to provide a more convenient numbering scheme, especially in cross-interface-applications.

Definition

```
G_Error_t
G_Io_Trigger_SoftwareOut_Read(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 G_Io_Trigger_SoftwareOut_Read_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Number of the software output channel that is to be queried (starting with **0**)

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_IO__TRIGGER__SOFTWARE_OUT__READ__RSP_FLAG__NONE`
No flag is set

`G_IO__TRIGGER__SOFTWARE_OUT__READ__RSP_FLAG__IS_SET`
The software out channel is set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Trigger_SoftwareIn_Read

G_Io_Trigger_SoftwareIn_Read — Query software input state

Description

Use this command to query the state of an internal software input.

Definition

```
G_Error_t
G_Io_Trigger_SoftwareIn_Read(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 G_Io_Trigger_SoftwareIn_Read_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`Channel`
Software input channel (starting with **1**)

If you prefer channel numbers starting with **0**, use command G_Io_Trigger_SoftwareIn_Read2 !

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_IO__TRIGGER__SOFTWARE_IN__READ__RSP_FLAG__NONE`
No flag is set

`G_IO__TRIGGER__SOFTWARE_IN__READ__RSP_FLAG__IS_SET`
The software input channel is set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Trigger_SoftwareIn_Read2

G_Io_Trigger_SoftwareIn_Read2 — Query software input state

Description

Use this command to query the state of an internal software input.

In contrast to command G_Io_Trigger_SoftwareIn_Read the channel number defined by `channel` is starting with **0** to provide a more convenient numbering scheme, especially in cross-interface-applications.

Definition

```
G_Error_t
G_Io_Trigger_SoftwareIn_Read2(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 G_Io_Trigger_SoftwareIn_Read_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`Channel`
Software input channel (starting with **0**)

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_IO__TRIGGER__SOFTWARE_IN__READ__RSP_FLAG__NONE`
No flag is set

`G_IO__TRIGGER__SOFTWARE_IN__READ__RSP_FLAG__IS_SET`
The software input channel is set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Trigger_SoftwareIn_TsFifo_Init

G_Io_Trigger_SoftwareIn_TsFifo_Init — Init timestamp FIFO for a software input

Description

Use this command to init timestamp FIFO for a software input (allocate resources).

The timestamp FIFO records events (level changes) of a specific software input and assigns a timestamp to each event.

The timestamps can then be queried with G_Io_Trigger_SoftwareIn_TsFifo_Read .

Definition

```
G_Error_t
G_Io_Trigger_SoftwareIn_TsFifo_Init(
 const G_PortHandle_t  portHandle,
 const G_Io_Trigger_SoftwareIn_TsFifo_Init_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_IO__TRIGGER__SOFTWARE_IN__TS_FIFO__INIT__CMD_FLAG__NONE`
No flag is set

`G_IO__TRIGGER__SOFTWARE_IN__TS_FIFO__INIT__CMD_FLAG__NO_OFFSET`
0: `Channel` starting with 1

1: `Channel` starting with 0

`Channel`
Software input channel (offset depending on flag `G_IO__TRIGGER__SOFTWARE_IN__TS_FIFO__INIT__CMD_FLAG__NO_OFFSET`)

`Edge`
Edge of software input state change for triggering timestamp generation

The following values are possible:

`G_IO__TRIGGER__EDGE__UNKNOWN`
Edge is unknown

`G_IO__TRIGGER__EDGE__NONE`
No edge

`G_IO__TRIGGER__EDGE__FALLING`
Falling edge

`G_IO__TRIGGER__EDGE__RISING`
Rising edge

`G_IO__TRIGGER__EDGE__BOTH`
Rising and falling edge

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`FifoDepth`
Maximum number of entries the FIFO can store

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Trigger_SoftwareIn_TsFifo_Read

G_Io_Trigger_SoftwareIn_TsFifo_Read — Read timestamps from timestamp FIFO

Description

Use this command to read timestamps from the timestamp FIFO.

Definition

```
G_Error_t
G_Io_Trigger_SoftwareIn_TsFifo_Read(
 const G_PortHandle_t  portHandle,
 const G_Io_Trigger_SoftwareIn_TsFifo_Read_Cmd_t * const cmd,
 u16_t * const numberOfEntries,
 G_Io_Trigger_SoftwareIn_TsFifo_Entry_t * const entries
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_IO__TRIGGER__SOFTWARE_IN__TS_FIFO__READ__CMD_FLAG__NONE`
No flag is set

`G_IO__TRIGGER__SOFTWARE_IN__TS_FIFO__READ__CMD_FLAG__NO_OFFSET`
0: `Channel` starting with 1

1: `Channel` starting with 0

`Channel`
Software input channel (offset depending on flag `G_IO__TRIGGER__SOFTWARE_IN__TS_FIFO__READ__CMD_FLAG__NO_OFFSET`)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`numberOfEntries`
in : size of buffer `entries`

out : number of returned entries

`entries`
Returns timestamp FIFO entries

You need to probvide a buffer that is big enough for at least `numberOfEntries` entries.

The structure contains the following elements:

`Flags`
Entry flags

The following values are possible:

`G_IO__TRIGGER__SOFTWARE_IN__TS_FIFO__FLAG__NONE`
No flag is set

`G_IO__TRIGGER__SOFTWARE_IN__TS_FIFO__FLAG__FIFO_OVERFLOW`
The internal FIFO buffer was too small to store all entries, some entries are lost.

`Edge`
Edge of trigger event

The following values are possible:

`G_IO__TRIGGER__EDGE__UNKNOWN`
Edge is unknown

`G_IO__TRIGGER__EDGE__NONE`
No edge

`G_IO__TRIGGER__EDGE__FALLING`
Falling edge

`G_IO__TRIGGER__EDGE__RISING`
Rising edge

`G_IO__TRIGGER__EDGE__BOTH`
Rising and falling edge

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`TimeStamp`
Timestamp of trigger event, in ns

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Io_Trigger_SoftwareIn_TsFifo_Reset

G_Io_Trigger_SoftwareIn_TsFifo_Reset — Reset timestamp FIFO for a software input

Description

Use this command to reset timestamp FIFO for a software input (deallocate resources).

Definition

```
G_Error_t
G_Io_Trigger_SoftwareIn_TsFifo_Reset(
 const G_PortHandle_t  portHandle,
 const G_Io_Trigger_SoftwareIn_TsFifo_Reset_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following parameters:

`Flags`
Command flags

The following values are possible:

`G_IO__TRIGGER__SOFTWARE_IN__TS_FIFO__RESET__CMD_FLAG__NONE`
No flag is set

`G_IO__TRIGGER__SOFTWARE_IN__TS_FIFO__RESET__CMD_FLAG__NO_OFFSET`
0: `Channel` starting with 1

1: `Channel` starting with 0

`Channel`
Software input channel (offset depending on flag `G_IO__TRIGGER__SOFTWARE_IN__TS_FIFO__RESET__CMD_FLAG__NO_OFFSET`)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

K-Line Functions

These **G-API** functions provide access to the K-Line-Functionality of your hardware. If the return value of type **G_Error_t** does not correspond to G_NO_ERROR, an error has occurred. A description of this error can be retrieved by calling G_GetLastErrorDescription .

After a power-on or software reset, all **K-Line** interfaces are in an inactive state (HIGH level).

The term **Initialization** used in this K-Line command description means the following: The tester sends an initialization pattern to establish communication.

1 Diag

Diagnostic functions of the K-Line interface

These functions are used to configure a K-Line diagnostics protocol. After the protocol has been configured, the diagnostics can be run via the functions provided by **g_api_diag.dll** . See Diagnostic Functions for a list of possible diagnostics commands.

If applicable, each protocol provides a **Tester Present Service** . This service is used to maintain the communication by transmitting specific messages if no messages were sent or received during a defined period of time.

G_KLine_Diag_Config_Kw2000

G_KLine_Diag_Config_Kw2000 — Configure K-Line interface for **Kw2000** diagnostics

Description

Use this command to configure the K-Line interface for **Kw2000** diagnostics.

For configuring the diagnostics with **default values**, only few parameters are important. If the default values do not match your configuration, use the command flags to select the values to be configured manually.

Where applicable, the default value of each parameter is marked **bold** in the parameter description.

Definition

```
G_Error_t
G_KLine_Diag_Config_Kw2000(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_KLine_Diag_Config_Kw2000_CmdFlags_t  cmdFlags,
 const G_KLine_Diag_Config_Kw2000_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_KLINE__DIAG__CONFIG__KW2000__CMD_FLAG__NONE`
No flag is set

The diagnostics will be configured with default values where applicable.

`G_KLINE__DIAG__CONFIG__KW2000__CMD_FLAG__VERIFY_CHECKSUM`
The checksum of each received frame will be verified. If an invalid checksum value is received, the diagnostics will be stopped.

`G_KLINE__DIAG__CONFIG__KW2000__CMD_FLAG__SEPARATE_LENGTH_BYTE`
Insert a separate length byte into the header of the frame

`G_KLINE__DIAG__CONFIG__KW2000__CMD_FLAG__NO_ADDRESS_BYTES`
The header of the frame will not include any address bytes

`G_KLINE__DIAG__CONFIG__KW2000__CMD_FLAG__SET_SOURCE_ADDRESS`
Use `SourceAddress` parameter to set the source address value, otherwise use default value

`G_KLINE__DIAG__CONFIG__KW2000__CMD_FLAG__SET_TIMING`
Use parameters of structure `Timing` to set timing values, otherwise use default values

`G_KLINE__DIAG__CONFIG__KW2000__CMD_FLAG__SET_TESTER_PRESENT`
Use parameters of structure `TesterPresent` to set tester present values, otherwise use default values

G_KLINE__DIAG__CONFIG__KW2000__CMD_FLAG__SET_INIT
Use parameters of structure Init.Slow or Init.Fast to set init values depending on the value of Init.InitType, otherwise use default values (also dependening on Init.InitType)

G_KLINE__DIAG__CONFIG__KW2000__CMD_FLAG__SET_SPECIFIC_HANDLING
Use parameters of structure SpecificHandling to set specific handling values, otherwise use default values

parameters
Structure with command parameters

The structure contains the following elements:

TargetAddress
Address of the target (ECU) - necessary

InitMode
Initialization mode - necessary

The following values are possible:

G_KLINE__DIAG__KW2000__INIT_MODE__SLOW
Slow initialization (**5 baud init**)

G_KLINE__DIAG__KW2000__INIT_MODE__FAST
Fast initialization

SourceAddress
Source address of the tester

Optional, used if flag G_KLINE__DIAG__CONFIG__KW2000__CMD_FLAG__SET_SOURCE_ADDRESS is set.

Default value: **0xF1**

reserved
reserved parameter (must be initialized with **0**)

Timing
Structure with timing parameters

Optional, used if flag G_KLINE__DIAG__CONFIG__KW2000__CMD_FLAG__SET_TIMING is set, otherwise the default value for each element of the structure is assumed.

The structure contains the following elements:

P1min
Minimum interbyte time for responses of the ECU in milliseconds

Default value: **0 milliseconds**

P1max
Maximum interbyte time for responses of the ECU in milliseconds

Default value: **20 milliseconds**

P2min
Minimum time gap between the request of the tester and the response of the ECU, or minimum time gap between two responses of the ECU in milliseconds

Default value: **0 milliseconds**

`P2max`
Maximum time gap between the request of the tester and the response of the ECU, or maximum time gap between two responses of the ECU in milliseconds

Default value: **1000 milliseconds**

`P3min`
Minimum time gap between the response of the ECU and a new request of the tester in milliseconds

Default value: **0 milliseconds**

`P3max`
Maximum time gap between the response of the ECU and a new request of the tester in milliseconds

Default value: **5000 milliseconds**

`P4min`
Minimum interbyte gap time for requests of the tester in milliseconds

Default value: **0 milliseconds**

`P4max`
Maximum interbyte gap time for requests of the tester in milliseconds

Default value: **20 milliseconds**

`TesterPresent`
Structure with **TesterPresent** parameters

Optional, used if flag `G_KLINE__DIAG__CONFIG__KW2000__CMD_FLAG__SET_TESTER_PRESENT` is set, otherwise the default value for each element of the structure is assumed.

The structure contains the following elements:

`Flags`
TesterPresent flags

The following values are possible and can be combined:

`G_KLINE__DIAG__CONFIG__KW2000__TESTER_PRESENT_FLAG__NONE`
No flag is set (default value)

`G_KLINE__DIAG__CONFIG__KW2000__TESTER_PRESENT_FLAG__USE_RR_PARAM`
Use **TesterPresent ResponseRequired Parameter**

If set, the value of `RRParameter` is used as **TesterPresent Response Required Parameter** value

`SourceAddress`
TesterPresent source address

Default value = value for general source address

`TargetAddress`
TesterPresent target address

Default value = value for general target address

`RRParameter`
Tester Present Response Required Parameter

Optional, used if tester present flag G_KLINE__DIAG__CONFIG__KW2000__TESTER_PRESENT_FLAG__USE_RR_PARAM is set

reserved
reserved parameter (must be initialized with **0**)

Init_Slow
Structure with parameters for **slow initialization** (5 baud init)

Optional, used if flag G_KLINE__DIAG__CONFIG__KW2000__CMD_FLAG__SET_INIT is set, and InitMode = G_KLINE__DIAG__KW2000__INIT_MODE__SLOW.

If flag G_KLINE__DIAG__CONFIG__KW2000__CMD_FLAG__SET_INIT is not set, the default value for each element of the structure is assumed.

The Structure contains the following elements:

TargetAddress
Target address for slow initialization (5 baud address)

Default value = value for general target address

Parity
Parity for sending the address byte

The following values are possible:

G_KLINE__DIAG__CONFIG__KW2000__INIT__SLOW__PARITY__EVEN
Even parity

G_KLINE__DIAG__CONFIG__KW2000__INIT__SLOW__PARITY__ODD
Odd parity (default)

reserved1
reserved parameter (must be initialized with **0**)

reserved2
reserved parameter (must be initialized with **0**)

W1min
Minimum time gap between the end of the address byte and the start of the synchronization pattern in milliseconds

Default value: **0 milliseconds**

W1max
Maximum time gap between the end of the address byte and the start of the synchronization pattern in milliseconds

Default value: **300 milliseconds**

W2min
Minimum time gap between the end of the synchronization pattern and the start of **Keybyte 1** in milliseconds

Default value: **0 milliseconds**

W2max
Maximum time gap between the end of the synchronization pattern and the start of **Keybyte 1** in milliseconds

Default value: **20 milliseconds**

`W3min`
Minimum time gap between **Keybyte 1** and **Keybyte 2** in milliseconds

Default value: **0 milliseconds**

`W3max`
Maximum time gap between **Keybyte 1** and **Keybyte 2** in milliseconds

Default value: **20 milliseconds**

`W4min`
Minimum time gap between **Keybyte 2** of the ECU and its inversion by the tester as well as minimum time gap between the inverted **Keybyte 2** of the tester and the inverted address byte of the ECU in milliseconds

Default value: **25 milliseconds**

`W4max`
Maximum time gap between **Keybyte 2** of the ECU and its inversion by the tester as well as maximum time gap between the inverted **Keybyte 2** of the tester and the inverted address byte of the ECU in milliseconds

Default value: **50 milliseconds**

`W5min`
Minimum time gap before the tester starts to send the address byte in milliseconds

Default value: **300 milliseconds**

`W5max`
Maximum time gap before the tester starts to send the address byte in milliseconds

Default value: **300 milliseconds**

`Init_Fast`
Structure with parameters for **fast initialization**

Optional, used if flag `G_KLINE__DIAG__CONFIG__KW2000__CMD_FLAG__SET_INIT` is set, and `InitMode` = `G_KLINE__DIAG__KW2000__INIT_MODE__FAST`.

If flag `G_KLINE__DIAG__CONFIG__KW2000__CMD_FLAG__SET_INIT` is not set, the default value for each element of the structure is assumed.

The structure contains the following elements:

`SourceAddress`
Source address for fast initialization

Default value = value for general source address

`TargetAddress`
Target address for fast initialization

Default value = value for general target address

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`W5min`
Minimum time gap before the tester starts to send the address byte in milliseconds

Default value: **300 milliseconds**

`W5max`
Maximum time gap before the tester starts to send the address byte in milliseconds

Default value: **300 milliseconds**

`TWuPmin`
Minimum time period for the **Wake Up Pattern** in milliseconds

Default value: **50 milliseconds**

`TWuPmax`
Maximum time period for the **Wake Up Pattern** in milliseconds

Default value: **50 milliseconds**

`TIniLmin`
Minimum time period for the **Low** phase of the **Wake up Pattern** in milliseconds

Default value: **25 milliseconds**

`TIniLmax`
Maximum time period for the **Low** phase of the **Wake up Pattern** in milliseconds

Default value: **25 milliseconds**

`P1min`
Minimum interbyte time for responses of the ECU in milliseconds

Default value: **0 milliseconds**

`P1max`
Maximum interbyte time for responses of the ECU in milliseconds

Default value: **20 milliseconds**

`P2min`
Minimum time gap between the request of the tester and the response of the ECU in milliseconds

Default value: **0 milliseconds**

`P2max`
Maximum time gap between the request of the tester and the response of the ECU in milliseconds

Default value: **1000 milliseconds**

`P3min`
Minimum time gap between the response of the ECU and a new request of the tester in milliseconds

Default value: **0 milliseconds**

`P3max`
Maximum time gap between the response of the ECU and a new request of the tester in milliseconds

Default value: **5000 milliseconds**

`P4min`
Minimum interbyte gap time for requests of the tester in milliseconds

Default value: **0 milliseconds**

`P4max`
Maximum interbyte gap time for requests of the tester in milliseconds

Default value: **20 milliseconds**

`BaudRate`
Baud rate for fast initialization in Hz

Default value: **10400 Hz**

`SpecificHandling`
Structure with specific handling parameters

Optional, used if flag `G_KLINE__DIAG__CONFIG__KW2000__CMD_FLAG__SET_SPECIFIC_HANDLING` is set, otherwise the default value for each parameter is assumed.

The structure contains the following elements:

`BusyRepeatRequestMax`
Maximum number of **Busy Repeat Request (0x21)** responses to a request

If the maximum number is exceeded, a response with an error code will be sent to the host application and the request will not be repeated.

Value **0xFFFF** : unlimited

Default value: **0xFFFF** (unlimited)

`RoutineNotCompleteMax`
Maximum number of **Routine Not Complete (0x23)** responses to a request

If the maximum number is exceeded, a response with an error code will be sent to the host and the request will not be repeated.

Value **0xFFFF** : unlimited

Default value: **0xFFFF** (unlimited)

`RequestCorrectlyReceivedResponsePendingMax`
Maximum number of **Request Correctly Received Response Pending (0x78)** responses to a request

If the maximum number is exceeded, the communication will be aborted and a **NO_RESPONSE** error will be generated.

Value **0xFFFF** : unlimited

Default value: **0xFFFF** (unlimited)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

In the following example, the **K-Line** interface is configured for **Kw2000** diagnostics with **fast initialization** and to verify the checksum. Despite of the **fast initialization** parameters, default values are used.

```
const u8_t  channel = 0;
```

```
G_KLine_Diag_Config_Kw2000_CmdFlags_t  cmdFlags;
G_KLine_Diag_Config_Kw2000_Parameters_t  param;
G_Error_t  rc;

cmdFlags =
 G_KLINE__DIAG__CONFIG__KW2000__CMD_FLAG__VERIFY_CHECKSUM |
 G_KLINE__DIAG__CONFIG__KW2000__CMD_FLAG__SET_INIT;

// set all parameters to '0'
memset(&param, 0, sizeof(param));

param.TargetAddress = 0x69;

// set init fast parameters
param.InitMode = G_KLINE__DIAG__KW2000__INIT_MODE__FAST;

param.Init_Fast.SourceAddress = 0xF1;
param.Init_Fast.TargetAddress = 0x69;

param.Init_Fast.W5min = 300;
param.Init_Fast.W5max = 300;

param.Init_Fast.TWuPmin = 50;
param.Init_Fast.TWuPmax = 50;

param.Init_Fast.TIniLmin = 25;
param.Init_Fast.TIniLmax = 25;

param.Init_Fast.P1min = 0;
param.Init_Fast.P1max = 20;

param.Init_Fast.P2min = 25;
param.Init_Fast.P2max = 50;

param.Init_Fast.P3min = 55;
param.Init_Fast.P3max = 2000;

param.Init_Fast.P4min = 5;
param.Init_Fast.P4max = 20;

param.Init_Fast.BaudRate = 10400;

rc =
 G_KLine_Diag_Config_Kw2000(
  PortHandle1,
  channel,
  cmdFlags,
  &param
 );
```

G_KLine_Diag_Config_Kw1281

G_KLine_Diag_Config_Kw1281 — Configure K-Line interface for **Kw1281** diagnostics

Description

Use this command to configure the K-Line interface for **Kw1281** diagnostics.

For configuring the diagnostics with **default values**, only few parameters are important. If the default values do not match your configuration, use the command flags to select the values to be configured manually.

Where applicable, the default value of each parameter is marked **bold** in the parameter description.

Definition

```
G_Error_t
G_KLine_Diag_Config_Kw1281(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_KLine_Diag_Config_Kw1281_CmdFlags_t  cmdFlags,
 const G_KLine_Diag_Config_Kw1281_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_KLINE__DIAG__CONFIG__KW1281__CMD_FLAG__NONE`
No flag is set

The diagnostics will be configured with default values where applicable.

`G_KLINE__DIAG__CONFIG__KW1281__CMD_FLAG__SET_TIMING`
Use parameters of structure `Timing` to set timing values, otherwise use default values

`G_KLINE__DIAG__CONFIG__KW1281__CMD_FLAG__SET_INIT`
Use parameters of structure `Init` to set initialization values, otherwise use default values

`G_KLINE__DIAG__CONFIG__KW1281__CMD_FLAG__SET_SPECIFIC_HANDLING`
Use parameters of structure `SpecificHandling` to set specific handling values, otherwise use default values

`parameters`
Structure with **Kw1281** parameters

The structure contains the following elements:

`TargetAddress`
Target address for initialization (5 baud address)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Timing`
Structure with timing parameters

Optional, used if flag `G_KLINE__DIAG__CONFIG__KW1281__CMD_FLAG__SET_TIMING` is set, otherwise default values are used

The structure contains the following elements:

`T7min`
Minimum time gap between the bytes within one block, in milliseconds

Default value: **1 milliseconds**

`T7max`
Maximum time gap between the bytes within one block, in milliseconds

Default value: **50 milliseconds**

`T8min`
Minimum time for the repeated reception of the first byte of a block (if the slave has not received the last byte of a block), in milliseconds

Default value: **55 milliseconds**

`T8max`
Maximum time for the repeated reception of the first byte of a block (if the slave has not received the last byte of a block), in milliseconds

Default value: **55 milliseconds**

`T9min`
Minimum time gap between the end of a block and the start of the next block, in milliseconds

Default value: **1 milliseconds**

`T9max`
Maximum time gap between the end of a block and the start of the next block, in milliseconds

Default value: **1100 milliseconds**

`Init`
Structure with initialization parameters

Optional, used if flag `G_KLINE__DIAG__CONFIG__KW1281__CMD_FLAG__SET_INIT` is set, otherwise default values are used

The structure contains the following elements:

`Parity`
Parity of the address byte

The following values are possible:

`G_KLINE__DIAG__CONFIG__KW1281__INIT__PARITY__EVEN`
Even parity

`G_KLINE__DIAG__CONFIG__KW1281__INIT__PARITY__ODD`
Odd parity (default)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`T0min`
Minimum idle time before the start of initialization, in milliseconds

Default value: **0 milliseconds**

`T0max`
Maximum idle time before the start of initialization, in milliseconds

Default value: **0 milliseconds**

`T1min`
Minimum time gap between the correct initialization and the start of the synchronous byte, in milliseconds

Default value: **80 milliseconds**

`T1max`
Maximum time gap between the correct initialization and the start of the synchronous byte, in milliseconds

Default value: **231 milliseconds**

`T2min`
Minimum time gap between the synchronous byte and **Keybyte 1** , in milliseconds

Default value: **5 milliseconds**

`T2max`
Maximum time gap between the synchronous byte and **Keybyte 1** , in milliseconds

Default value: **20 milliseconds**

`T3min`
Minimum time gap between **Keybyte 1** and **Keybyte 2** , in milliseconds

Default value: **1 milliseconds**

`T3max`
Maximum time gap between **Keybyte 1** and **Keybyte 2** , in milliseconds

Default value: **20 milliseconds**

`T4min`
Minimal time gap between **Keybyte 2** and complement of **Keybyte 2** , in milliseconds

Default value: **25 milliseconds**

`T4max`
Maximum time gap between **Keybyte 2** and complement of **Keybyte 2** , in milliseconds

Default value: **50 milliseconds**

`T5min`
Minimum time gap between the complement of **Keybyte 2** and the repeated output of the synchronous byte (if the complement of **Keybyte 2** has not been received correctly by the control unit), in milliseconds

Default value: **25 milliseconds**

`T5max`
Maximum time gap between the complement of **Keybyte 2** and the repeated output of the synchronous byte (if the complement of **Keybyte 2** has not been received correctly by the control unit), in milliseconds

Default value: **50 milliseconds**

`T6min`
Minimum time gap between the complement of **Keybyte 2** and the start of the ECU identification, in milliseconds

Default value: **1 milliseconds**

`T6max`
Maximum time gap between the complement of **Keybyte 2** and the start of the ECU identification, in milliseconds

Default value: **50 milliseconds**

`SpecificHandling`
Structure with specific handling parameters

Optional, used if flag `G_KLINE__DIAG__CONFIG__KW1281__CMD_FLAG__SET_SPECIFIC_HANDLING` is set, otherwise the default value for each parameter is assumed.

The structure contains the following elements:

`MasterBlockRetry`
Maximum number of new attempts if errors occur during the transmission of a block

Input of the maximum number of repeated attempts to send a block, if errors occur during the transmission of a block (protocol driver is master).

Value **0xFFFF** : unlimited

Default value: **5**

`SlaveBlockRetry`
Maximum number of new attempts if errors occur during the reception of a block

Input of the maximum number of repeated attempts to receive a block, if errors occur during the reception of a block (protocol driver is slave).

(e.g. timeout during the reception of the next byte within one block of the master)

Value **0xFFFF** : unlimited

Default value: **5**

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_KLine_Diag_Config_Iso9141Ford

G_KLine_Diag_Config_Iso9141Ford — Configure K-Line interface for **Iso9141Ford** diagnostics

Description

Use this command to configure the K-Line interface for **Iso9141Ford** diagnostics.

For configuring the diagnostics with **default values**, only few parameters are important. If the default values do not match your configuration, use the command flags to select the values to be configured manually.

Where applicable, the default value of each parameter is marked **bold** in the parameter description.

Definition

```
G_Error_t
G_KLine_Diag_Config_Iso9141Ford(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_KLine_Diag_Config_Iso9141Ford_CmdFlags_t  cmdFlags,
 const G_KLine_Diag_Config_Iso9141Ford_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_KLINE__DIAG__CONFIG__ISO9141_FORD__CMD_FLAG__NONE`
No flag is set

`G_KLINE__DIAG__CONFIG__ISO9141_FORD__CMD_FLAG__VERIFY_CHECKSUM`
The checksum of each received frame will be verified. If an invalid checksum value is received, the diagnostics will be stopped.

`G_KLINE__DIAG__CONFIG__ISO9141_FORD__CMD_FLAG__SET_SOURCE_ADDRESS`
Use parameter `SourceAddress` to set the source address value, otherwise use default value

`G_KLINE__DIAG__CONFIG__ISO9141_FORD__CMD_FLAG__SET_TIMING`
Use parameters of structure `Timing` to set timing values, otherwise use default values

`G_KLINE__DIAG__CONFIG__ISO9141_FORD__CMD_FLAG__SET_SPECIFIC_HANDLING`
Use parameters of structure `SpecificHandling` to set specific handling values, otherwise use default values

`parameters`
Structure with Iso9141Ford parameters

The structure contains the following elements:

`TargetAddress`
Address of the target device (ECU) during diagnostics

`SourceAddress`
Tester address during diagnostics

Optional, used if flag `G_KLINE__DIAG__CONFIG__ISO9141_FORD__CMD_FLAG__SET_SOURCE_ADDRESS` is set, otherwise the default value is used.

Default value: **0xF0**

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Timing`
Structure with timing parameters

Optional, used if flag `G_KLINE__DIAG__CONFIG__ISO9141_FORD__CMD_FLAG__SET_TIMING` is set, otherwise default values are used

The structure contains the following elements:

`ReceiveInterByteGapMin`
Minimum interbyte time for responses of the ECU, in milliseconds

Default value: **0 milliseconds**

`ReceiveInterByteGapMax`
Maximum interbyte time for responses of the ECU, in milliseconds

Default value: **22 milliseconds**

`ResponseInterMsgGapMin`
Minimum time gap between the request of the tester and the response of the ECU, or minimum time gap between two responses of the ECU, in milliseconds

Default value: **0 milliseconds**

`ResponseInterMsgGapMax`
Maximum time gap between the request of the tester and the response of the ECU, or minimum time gap between two responses of the ECU, in milliseconds

Default value: **50 milliseconds**

`RequestInterMsgGapMin`
Minimum time gap between the response of the ECU and a new request of the tester, in milliseconds

Default value: **55 milliseconds**

`RequestInterMsgGapMax`
Maximum time gap between the response of the ECU and a new request of the tester, in milliseconds

Default value: **4000 milliseconds**

`TransmitInterByteGapMin`
Minimum interbyte time for requests of the tester, in milliseconds

Default value: **5 milliseconds**

`TransmitInterByteGapMax`
Maximum interbyte time for requests of the tester, in milliseconds

Default value: **20 milliseconds**

`SpecificHandling`
Structure with specific handling parameters

Optional, used if flag `G_KLINE__DIAG__CONFIG__ISO9141_FORD__CMD_FLAG__SET_SPECIFIC_HANDLING` is set, otherwise the default value for each parameter is assumed.

The structure contains the following elements:

`BusyRepeatRequestMax`
Maximum number of new attempts if errors occur during the transmission of a block

Value **0xFFFF** : unlimited

Default value: **5**

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_KLine_Diag_Config_WBus

G_KLine_Diag_Config_WBus — Configure K-Line interface for **WBus** diagnostics

Description

Use this command to configure the K-Line interface for **WBus** diagnostics.

For configuring the diagnostics with **default values**, only few parameters are important. If the default values do not match your configuration, use the command flags to select the values to be configured manually.

Where applicable, the default value of each parameter is marked **bold** in the parameter description.

Definition

```
G_Error_t
G_KLine_Diag_Config_WBus(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_KLine_Diag_Config_WBus_CmdFlags_t  cmdFlags,
 const G_KLine_Diag_Config_WBus_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_KLINE__DIAG__CONFIG__WBUS__CMD_FLAG__NONE`
No flag is set

`G_KLINE__DIAG__CONFIG__WBUS__CMD_FLAG__VERIFY_CHECKSUM`
The checksum of each received frame will be verified. If an invalid checksum value is received, the diagnostics will be stopped.

`G_KLINE__DIAG__CONFIG__WBUS__CMD_FLAG__SET_SOURCE_ADDRESS`
Use parameter `SourceAddress` to set the source address value, otherwise use default value

`G_KLINE__DIAG__CONFIG__WBUS__CMD_FLAG__SET_REPETITIONS`
Use parameter `Repetitions` to set the number of times the tester repeats a request if an error occurred while sending, or a **Negative Acknowledge** with **Repeat Request (0x21)** was received, otherwise use default value

`G_KLINE__DIAG__CONFIG__WBUS__CMD_FLAG__SET_BAUD_RATE`
Use parameter `BaudRate` to set the baud rate value, otherwise use default value

`G_KLINE__DIAG__CONFIG__WBUS__CMD_FLAG__SET_TIMING`
Use parameters of structure `Timing` to set timing values, otherwise use default values

`G_KLINE__DIAG__CONFIG__WBUS__CMD_FLAG__SET_WAKE_UP`
Use parameters of structure `WakeUp` to set wake up values, otherwise use default values

`parameters`
Structure with **WBus** parameters

The structure contains the following elements:

`TargetAddress`
Address of the target device (ECU)

`SourceAddress`
Tester address during diagnostics

Optional, used if flag `G_KLINE__DIAG__CONFIG__WBUS__CMD_FLAG__SET_SOURCE_ADDRESS` is set, otherwise the default value is used.

Default value: **0xF1**

`Repetitions`
Number of times the tester repeats a request if an error occurred while sending, or a **Negative Acknowledge** with **Repeat Request (0x21)** was received

Optional, used if flag `G_KLINE__DIAG__CONFIG__WBUS__CMD_FLAG__SET_REPETITIONS` is set, otherwise the default value is used.

Default value: **2**

`reserved`
reserved parameter (must be initialized with **0**)

`BaudRate`
Baud rate of the communication in baud

Optional, used if flag `G_KLINE__DIAG__CONFIG__WBUS__CMD_FLAG__SET_BAUD_RATE` is set, otherwise the default value is used.

Default value: **2400 baud**

`Timing`
Structure with timing parameters

Optional, used if flag `G_KLINE__DIAG__CONFIG__WBUS__CMD_FLAG__SET_TIMING` is set, otherwise default values are used

The structure contains the following elements:

`ResponseTimeout`
Timeout for an ECU to respond to a request, in microseconds

Default value: **100000 microseconds**

`ResponseByteTimeout`
Maximum time between two bytes of a received message, in microseconds

If exceeded, the last received byte is interpreted as the start of a new message.

Default value: **4500 microseconds**

`RetryTime`
Time until the tester repeats a request if an error occurred while sending, or a **Negative Acknowledge** with **Repeat Request (0x21)** was received, in microseconds

Default value: **100000 microseconds**

`BusIdleTime`
Minimum time the bus has to be idle before the next request can be sent, in microseconds

Default value: **41000 microseconds**

`WakeUp`
Structure with wake up parameters

Optional, used if flag `G_KLINE__DIAG__CONFIG__WBUS__CMD_FLAG__SET_WAKE_UP` is set, otherwise default values are used

The structure contains the following elements:

`Flags`
Wake up flags

The following values are possible and can be combined:

`G_KLINE__DIAG__CONFIG__WBUS__WAKE_UP_FLAG__NONE`
No flag is set

`G_KLINE__DIAG__CONFIG__WBUS__WAKE_UP_FLAG__WAKE_UP_ON_START`
Send a wake up pulse at the start of the diagnostics

`G_KLINE__DIAG__CONFIG__WBUS__WAKE_UP_FLAG__WAKE_UP_AFTER_BUS_IDLE`
Send a wake up pulse before sending the next request, when the tester is for at least `BusIdleTimeBeforeWakeUp` microseconds in **bus idle state**

`TWuP`
Duration of the wake up pulse, in microseconds

Default value: **50000 microseconds**

`TIniL`
Duration of the **Low Level** of the wake up pulse, in microseconds

Default value: **25000 microseconds**

`BusIdleTimeBeforeWakeUp`
Minimum time without bus activity before sending a wake up pulse preceding to a request, in microseconds

Default value: **60000000 microseconds**

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

2 FIFO

FIFO functions of the K-Line interface

These functions provide control over the **FIFO** functionality of the K-Line interface.

FIFO is an acronym for **First In, First Out** and in conjunction with a K-Line interface of your **GOEPEL-electronic** device, it represents a queue of bytes in the transmit and receive buffer. This enables you to send and receive data independent of any diagnostics protocol.

G_KLine_Fifo_Reset

G_KLine_Fifo_Reset — Reset FIFO buffer

Description

This command resets the FIFO buffer of the K-Line interface.

Definition

```
G_Error_t
G_KLine_Fifo_Reset(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_KLine_Fifo_Init

G_KLine_Fifo_Init — Initialize FIFO buffer

Description

This command initializes the FIFO buffer of the K-Line interface.

Before the FIFO functionality of the interface can be used, it has to be initialized with this command.

Definition

```
G_Error_t
G_KLine_Fifo_Init(
 const G_PortHandle_t  portHandle,
 const G_KLine_Fifo_Init_CmdFlags_t  cmdFlags,
 const G_KLine_Fifo_Init_Parameters_t * const parameters,
 u32_t * const actualBaudRate
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_KLINE__FIFO__INIT__CMD_FLAG__NONE`
No flag is set

`parameters`
Structure with FIFO parameters

The structure contains the following elements:

`BaudRate`
K-Line baud rate in baud

`TxBufferSize`
Size of the transmit buffer in bytes (0 = default size)

The bigger the size of the transmit buffer, the more data can be queued. This is important if for example a large amount of data has to be transmitted with a slow baud rate.

If the buffer is too small to hold the transmit data, the function will return with an error.

`RxBufferSize`
Size of the receive buffer in bytes (0 = default size)

The bigger the size of the receive buffer, the more data can be queued. This is important if the buffer is not read frequently.

If the buffer is too small to hold the received data, the data will be lost and a buffer overrun will be signaled.

`UartMode`
UART mode of the K-Line interface

The following values are possible:

`G_KLINE__FIFO__UART_MODE__8_BIT_DATA`
Data is transmitted as **8 bits**

`G_KLINE__FIFO__UART_MODE__7_BIT_DATA__1_BIT_PARITY`
Data is transmitted as **7 bits** and **1 bit parity**

`G_KLINE__FIFO__UART_MODE__8_BIT_DATA__1_BIT_PARITY`
Data is transmitted as **8 bits** and **1 bit parity**

`Parity`
Parity of the transmitted data

Only applicable when `UartMode` = `G_KLINE__FIFO__UART_MODE__7_BIT_DATA__1_BIT_PARITY` or `UartMode` = `G_KLINE__FIFO__UART_MODE__8_BIT_DATA__1_BIT_PARITY`

The following values are possible:

`G_KLINE__FIFO__PARITY__EVEN`
The parity bit is set if the number of active data bits is **even**

`G_KLINE__FIFO__PARITY__ODD`
The parity bit is set if the number of active data bits is **odd**

`StopBits`
Number of stop bits that are appended

The following values are possible:

`G_KLINE__FIFO__STOP_BITS__ONE`
One stop bit is appended

`G_KLINE__FIFO__STOP_BITS__TWO`
Two stop bits are appended

`reserved`
reserved parameter (must be initialized with **0**)

`actualBaudRate`
Returns the actual baud rate

Because not all desired baud rate values can be set exactly, this parameter returns the actual baud rate value set. Therefore it may differ from your desired baud rate but represents the closest possible match on this device.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_KLine_Fifo_Tx_Write

G_KLine_Fifo_Tx_Write — Write data to the transmit queue of the FIFO

Description

Use this command to write data to the transmit queue of the FIFO.

The data will be queued and transmitted sequentially.

Definition

```
G_Error_t
G_KLine_Fifo_Tx_Write(
 const G_PortHandle_t  portHandle,
 const u32_t  numberOfBytes,
 const u8_t * const data
);
```

Parameters

`portHandle`
: Handle to the communication port

`numberOfBytes`
: Number of bytes to be written to the transmit queue

`data`
: Data bytes to be written to the transmit queue

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

In this example, 4 bytes are written to the transmit queue of the FIFO.

```
const u8_t  data[] = {0, 1, 2, 3};
G_Error_t  rc;

rc =
 G_KLine_Fifo_Tx_Write(
  PortHandle,
  4,
  data
 );
```

G_KLine_Fifo_Tx_GetState

G_KLine_Fifo_Tx_GetState — Query the state of the transmit queue of the FIFO

Description

Use this command to query the state of the transmit queue of the FIFO.

Definition

```
G_Error_t
G_KLine_Fifo_Tx_GetState(
 const G_PortHandle_t  portHandle,
 G_KLine_Fifo_Tx_GetState_RspFlags_t * const rspFlags,
 u32_t * const fifoSize,
 u32_t * const fifoFillingLevel
);
```

Parameters

`portHandle`
Handle to the communication port

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_KLINE__FIFO__TX__GET_STATE__RSP_FLAG__NONE`
No flag is set

`G_KLINE__FIFO__TX__GET_STATE__RSP_FLAG__FIFO_IS_EMPTY`
The transmit queue is empty

`G_KLINE__FIFO__TX__GET_STATE__RSP_FLAG__FIFO_IS_FULL`
The transmit queue is full

`fifoSize`
Returns the size of the transmit queue in bytes

`fifoFillingLevel`
Returns the number of bytes that are currently in the transmit queue

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_KLine_Fifo_Rx_Read

G_KLine_Fifo_Rx_Read — Read data from receive queue of the FIFO

Description

Use this command to read received data from the receive queue of the FIFO.

To avoid buffer overruns, it is advisable to frequently read data from the receive queue.

Definition

```
G_Error_t
G_KLine_Fifo_Rx_Read(
 const G_PortHandle_t  portHandle,
 G_KLine_Fifo_Rx_Read_RspFlags_t * const rspFlags,
 u32_t * const numberOfBytes,
 u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_KLINE__FIFO__RX__READ__RSP_FLAG__NONE`
No flag is set

`G_KLINE__FIFO__RX__READ__RSP_FLAG__FIFO_NOT_EMPTY`
The receive queue of the FIFO contains data

`G_KLINE__FIFO__RX__READ__RSP_FLAG__FIFO_OVERRUN`
A buffer overrun occurred - data was lost

`numberOfBytes`
in : size of your buffer (`data`) in bytes

out : size of returned data in bytes

If the size of the available data exceeds the size of your buffer, the command will return with errorcode **0x020F0003** (`G_ERROR__DLL__KLINE__RESPONSE_BUFFER_TOO_SMALL`) and `data` will contain all bytes that could be returned regarding the size of your buffer.

Subsequent calls will then return the rest of the data with regards to your buffer size. (See example)

`data`
Returns data bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

In this example, the receive queue is read with a too small user buffer size and therefore, subsequent calls are necessary to get all available data.

```
G_KLine_Fifo_Rx_Read_RspFlags_t  rspFlags;
u8_t * buffer;
u32_t  numberOfBytes;
G_Error_t  rc;

// allocate buffer for 200 bytes
numberOfBytes = 200;
buffer = malloc(numberOfBytes);

if (buffer == NULL) {
 return -1;
}

rc =
 G_KLine_Fifo_Rx_Read(
  PortHandle,
  &rspFlags,
  &numberOfBytes,
  buffer
 );

if (rc == G_ERROR__DLL__KLINE__RESPONSE_BUFFER_TOO_SMALL) {
 // the buffer was too small to return all available bytes
 // allocate a bigger buffer and get remaining bytes

 free(buffer);
 numberOfBytes = 10000;
 buffer = malloc(numberOfBytes);

 if (bigBuffer == NULL) {
  return -1;
 }

 rc =
  G_KLine_Fifo_Rx_Read(
   PortHandle,
   &rspFlags,
   &numberOfBytes,
   buffer
  );

 if (rc != G_NO_ERROR) {
  // another error has occurred -> quit
  free(buffer);
  return -1;
 }

 // all data could be read and is available in 'buffer'
}
```

G_KLine_Fifo_Rx_GetState

G_KLine_Fifo_Rx_GetState — Query state of the receive queue

Description

Use this command to query the state of the receive queue of the FIFO.

Definition

```
G_Error_t
G_KLine_Fifo_Rx_GetState(
 const G_PortHandle_t  portHandle,
 G_KLine_Fifo_Rx_GetState_RspFlags_t * const rspFlags,
 u32_t * const fifoSize,
 u32_t * const fifoFillingLevel
);
```

Parameters

`portHandle`
Handle to the communication port

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_KLINE__FIFO__RX__GET_STATE__RSP_FLAG__NONE`
No flag is set

`G_KLINE__FIFO__RX__GET_STATE__RSP_FLAG__FIFO_IS_EMPTY`
The receive queue is empty

`G_KLINE__FIFO__RX__GET_STATE__RSP_FLAG__FIFO_IS_FULL`
The receive queue is full

`G_KLINE__FIFO__RX__GET_STATE__RSP_FLAG__FIFO_OVERRUN`
A buffer overrun occurred

Not all received data could be queued, because the buffer is full. The data is lost.

`fifoSize`
Size of the receive queue buffer in bytes

`fifoFillingLevel`
Number of bytes within receive queue buffer

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

3 Monitor

Monitor functions of the K-Line interface

These functions provide control over the **Monitor** functionality of the K-Line interface.

There are two different types of monitors for the K-Line interface. The **Regular Monitor** is used to monitor single byte data. The **Advanced Monitor** is used for item based monitoring. Since this monitor is independent of any diagnostics protocol, you need to configure under which conditions a set of bytes is interpreted as an item.

A common command sequence of a monitor application would look like this:

- Configure monitor with G_KLine_Monitor_Config_Regular or G_KLine_Monitor_Config_Advanced
- Start monitor with G_KLine_Monitor_Start
- Query monitor data with G_KLine_Monitor_GetItems_Regular or G_KLine_Monitor_GetItems_Advanced
- Stop monitor with G_KLine_Monitor_Stop or switch off monitor with G_KLine_Monitor_Config_Off

G_KLine_Monitor_Config_Off

G_KLine_Monitor_Config_Off — Disable monitor

Description

Use this function to disable the K-Line monitor.

The monitoring will be stopped and all recources will be freed.

This function is applicable for the **Regular** and the **Advanced Monitor** .

Definition

```
G_Error_t
G_KLine_Monitor_Config_Off(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_KLine_Monitor_Config_Regular

G_KLine_Monitor_Config_Regular — Configure Regular Monitor

Description

Use this function to configure the **Regular K-Line Monitor** .

The **Regular K-Line Monitor** can monitor single bytes. For item based monitoring, use the **Advanced K-Line Monitor** .

Definition

```
G_Error_t
G_KLine_Monitor_Config_Regular(
 const G_PortHandle_t  portHandle,
 const G_KLine_Monitor_Config_Regular_CmdFlags_t  cmdFlags,
 const u32_t  bufferSize
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_KLINE__MONITOR__CONFIG__REGULAR__CMD_FLAG__NONE`
No flag is set

`bufferSize`
Size of the internal monitor data buffer in bytes (0 = default size)

To avoid the risk of buffer overruns, a fairly large size (10000 bytes or more) is recommended.

By setting `bufferSize` = 0, the default value (20000 bytes) is assumed.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_KLine_Monitor_Config_Advanced

G_KLine_Monitor_Config_Advanced — Configure Advanced Monitor

Description

Use this command to configure the **Advanced K-Line Monitor**.

The **Advanced K-Line Monitor** provides item based monitoring. Since it is executed independent from any diagnostics protocol, you need to specify the conditions under which a set of data bytes is interpreted as an item.

Definition

```
G_Error_t
G_KLine_Monitor_Config_Advanced(
 const G_PortHandle_t  portHandle,
 const G_KLine_Monitor_Config_Advanced_CmdFlags_t  cmdFlags,
 const u8_t  byteCount,
 const u32_t  time,
 const u32_t  bufferSize
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_KLINE__MONITOR__CONFIG__ADVANCED__CMD_FLAG__NONE`
No flag is set

Since at least one condition for dividing data into items is necessary, this value is **invalid**.

`G_KLINE__MONITOR__CONFIG__ADVANCED__CMD_FLAG__DIVIDE_BY_BYTE_COUNT`
Divide items by byte count

When the number of monitored bytes has reached the value specified by `byteCount`, the next byte will be interpreted as the start of a new item.

Use this divide condition if your items have a fixed number of bytes.

`G_KLINE__MONITOR__CONFIG__ADVANCED__CMD_FLAG__DIVIDE_BY_TIME`
Divide items by time

When the time gap between two bytes exceeds the value specified by `time`, the second byte will be interpreted as the start of a new item.

Use this divide condition if the time gap between the bytes **within an item** differs significantly from the time gap **between two items**.

`G_KLINE__MONITOR__CONFIG__ADVANCED__CMD_FLAG__DIVIDE_BY_DIRECTION`
Divide items by direction

When two bytes differ in direction (**TX** vs. **RX**), the second byte is interpreted as the start of a new item.

This divide condition is applicable in most cases.

`byteCount`
Byte count for divide condition **Divide By Byte Count**

Optional, used if flag `G_KLINE__MONITOR__CONFIG__ADVANCED__CMD_FLAG__DIVIDE_BY_BYTE_COUNT` is set

`time`
Time value for divide condition **Divide By Time** , in microseconds

Optional, used if flag `G_KLINE__MONITOR__CONFIG__ADVANCED__CMD_FLAG__DIVIDE_BY_TIME` is set

`bufferSize`
Size of the internal monitor data buffer in bytes (0 = default size)

To avoid the risk of buffer overruns, a fairly large size (10000 bytes or more) is recommended.

By setting `bufferSize` = 0, the default value (20000 bytes) is assumed.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_KLine_Monitor_Start

G_KLine_Monitor_Start — Start K-Line monitor

Description

Use this command to start the K-Line monitor.

Prior to this command call, the monitor needs to be configured with G_KLine_Monitor_Config_Regular or G_KLine_Monitor_Config_Advanced .

Definition

```
G_Error_t
G_KLine_Monitor_Start(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_KLine_Monitor_Stop

G_KLine_Monitor_Stop — Stop K-Line monitor

Description

Use this command to stop the K-Line monitor.

By stopping the monitor, all recources will not be freed and the monitor configuration stays active.

To resume monitoring, use G_KLine_Monitor_Start .

Definition

```
G_Error_t
G_KLine_Monitor_Stop(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_KLine_Monitor_GetItems_Regular

G_KLine_Monitor_GetItems_Regular — Query regular monitor items

Description

Use this command to query items of the **Regular K-Line Monitor** .

Definition

```
G_Error_t
G_KLine_Monitor_GetItems_Regular(
 const G_PortHandle_t  portHandle,
 u32_t * const numberOfItems,
 G_KLine_Monitor_Item_t * const items
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfItems`
in : size of your buffer `items` in number of items

out : number of items returned in `items`

`items`
Returns regular monitor items

A regular monitor item consists of the following elements:

`Timestamp`
Timestamp in nanoseconds

`Flags`
Item Flags

The following values are possible and can be combined:

`G_KLINE__MONITOR__ITEM__FLAG__NONE`
No flag is set

`G_KLINE__MONITOR__ITEM__FLAG__TX`
The item direction was **TX** (the item was transmitted)

If this flag is not set, the item direction was **RX** (the item was received).

`G_KLINE__MONITOR__ITEM__FLAG__COLLISION`
There was a collision on the bus. The returned data may be invalid.

`G_KLINE__MONITOR__ITEM__FLAG__EVENT`
A monitor event was received

`G_KLINE__MONITOR__ITEM__FLAG__BUFFER_OVERRUN`
A buffer overrun occurred

Not all monitored data could be stored in the internal buffer, because it was full.

`Data`
Item data (1 byte)

`reserved`
Reserved byte

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_KLine_Monitor_GetItems_Advanced

G_KLine_Monitor_GetItems_Advanced — Query advanced monitor items

Description

Use this command to query items of the **Advanced K-Line Monitor** .

Since advanced monitor items do not have a fixed length, they are copied into the raw data buffer `data` one after another with respect to the alignment.

The alignment size is a multiple of **4** and therefore a monitor item, e.g. consisting of 18 bytes, will demand a size of 20 bytes in the raw data buffer.

The parameter `ItemSize` specifies the size of the item in bytes including the alignment bytes.

Definition

```
G_Error_t
G_KLine_Monitor_GetItems_Advanced(
 const G_PortHandle_t  portHandle,
 u32_t * const numberOfBytes,
 u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfBytes`
in : size of your buffer `data` in bytes

out : number of bytes returned in `data`

`data`
Returns raw item data

The items have to be extracted from this buffer (see Example for details).

An **Advanced K-Line Monitor Item** contains the following elements:

`ItemSize`
Size of the item in bytes (including alignment bytes)

`Flags`
Item Flags

The following values are possible and can be combined:

`G_KLINE__MONITOR__ITEM__FLAG__NONE`
No flag is set

`G_KLINE__MONITOR__ITEM__FLAG__TX`
The item direction was **TX** (the item was transmitted)

If this flag is not set, the item direction was **RX** (the item was received).

`G_KLINE__MONITOR__ITEM__FLAG__COLLISION`
There was a collision on the bus. The returned data may be invalid.

`G_KLINE__MONITOR__ITEM__FLAG__EVENT`
A monitor event was received

`G_KLINE__MONITOR__ITEM__FLAG__BUFFER_OVERRUN`
A buffer overrun occurred

Not all monitored data could be stored in the internal buffer, because it was full.

`DataLength`
Data length of the item in bytes (number of bytes returned, starting at `Data`)

`Timestamp`
Timestamp in nanoseconds

`Data`
Buffer with data bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

In this example, advanced monitor items are queried and extracted from the raw data buffer.

```
u32_t  numberOfBytes;
u8_t  data[1000];
u8_t  offset;
const G_KLine_Monitor_Item_Advanced_t * item;
u32_t  i;
G_Error_t  rc;

// configure monitor
rc =
 G_KLine_Monitor_Config_Advanced(
  PortHandle,
  G_KLINE__MONITOR__CONFIG__ADVANCED__CMD_FLAG__DIVIDE_BY_DIRECTION,
  0,  // byte count
  0,  // time
  20000  // bufferSize
 );

if (rc != G_NO_ERROR) {
 return -1;
}

// start monitor
rc = G_KLine_Monitor_Start(PortHandle);

if (rc != G_NO_ERROR) {
 return -1;
}

// wait 1s to let items be received
Sleep(1000);

// query items
numberOfBytes = 1000;
```

```
rc =
 G_KLine_Monitor_GetItems_Advanced(
  PortHandle,
  &numberOfBytes,
  data
 );

if (rc != G_NO_ERROR) {
 return -1;
}

// extract items from raw data buffer
item = (G_KLine_Monitor_Item_Advanced_t *) data;

for (offset = 0; offset < numberOfBytes;) {
 printff("Timestamp: "%I64d\n", item->Timestamp);
 printf("Flags: %0x\n", item->Flags);
 printf("DataLength: %u\n", item->DataLength);

 printf("Data: ");

 for (i = 0; i < item->DataLength; i++) {
  printf(%0x ", item->Data[i]);
 }

 printf("\n-----------------\n");

 // move pointer to next item
 offset += item->ItemSize;
 item = (G_KLine_Monitor_Item_Advanced_t *) &data[offset];
}

// reset monitor
rc = G_KLine_Monitor_Config_Off(PortHandle);

if (rc != G_NO_ERROR) {
 return -1;
}
```

G_KLine_Monitor_EnableEvent

G_KLine_Monitor_EnableEvent — Enable monitor event

Description

This command enables the monitor to signal the event specified by `eventHandle` as soon as monitor data is received.

For general notes on **G-API** event handling, see Section 5, "Events".

Definition

```
G_Error_t
G_KLine_Monitor_EnableEvent(
 const G_PortHandle_t portHandle,
 const G_Common_EventHandle_t * const eventHandle
);
```

Parameters

`portHandle`
Handle to the communication port

`eventHandle`
Handle to the event

Each time monitor data is received, the specified handle is signaled.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_KLine_Monitor_DisableEvent

G_KLine_Monitor_DisableEvent — Disable monitor event

Description

This command disables a monitor event.

After this function has been executed, no event will be signaled when new monitor data is received.

For general notes on **G-API** event handling, see Section 5, "Events".

Definition

```
G_Error_t
G_KLine_Monitor_DisableEvent(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

4 Node

Node specific functions of the K-Line interface

These functions are used to control K-Line node features.

G_KLine_Node_Property_SetById

G_KLine_Node_Property_SetById — Set node property value by ID

Description

Use this command to set a K-Line node property value.

The current value of a node property can be queried with G_KLine_Node_Property_GetById .

Definition

```
G_Error_t
G_KLine_Node_Property_SetById(
 const G_PortHandle_t  portHandle,
 const G_KLine_Node_PropertyId_t  id,
 const u32_t  value
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Property ID

The value of `id` specifies the node property to be set.

The following values are possible:

`G_KLINE__NODE__PROPERTY_ID__UNKNOWN`
This value is just a placeholder

`G_KLINE__NODE__PROPERTY_ID__ENABLE_RX_TX_SPLIT`
Enable splitting of receive- and transmit-path of the K-Line transceiver (only available on selected K-Line transceivers)

`G_KLINE__NODE__PROPERTY_ID__ENABLE_BUS_TERMINATION`
Enable bus termination

`G_KLINE__NODE__PROPERTY_ID__ENABLE_INTERNAL_VBAT`
Enable transceiver supply voltage

`value`
Property value to be set

For boolean properties the value **0** equals **false** and a value **> 0** equals **true** .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_KLine_Node_Property_GetById

G_KLine_Node_Property_GetById — Query node property value by ID

Description

Use this command to query a K-Line node property value.

The value of a node property can be set with G_KLine_Node_Property_SetById .

Definition

```
G_Error_t
G_KLine_Node_Property_GetById(
 const G_PortHandle_t  portHandle,
 const G_KLine_Node_PropertyId_t  id,
 u32_t * const value
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Property ID

The value of `id` specifies the node property to be queried (see G_KLine_Node_PropertyId_t).

`value`
Returns property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

LIN Functions

These **G-API** functions provide access to the LIN-Functionality of your Hardware. If the return value of type **G_Error_t** is not equal to G_NO_ERROR, an error has occurred. A description of this error can be retrieved by calling G_GetLastErrorDescription .

1 Overview

1.1 Initial State

In the initial state of your Hardware, the master task is deactivated, while the slave task is activated (for monitoring). The slave task operates with an automatic baud rate recognition, a **BreakDetection-Threshold** of **9.5** bit times and with the classic checksum model.[1] This way, the LIN bus can be monitored by using the LIN bus monitor without providing the baud rate.

The firmware internal **ResponseSpace** and **InterByteSpace** variables are initialized by **0** . However, on the LIN bus **ResponseSpace** and **InterByteSpace** appear with **0..1** bit times, as the UART is used for transmitting.

The firmware operates with a value of **9.5** bit times for the **BreakDetectionThreshold** in contrast to the LIN Specification (11 bit times) for bus monitoring, to be able to detect breaks shorter than 11 bit times. At any time, the value for the **BreakDetectionThreshold** can be modified by the G_Lin_SlaveTask_SetBreakDetectionThreshold command.

1.2 Structure of a LIN Cluster

The following information is taken from the LIN Specification 2.0.

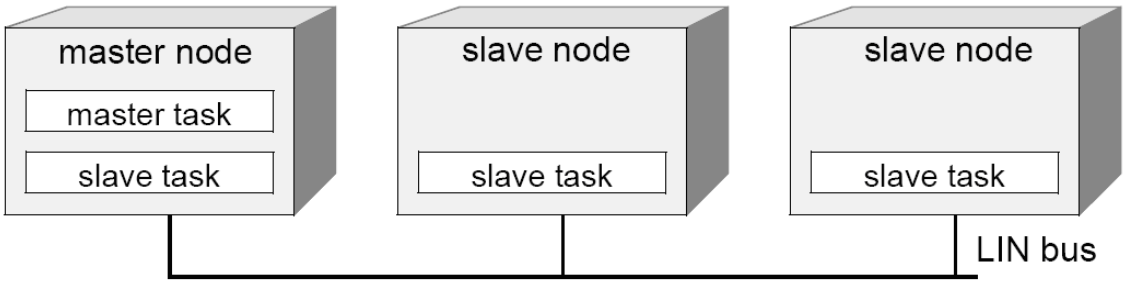

Figure 1 Structure of a LIN Cluster

A LIN cluster consists of one **master task** and several **slave tasks** . The LIN master (**master node**) includes one **master task** and one **slave task** . Each LIN slave (**slave node**) includes only one **slave task** .

1.3 Structure of a LIN Frame

The following information is taken from the LIN Specification 2.0.

[1] **Classic Checksum** = checksum via data bytes

Enhanced Checksum = checksum via identifier byte and data bytes

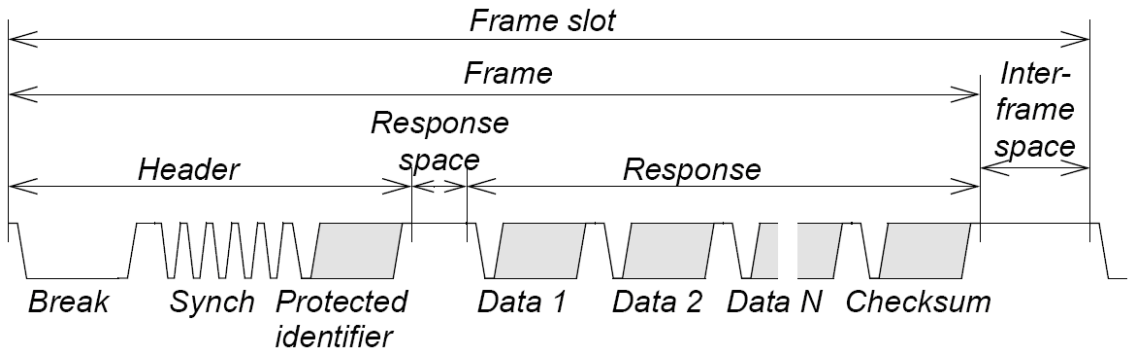

Figure 2 Structure of a LIN Frame

Essentially a LIN **Frame** (LIN message) consists of the LIN Frame **Header** and the LIN Frame **Response** . The **Header** is exclusively sent by the **master task** of the LIN master (**master node**). The **Response** is sent by a **slave task** of the **master node** or of a **slave node** .

The pause between **Header** and **Response** is called **Response space** .

The following example shows a LIN message of two data bytes, consisting of **Header** and **Response** . All time-parameters whithin a LIN message changeable by the firmware can be seen (**BreakTime** , **BreakDelimiterTime** , **ArbitrationTime** , **ResponseSpace** and **InterByteSpace**):

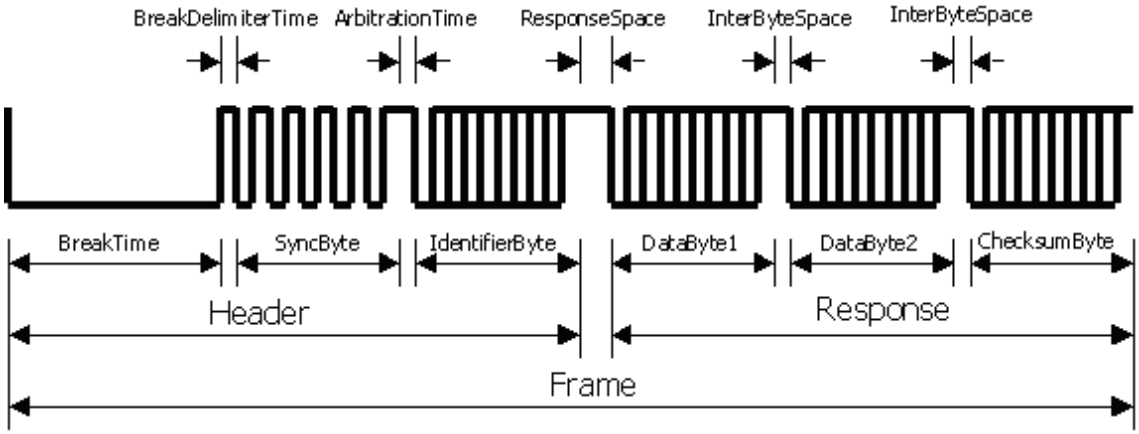

Figure 3 LIN Message

The continous lines above and below the LIN signal indicate that this signal can have **VBat** (high) level as well as **Gnd** (low) level during the corresponding times.

2 Common

These functions provide control over common LIN-Interface parameters.

G_Lin_InitInterface

G_Lin_InitInterface — Initialise LIN Interface

Description

This command resets the selected LIN interface without software reset to the initial state. Additionally, further configuration possibilities are offered.

Definition

```
G_Error_t
G_Lin_InitInterface(
  const G_PortHandle_t portHandle,
  const G_Lin_InitInterface_CmdFlags_t cmdFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
command flags

the following values are possible and can be combined:

- `G_LIN__INIT_INTERFACE__CMD_FLAG__NONE`

  no flag set
- `G_LIN__INIT_INTERFACE__CMD_FLAG__DONT_USE_UART_FOR_TX`

  do not use UART for transmitting
- `G_LIN__INIT_INTERFACE__CMD_FLAG__RESET_RELAYS`

  all relays will be reset
- `G_LIN__INIT_INTERFACE__CMD_FLAG__ENABLE_BLINKING`

  enable periodic blinking of the LEDs

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
G_Lin_InitInterface_CmdFlags_t flags =
  G_LIN__INIT_INTERFACE__CMD_FLAG__RESET_RELAYS |
  G_LIN__INIT_INTERFACE__CMD_FLAG__ENABLE_BLINKING;

rc =
  G_Lin_InitInterface(
  portHandle,
  flags
);
```

The Interface which is dedicated to `portHandle` is reset including its relays, and the blinking of the LEDs is activated.

3 Node

These functions provide control over node-specific interface parameters.

G_Lin_Node_Config

G_Lin_Node_Config — Set Common Interface Properties

Description

The common interface properties are set by this command. More detailed properties can be set by command G_Lin_Node_Config2.

Definition

```
G_Error_t
G_Lin_Node_Config(
 const G_PortHandle_t  portHandle,
 const G_Lin_Node_Config_CmdFlags_t  cmdFlags,
 const u32_t  baudRate
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
command flags

The following values are possible and can be combined:

- `G_LIN__NODE__CONFIG__CMD_FLAG__NONE`

  no flag set
- `G_LIN__NODE__CONFIG__CMD_FLAG__ENABLE_MASTER_TASK`

  enable Master task
- `G_LIN__NODE__CONFIG__CMD_FLAG__ENABLE_SLAVE_TASK`

  enable Slave task
- `G_LIN__NODE__CONFIG__CMD_FLAG__ENABLE_BAUD_RATE_DETECTION`

  enable detection of the Baud rate

For monitoring, the slave task must be activated. That means, the **G_LIN__NODE__CONFIG__CMD_FLAG__ENABLE_SLAVE_TASK** -flag has to be set.

When the **G_LIN__NODE__CONFIG__CMD_FLAG__ENABLE_BAUD_RATE_DETECTION** - flag is set, the baud rate will be detected automatically and the value under `baudRate` will be disregarded.

`baudRate`
Baud rate in Hertz

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
G_Lin_Node_Config_CmdFlags_t flags =
```

```
 G_LIN__NODE__CONFIG__CMD_FLAG__ENABLE_MASTER_TASK |
 G_LIN__NODE__CONFIG__CMD_FLAG__ENABLE_BAUD_RATE_DETECTION;

rc =
 G_Lin_Node_Config(
 portHandle,
 cmdFlags,
 0
);
```

This command call enables the master task and the detection of the baud rate. Because of the automatic baud rate detection there is no need to provide a valid baud rate value with the third parameter.

G_Lin_Node_Config2

G_Lin_Node_Config2 — Set Interface Properties

Description

The interface properties are set by this command.

Definition

```
G_Error_t
G_Lin_Node_Config2(
 const G_PortHandle_t  portHandle,
 const G_Lin_Node_Config2_Parameters_t * const cmdParameters
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdParameters`
Pointer to command parameter structure

the command parameter structure contains the following elements:

- `CmdFlags`
  Command flags

  the following values are possible and can be combined:
  - `G_LIN__NODE__CONFIG__CMD_FLAG__NONE`

    no flag set
  - `G_LIN__NODE__CONFIG__CMD_FLAG__ENABLE_MASTER_TASK`

    enable master task
  - `G_LIN__NODE__CONFIG__CMD_FLAG__ENABLE_SLAVE_TASK`

    enable slave task
  - `G_LIN__NODE__CONFIG__CMD_FLAG__ENABLE_BAUD_RATE_DETECTION`

    enable detection of the baud rate

  

  For monitoring, the slave task must be activated. That means, the **G_LIN__NODE__CONFIG__CMD_FLAG__ENABLE_SLAVE_TASK** -flag has to be set.

  

  When the **G_LIN__NODE__CONFIG__CMD_FLAG__ENABLE_BAUD_RATE_DETECTION** -flag is set, the baud rate will be detected automatically and the value under `baudRate` will be disregarded.

`Master`
Structure with parameters for master task

the structure contains the following elements:

- `BaudRate`

  Baud rate in Hertz
- `BreakTime`

  Break time in nanoseconds

(duration of the low level of the **Break**, that signals the start of a new LIN frame, see Figure 3, "LIN Message")

(generally **13 Bit-times**, e.g. 677083 for 19200 Baud)

- `BreakDelimiterTime`

  Break delimiter time in nanoseconds

  (time between the end of the low level of the **Break** and the beginning of the StartBit of the **SyncByte** or duration of the high level of the **BreakDelimiter**, see Figure 3, "LIN Message")

  (generally **1 Bit-time**, e.g. 52083 for 19200 Baud)
- `ArbitrationTime`

  only for **Advanced Library**, otherwise to be initialized with **0**

  arbitration time in nanoseconds

  (time between the end of the StopBit of the **SyncByte** and the beginning of the StartBit of the **IdentifierByte**, see Figure 3, "LIN Message")

  (generally 0)

`Slave`

Structure with parameters for Slave Task

the structure contains the following elements:

- `BaudRate`

  Baud Rate in Hertz
- `ResponseSpace`

  only for **Advanced Library**, otherwise to be initialized with **0**

  response space in nanoseconds

  (time between the end of the StopBit of the **IdentifierByte** and the beginning of the StartBit of the **Response**, or time between **Header** and **Response**, see Figure 3, "LIN Message")

  (generally 0)
- `InterByteSpace`

  only for **Advanced Library**, otherwise to be initialized with **0**

  response space in nanoseconds

  (time between the end of the StopBit of a **Response DataByte** and the beginning of the StartBit of the next **Response DataByte**, see Figure 3, "LIN Message")

  (generally 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
G_Lin_Node_Config2_Parameters_t cmdParameters;
```

```
cmdParameters.Flags =
  G_LIN__NODE__CONFIG__CMD_FLAG__ENABLE_MASTER_TASK;

cmdParameters.Master.BaudRate = 19200;
cmdParameters.Master.BreakTime = 677083;
cmdParameters.Master.BreakDelimiterTime = 52083;
cmdParameters.Master.ArbitrationTime = 0;

cmdParameters.Slave.BaudRate = 0;
cmdParameters.Slave.ResponseSpace = 0;
cmdParameters.Slave.InterByteSpace = 0;

rc =
  G_Lin_Node_Config2(
  portHandle,
  &cmdParameters
);
```

In this example, the master task is enabled with a baud rate of **19200 Hz** .

G_Lin_Node_SetBaudRate

G_Lin_Node_SetBaudRate — Set Baud Rate

Description

With this command, you can set the baud rate parameter explicitly without having to call G_Lin_Node_Config.

Definition

```
G_Error_t
G_Lin_Node_SetBaudRate(
 const G_PortHandle_t  portHandle,
 const u32_t  baudRate
);
```

Parameters

`portHandle`
Handle to the communication port

`baudRate`
BaudRate in **Baud** (generally 19200)

in case of 0 the automatic baud rate detection is enabled

The BaudRate range of values is **700..125000 .** That corresponds to a **minimum BaudRate** of **700 Baud** and a **maximum BaudRate** of **125kBaud .**

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
rc =
 G_Lin_Node_SetBaudRate(
 portHandle,
 19200
);
```

This command call sets the baud rate to 19200 Baud.

G_Lin_Node_SetChecksumModel

G_Lin_Node_SetChecksumModel — Set Checksum Model

Description

Use this command to set the checksum model.

Definition

```
G_Error_t
G_Lin_Node_SetChecksumModel(
 const G_PortHandle_t  portHandle,
 const G_Lin_Node_SetChecksumModel_CmdFlags_t  cmdFlags,
 const G_Lin_ChecksumModel_t  checksumModel,
 const u8_t  numberOfIds,
 const u8_t * const ids
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

the following values are possible and can be combined:

- `G_LIN__NODE__SET_CHECKSUM_MODEL__CMD_FLAG__NONE`

  no command flag set

  (Setting is valid for all identifiers indicated by `ids`)
- `G_LIN__NODE__SET_CHECKSUM_MODEL__CMD_FLAG__SELECT_ALL`

  Setting is valid for all identifiers

`checksumModel`
Checksum model

the following values are possible:

- `G_LIN__CHECKSUM_MODEL__CLASSIC`

  Classic checksum model

  (Checksum via data bytes)
- `G_LIN__CHECKSUM_MODEL__ENHANCED`

  Enhanced checksum model

  (Checksum via identifier and data bytes)

`numberOfIds`
Number of identifiers indicated by `ids`

`ids`
Pointer to identifier list (one byte per identifier)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
rc =
 G_Lin_Node_SetChecksumModel(
  portHandle,
  G_LIN__NODE__SET_CHECKSUM_MODEL__CMD_FLAG__NONE,
  G_LIN__CHECKSUM_MODEL__CLASSIC,
  2,
  {0x08, 0x15}
 );
```

This command call sets the classic checksum model for identifiers **0x08** and **0x15**.

G_Lin_Node_SendWakeUp

G_Lin_Node_SendWakeUp — Send Wake Up Request

Description

Use this command to send one wake up request.

Definition

```
G_Error_t
G_Lin_Node_SendWakeUp(
 const G_PortHandle_t  portHandle,
 const u32_t  wakeUpTime
);
```

Parameters

`portHandle`
Handle to the communication port

`wakeUpTime`
duration of the dominant bus level in nanoseconds

(**0** = default **WakeUpTime**)

In the case of **0** as **WakeUpTime** , the default **WakeUpTime** is eight bit times. According to **LIN 2.0 Specification** , the **WakeUpTime** should have a value of **250 microseconds** to **5 milliseconds** .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Node_SendGoToSleep

G_Lin_Node_SendGoToSleep — Send Go To Sleep Command

Description

This function sends a **Go To Sleep** command.

The condition for sending a **Go To Sleep** command is that the master sends the **0x3C** identifier (diagnostics master request frame).

The first data byte is **0**.

Definition

```
G_Error_t
G_Lin_Node_SendGoToSleep(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Node_Property_Set

G_Lin_Node_Property_Set — Set node properties

Description

This command is used to set properties of the node.

Definition

```
G_Error_t
G_Lin_Node_Property_Set(
 const G_PortHandle_t  portHandle,
 const G_Lin_Node_PropertyId_t  propertyId,
 const u32_t  value
);
```

Parameters

`portHandle`
Handle to the communication port

`propertyId`
Property to be set

The following values are possible:

Relative times are in **%TBit** (percent of one bit time, e.g. **1300** means **13 bit times**).

`G_LIN__NODE__PROPERTY_ID__BREAK_TIME__RELATIVE`
Set relative BreakTime

`G_LIN__NODE__PROPERTY_ID__BREAK_TIME__ABSOLUTE_NS`
Set absolute BreakTime in nanoseconds

`G_LIN__NODE__PROPERTY_ID__BREAK_TIME__IS_RELATIVE`
Set BreakTime reference to be **relative** (`value = 1`) or **absolute** (`value = 0`)

`G_LIN__NODE__PROPERTY_ID__BREAK_DELIMITER_TIME__RELATIVE`
Set relative BreakDelimiterTime

`G_LIN__NODE__PROPERTY_ID__BREAK_DELIMITER_TIME__ABSOLUTE_NS`
Set absolute BreakDelimiterTime in nanoseconds

`G_LIN__NODE__PROPERTY_ID__BREAK_DELIMITER_TIME__IS_RELATIVE`
Set BreakDelimiterTime reference to be **relative** (`value = 1`) or **absolute** (`value = 0`)

`G_LIN__NODE__PROPERTY_ID__ARBITRATION_TIME__RELATIVE`
Set relative ArbitrationTime

`G_LIN__NODE__PROPERTY_ID__ARBITRATION_TIME__ABSOLUTE_NS`
Set absolute ArbitrationTime in nanoseconds

`G_LIN__NODE__PROPERTY_ID__ARBITRATION_TIME__IS_RELATIVE`
Set ArbitrationTime reference to be **relative** (`value = 1`) or **absolute** (`value = 0`)

`G_LIN__NODE__PROPERTY_ID__RESPONSE_SPACE__RELATIVE`
Set relative ResponseSpace

`G_LIN__NODE__PROPERTY_ID__RESPONSE_SPACE__ABSOLUTE_NS`
Set absolute ResponseSpace in nanoseconds

`G_LIN__NODE__PROPERTY_ID__RESPONSE_SPACE__IS_RELATIVE`
Set ResponseSpace reference to be **relative** (`value = 1`) or **absolute** (`value = 0`)

`G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE__RELATIVE`
Set relative InterByteSpace

`G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE__ABSOLUTE_NS`
Set absolute InterByteSpace in nanoseconds

`G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE__IS_RELATIVE`
Set InterByteSpace reference to be **relative** (`value = 1`) or **absolute** (`value = 0`)

`G_LIN__NODE__PROPERTY_ID__DEFAULT_WAKE_UP_TIME__RELATIVE`
Set relative DefaultWakeUpTime

`G_LIN__NODE__PROPERTY_ID__DEFAULT_WAKE_UP_TIME__ABSOLUTE_US`
Set absolute DefaultWakeUpTime in microseconds

`G_LIN__NODE__PROPERTY_ID__DEFAULT_WAKE_UP_TIME__IS_RELATIVE`
Set DefaultWakeUpTime reference to be **relative** (`value = 1`) or **absolute** (`value = 0`)

`G_LIN__NODE__PROPERTY_ID__BAUD_RATE`
Set baud rate in baud

`G_LIN__NODE__PROPERTY_ID__BAUD_RATE_DETECTION__IS_ENABLED`
Enable (`value = 1`) or disable (`value = 0`) baud rate detection

`G_LIN__NODE__PROPERTY_ID__UART__USE_FOR_TRANSMIT`
Use UART for transmit (`value = 1`) or dont (`value = 0`)

`G_LIN__NODE__PROPERTY_ID__MASTER_TASK__IS_ENABLED`
Enable (`value = 1`) or disable (`value = 0`) master task

`G_LIN__NODE__PROPERTY_ID__SLAVE_TASK__IS_ENABLED`
Enable (`value = 1`) or disable (`value = 0`) slave task

`G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_0__RELATIVE`
Set relative InterByteSpace for Byte **0**

`G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_1__RELATIVE`
Set relative InterByteSpace for Byte **1**

`G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_2__RELATIVE`
Set relative InterByteSpace for Byte **2**

`G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_3__RELATIVE`
Set relative InterByteSpace for Byte **3**

`G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_4__RELATIVE`
Set relative InterByteSpace for Byte **4**

`G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_5__RELATIVE`
Set relative InterByteSpace for Byte **5**

`G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_6__RELATIVE`
Set relative InterByteSpace for Byte **6**

`G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_7__RELATIVE`
Set relative InterByteSpace for Byte **7**

G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_0__ABSOLUTE_NS
Set absolute InterByteSpace for Byte **0** in nanoseconds

G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_1__ABSOLUTE_NS
Set absolute InterByteSpace for Byte **1** in nanoseconds

G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_2__ABSOLUTE_NS
Set absolute InterByteSpace for Byte **2** in nanoseconds

G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_3__ABSOLUTE_NS
Set absolute InterByteSpace for Byte **3** in nanoseconds

G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_4__ABSOLUTE_NS
Set absolute InterByteSpace for Byte **4** in nanoseconds

G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_5__ABSOLUTE_NS
Set absolute InterByteSpace for Byte **5** in nanoseconds

G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_6__ABSOLUTE_NS
Set absolute InterByteSpace for Byte **6** in nanoseconds

G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_7__ABSOLUTE_NS
Set absolute InterByteSpace for Byte **7** in nanoseconds

G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_0__IS_RELATIVE
Set InterByteSpace reference of byte **0** to be **relative** (value = **1**) or **absolute** (value = **0**)

G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_1__IS_RELATIVE
Set InterByteSpace reference of byte **1** to be **relative** (value = **1**) or **absolute** (value = **0**)

G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_2__IS_RELATIVE
Set InterByteSpace reference of byte **2** to be **relative** (value = **1**) or **absolute** (value = **0**)

G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_3__IS_RELATIVE
Set InterByteSpace reference of byte **3** to be **relative** (value = **1**) or **absolute** (value = **0**)

G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_4__IS_RELATIVE
Set InterByteSpace reference of byte **4** to be **relative** (value = **1**) or **absolute** (value = **0**)

G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_5__IS_RELATIVE
Set InterByteSpace reference of byte **5** to be **relative** (value = **1**) or **absolute** (value = **0**)

G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_6__IS_RELATIVE
Set InterByteSpace reference of byte **6** to be **relative** (value = **1**) or **absolute** (value = **0**)

G_LIN__NODE__PROPERTY_ID__INTER_BYTE_SPACE_7__IS_RELATIVE
Set InterByteSpace reference of byte **7** to be **relative** (value = **1**) or **absolute** (value = **0**)

G_LIN__NODE__PROPERTY_ID__IGNORE_GO_TO_SLEEP_COMMAND
Ignore GoToSleep-Command (value = **1**) or dont (value = **0**)

G_LIN__NODE__PROPERTY_ID__IGNORE_GO_TO_SLEEP_COMMAND_COUNTER
Number of times the GoToSleep-Command is ignored

`G_LIN__NODE__PROPERTY_ID__ENABLE_BUS_TERMINATION`
Enable (`value = 1`) or disable (`value = 0`) bus termination

Only available on dedicated devices.

`G_LIN__NODE__PROPERTY_ID__ENABLE_INTERNAL_VBAT`
Enable (`value = 1`) or disable (`value = 0`) internal transceiver supply voltage

Only available on dedicated devices.

`G_LIN__NODE__PROPERTY_ID__TRANSCEIVER_OUTPUT_INHIBIT`
Query the state of the transceiver Inhibit Output

Only available on dedicated transceivers.

When setting properties with a higher resolution as the hardware resolution, reading back these properties with G_Lin_Node_Property_Get brings the used property values (e.g. setting a time of **2007 nanoseconds** will result in a hardware setting of **2000 nanoseconds**, because hardware resolution is **25 nanoseconds**).

`value`
Value for selected property to be set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Node_Property_Get

G_Lin_Node_Property_Get — Get node properties

Description

This command is used to query properties of the node.

Definition

```
G_Error_t
G_Lin_Node_Property_Get(
 const G_PortHandle_t  portHandle,
 const G_Lin_Node_PropertyId_t  propertyId,
 u32_t * const value
);
```

Parameters

`portHandle`
: Handle to the communication port

`propertyId`
: See propertyId

`value`
: Returns the value of the selected property

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Node_InternalVBat_Enable

G_Lin_Node_InternalVBat_Enable — Enable internal transceiver supply voltage

Description

Use this command to enable the internal transceiver supply voltage.

The internal transceiver supply voltage is enabled by default. Use G_Lin_Node_InternalVBat_Disable to disable the internal transceiver supply voltage.

Internal transceiver supply voltage is only available on dedicated devices.

The internal transceiver supply voltage setting is stored permanently.

When connecting an external transceiver supply voltage, the internal transceiver supply voltage must be disabled.

Definition

```
G_Error_t
G_Lin_Node_InternalVBat_Enable(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Node_InternalVBat_Disable

G_Lin_Node_InternalVBat_Disable — Disable transceiver supply voltage

Description

Use this command to disable the transceiver supply voltage.

Use G_Can_Node_InternalVBat_Enable to re-enable the transceiver supply voltage.

Internal transceiver supply voltage is only available on dedicated devices.

The transceiver supply voltage setting is stored permanently.

Definition

```
G_Error_t
G_Lin_Node_InternalVBat_Disable(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

4 MasterTask

These functions provide control over the master task of a LIN Interface.

G_Lin_MasterTask_FillScheduleTable

G_Lin_MasterTask_FillScheduleTable — Fill Schedule Table

Description

Use this command to fill a LIN **Schedule Table** .

Altogether there are **16** Schedule tables with **256** entries each.

Definition

```
G_Error_t
G_Lin_MasterTask_FillScheduleTable(
 const G_PortHandle_t  portHandle,
 const G_Lin_MasterTask_FillScheduleTable_CmdFlags_t  cmdFlags,
 const u8_t  scheduleTableNumber,
 const u16_t  numberOfItems,
 const G_Lin_ScheduleTableItem_t * const items
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

- `G_LIN__MASTER_TASK__FILL_SCHEDULE_TABLE__CMD_FLAG__NONE`

  No command flag set

  (The protected identifier is calculated according to the specification by the firmware.)
- `G_LIN__MASTER_TASK__FILL_SCHEDULE_TABLE__CMD_FLAG__ID_AS_ID_CODE`

  The protected identifier is assumed as submitted.

  (Sendig an invalid identifier is possible then!)

`scheduleTableNumber`
Number of the **Schedule Table** (0..15)

`numberOfItems`
Number of table entries

`items`
Pointer to table entries

A **Schedule Table Item** consists of the following elements:

- `Id`

  Identifier
- reserved1
- reserved2
- reserved3
- `Delay`

Delay to the next LIN frame in nanoseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_MasterTask_FillScheduleTable_WithFrameType

G_Lin_MasterTask_FillScheduleTable_WithFrameType — Fill Schedule Table (including enhanced parameters)

Description

Use this command to fill a LIN **Schedule Table** with more control over the schedule items than in command G_Lin_MasterTask_FillScheduleTable.

Altogether there are 16 Schedule tables with 256 entries each.

Definition

```
G_Error_t
G_Lin_MasterTask_FillScheduleTable_WithFrameType(
 const G_PortHandle_t  portHandle,
 const G_Lin_MasterTask_FillScheduleTable_CmdFlags_t  cmdFlags,
 const u8_t  scheduleTableNumber,
 const u16_t  numberOfItems,
 const G_Lin_ScheduleTableItem_WithFrameType_t * const items
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

- `G_LIN__MASTER_TASK__FILL_SCHEDULE_TABLE__CMD_FLAG__NONE`

  No command flag set

  (The protected identifier is calculated according to the specification by the firmware.)
- `G_LIN__MASTER_TASK__FILL_SCHEDULE_TABLE__CMD_FLAG__ID_AS_ID_CODE`

  The protected identifier is assumed as submitted.

  (Sendig an invalid identifier is possible then!)

`scheduleTableNumber`
Number of the **Schedule Table (0..15)**

`numberOfItems`
Number of table entries

`items`
Pointer to table entries

- `Id`

  Identifier
- `FrameType`

  The following values for **FrameType** are possible:
  - `G_LIN__FRAME_TYPE__UNCONDITIONAL`

"Normal" frame type

- `G_LIN__FRAME_TYPE__EVENT_TRIGGERED`

  Event triggered frame type

- `G_LIN__FRAME_TYPE__SPORADIC`

  Sporadic frame type

- `G_LIN__FRAME_TYPE__DIAGNOSTIC`

  Diagnostic frame type (`0x3C` and `0x3D`)

- `FrameListIndex`

  List index for **Event Triggered Frames** and **Sporadic Frames** (starting with **0**)

  

  Similar to the frame list for **Unconditional Frames** of a LIN Description File (**LDF** file), there is possibly an **Event Triggered Frames List** for Event Triggered Frames or a **Sporadic Frames List** for Sporadic Frames.

  `FrameListIndex` is the index in these lists.

- `reserved`

  reserved byte (has to be **0**)

- `Delay`

  Delay to the next LIN frame in nanoseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_MasterTask_ClearScheduleTable

G_Lin_MasterTask_ClearScheduleTable — Clear Schedule Table

Description

Use this command to empty the **Schedule Table** indicated by `scheduleTableNumber`.

Definition

```
G_Error_t
G_Lin_MasterTask_ClearScheduleTable(
 const G_PortHandle_t  portHandle,
 const u8_t  scheduleTableNumber
);
```

Parameters

`portHandle`
Handle to the communication port

`scheduleTableNumber`
Number of the **Schedule Table**

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_MasterTask_ChangeScheduleTable

G_Lin_MasterTask_ChangeScheduleTable — Change to another schedule table

Description

Use this command to change to another schedule table.

Changing to another Schedule table is always carried out after proceeding the current table.

By setting the `G_LIN__MASTER_TASK__C_S_T__CMD_FLAG__CHANGE_IMMEDIATELY` flag, an immediate change can be enforced.

Definition

```
G_Error_t
G_Lin_MasterTask_ChangeScheduleTable(
 const G_PortHandle_t  portHandle,
 const u8_t  scheduleTableNumber,
 const G_Lin_MasterTask_CST_CmdFlags_t  cmdFlags,
 const u32_t  numberOfCycles
);
```

Parameters

`portHandle`
Handle to the communication port

`scheduleTableNumber`
Number of the schedule table to change to

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_LIN__MASTER_TASK__C_S_T__CMD_FLAG__ZERO`
No flag is set

`G_LIN__MASTER_TASK__C_S_T__CMD_FLAG__START_SCHEDULER_BEFORE_CHANGE`
The scheduler is started before changing to another schedule table

`G_LIN__MASTER_TASK__C_S_T__CMD_FLAG__STOP_SCHEDULER_AFTER_CHANGE`
The scheduler is stopped after `numberOfCycles` runs of the schedule table

`G_LIN__MASTER_TASK__C_S_T__CMD_FLAG__CHANGE_BACK_TO_PREVIOUS_TABLE`
The scheduler changes back to the previous schedule table after `numberOfCycles` runs of the schedule table

`G_LIN__MASTER_TASK__C_S_T__CMD_FLAG__CHANGE_IMMEDIATELY`
The scheduler changes immediately to the specified schedule table

`G_LIN__MASTER_TASK__C_S_T__CMD_FLAG__START_SCHEDULER_ON_WAKE_UP`
The scheduler is started automatically after a **wake up**

`numberOfCycles`
Number of cycles

Necessary if flag `G_LIN__MASTER_TASK__C_S_T__CMD_FLAG__STOP_SCHEDULER_AFTER_CHANGE` or `G_LIN__MASTER_TASK__C_S_T__CMD_FLAG__CHANGE_BACK_TO_PREVIOUS_TABLE` is set.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_MasterTask_GetSchedulerState

G_Lin_MasterTask_GetSchedulerState — Get state of scheduler

Description

Use this command to query the state of the scheduler.

Definition

```
G_Error_t
G_Lin_MasterTask_GetSchedulerState(
 const G_PortHandle_t  portHandle,
 G_Lin_MasterTask_GetSchedulerState_RspFlags_t * const rspFlags,
 u16_t * const lastScheduleTableNumber,
 u16_t * const lastScheduleTableEntry,
 u16_t * const nextScheduleTableNumber,
 u16_t * const nextScheduleTableEntry
);
```

Parameters

`portHandle`
Handle to the communication port

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_LIN__MASTER_TASK__GET_SCHEDULER_STATE__RSP_FLAG__ZERO`
No flag is set

`G_LIN__MASTER_TASK__GET_SCHEDULER_STATE__RSP_FLAG__INITIALIZED`
The scheduler is initialized

`G_LIN__MASTER_TASK__GET_SCHEDULER_STATE__RSP_FLAG__STARTED`
The scheduler is started

`lastScheduleTableNumber`
Returns the number of the **Schedule table** proceeded last

`lastScheduleTableEntry`
Returns the number of the **Schedule table entry** proceeded last

`nextScheduleTableNumber`
Returns the number of the **Schedule table** proceeded next

`nextScheduleTableEntry`
Returns the number of the **Schedule table entry** proceeded next

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_MasterTask_Start

G_Lin_MasterTask_Start — Start Transmitting

Description

The master task starts processing the indicated LIN **Schedule Table** .

That means the corresponding LIN frame headers are sent.

Definition

```
G_Error_t
G_Lin_MasterTask_Start(
 const G_PortHandle_t  portHandle,
 const u8_t  scheduleTableNumber
);
```

Parameters

`portHandle`
Handle to the communication port

`scheduleTableNumber`
Number of the **Schedule Table**

The indicated **Schedule Table** is always processed starting with the FIRST entry.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_MasterTask_Stop

G_Lin_MasterTask_Stop — Stop Transmitting

Description

The master task stops to send LIN frame headers.

Definition

```
G_Error_t
G_Lin_MasterTask_Stop(
const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_MasterTask_SetWakeUpDelimiterTime

G_Lin_MasterTask_SetWakeUpDelimiterTime — Set WakeUpDelimiterTime

Description

The **WakeUpDelimiterTime** determines the point in time when the Master Task (if activated) restarts processing the **Schedule Table** (i.e. sending) after the end of the dominant level of a **WakeUp**.

According to the **LIN 2.0 Specification**, all Slaves should be ready to receive LIN messages **100 milliseconds** after a **WakeUp**. That means **100 milliseconds** after the end of a **WakeUp** the Master task should start communication again.

Definition

```
G_Error_t
G_Lin_MasterTask_SetWakeUpDelimiterTime(
 const G_PortHandle_t  portHandle,
 const u32_t  wakeUpDelimiterTime
);
```

Parameters

`portHandle`
Handle to the communication port

`wakeUpDelimiterTime`
WakeUpDelimiterTime

In case of `0`, the default **WakeUpDelimiterTime** of four bit times is used.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_MasterTask_SendHeader

G_Lin_MasterTask_SendHeader — Send LIN frame header

Description

This command sends a LIN frame header without any relation to a schedule table.

Per default, the **Protected Identifier (PID)** of the frame consists of 6 bits `id` + 2 bits parity.

If command flag `G_LIN__MASTER_TASK__SEND_HEADER__CMD_FLAG__ID_AS_PID` is set, the **PID** equals the value of `id`.

Definition

```
G_Error_t
G_Lin_MasterTask_SendHeader(
 const G_PortHandle_t  portHandle,
 const G_Lin_MasterTask_SendHeader_CmdFlags_t  cmdFlags,
 const u8_t  id
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_LIN__MASTER_TASK__SEND_HEADER__CMD_FLAG__NONE`
No flag is set

`G_LIN__MASTER_TASK__SEND_HEADER__CMD_FLAG__ID_AS_PID`
The PID equals the value of `id`

`id`
Frame header ID

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

5 SlaveTask

These functions provide control over the slave task of a LIN Interface.

G_Lin_SlaveTask_SetSlaveCommunicationState

G_Lin_SlaveTask_SetSlaveCommunicationState — Set SlaveCommunicationState

Description

This command sets the slave controller to the sleep or the awake state.

Definition

```
G_Error_t
G_Lin_SlaveTask_SetSlaveCommunicationState(
 const G_PortHandle_t  portHandle,
 const G_Lin_SlaveCommunicationState_t  state
);
```

Parameters

`portHandle`
Handle to the communication port

`state`
Communication state

The following values are possible:

- `G_LIN__SLAVE_COMMUNICATION_STATE__SLEEP`

  Sleep state
- `G_LIN__SLAVE_COMMUNICATION_STATE__AWAKE`

  Awake state

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_SlaveTask_SetBreakDetectionThreshold

G_Lin_SlaveTask_SetBreakDetectionThreshold — Set BreakDetectionThreshold

Description

This command sets the **BreakDetectionThreshold** parameter.

Definition

```
G_Error_t
G_Lin_SlaveTask_SetBreakDetectionThreshold(
 const G_PortHandle_t  portHandle,
 const u16_t  breakDetectionThreshold
);
```

Parameters

`portHandle`
: Handle to the communication port

`breakDetectionThreshold`
: **BreakDetectionThreshold** in percent of the bit-time (generally `950`)

  BrekDetectionThreshold for **GOEPEL electronic** products:

  `9.5` bit-times (in contrast to the LIN-Specification with 11 bit-times) for bus monitoring, to be able to detect breaks shorter than 11 bit-times.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

6 FrameResponses

These functions provide control over the frame responses of a LIN Interface.

G_Lin_FrameResponses_DefineFrameResponse

G_Lin_FrameResponses_DefineFrameResponse — Frame Response Definition

Description

Use this command to define the LIN **Frame response** indicated by **ID**.

Definition

```
G_Error_t
G_Lin_FrameResponses_DefineFrameResponse(
 const G_PortHandle_t  portHandle,
 const u8_t  id,
 const G_Lin_FrameResponses_SendState_t  sendState,
 const u8_t  group,
 const u8_t  messageCount,
 const u8_t  length,
 const u8_t  data[8]
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Identifier (0x00..0x3F)

`sendState`
Send state of LIN **Frame response**

The following values are possible:

- `G_LIN__FRAME_RESPONSES__SEND_STATE__DONT_SEND_MSG`

  LIN **Frame response** will not be sent
- `G_LIN__FRAME_RESPONSES__SEND_STATE__SEND_MSG`

  LIN **Frame response** will be sent

`group`
Group number of the **Frame response**

In the current version of the **G-API** only values `0` and `1` are supported.

All **Frame responses** belonging to the same group can be started and stopped alltogether by using the commands G_Lin_FrameResponses_StartGroup and G_Lin_FrameResponses_StopGroup .

Frame responses defined with a group number of `0` do not belong to a group.

`messageCount`
`0`: Always send LIN **Frame response**

`1 <= N <= 255`: Send LIN **Frame Response** `N` times

`length`
Data length (0..8)

`data`
Data bytes (0..7)

The sending of LIN frame headers is NOT affected by this command.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_FrameResponses_DeleteFrameResponse

G_Lin_FrameResponses_DeleteFrameResponse — Delete a Frame response

Description

After calling this command, the output of the LIN Frame response indicated by **ID** is stopped and the LIN Frame response is deleted from the internal administration.

Another LIN Frame response output is only possible after executing G_Lin_FrameResponses_DefineFrameResponse.

Definition

```
G_Error_t
G_Lin_FrameResponses_DeleteFrameResponse(
 const G_PortHandle_t  portHandle,
 const u8_t  id
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Identifier (0x00..0x3F)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_FrameResponses_DeleteAllFrameResponses

G_Lin_FrameResponses_DeleteAllFrameResponses — Delete all Frame responses

Description

After calling this command, all Frame responses are deleted.

Definition

```
G_Error_t
G_Lin_FrameResponses_DeleteAllFrameResponses(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_FrameResponses_StartGroup

G_Lin_FrameResponses_StartGroup — Start a group of Frame responses

Description

After calling this command, all Frame responses of the indicated group will be started.

Frame responses defined with a group number of `0` can not be started with this command because they are not belonging to a group.

Definition

```
G_Error_t
G_Lin_FrameResponses_StartGroup(
 const G_PortHandle_t  portHandle,
 const u8_t  group
);
```

Parameters

`portHandle`
Handle to the communication port

`group`
Group number of the **Frame Response Group**

In the current version of the **G-API** only values `0` and `1` are supported.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_FrameResponses_StopGroup

G_Lin_FrameResponses_StopGroup — Stop a group of Frame responses

Description

After calling this command, all Frame responses of the indicated group will be stopped.

Frame responses defined with a group number of `0` can not be stopped with this command because they are not belonging to a group.

Definition

```
G_Error_t
G_Lin_FrameResponses_StopGroup(
 const G_PortHandle_t  portHandle,
 const u8_t  group
);
```

Parameters

`portHandle`
Handle to the communication port

`group`
Group number of the **Frame Response Group**

In the current version of the **G-API** only values `0` and `1` are supported.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_FrameResponses_GetState

G_Lin_FrameResponses_GetState — Get frame response state

Description

Query the state of the frame response indicated by `id`.

Definition

```
G_Error_t
G_Lin_FrameResponses_GetState(
 const G_PortHandle_t  portHandle,
 const u8_t  id,
 G_Lin_FrameResponses_GetState_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Identifier (0x00..0x3F)

`rsp`
Returns frame response state

The structure contains the following elements:

`Id`
Identifier

`SendState`
Send state

The following values are possible:

`G_LIN__FRAME_RESPONSES__SEND_STATE__DONT_SEND`
The frame response will not be sent

`G_LIN__FRAME_RESPONSES__SEND_STATE__SEND`
The frame response will be sent

`Group`
Group number of the frame response

`MessageCount`
- `0`: Always send frame response
- `1 <= N <= 255`: Send frame response `N` times

`Flags`
Response Flags

The following values are possible and can be combined:

`G_LIN__FRAME_RESPONSES__GET_STATE__RSP_FLAG__ZERO`
No flag is set

`G_LIN__FRAME_RESPONSES__GET_STATE__RSP_FLAG__CHANGE_STATE_BEFORE_SEND`
The `SendState` of the frame response will be set to `G_LIN__FRAME_RESPONSES__SEND_STATE__SEND` before sending

`G_LIN__FRAME_RESPONSES__GET_STATE__RSP_FLAG__CHANGE_STATE_AFTER_SEND`
The `SendState` of the frame response will be set to `G_LIN__FRAME_RESPONSES__SEND_STATE__DONT_SEND` after it has been sent

`G_LIN__FRAME_RESPONSES__GET_STATE__RSP_FLAG__PROCESSED_ONLY`
The frame response will be processed but not sent

`G_LIN__FRAME_RESPONSES__GET_STATE__RSP_FLAG__DEFINED`
The frame response is defined

`Length`
Data length in bytes

`MessageCounter`
Defines the number of times the frame response will be sent

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Data`
Data bytes (0..7)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_FrameResponses_ChangeProperties

G_Lin_FrameResponses_ChangeProperties — Change frame response properties

Description

Change properties of the frame response indicated by `id`.

Definition

```
G_Error_t
G_Lin_FrameResponses_ChangeProperties(
 const G_PortHandle_t  portHandle,
 const u8_t  id,
 const G_Lin_FrameResponses_SendState_t  sendState,
 const u8_t  group,
 const u8_t  messageCount
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Identifier (0x00..0x3F)

`SendState`
Send state

The following values are possible:

`G_LIN__FRAME_RESPONSES__SEND_STATE__DONT_SEND`
The frame response will not be sent

`G_LIN__FRAME_RESPONSES__SEND_STATE__SEND`
The frame response will be sent

`group`
Group number of the frame response

In the current version of the **G-API** only values `0` and `1` are supported.

All **Frame responses** belonging to the same group can be started and stopped alltogether by using the commands G_Lin_FrameResponses_StartGroup and G_Lin_FrameResponses_StopGroup .

Frame responses defined with a group number of `0` do not belong to a group.

`messageCount`
- `0`: Always send frame response
- `1 <= N <= 255`: Send frame response `N` times

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_FrameResponses_ChangeData

G_Lin_FrameResponses_ChangeData — Change frame response data and message count

Description

Change data part and message count of the frame response indicated by `id`.

Definition

```
G_Error_t
G_Lin_FrameResponses_ChangeData(
 const G_PortHandle_t  portHandle,
 const u8_t  id,
 const u8_t  messageCount,
 const u8_t  length,
 const u8_t * const mask,
 const u8_t * const data
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier (0x00..0x3F)

`messageCount`
- `0`: Always send frame response
- `1 <= N <= 255`: Send frame response `N` times

`length`
: Data length (0..8)

`mask`
: Mask bytes (0..7)

`data`
: Data bytes (0..7)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_FrameResponses_Suspend

G_Lin_FrameResponses_Suspend — Suspend frame responses

Description

Use this command to simulate failures of LIN frame responses.

The LIN frame response indicated by `id` is not sent for `count` times.

Definition

```
G_Error_t
G_Lin_FrameResponses_Suspend(
 const G_PortHandle_t  portHandle,
 const u8_t  id,
 const u16_t  count
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier (0x00..0x3F)

`count`
: Number of suspensions of the frame response

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_FrameResponses_Disturb

G_Lin_FrameResponses_Disturb — Disturb frame responses

Description

Use this command to simulate temporary disturbances of frame responses.

The LIN frame responses indicated by `id` are sent temporarily for `count` times with the data length defined by `length` and the data defined by `data` according to the corresponding bit mask defined by `mask`. Then the initial data is sent again with the initial data length.

Definition

```
G_Error_t
G_Lin_FrameResponses_Disturb(
 const G_PortHandle_t  portHandle,
 const u8_t  id,
 const u16_t  count,
 const u8_t  length,
 const u8_t * const mask,
 const u8_t * const data
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier (0x00..0x3F)

`count`
: Number of disturbed frame responses

`length`
: Data length of the disturbed frame responses

`data`
: Data of the disturbed frame responses

`mask`
: Data Mask of the disturbed frame responses

  The data is assumed in accordance with the set mask byte bit positions. In the case no mask byte bits are set, the original data is assumed.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_FrameResponses_Ramp_Define

G_Lin_FrameResponses_Ramp_Define — Define frame response data ramp

Description

Use this command to define and start a new ramp or to modify a running one within the LIN frame response indicated by `id` (a ramp serves to create alternately increasing/ decreasing signal courses).

Definition

```
G_Error_t
G_Lin_FrameResponses_Ramp_Define(
 const G_PortHandle_t  portHandle,
 const G_Lin_FrameResponses_Ramp_Define_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
The structure contains the following elements:

`Id`
Identifier (0x00..0x3F)

`Mode`
Mode of the ramp

The following values are possible:

`G_LIN__FRAME_RESPONSES__RAMP__MODE__TRIANGULAR`
Triangular ramp

`G_LIN__FRAME_RESPONSES__RAMP__MODE__SAWTOOTH`
Sawtooth ramp

`G_LIN__FRAME_RESPONSES__RAMP__MODE__SAWTOOTH_WITH_OVERFLOW_BIT`
Sawtooth ramp with overflow bit

`Direction`
Direction of the ramp

The following values are possible:

`G_LIN__FRAME_RESPONSES__RAMP__DIRECTION__DOWN`
The starting direction of the ramp is **downwards**

`G_LIN__FRAME_RESPONSES__RAMP__DIRECTION__UP`
The starting direction of the ramp is **upwards**

`reserved`
reserved parameter (must be initialized with **0**)

`Signal`
Signal definition of the ramp (see G_Common_Signal_t)

`MinValue`
Minimum value of the ramp

`MaxValue`
Maximum value of the ramp

`UpdateRate`
Update rate of the ramp
- **1** : The ramp is updated with each frame response
- **2** : The ramp is updated with each second frame response
- etc.

`NumberOfFlanks`
Number of flanks to be used

(0 = No limitation)

`Increment`
Increment of the ramp as a floating point value

`RampSum`
Relative ramp sum (sum of the partial increments starting with the execution of this command, therefore relative)

(0 = No limitation)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_FrameResponses_Ramp_ResetRampSum

G_Lin_FrameResponses_Ramp_ResetRampSum — Reset ramp sum

Description

Use this command to reset the ramp sum of a ramp defined by `signal` within the frame response defined by `id`.

Definition

```
G_Error_t
G_Lin_FrameResponses_Ramp_ResetRampSum(
 const G_PortHandle_t  portHandle,
 const u8_t  id,
 const G_Common_Signal_t  signal
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier (0x00..0x3F)

`signal`
: Signal definition of the ramp (see G_Common_Signal_t)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_FrameResponses_Ramp_GetState

G_Lin_FrameResponses_Ramp_GetState — Query ramp state

Description

Query the state of the ramp defined by `signal` within the LIN frame response defined by `id` with this command.

Definition

```
G_Error_t
G_Lin_FrameResponses_Ramp_GetState(
 const G_PortHandle_t  portHandle,
 const u8_t  id,
 const G_Common_Signal_t  signal,
 G_Lin_FrameResponses_Ramp_State_Flags_t * const rampState,
 u32_t * const rampSum
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Identifier (0x00..0x3F)

`signal`
Signal definition of the ramp (see G_Common_Signal_t)

`rampState`
Returns the state of the ramp

The following values are possible and can be combined:

`G_LIN__FRAME_RESPONSES__RAMP__STATE__FLAG__NONE`
No flag is set

`G_LIN__FRAME_RESPONSES__RAMP__STATE__FLAG__RUNNING`
The ramp is running

`G_LIN__FRAME_RESPONSES__RAMP__STATE__FLAG__DIRECTION_IS_UP`
The ramp is currently running upwards

If the flag is not set, the ramp is running downwards.

`rampSum`
Current internal ramp sum

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_FrameResponses_Ramp_Delete

G_Lin_FrameResponses_Ramp_Delete — Delete a ramp

Description

Delete the ramp defined by `signal` within the LIN frame response defined by `id` with this command.

Definition

```
G_Error_t
G_Lin_FrameResponses_Ramp_Delete(
 const G_PortHandle_t  portHandle,
 const u8_t  id,
 const G_Common_Signal_t  signal
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier (0x00..0x3F)

`signal`
: Signal definition of the ramp (see G_Common_Signal_t)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_FrameResponses_MultiplexBytes_Define

G_Lin_FrameResponses_MultiplexBytes_Define — Define multiplex bytes

Description

This command defines multiplex bytes for a LIN frame response.

Multiplexing the data bytes of a LIN frame response serves to transfer more information as usually possible with only eight data bytes. Generally the multiplex code located fixed in a certain data byte explaines the meaning of the remaining data bytes. This way, time multiplexing is carried out.

The multiplex code itself is defined together with the actual multiplex data within parameter `Data`.

Definition

```
G_Error_t
G_Lin_FrameResponses_MultiplexBytes_Define(
 const G_PortHandle_t portHandle,
 const u8_t  id,
 const u8_t  updateRate,
 const u8_t  numberOfIndices,
 const G_Data_t + const mask,
 const G_Data_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Identifier (0x00..0x3F)

`updateRate`
Update rate of the Multiplexbytes

1. With each frame response

2. With every second frame response

3. Witch every third frame response
etc.

`numberOfIndices`
Number of multiplex indices (1..16)

`mask`
Array with mask bytes (0..7)

(Data is assumed in accordance with the set mask byte bit positions)

Mask bytes have no indices, therefore the same mask is applied to every data byte index.

`data`
Array with data bytes

The elements of the array represent the indices (indexes) of multiplex bytes.
- byte **0..7** = index **1**

- byte **8..15** = index **2**
- etc.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
G_Data_t  mask;
G_Data_t  data;
G_Error_t  rc;

mask.U64.High = 0xFFFFFF;
mask.U64.Low = 0;

data.U8[0] = 0x42;
data.U8[1] = 0x11;
data.U8[2] = 0x22;
data.U8[3] = 0x33;
data.U8[4] = 0x44;
data.U8[5] = 0x55;
data.U8[6] = 0x66;
data.U8[7] = 0x77;

rc =
 G_Lin_FrameResponses_MultiplexBytes_Define(
  portHandle,
  0x01,
  1,
  1,
  &mask,
  data
 );
```

Defining multiplex bytes with a byte mask for a LIN frame response with identifier **0x01**

G_Lin_FrameResponses_MultiplexBytes_Stop

G_Lin_FrameResponses_MultiplexBytes_Stop — Stop multiplexing bytes

Description

The multiplexed bytes of the LIN frame response defined by `id` are stopped by this command. After stopping the multiplexed bytes, data is assumed in accordance with the defined byte mask.

Definition

```
G_Error_t
G_Lin_FrameResponses_MultiplexBytes_Stop(
 const G_PortHandle_t portHandle,
 const u8_t  id,
 const G_Can_Data_t * const mask,
 const G_Can_Data_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Identifier (**0x00..0x3F**)

`mask`
Mask bytes (0..7)

(Data is assumed in accordance with the set mask byte bit positions)

`data`
Data bytes (0..7)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

7 Monitor

These functions provide control over the bus monitoring of a LIN Interface.

There are two different monitor modes:

- Buffer Mode

  The LIN messages are stored in an internal ring buffer after passing the monitor filter as sent or received on the bus. The size of this internal ring buffer is defined at the point the monitor is started.
- List Mode

  A list entry exists for each identifier. This list entry is updated by receiving or transmitting the identifier. It can be queried explicitly at any time.

For example on using the **LIN Monitor** , a few samples are included in the **samples directory** of the **G-API** installation.

For monitoring, the **Slave Task** has to be activated.
See G_Lin_Node_Config for details!

Monitor Items

Monitor Items — An explanation of the different LIN monitor items.

Small Buffer Monitor Item

A small LIN buffer monitor item has the type **G_Lin_Monitor_BufferMode_Small_Item_t** and consists of the following elements:

`Flags`
Item flags

The following values are possible and can be combined:

- `G_LIN__MONITOR__ITEM_FLAG__NONE`

  No command flag set
- `G_LIN__MONITOR__ITEM_FLAG__IDENTIFIER_PARITY_ERROR`

  Identifier parity error
- `G_LIN__MONITOR__ITEM_FLAG__CHECKSUM_ERROR`

  Checksum error
- `G_LIN__MONITOR__ITEM_FLAG__INCONSISTENT_SYNC_BYTE`

  Inconsistent SyncByte
- `G_LIN__MONITOR__ITEM_FLAG__BIT_ERROR`

  Bit error
- `G_LIN__MONITOR__ITEM_FLAG__EVENT`

  Event (see `IdCode`)
- `G_LIN__MONITOR__ITEM_FLAG__WAKE_UP`

  WakeUp
- `G_LIN__MONITOR__ITEM_FLAG__SENT_LIN_FRAME_RSP`

  Sent LIN Frame response (TX)
- `G_LIN__MONITOR__ITEM_FLAG__BUFFER_OVERRUN`

  Buffer overrun in firmware layer
- `G_LIN__MONITOR__ITEM_FLAG__API_BUFFER_OVERRUN`

  Buffer overrun in api layer

`Length`
Data length including checksum (0..9)

`IdCode`
Generally the Identifier byte (consisting of identifier + parity bits), BUT if flag `G_LIN__MONITOR__ITEM_FLAG__EVENT` is set, **`IdCode`** specifies one of the following events:

- **`IdCode`** = `0`:

  rising edge at the trigger input
- **`IdCode`** = `1`:

  falling edge at the trigger input

`reserved`
reserved parameter (must be initialized with **`0`**)

`Data`
Data bytes and checksum (0..9)

`StartTime`
Start time stamp in nanoseconds

`BitTimeX8`
Bit time measured over eight bit times in nanoseconds

Normal Buffer Monitor Item

A normal LIN buffer monitor item has the type **G_Lin_Monitor_BufferMode_Normal_Item_t** and consists of the following elements:

`Flags`
Item flags

The following values are possible and can be combined:

- `G_LIN__MONITOR__ITEM_FLAG__NONE`

  No command flag set
- `G_LIN__MONITOR__ITEM_FLAG__IDENTIFIER_PARITY_ERROR`

  Identifier parity error
- `G_LIN__MONITOR__ITEM_FLAG__CHECKSUM_ERROR`

  Checksum error
- `G_LIN__MONITOR__ITEM_FLAG__INCONSISTENT_SYNC_BYTE`

  Inconsistent SyncByte
- `G_LIN__MONITOR__ITEM_FLAG__BIT_ERROR`

  Bit error
- `G_LIN__MONITOR__ITEM_FLAG__EVENT`

  Event (see `IdCode`)
- `G_LIN__MONITOR__ITEM_FLAG__WAKE_UP`

  WakeUp
- `G_LIN__MONITOR__ITEM_FLAG__SENT_LIN_FRAME_RSP`

  Sent LIN Frame response (TX)
- `G_LIN__MONITOR__ITEM_FLAG__BUFFER_OVERRUN`

  Buffer overrun in firmware layer
- `G_LIN__MONITOR__ITEM_FLAG__API_BUFFER_OVERRUN`

  Buffer overrun in api layer

`Length`
Data length including checksum (0..9)

`IdCode`
Generally the Identifier byte (consisting of identifier + parity bits), BUT if flag `G_LIN__MONITOR__ITEM_FLAG__EVENT` is set, **IdCode** specifies one of the following events:

- **IdCode** = `0`:

  rising edge at the trigger input
- **IdCode** = `1`:

  falling edge at the trigger input

`reserved`
reserved parameter (must be initialized with **0**)

`Data`
Data bytes and checksum (0..9)

`StartTime`
Start time stamp in nanoseconds

`BitTimeX8`
Bit time measured over eight bit times in nanoseconds

`TimeStamps`
Structure with additional timestamps

The structure consists of the following elements:

- `BreakDelimiter`

  Time stamp of the rising edge of the **BreakDelimiter**
- `SyncByte`

  Time stamp of the start bit of the **SyncByte** in nanoseconds
- `Identifier`

  Time stamp of the start bit of the **Identifier** in nanoseconds
- `Data` (`0`..`9`)

  Time stamps of the start bits of the data bytes (0..9) in nanoseconds

Monitoring with **Normal Monitor Items** on **smartCAR** - Hardware is only possible with the enhanced instruction set.

Advanced Monitor Buffer Item

An advanced LIN buffer monitor item has the type **G_Lin_Monitor_BufferMode_Advanced_Item_t** and consists of the following elements:

`Flags`
Item flags

The following values are possible and can be combined:

- `G_LIN__MONITOR__ITEM_FLAG__NONE`

  No command flag set
- `G_LIN__MONITOR__ITEM_FLAG__IDENTIFIER_PARITY_ERROR`

  Identifier parity error
- `G_LIN__MONITOR__ITEM_FLAG__CHECKSUM_ERROR`

  Checksum error
- `G_LIN__MONITOR__ITEM_FLAG__INCONSISTENT_SYNC_BYTE`

  Inconsistent SyncByte
- `G_LIN__MONITOR__ITEM_FLAG__BIT_ERROR`

  Bit error
- `G_LIN__MONITOR__ITEM_FLAG__EVENT`

  Event (see `IdCode`)
- `G_LIN__MONITOR__ITEM_FLAG__WAKE_UP`

  WakeUp
- `G_LIN__MONITOR__ITEM_FLAG__SENT_LIN_FRAME_RSP`

Sent LIN Frame response (TX)

- `G_LIN__MONITOR__ITEM_FLAG__BUFFER_OVERRUN`

  Buffer overrun in firmware layer
- `G_LIN__MONITOR__ITEM_FLAG__API_BUFFER_OVERRUN`

  Buffer overrun in api layer

`Length`
Data length including checksum (0..9)

`IdCode`
Generally the Identifier byte (consisting of identifier + parity bits), BUT if flag `G_LIN__MONITOR_ _ITEM_FLAG__EVENT` is set, **IdCode** specifies one of the following events:

- **IdCode** = `0`:

  rising edge at the trigger input
- **IdCode** = `1`:

  falling edge at the trigger input

`reserved`
reserved parameter (must be initialized with **0**)

`Data`
Data bytes and checksum (0..9)

`StartTime`
Start time stamp in nanoseconds

`BitTimeX8`
Bit time measured over eight bit times in nanoseconds

`TimeStamps`
Structure with additional timestamps

The structure consists of the following elements:

- `BreakDelimiter`

  Time stamp of the rising edge of the **BreakDelimiter** in nanoseconds
- `SyncByte`

  Time stamp of the start bit of the **SyncByte** in nanoseconds
- `Identifier`

  Time stamp of the start bit of the **Identifier** in nanoseconds
- `Data`

  Structure with additional data time stamps

  The structure consists of the following elements:
  - `StartBit` (`0..9`)

    Time stamps of the start bits of the data bytes (**0..9**) in nanoseconds
  - `LastFallingEdge` (0..9)

    Time stamps of the last falling edges of the data bytes (0..9) in nanoseconds
  - `LastRisingEdge` (0..9)

    Time stamps of the last rising edges of the data bytes (0..9) in nanoseconds

Monitoring with **Advanced Monitor Items** is only possible with the **Advanced Library .**

Small List Monitor Item

A small LIN list monitor item has the type **G_Lin_Monitor_ListMode_Small_Item_t** and consists of the following elements:

`FrameCounter`
Frame counter

`Flags`
Item flags

The following values are possible and can be combined:

- `G_LIN__MONITOR__ITEM_FLAG__NONE`

  No command flag set
- `G_LIN__MONITOR__ITEM_FLAG__IDENTIFIER_PARITY_ERROR`

  Identifier parity error
- `G_LIN__MONITOR__ITEM_FLAG__CHECKSUM_ERROR`

  Checksum error
- `G_LIN__MONITOR__ITEM_FLAG__INCONSISTENT_SYNC_BYTE`

  Inconsistent SyncByte
- `G_LIN__MONITOR__ITEM_FLAG__BIT_ERROR`

  Bit error
- `G_LIN__MONITOR__ITEM_FLAG__EVENT`

  Event (see `IdCode`)
- `G_LIN__MONITOR__ITEM_FLAG__WAKE_UP`

  WakeUp
- `G_LIN__MONITOR__ITEM_FLAG__SENT_LIN_FRAME_RSP`

  Sent LIN Frame response (TX)
- `G_LIN__MONITOR__ITEM_FLAG__BUFFER_OVERRUN`

  Buffer overrun in firmware layer
- `G_LIN__MONITOR__ITEM_FLAG__API_BUFFER_OVERRUN`

  Buffer overrun in api layer

`Id`
Identifier (0x00..0x3F)

`IdCode`
Generally the Identifier byte (consisting of identifier + parity bits), BUT if flag `G_LIN__MONITOR__ITEM_FLAG__EVENT` is set, **IdCode** specifies one of the following events:

- **IdCode** = `0`:

  rising edge at the trigger input
- **IdCode** = `1`:

  falling edge at the trigger input

`Length`
Data length including checksum (`0`..`9`)

`Data`
Data bytes and checksum (0..9)

`StartTime`
Start time stamp in nanoseconds

`BitTimeX8`
Bit time measured over eight bit times in nanoseconds

Normal List Monitor Item

A normal LIN list monitor item has the type **G_Lin_Monitor_ListMode_Normal_Item_t** and consists of the following elements:

`FrameCounter`
Frame counter

`Flags`
Item flags

The following values are possible and can be combined:

- `G_LIN__MONITOR__ITEM_FLAG__NONE`

  No command flag set
- `G_LIN__MONITOR__ITEM_FLAG__IDENTIFIER_PARITY_ERROR`

  Identifier parity error
- `G_LIN__MONITOR__ITEM_FLAG__CHECKSUM_ERROR`

  Checksum error
- `G_LIN__MONITOR__ITEM_FLAG__INCONSISTENT_SYNC_BYTE`

  Inconsistent SyncByte
- `G_LIN__MONITOR__ITEM_FLAG__BIT_ERROR`

  Bit error
- `G_LIN__MONITOR__ITEM_FLAG__EVENT`

  Event (see `IdCode`)
- `G_LIN__MONITOR__ITEM_FLAG__WAKE_UP`

  WakeUp
- `G_LIN__MONITOR__ITEM_FLAG__SENT_LIN_FRAME_RSP`

  Sent LIN Frame response (TX)
- `G_LIN__MONITOR__ITEM_FLAG__BUFFER_OVERRUN`

  Buffer overrun in firmware layer
- `G_LIN__MONITOR__ITEM_FLAG__API_BUFFER_OVERRUN`

  Buffer overrun in api layer

`Id`
Identifier (0x00..0x3F)

`IdCode`
Generally the Identifier byte (consisting of identifier + parity bits), BUT if flag `G_LIN__MONITOR__ITEM_FLAG__EVENT` is set, **IdCode** specifies one of the following events:

- **IdCode** = `0`:

  rising edge at the trigger input

- **IdCode** = 1:

  falling edge at the trigger input

`Length`
Data length including checksum (0..9)

`Data`
Data bytes and checksum (0..9)

`StartTime`
Start time stamp in nanoseconds

`BitTimeX8`
Bit time measured over eight bit times in nanoseconds

`TimeStamps`
Structure with additional timestamps

The structure consists of the following elements:

- `BreakDelimiter`

  Time stamp of the rising edge of the **BreakDelimiter**
- `SyncByte`

  Time stamp of the start bit of the **SyncByte** in nanoseconds
- `Identifier`

  Time stamp of the start bit of the **Identifier** in nanoseconds
- `Data` (0..9)

  Time stamps of the start bits of the data bytes (0..9) in nanoseconds

Monitoring with **Normal Monitor Items** on **smartCAR** - Hardware is only possible with the enhanced instruction set.

G_Lin_Monitor_DefineFilter

G_Lin_Monitor_DefineFilter — Receiving Filter Definition

Description

With this command, a monitor filter can be defined and selected identifiers can be blocked or be received (accepted) explicitly.

Definition

```
G_Error_t
G_Lin_Monitor_DefineFilter(
 const G_PortHandle_t  portHandle,
 const G_Lin_Monitor_DefineFilter_Mode_t  mode,
 const u8_t  startId,
 const u8_t  endId
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
Filter mode

The following values are possible:

- `G_LIN__MONITOR__DEFINE_FILTER_MODE__ACCEPT_ALL`

  All identifiers will be received (default mode)
- `G_LIN__MONITOR__DEFINE_FILTER_MODE__BLOCK_ALL`

  All identifiers are blocked
- `G_LIN__MONITOR__DEFINE_FILTER_MODE__SINGLE_ACCEPTANCE_RANGE`

  Only identifiers within the range defined by `startId` and `endId` will be received
- `G_LIN__MONITOR__DEFINE_FILTER_MODE__SINGLE_BLOCKING_RANGE`

  Identifiers within the range defined by `startId` and `endId` will be blocked
- `G_LIN__MONITOR__DEFINE_FILTER_MODE__ADD_ACCEPTANCE_RANGE`

  Identifiers within the range defined by `startId` and `endId` will additionally be received
- `G_LIN__MONITOR__DEFINE_FILTER_MODE__ADD_BLOCKING_RANGE`

  Identifiers within the range defined by `startId` and `endId` will additionally be blocked

`startId`
Identifier that defines the beginning of the range

`endId`
Identifier that defines the end of the range

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
mode =
 G_LIN__MONITOR__DEFINE_FILTER_MODE__SINGLE_ACCEPTANCE_RANGE;

rc =
 G_Lin_Monitor_DefineFilter(
  portHandle,
  mode,
  0x10,
  0x20
 );

mode =
 G_LIN__MONITOR__DEFINE_FILTER_MODE__ADD_ACCEPTANCE_RANGE;

rc =
 G_Lin_Monitor_DefineFilter(
  portHandle,
  mode,
  0x30,
  0x35
 );
```

With the first call of `G_Lin_Monitor_DefineFilter`, a monitor filter for all identifiers in the range of `0x10` - `0x20` is defined. In addition to this, the range is enhanced for identifiers in the range of `0x30` - `0x35` with the second call.

G_Lin_Monitor_BufferMode_Small_Start

G_Lin_Monitor_BufferMode_Small_Start — Monitor Activation for small monitor items

Description

This command serves to activate the Buffer Reception for small monitor items (see Monitor Items).

The LIN messages are stored in an internal ring buffer after passing the monitor filter.

For monitoring, the **Slave Task** has to be activated.
See G_Lin_Node_Config for details!

Definition

```
G_Error_t
G_Lin_Monitor_BufferMode_Small_Start(
 const G_PortHandle_t  portHandle,
 const G_Lin_Monitor_BufferMode_Mode_t  mode,
 const u32_t  bufferSize
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
Specifies the type of messages that will be monitored

The following values are possible:

- `G_LIN__MONITOR__BUFFER_MODE__MODE__NOTHING`

  Nothing is monitored.
- `G_LIN__MONITOR__BUFFER_MODE__MODE__RX`

  Only **received messages** are monitored.
- `G_LIN__MONITOR__BUFFER_MODE__MODE__RX_AND_TX`

  Received messages and **transmitted messages** are monitored.
- `G_LIN__MONITOR__BUFFER_MODE__MODE__WAKE_UP`

  Only **WakeUp-Events** are monitored.
- `G_LIN__MONITOR__BUFFER_MODE__MODE__WAKE_UP_AND_RX`

  Only **WakeUp-Events** and **received messages** are monitored.
- `G_LIN__MONITOR__BUFFER_MODE__MODE__WAKE_UP_AND_TX`

  Only **WakeUp-Events** and **transmitted messages** are monitored.
- `G_LIN__MONITOR__BUFFER_MODE__MODE__WAKE_UP_AND_RX_AND_TX`

  WakeUp-Events , **received messages** and **transmitted messages** are monitored.

`bufferSize`
Size of the internal buffer for monitor items.

The minimum size that is needed is hardware-dependent.

The bigger the internal buffer, the lesser a call of G_Lin_Monitor_BufferMode_Small_GetItems is necessary for avoiding buffer overruns.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_BufferMode_Small_Stop

G_Lin_Monitor_BufferMode_Small_Stop — Stop the buffer monitor

Description

The monitoring of small buffer items is stopped.

Definition

```
G_Error_t
G_Lin_Monitor_BufferMode_Small_Stop(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_BufferMode_Small_GetItems

G_Lin_Monitor_BufferMode_Small_GetItems — Query small LIN buffer items

Description

With this command you can query small LIN monitor buffer items from the monitor buffer.

Definition

```
G_Error_t
G_Lin_Monitor_BufferMode_Small_GetItems(
 const G_PortHandle_t  portHandle,
 const u32_t  bufferSize,
 u32_t * const numberOfItems,
 G_Lin_Monitor_BufferMode_Small_Item_t * const items
);
```

Parameters

`portHandle`
Handle to the communication port

`bufferSize`
Size of the user buffer for monitor items indicated by `items`

`numberOfItems`
Pointer to **u32_t**-Variable, where the number of the returned monitor items is stored

`items`
Pointer to buffer for small LIN monitor items

See Monitor Items for details.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
const u32_t  bufferSize = 40000;
u32_t  numberOfItems;
G_Lin_Monitor_BufferMode_Small_Item_t * items;
G_Error_t  rc;

items = malloc(bufferSize);

if (items == NULL) {
 return -1;
}

rc =
 G_Lin_Monitor_BufferMode_Small_GetItems(
  portHandle,
  bufferSize,
  &numberOfItems,
  items
```

```
);
```

A 40000 Byte buffer is allocated and after the call of **G_Lin_Monitor_BufferMode_Small_GetItems** the retrieved items are available in the buffer indicated by `items`.

G_Lin_Monitor_BufferMode_Small_Async_Start

G_Lin_Monitor_BufferMode_Small_Async_Start — Start asynchronous monitoring

Description

This command enables the asynchronous reception of small LIN monitor items.

Once a monitor item is received, the callback function declared by `callback` is called.

For monitoring, the **Slave Task** has to be activated.
See G_Lin_Node_Config for details!

Definition

```
G_Error_t
G_Lin_Monitor_BufferMode_Small_Async_Start(
 const G_PortHandle_t  portHandle,
 const G_Lin_Monitor_BufferMode_Mode_t  mode,
 const G_Lin_Monitor_BufferMode_Small_Async_Callback_t callback
);
```

Parameters

`portHandle`
: Handle to the communication port

`mode`
: Specifies the type of messages that will be monitored (see Lin Monitor Buffer Mode)

`callback`
: Function pointer of the callback function (see G_Lin_Monitor_BufferMode_Small_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_BufferMode_Small_Async_Callback

G_Lin_Monitor_BufferMode_Small_Async_Callback — Callback function for asynchronous monitoring

Description

This function is automatically called when the asynchronous monitoring has been activated by G_Lin_Monitor_BufferMode_Small_Async_Start and small LIN monitor items have been received.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Lin_Monitor_BufferMode_Small_Async_Callback(
 const G_PortHandle_t  portHandle,
 const u32_t numberOfItems,
 const G_Lin_Monitor_BufferMode_Small_Item_t * const items
);
```

Parameters

`portHandle`
: Handle to the communication port

`numberOfItems`
: Number of monitor items

`items`
: Pointer to monitor items

  See Monitor Items for details.

This function has to be defined by the user.

G_Lin_Monitor_BufferMode_Small_Async_Stop

G_Lin_Monitor_BufferMode_Small_Async_Stop — Stop asynchronous monitoring

Description

This function stops the asynchronous monitoring of small LIN monitor items.

Definition

```
G_Error_t
G_Lin_Monitor_BufferMode_Small_Async_Stop(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_BufferMode_Normal_Start

G_Lin_Monitor_BufferMode_Normal_Start — Monitor Activation for normal monitor items

Description

This command serves to activate the Buffer Reception for normal monitor items (see Monitor Items).

The LIN messages are stored in an internal ring buffer after passing the monitor filter.

For monitoring, the **Slave Task** has to be activated.
See G_Lin_Node_Config for details!

Monitoring with **Normal Monitor Items** on **smartCAR** - Hardware is only possible with the enhanced instruction set.

Definition

```
G_Error_t
G_Lin_Monitor_BufferMode_Normal_Start(
 const G_PortHandle_t  portHandle,
 const G_Lin_Monitor_BufferMode_Mode_t  mode,
 const u32_t  bufferSize
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
see Lin Monitor Buffer Mode

`bufferSize`
Size of the internal buffer for monitor items

The minimum size that is needed is hardware-dependent.

The bigger the internal buffer, the lesser a call of G_Lin_Monitor_BufferMode_Normal_GetItems is necessary for avoiding buffer overruns.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_BufferMode_Normal_Stop

G_Lin_Monitor_BufferMode_Normal_Stop — Stop the buffer monitor

Description

The monitoring of normal buffer items is stopped.

Monitoring with **Normal Monitor Items** on **smartCAR** - Hardware is only possible with the enhanced instruction set.

Definition

```
G_Error_t
G_Lin_Monitor_BufferMode_Normal_Stop(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_BufferMode_Normal_GetItems

G_Lin_Monitor_BufferMode_Normal_GetItems — Query normal LIN buffer items

Description

With this command you can query normal LIN monitor buffer items from the monitor buffer.

Monitoring with **Normal Monitor Items** on **smartCAR** - Hardware is only possible with the enhanced instruction set.

Definition

```
G_Error_t
G_Lin_Monitor_BufferMode_Normal_GetItems(
 const G_PortHandle_t  portHandle,
 const u32_t  bufferSize,
 u32_t * const numberOfItems,
 G_Lin_Monitor_BufferMode_Normal_Item_t * const items
);
```

Parameters

`portHandle`
Handle to the communication port

`bufferSize`
Size of the user buffer for monitor items indicated by `items`.

`numberOfItems`
Pointer to **u32_t**-Variable, where the number of the returned monitor items is stored

`items`
Pointer to buffer for normal LIN monitor items

See Monitor Items for details.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_BufferMode_Normal_Async_Start

G_Lin_Monitor_BufferMode_Normal_Async_Start — Start asynchronous monitoring

Description

This command enables the asynchronous reception of normal LIN monitor items.

Once a monitor item is received, the callback function declared by `callback` is called.

For monitoring, the **Slave Task** has to be activated.
See G_Lin_Node_Config for details!

Monitoring with **Normal Monitor Items** on **smartCAR** - Hardware is only possible with the enhanced instruction set.

Definition

```
G_Error_t
G_Lin_Monitor_BufferMode_Normal_Async_Start(
 const G_PortHandle_t  portHandle,
 const G_Lin_Monitor_BufferMode_Mode_t  mode,
 const G_Lin_Monitor_BufferMode_Normal_Async_Callback_t callback
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
see Lin Monitor Buffer Mode

`callback`
Function pointer of the callback function (see G_Lin_Monitor_BufferMode_Normal_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_BufferMode_Normal_Async_Callback

G_Lin_Monitor_BufferMode_Normal_Async_Callback — Callback function for asynchronous monitoring

Description

This function is automatically called when the asynchronous monitoring has been activated by G_Lin_Monitor_BufferMode_Normal_Async_Start and normal LIN monitor items have been received.

Monitoring with **Normal Monitor Items** on **smartCAR** - Hardware is only possible with the enhanced instruction set.

Definition

```
void
G_Lin_Monitor_BufferMode_Normal_Async_Callback(
 const G_PortHandle_t  portHandle,
 const u32_t numberOfItems,
 const G_Lin_Monitor_BufferMode_Normal_Item_t * const items
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfItems`
Number of monitor items

`items`
Pointer to monitor items

See Monitor Items for details.

This function has to be defined by the user.

G_Lin_Monitor_BufferMode_Normal_Async_Stop

G_Lin_Monitor_BufferMode_Normal_Async_Stop — Stop asynchronous monitoring

Description

This function stops the asynchronous monitoring of normal LIN monitor items.

Monitoring with **Normal Monitor Items** on **smartCAR** - Hardware is only possible with the enhanced instruction set.

Definition

```
G_Error_t
G_Lin_Monitor_BufferMode_Normal_Async_Stop(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_BufferMode_Advanced_Start

G_Lin_Monitor_BufferMode_Advanced_Start — Monitor Activation for advanced monitor items

Description

This command serves to activate the Buffer Reception for advanced monitor items (see Monitor Items).

The LIN messages are stored in an internal ring buffer after passing the monitor filter.

For monitoring, the **Slave Task** has to be activated.
See G_Lin_Node_Config for details!

Monitoring with **Advanced Monitor Items** is only possible with the **Advanced Library** .

Definition

```
G_Error_t
G_Lin_Monitor_BufferMode_Advanced_Start(
 const G_PortHandle_t  portHandle,
 const G_Lin_Monitor_BufferMode_Mode_t  mode,
 const u32_t  bufferSize
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
see Lin Monitor Buffer Mode

`bufferSize`
Size of the internal buffer for monitor items

The minimum size that is needed is hardware-dependent.

The bigger the internal buffer, the lesser a call of G_Lin_Monitor_BufferMode_Advanced_GetItems is necessary for avoiding buffer over-runs.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_BufferMode_Advanced_Stop

G_Lin_Monitor_BufferMode_Advanced_Stop — Stop the buffer monitor

Description

The monitoring of advanced buffer items is stopped.

Monitoring with **Advanced Monitor Items** is only possible with the **Advanced Library .**

Definition

```
G_Error_t
G_Lin_Monitor_BufferMode_Advanced_Stop(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_BufferMode_Advanced_GetItems

G_Lin_Monitor_BufferMode_Advanced_GetItems — Query advanced LIN buffer items

Description

With this command you can query advanced LIN monitor buffer items from the monitor buffer.

Monitoring with **Advanced Monitor Items** is only possible with the **Advanced Library .**

Definition

```
G_Error_t
G_Lin_Monitor_BufferMode_Advanced_GetItems(
 const G_PortHandle_t  portHandle,
 const u32_t  bufferSize,
 u32_t * const numberOfItems,
 G_Lin_Monitor_BufferMode_Advanced_Item_t * const items
);
```

Parameters

`portHandle`
Handle to the communication port

`bufferSize`
Size of the user buffer for monitor items indicated by `items`.

`numberOfItems`
Pointer to **u32_t**-Variable, where the number of the returned monitor items is stored

`items`
Pointer to buffer for advanced LIN monitor items

See Monitor Items for details.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_BufferMode_Advanced_Async_Start

G_Lin_Monitor_BufferMode_Advanced_Async_Start — Start asynchronous monitoring

Description

This command enables the asynchronous reception of advanced LIN monitor items.

Once a monitor item is received, the callback function declared by `callback` is called.

For monitoring, the **Slave Task** has to be activated.
See G_Lin_Node_Config for details!

Monitoring with **Advanced Monitor Items** is only possible with the **Advanced Library** .

Definition

```
G_Error_t
G_Lin_Monitor_BufferMode_Advanced_Async_Start(
 const G_PortHandle_t  portHandle,
 const G_Lin_Monitor_BufferMode_Mode_t  mode,
 const G_Lin_Monitor_BufferMode_Advanced_Async_Callback_t callback
);
```

Parameters

`portHandle`
: Handle to the communication port

`mode`
: see Lin Monitor Buffer Mode

`callback`
: Function pointer of the callback function (see G_Lin_Monitor_BufferMode_Advanced_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_BufferMode_Advanced_Async_Callback

G_Lin_Monitor_BufferMode_Advanced_Async_Callback — Callback function for asynchronous monitoring

Description

This function is automatically called when the asynchronous monitoring has been activated by G_Lin_Monitor_BufferMode_Advanced_Async_Callback and advanced LIN monitor items have been received.

Monitoring with **Advanced Monitor Items** is only possible with the **Advanced Library** .

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Lin_Monitor_BufferMode_Advanced_Async_Callback(
 const G_PortHandle_t  portHandle,
 const u32_t numberOfItems,
 const G_Lin_Monitor_BufferMode_Advanced_Item_t * const items
);
```

Parameters

`portHandle`
: Handle to the communication port

`numberOfItems`
: Number of monitor items

`items`
: Pointer to monitor items

  See Monitor Items for details.

This function has to be defined by the user.

G_Lin_Monitor_BufferMode_Advanced_Async_Stop

G_Lin_Monitor_BufferMode_Advanced_Async_Stop — Stop asynchronous monitoring

Description

This function stops the asynchronous monitoring of advanced LIN monitor items.

Monitoring with **Advanced Monitor Items** is only possible with the **Advanced Library** .

Definition

```
G_Error_t
G_Lin_Monitor_BufferMode_Advanced_Async_Stop(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_BufferMode_EnableEvent

G_Lin_Monitor_BufferMode_EnableEvent — Enable monitor event

Description

This command enables the monitor to signal the event specified by `eventHandle` as soon as monitor data is received.

For general notes on **G-API** event handling, see Section 5, "Events".

Definition

```
G_Error_t
G_Lin_Monitor_BufferMode_EnableEvent(
 const G_PortHandle_t portHandle,
 const G_Common_EventHandle_t * const eventHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

`eventHandle`
: Handle to the event

  Each time monitor data is received, the specified handle is signaled.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_BufferMode_DisableEvent

G_Lin_Monitor_BufferMode_DisableEvent — Disable monitor event

Description

This command disables a monitor event.

After this function has been executed, no event will be signaled when new monitor data is received.

For general notes on **G-API** event handling, see Section 5, “Events”.

Definition

```
G_Error_t
G_Lin_Monitor_BufferMode_DisableEvent(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_ListMode_Small_Start

G_Lin_Monitor_ListMode_Small_Start — Start list monitor for small items

Description

This function starts the monitor in **list mode** for small monitor items.

In list mode, a list entry exists for each identifier. This list entry is updated by reception or transmission of messages with this identifier. It can be queried explicitly at any time.

For monitoring, the **Slave Task** has to be activated.
See G_Lin_Node_Config for details!

Definition

```
G_Error_t
G_Lin_Monitor_ListMode_Small_Start(
  const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_ListMode_Small_Stop

G_Lin_Monitor_ListMode_Small_Stop — Stop list monitor

Description

This function stops the monitor.

Definition

```
G_Error_t
G_Lin_Monitor_ListMode_Small_Stop(
  const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_ListMode_Small_GetItem

G_Lin_Monitor_ListMode_Small_GetItem — Get small list item

Description

This function is used to query a small monitor list item.

Definition

```
G_Error_t
G_Lin_Monitor_ListMode_Small_GetItem(
 const G_PortHandle_t  portHandle,
 const u8_t  id,
 G_Lin_Monitor_ListMode_Small_Item_t * const item
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Identifier (0x00..0x3F)

`item`
: Pointer to buffer for list entry

(see Small List Monitor Item)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_ListMode_Small_GetItem_Async

G_Lin_Monitor_ListMode_Small_GetItem_Async — Asynchronous query of small LIN list monitor items

Description

This command initiates an asynchronous query of small LIN list monitor items.

In contrast to the command G_Lin_Monitor_ListMode_Small_GetItem, this command returns immediately (that means without waiting for the response). As soon as the corresponding response is available, the callback function set by G_Lin_Monitor_ListMode_Small_GetItem_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Lin_Monitor_ListMode_Small_GetItem_Async(
 const G_PortHandle_t  portHandle,
 const u8_t  id
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: ID of the list item to be queried (0x00..0x3F)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_ListMode_Small_GetItem_Async_Callback

G_Lin_Monitor_ListMode_Small_GetItem_Async_Callback — Callback function for asynchronous monitoring

Description

This function is automatically called when the asynchronous query of small LIN monitor items has been initiated with function G_Lin_Monitor_ListMode_Small_GetItem_Async.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Lin_Monitor_ListMode_Small_GetItem_Async_Callback(
 const G_PortHandle_t  portHandle,
 const G_Lin_Monitor_ListMode_Small_Item_t * const item
);
```

Parameters

`portHandle`
Handle to the communication port

`item`
List entry

(see Small List Monitor Item)

This function has to be defined by the user.

G_Lin_Monitor_ListMode_Small_GetItem_Async_AddCallback

G_Lin_Monitor_ListMode_Small_GetItem_Async_AddCallback — Set the callback function for the asynchronous query of a small list monitor item

Description

This command sets a callback function for the asynchronous query of a small list monitor item with command G_Lin_Monitor_ListMode_Small_GetItem_Async.

Definition

```
G_Error_t
G_Lin_Monitor_ListMode_Small_GetItem_Async_AddCallback(
 const G_PortHandle_t  portHandle,
 const G_Lin_Monitor_ListMode_Small_GetItem_Async_Callback_t  callback
);
```

Parameters

`portHandle`
: Handle to the communication port

`callback`
: Function pointer of the callback function (see G_Lin_Monitor_ListMode_Small_GetItem_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_ListMode_Small_GetItem_Async_RemoveCallback

G_Lin_Monitor_ListMode_Small_GetItem_Async_RemoveCallback — Remove the callback function for the asynchronous query of a small list monitor item

Description

This command removes the callback function for the asynchronous query of a small list monitor item.

Definition

```
G_Error_t
G_Lin_Monitor_ListMode_Small_GetItem_Async_RemoveCallback(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_ListMode_Normal_Start

G_Lin_Monitor_ListMode_Normal_Start — Start list monitor for normal items

Description

This function starts the monitor in **list mode** for normal monitor items.

In list mode, a list entry exists for each identifier. This list entry is updated by reception or transmission of messages with this identifier. It can be queried explicitly at any time.

For monitoring, the **Slave Task** has to be activated.
See G_Lin_Node_Config for details!

Definition

```
G_Error_t
G_Lin_Monitor_ListMode_Normal_Start(
  const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_ListMode_Normal_Stop

G_Lin_Monitor_ListMode_Normal_Stop — Stop list monitor

Description

This function stops the monitor.

Definition

```
G_Error_t
G_Lin_Monitor_ListMode_Normal_Stop(
  const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_ListMode_Normal_GetItem

G_Lin_Monitor_ListMode_Normal_GetItem — Get normal list item

Description

This function is used to query a normal monitor list item.

Definition

```
G_Error_t
G_Lin_Monitor_ListMode_Normal_GetItem(
 const G_PortHandle_t  portHandle,
 const u8_t  id,
 G_Lin_Monitor_ListMode_Small_Item_t * const item
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Identifier (0x00..0x3F)

`item`
Pointer to buffer for list entry

(see Normal List Monitor Item)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_ListMode_Normal_GetItem_Async

G_Lin_Monitor_ListMode_Normal_GetItem_Async — Asynchronous query of normal LIN list monitor items

Description

This command initiates an asynchronous query of normal LIN list monitor items.

In contrast to G_Lin_Monitor_ListMode_Normal_GetItem, this command returns immediately (that means without waiting for the response). As soon as the corresponding response is available, the call-back function set by G_Lin_Monitor_ListMode_Normal_GetItem_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Lin_Monitor_ListMode_Normal_GetItem_Async(
 const G_PortHandle_t  portHandle,
 const u8_t  id
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: ID of the list item to query (0x00..0x3F)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_ListMode_Normal_GetItem_Async_Callback

G_Lin_Monitor_ListMode_Normal_GetItem_Async_Callback — Callback function for asynchronous monitoring

Description

This function is automatically called when the asynchronous query of normal LIN monitor items has been initiated with function G_Lin_Monitor_ListMode_Normal_GetItem_Async.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Lin_Monitor_ListMode_Normal_GetItem_Async_Callback(
 const G_PortHandle_t  portHandle,
 const G_Lin_Monitor_ListMode_Small_Item_t * const item
);
```

Parameters

`portHandle`
: Handle to the communication port

`item`
: List entry

(see Normal List Monitor Item)

This function has to be defined by the user.

G_Lin_Monitor_ListMode_Normal_GetItem_Async_AddCallback

G_Lin_Monitor_ListMode_Normal_GetItem_Async_AddCallback — Set the callback function for the asynchronous query of a small list monitor item

Description

This command sets a callback function for the asynchronous query of a small list monitor item with command G_Lin_Monitor_ListMode_Normal_GetItem_Async

Definition

```
G_Error_t
G_Lin_Monitor_ListMode_Normal_GetItem_Async_AddCallback(
 const G_PortHandle_t  portHandle,
 const G_Lin_Monitor_ListMode_Normal_GetItem_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`callback`
Function pointer of the callback function (see G_Lin_Monitor_ListMode_Normal_GetItem_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Monitor_ListMode_Normal_GetItem_Async_RemoveCallback

G_Lin_Monitor_ListMode_Normal_GetItem_Async_RemoveCallback — Remove the callback function for the asynchronous query of a normal list monitor item

Description

This command removes the callback function for the asynchronous query of a normal list monitor item.

Definition

```
G_Error_t
G_Lin_Monitor_ListMode_Normal_GetItem_Async_RemoveCallback(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

8 Relays

These functions provide control over the relays of the LIN hardware.

For a list of the available relays and their function refer to the documentation of your LIN hardware!

G_Lin_Relays_Set

G_Lin_Relays_Set — Set one or more relays

Description

This function sets one or more of the relays of the LIN hardware.

Definition

```
G_Error_t
G_Lin_Relays_Set(
  const G_PortHandle_t  portHandle,
  const G_Lin_Relays_Set_CmdFlags_t  cmdFlags,
  const u8_t numberOfRelays,
  const u8_t * const relays
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

- `G_LIN__RELAYS__SET__CMD_FLAG__NONE`

  No command flag set
- `G_LIN__RELAYS__SET__CMD_FLAG__SELECT_ALL`

  All relays will be set.

`numberOfRelays`
Number of relays to be set (only relevant if `G_LIN__RELAYS__SET__CMD_FLAG__SELECT_ALL` is not set)

`relays`
Relays list (only relevant if `G_LIN__RELAYS__SET__CMD_FLAG__SELECT_ALL` is not set)

Every byte represents a relay-number

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Relays_Reset

G_Lin_Relays_Reset — Reset one or more relays

Description

This function resets one or more of the relays of the LIN hardware.

Definition

```
G_Error_t
G_Lin_Relays_Reset(
  const G_PortHandle_t  portHandle,
  const G_Lin_Relays_Reset_CmdFlags_t  cmdFlags,
  const u8_t numberOfRelays,
  const u8_t * const relays
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

- `G_LIN__RELAYS__RESET__CMD_FLAG__NONE`

  No command flag set
- `G_LIN__RELAYS__RESET__CMD_FLAG__SELECT_ALL`

  All relays will be reset.

`numberOfRelays`
Number of relays to reset (only relevant if `G_LIN__RELAYS__SET__CMD_FLAG__SELECT_ALL` is not set)

`relays`
Relays list (only relevant if `G_LIN__RELAYS__SET__CMD_FLAG__SELECT_ALL` is not set)

Every byte represents a relay-number.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Relays_SetDirect

G_Lin_Relays_SetDirect — Configure all relays

Description

This function configures all relays of the LIN hardware.

Definition

```
G_Error_t
G_Lin_Relays_SetDirect(
  const G_PortHandle_t  portHandle,
  const u32_t  relays
);
```

Parameters

`portHandle`
: Handle to the communication port

`relays`
: Relays list

  Every bit represents a relays number.

  If a bit is set/reset the corresponding relays will be set/reset.
  - Bit 0: Relay 1
  - Bit 1: Relay 2
  - Bit 2: Relay 3
  - etc.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

9 Pull Up Resistor

These functions provide control over the pull up resistors of your LIN hardware.

The pull up resistors are used to change between **Master** and **Slave** configuration of the Interface.

G_Lin_PullUpResistor_Enable

G_Lin_PullUpResistor_Enable — Set pull up resistor to enable **master mode**

Description

This function sets the pull up resistor of the LIN interface to enable **master mode.**

Definition

```
G_Error_t
G_Lin_PullUpResistor_Enable(
  const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_PullUpResistor_Disable

G_Lin_PullUpResistor_Disable — Reset pull up resistor to enable **slave mode**

Description

This function resets the pull up resistor of the LIN interface to enable **slave mode.**

Definition

```
G_Error_t
G_Lin_PullUpResistor_Disable(
  const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

10 Diag

These functions provide control over the diagnostics functionality of your LIN hardware.

G_Lin_Diag_Config_Raw

G_Lin_Diag_Config_Raw — Configure a multisession channel for diagnostics in **Raw Mode** .

Description

This function configures a multisession channel for diagnostics in **Raw Mode**.

Definition

```
G_Error_t
G_Lin_Diag_Config_Raw(
  const G_PortHandle_t  portHandle,
  const u8_t  channel,
  const G_Lin_Diag_TxMethod_t  txMethod,
  const G_Lin_Diag_Config_Raw_Parameters_t * const rawParameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`txMethod`
Sending or Scheduling mode for MasterRequest IDs and SlaveResponse IDs

The following values are possible:

- `G_LIN__DIAG__TX_METHOD__DIAG_ID_IN_SCHEDULE_TABLE`

  Diagnostic identifiers (MasterRequest Identifier and SlaveResponse Identifier) are contained in the **Schedule Table**
- `G_LIN__DIAG__TX_METHOD__SPORADIC_FRAME_IN_SCHEDULE_TABLE`

  Sending the MasterRequest identifier or the SlaveResponse identifier in a **Sporadic Frame Slot** , unless a **Spodadic Frame** is sent at this moment (regarding **Frame Slot** see also Section 1.3, "Structure of a LIN Frame")
- `G_LIN__DIAG__TX_METHOD__SEND_ONCE_AT_END_OF_SCHEDULE_TABLE`

  Sending a MasterRequest identifier or SlaveResponse identifier ONCE at the end of the **Schedule Table**
- `G_LIN__DIAG__TX_METHOD__SEND_ALL_AT_END_OF_SCHEDULE_TABLE`

  Sending of ALL MasterRequest IDs and SlaveResponse IDs at the end of the **Schedule Table** , as long as the Diagnostic Request is sent and the Diagnostic Response is received completely.
- `G_LIN__DIAG__TX_METHOD__INTERRUPT_SCHEDULER`

  The normal **Schedule Table** is interrupted for a Diagnostic Request and its belonging Diagnostic Response as long as all corresponding MasterRequest IDs and SlaveResponse IDs are sent

`rawParameters`
Pointer to a structure of type **G_Lin_Diag_Config_Raw_Parameters_t**

The structure contains the following elements:

- P2max

P2max timeout in milliseconds (maximum time between end of the request and beginning of the response, e.g. **200 milliseconds**)

- P3max

  P3max timeout in milliseconds (maximum time between end of the request and beginning of the response during **ResponsePending** , e.g. **5100 milliseconds**)

- Repetitions

  Number of repetitions of the request, if the ECU does not react within the **P2max** or **P3max** timeouts (e.g. **2**)

- reserved1
- reserved2
- DefaultMasterData

  Structure with the following elements:

  - Mode

    Mode of the **Default Master Request Frame**

    The following values are possible:

    - `G_LIN__DIAG__CONFIG__RAW__DEFAULT_MASTER_DATA__MODE__DISABLED`

      Default Master Request Frame disabled

    - `G_LIN__DIAG__CONFIG__RAW__DEFAULT_MASTER_DATA__MODE__ENABLED`

      Default Master Request Frame enabled

  - reserved1
  - reserved2
  - reserved3
  - Data[8]

    Data of **Default Master Request Frame**

- DefaultSlaveData

  Structure with the following elements:

  - Mode

    Mode of the **Default Slave Response Frame**

    The following values are possible:

    - `G_LIN__DIAG__CONFIG__RAW__DEFAULT_SLAVE_DATA__MODE__DISABLED`

      Default Slave Response Frame disabled

    - `G_LIN__DIAG__CONFIG__RAW__DEFAULT_SLAVE_DATA__MODE__ENABLED`

      Default Slave Response Frame enabled

  - reserved1
  - reserved2
  - reserved3
  - Data[8]

    Data of **Default Slave Response Frame**

- TesterPresent

  Structure with the following elements:

  - Mode

    TesterPresent mode

    The following values are possible:

- G_LIN__DIAG__TESTER_PRESENT__MODE__DISABLED

  TesterPresent service disabled
- G_LIN__DIAG__TESTER_PRESENT__MODE__ENABLED

  TesterPresent service enabled

- ResponseMode

  TesterPresent response mode

  The following values are possible:
  - G_LIN__DIAG__TESTER_PRESENT__RSP_MODE__NO_RSP_REQUIRED

    Do not wait for a response to the **TesterPresent** service.
  - G_LIN__DIAG__TESTER_PRESENT__RSP_MODE__RSP_REQUIRED

    Wait for a response to the **TesterPresent** service.
- CycleTime

  Cycle for **TesterPresent** in milliseconds (e.g. **1000 milliseconds**)
- Data[8]

  Data of the **TesterPresent** service

- RxEndCondition

  End recognition of diagnostics responses in the case of multi frames

  The following values are possible:
  - G_LIN__DIAG__CONFIG__RAW__RX_END_CONDITION__SINGLE_FRAMES

    Only single frames (no multi frames)
  - G_LIN__DIAG__CONFIG__RAW__RX_END_CONDITION__NO_RESPONSE

    Empty slot (no response from the slave)
  - G_LIN__DIAG__CONFIG__RAW__RX_END_CONDITION__DEFAULT_DATA

    Default slave response frame
  - G_LIN__DIAG__CONFIG__RAW__RX_END_CONDITION__SAME_DATA

    Same frame
- reserved3
- reserved4
- NumberOfSpecialResponses

  Number of special diagnostics responses (**N**)
- SpecialResponses[8]

  Special diagnostics response entries

  Any received diagnostics response frame is compared with the **SpecialResponses** . This happens by a logical AND of the received data with **Mask** followed by binary comparision of the result with **Data** .

  Each Special diagnostics response entry contains the following members:
  - Mask[8]

    Mask bytes **0..7**
  - Data[8]

    Data bytes **0..7** (Data is compared with the received data in accordance with the set mask bytes bits)

- Flags

  Special diagnostics response entry flags

  The following values are possible and can be combined:
  - `G_LIN__DIAG__CONFIG__RAW__SPECIAL_RSP__FLAG__NONE`

    No flag set
  - `G_LIN__DIAG__CONFIG__RAW__SPECIAL_RSP__FLAG__REPEAT_REQUEST`

    Repeat request
  - `G_LIN__DIAG__CONFIG__RAW__SPECIAL_RSP__FLAG__CHANGE_TIMING`

    Change timing (**P2max** to **P3max**)
  - `G_LIN__DIAG__CONFIG__RAW__SPECIAL_RSP__FLAG__DEFAULT_FRAME`

    Default frame
  - `G_LIN__DIAG__CONFIG__RAW__SPECIAL_RSP__FLAG__LAST_FRAME`

    Last frame
  - `G_LIN__DIAG__CONFIG__RAW__SPECIAL_RSP__FLAG__IGNORE_IF_NOT_MATCH`

    Ignoring the received frames in the case data does not match according to **Mask** and **Data**

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Diag_Config_Lin_2_0

G_Lin_Diag_Config_Lin_2_0 — Configure a multisession channel for diagnostics in **Raw Mode**.

Description

This function configures a multisession channel for diagnostics in **Raw Mode**.

Definition

```
G_Error_t
G_Lin_Diag_Config_Lin_2_0(
  const G_PortHandle_t  portHandle,
  const u8_t  channel,
  const G_Lin_Diag_TxMethod_t  txMethod,
  const G_Lin_Diag_Config_Lin_2_0_Parameters_t * const lin_2_0_Parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`txMethod`
Sending or Scheduling mode for MasterRequest IDs and SlaveResponse IDs (see TxMethod)

`lin_2_0_Parameters`
Pointer to a structure of type **G_Lin_Diag_Config_Lin_2_0_Parameters_t**

The structure contains the following elements:

- NAD

  Address of the control unit (**N**ODE **AD**DRESS)
- reserved
- P2max

  P2max timeout in milliseconds (maximum time between end of the request and beginning of the response, e.g. **200 milliseconds**)
- P3max

  P3max timeout in milliseconds (maximum time between end of the request and beginning of the response during **ResponsePending**, e.g. **5100 milliseconds**)
- Repetitions

  Number of repetitions of the request, if the ECU does not react within the **P2max** or **P3max** timeouts (e.g. **2**)
- TesterPresent

  Structure with the following elements:
  - Mode

    TesterPresent mode (see TesterPresent Mode)
  - ResponseMode

    TesterPresent response mode (see TesterPresent ResponseMode)
  - CycleTime

Cycle for **TesterPresent** in milliseconds (e.g. **1000 milliseconds**)

- reserved1
- reserved2
- reserved3
- Length

  Data length of **TesterPresent** service
- Data[8]

  Data of the **TesterPresent** service (starting with service identifier, generally **0x3E**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Diag_Config_Off

G_Lin_Diag_Config_Off — Stops a diagnostics session

Description

This function stops an active diagnostics session and frees the allocated resources.

Definition

```
G_Error_t
G_Lin_Diag_Config_Off(
  const G_PortHandle_t  portHandle,
  const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Diag_GetState

G_Lin_Diag_GetState — Query the LIN diagnostics state

Description

This function queries the LIN diagnostics state for the multisession channel defined by `Channel`.

Additionally, the firmware internal **LastErrorCode** can be reset.

Generally the firmware internal **LastErrorCode** is reset automatically by starting or stopping a diagnostics session.

Definition

```
G_Error_t
G_Lin_Diag_GetState(
  const G_PortHandle_t portHandle,
  const u8_t  channel,
  const G_Lin_Diag_GetState_CmdFlags_t  cmdFlags,
  G_Error_t * const lastErrorCode,
  G_Lin_Diag_Type_t * const type,
  G_Lin_Diag_State_t * const state,
  G_Lin_Diag_GetState_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`cmdFlags`
Command Flags

The following values are possible and can be combined:

- `G_LIN__DIAG__GET_STATE__CMD_FLAG__NONE`

  No command flag set
- `G_LIN__DIAG__GET_STATE__CMD_FLAG__RESET_LAST_ERROR`

  Reset the firmware internal **LastErrorCode**

`lastErrorCode`
Pointer to a variable of type **G_Error_t**. It returns the error code of the last error that has occurred in the firmware (or `G_NO_ERROR` = no error). For getting an error description, you can call G_Common_GetFirmwareErrorDescription.

`type`
Pointer to buffer for type of diagnostics session

The following values are possible:

- `G_LIN__DIAG__TYPE__OFF`

  No diagnostics session started
- `G_LIN__DIAG__TYPE__RAW`

Diagnostics in **RAW mode**

- `G_LIN__DIAG__TYPE__LIN_2_0` Diagnostics in **LIN 2.0 mode**

`state`

Pointer to buffer for state of diagnostics session

The following values are possible:

- `G_LIN__DIAG__STATE__NOT_INITIALIZED`

  Diagnostic session not initialized
- `G_LIN__DIAG__STATE__DISCONNECTED`

  Diagnostic session not connected
- `G_LIN__DIAG__STATE__CONNECT_IN_PROGRESS`

  Diagnostic session is currently connecting
- `G_LIN__DIAG__STATE__CONNECTED`

  Diagnostic session is connected
- `G_LIN__DIAG__STATE__DISCONNECT_IN_PROGRESS`

  Diagnostic session is currently disconnecting

`rspFlags`

Pointer to buffer for response flags

The following values are possible and can be combined:

- `G_LIN__DIAG__GET_STATE__RSP_FLAG__NONE`

  No flag set
- `G_LIN__DIAG__GET_STATE__RSP_FLAG__BUSY`

  A request or a response to a request has not been successfully sent yet.
- `G_LIN__DIAG__GET_STATE__RSP_FLAG__RX_BUFFER_NOT_EMPTY`

  The diagnostics response buffer is not empty.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Diag_GetState_Async

G_Lin_Diag_GetState_Async — Asynchronous query of the diagnostics state.

Description

This command initiates an asynchronous query of the LIN diagnostics state.

In contrast to G_Lin_Diag_GetState, this command returns immediately (that means without waiting for the response). As soon as the corresponding response is available, the callback function set by G_Lin_Diag_GetState_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Lin_Diag_GetState_Async(
  const G_PortHandle_t portHandle,
  const u8_t  channel,
  const G_Lin_Diag_GetState_CmdFlags_t  cmdFlags
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with **0**)

`cmdFlags`
: Command Flags (see LIN Diag GetState Command Flags)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Diag_GetState_Async_Callback

G_Lin_Diag_GetState_Async_Callback — Callback function for asynchronous querying of the diagnostics state

Description

This function is automatically called when the asynchronous query of the diagnostics state is initiated with function G_Lin_Diag_GetState_Async.

Definition

```
void
G_Lin_Diag_GetState_Async_Callback (
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Error_t  lastErrorCode,
 const G_Lin_Diag_Type_t  type,
 const G_Lin_Diag_State_t  state,
 const G_Lin_Diag_GetState_RspFlags_t  rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`lastErrorCode`
Pointer to a variable of type **G_Error_t**. It returns the error code of the last error that has occurred in the firmware (or `G_NO_ERROR` = no error). For getting an error description, you can call G_Common_GetFirmwareErrorDescription.

`type`
Type of diagnostics session

The following values are possible:

- `G_LIN__DIAG__TYPE__OFF`

  No diagnostics session started
- `G_LIN__DIAG__TYPE__RAW`

  Diagnostics in **RAW mode**
- `G_LIN__DIAG__TYPE__LIN_2_0` Diagnostics in **LIN 2.0 mode**

`state`
State of diagnostics session

The following values are possible:

- `G_LIN__DIAG__STATE__NOT_INITIALIZED`

  Diagnostic session not initialized
- `G_LIN__DIAG__STATE__DISCONNECTED`

  Diagnostic session not connected
- `G_LIN__DIAG__STATE__CONNECT_IN_PROGRESS`

Diagnostic session is currently connecting

- `G_LIN__DIAG__STATE__CONNECTED`

  Diagnostic session is connected

- `G_LIN__DIAG__STATE__DISCONNECT_IN_PROGRESS`

  Diagnostic session is currently disconnecting

`rspFlags`
Response flags

The following values are possible and can be combined:

- `G_LIN__DIAG__GET_STATE__RSP_FLAG__NONE`

  No flag set

- `G_LIN__DIAG__GET_STATE__RSP_FLAG__BUSY`

  A request or a response to a request has not been successfully sent yet.

- `G_LIN__DIAG__GET_STATE__RSP_FLAG__RX_BUFFER_NOT_EMPTY`

  The diagnostics response buffer is not empty.

This function has to be defined by the user.

G_Lin_Diag_GetState_Async_AddCallback

G_Lin_Diag_GetState_Async_AddCallback — Set the callback function for the asynchronous query of a small list monitor item

Description

This command sets a callback function for the asynchronous query of the diagnostics state with command G_Lin_Diag_GetState_Async.

Definition

```
G_Error_t
G_Lin_Diag_GetState_Async_AddCallback(
  const G_PortHandle_t portHandle,
  const u8_t channel,
  const G_Lin_Diag_GetState_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with **0**)

`callback`
Function pointer of the callback function (see G_Lin_Diag_GetState_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Diag_GetState_Async_RemoveCallback

G_Lin_Diag_GetState_Async_RemoveCallback — Remove the callback function for the asynchronous query of the diagnostics state

Description

This command removes the callback function for the asynchronous query of the diagnostics state.

Definition

```
G_Error_t
G_Lin_Diag_GetState_Async_RemoveCallback(
  const G_PortHandle_t portHandle,
  const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Diag_ChangeScheduleDelay

G_Lin_Diag_ChangeScheduleDelay — Change schedule delay

Description

This function changes the schedule delays for master request and slave response.

Definition

```
G_Error_t
G_Lin_Diag_ChangeScheduleDelay(
  const G_PortHandle_t portHandle,
  const u8_t  channel,
  const u32_t  masterRequestDelay,
  const u32_t  slaveResponseDelay
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with **0**)

`masterRequestDelay`
: Schedule delay for a master request in nanoseconds

`slaveResponseDelay`
: Schedule delay for a slave response in nanoseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Diag_ChangeTxTimeout

G_Lin_Diag_ChangeTxTimeout — Change sending timeout

Description

This function changes the sending timeout.

Ususally the sending timeout is 1000 milliseconds.

Definition

```
G_Error_t
G_Lin_Diag_ChangeTxTimeout(
  const G_PortHandle_t portHandle,
  const u8_t  channel,
  const u16_t  txTimeout
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with **0**)

`txTimeout`
: Sending timeout in milliseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

11 Advanced Library

These functions provide control over advanced library parameters.

To use these functions, the advanced library feature needs to be unlocked on your hardware.

G_Lin_AdvLib_DominantLevel_Set

G_Lin_AdvLib_DominantLevel_Set — Send Dominant level

Description

This command is used to send a dominant level of any length of time at any position within a LIN message.

As a consequence, another received message may be destroyed/ disturbed (e.g. destruction of the checksum or the SyncByte).

To use this function, the advanced library feature must be unlocked.

Definition

```
G_Error_t
G_Lin_AdvLib_DominantLevel_Set(
 const G_PortHandle_t portHandle,
 const G_Lin_AdvLib_DominantLevel_Set_CmdFlags_t  cmdFlags,
 const G_Lin_AdvLib_DominantLevel_RefPoint_t  referencePoint,
 const u8_t  id,
 const u32_t  offset,
 const u32_t  duration,
 const u16_t  numberOfCycles
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command Flags

The following values are possible and can be combined:

`G_LIN__ADV_LIB__DOMINANT_LEVEL__SET__CMD_FLAG__ZERO`
No flag set

`G_LIN__ADV_LIB__DOMINANT_LEVEL__SET__CMD_FLAG__ALL_IDS`
The dominant level is triggered for every identifer

Parameter `id` is disregarded.

`referencePoint`
Reference point for the start of the dominant level

The following values are possible:

`G_LIN__ADV_LIB__DOMINANT_LEVEL__REF_POINT__RE__BREAK_DELIMITER`
Rising edge of BreakDelimiter, not ID-selective

`G_LIN__ADV_LIB__DOMINANT_LEVEL__REF_POINT__FE__SYNC_BYTE`
Falling edge of the starting bit of SyncByte, not ID-selective

`G_LIN__ADV_LIB__DOMINANT_LEVEL__REF_POINT__FE__ID_BYTE`
Falling edge of the starting bit of Identifier Byte, not ID-selective

`G_LIN__ADV_LIB__DOMINANT_LEVEL__REF_POINT__FE__DATA_BYTE_0`
Falling edge of the starting bit of DataByte0

`G_LIN__ADV_LIB__DOMINANT_LEVEL__REF_POINT__FE__DATA_BYTE_1`
Falling edge of the starting bit of DataByte1

`G_LIN__ADV_LIB__DOMINANT_LEVEL__REF_POINT__FE__DATA_BYTE_2`
Falling edge of the starting bit of DataByte2

`G_LIN__ADV_LIB__DOMINANT_LEVEL__REF_POINT__FE__DATA_BYTE_3`
Falling edge of the starting bit of DataByte3

`G_LIN__ADV_LIB__DOMINANT_LEVEL__REF_POINT__FE__DATA_BYTE_4`
Falling edge of the starting bit of DataByte4

`G_LIN__ADV_LIB__DOMINANT_LEVEL__REF_POINT__FE__DATA_BYTE_5`
Falling edge of the starting bit of DataByte5

`G_LIN__ADV_LIB__DOMINANT_LEVEL__REF_POINT__FE__DATA_BYTE_6`
Falling edge of the starting bit of DataByte6

`G_LIN__ADV_LIB__DOMINANT_LEVEL__REF_POINT__FE__DATA_BYTE_7`
Falling edge of the starting bit of DataByte7

`G_LIN__ADV_LIB__DOMINANT_LEVEL__REF_POINT__FE__DATA_BYTE_8`
Falling edge of the starting bit of DataByte8

`G_LIN__ADV_LIB__DOMINANT_LEVEL__REF_POINT__FE__BREAK`
Falling edge of the Break, not ID-selective

(The earliest point in time for transmitting is the break detection threshold, see G_Lin_SlaveTask_SetBreakDetectionThreshold.)

`id`
Identifier (0x00..0x3F) of the message to be destroyed/ disturbed

`offset`
Starting-offset to `referencePoint` in nanoseconds

`duration`
Duration of the dominant level in nanoseconds

`numberOfCycles`
Number of cycles of dominant levels

The sending moment for the dominant level is falsified by the transceiver running times (receiving and transmitting running time).

The complete running time of a transceiver depends on its type. The range is between **5** and **15 microseconds** (generally **8** to **9 microseconds**).

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_AdvLib_DominantLevel_Reset

G_Lin_AdvLib_DominantLevel_Reset — Reset dominant level

Description

This command is used to stop sending a dominant level.

To use this function, the advanced library feature must be unlocked.

Definition

```
G_Error_t
G_Lin_AdvLib_DominantLevel_Reset(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_AdvLib_RecessiveLevel_Set

G_Lin_AdvLib_RecessiveLevel_Set — Send recessive level

Description

This command is used to send a recessive level of any length of time at any position within a LIN message.

As a consequence, another received message may be destroyed/ disturbed (e.g. destruction of the checksum or the SyncByte).

To use this function, the advanced library feature must be unlocked.

Definition

```
G_Error_t
G_Lin_AdvLib_RecessiveLevel_Set(
 const G_PortHandle_t portHandle,
 const G_Lin_AdvLib_RecessiveLevel_Set_CmdFlags_t  cmdFlags,
 const G_Lin_AdvLib_RecessiveLevel_RefPoint_t  referencePoint,
 const u8_t  id,
 const u32_t  offset,
 const u32_t  duration,
 const u16_t  numberOfCycles
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command Flags

The following values are possible and can be combined:

`G_LIN__ADV_LIB__RECESSIVE_LEVEL__SET__CMD_FLAG__ZERO`
No flag set

`G_LIN__ADV_LIB__RECESSIVE_LEVEL__SET__CMD_FLAG__ALL_IDS`
The recessive level is triggered for every identifer

Parameter `id` is disregarded.

`referencePoint`
Reference point for the start of the recessive level

The following values are possible:

`G_LIN__ADV_LIB__RECESSIVE_LEVEL__REF_POINT__RE__BREAK_DELIMITER`
Rising edge of BreakDelimiter, not ID-selective

`G_LIN__ADV_LIB__RECESSIVE_LEVEL__REF_POINT__FE__SYNC_BYTE`
Falling edge of the starting bit of SyncByte, not ID-selective

`G_LIN__ADV_LIB__RECESSIVE_LEVEL__REF_POINT__FE__ID_BYTE`
Falling edge of the starting bit of Identifier Byte, not ID-selective

`G_LIN__ADV_LIB__RECESSIVE_LEVEL__REF_POINT__FE__DATA_BYTE_0`
Falling edge of the starting bit of DataByte0

`G_LIN__ADV_LIB__RECESSIVE_LEVEL__REF_POINT__FE__DATA_BYTE_1`
Falling edge of the starting bit of DataByte1

`G_LIN__ADV_LIB__RECESSIVE_LEVEL__REF_POINT__FE__DATA_BYTE_2`
Falling edge of the starting bit of DataByte2

`G_LIN__ADV_LIB__RECESSIVE_LEVEL__REF_POINT__FE__DATA_BYTE_3`
Falling edge of the starting bit of DataByte3

`G_LIN__ADV_LIB__RECESSIVE_LEVEL__REF_POINT__FE__DATA_BYTE_4`
Falling edge of the starting bit of DataByte4

`G_LIN__ADV_LIB__RECESSIVE_LEVEL__REF_POINT__FE__DATA_BYTE_5`
Falling edge of the starting bit of DataByte5

`G_LIN__ADV_LIB__RECESSIVE_LEVEL__REF_POINT__FE__DATA_BYTE_6`
Falling edge of the starting bit of DataByte6

`G_LIN__ADV_LIB__RECESSIVE_LEVEL__REF_POINT__FE__DATA_BYTE_7`
Falling edge of the starting bit of DataByte7

`G_LIN__ADV_LIB__RECESSIVE_LEVEL__REF_POINT__FE__DATA_BYTE_8`
Falling edge of the starting bit of DataByte8

`G_LIN__ADV_LIB__RECESSIVE_LEVEL__REF_POINT__FE__BREAK`
Falling edge of the Break, not ID-selective

(The earliest point in time for transmitting is the break detection threshold, see G_Lin_SlaveTask_SetBreakDetectionThreshold.)

`id`
Identifier (0x00..0x3F) of the message to be destroyed/ disturbed

`offset`
Starting-offset to `referencePoint` in nanoseconds

`duration`
Duration of the recessive level in nanoseconds

`numberOfCycles`
Number of cycles of recessive levels

The sending moment for the recessive level is falsified by the transceiver running times (receiving and transmitting running time).

The complete running time of a transceiver depends on its type. The range is between **5** and **15 microseconds** (generally **8** to **9 microseconds**).

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_AdvLib_RecessiveLevel_Reset

G_Lin_AdvLib_RecessiveLevel_Reset — Reset recessive level

Description

This command is used to stop sending a recessive level.

To use this function, the advanced library feature must be unlocked.

Definition

```
G_Error_t
G_Lin_AdvLib_RecessiveLevel_Reset(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_AdvLib_WakeUpDelay_Set

G_Lin_AdvLib_WakeUpDelay_Set — Change starting behavior

Description

The command is used to change the starting behavior of the master after detecting a **WakeUp**.

To use this function, the advanced library feature must be unlocked.

Definition

```
G_Error_t
G_Lin_AdvLib_WakeUpDelay_Set(
 const G_PortHandle_t portHandle,
 const u16_t  numberOfWakeUpsForOneDetection,
 const u16_t  numberOfDelayedDetections
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfWakeUpsForOneDetection`
Defines the number of wake ups that are received before one wake up is recognized

`numberOfDelayedDetections`
Defines the number of times the wake up should be delayed (0 = infinitely)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_AdvLib_WakeUpDelay_Reset

G_Lin_AdvLib_WakeUpDelay_Reset — Reset starting behavior

Description

The command is used to reset the starting behavior of the master to its default state.

To use this function, the advanced library feature must be unlocked.

Definition

```
G_Error_t
G_Lin_AdvLib_WakeUpDelay_Reset(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_AdvLib_ErrorMode_Set

G_Lin_AdvLib_ErrorMode_Set — Generate frame errors

Description

The command is used to generate frame errors.

To use this function, the advanced library feature must be unlocked.

Definition

```
G_Error_t
G_Lin_AdvLib_ErrorMode_Set(
 const G_PortHandle_t portHandle,
 const G_Lin_AdvLib_ErrorMode_Set_CmdFlags_t  cmdFlags,
 const G_Lin_AdvLib_ErrorMode_t  mode,
 const u8_t  id,
 const u32_t  numberOfErrors,
 const G_Lin_AdvLib_ErrorMode_Set_ExParam_t * const extendedParameters
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_LIN__ADV_LIB__ERROR_MODE__SET__CMD_FLAG__ZERO`
No flag set

`G_LIN__ADV_LIB__ERROR_MODE__SET__CMD_FLAG__ALL_IDS`
Error mode is set for all IDs

The parameter `id` is disregarded.

`mode`
Errormode

The following parameters are possible:

`G_LIN__ADV_LIB__ERROR_MODE__SET__INVALID_ID_PARITY`
Identifier parity error generation

`G_LIN__ADV_LIB__ERROR_MODE__SET__INVALID_CHECKSUM_1`
Checksum error generation

The checksum is logically XORed with **0xFF** .

`G_LIN__ADV_LIB__ERROR_MODE__SET__INVALID_CHECKSUM_2`
Checksum error generation

The checksum is logically XORed with **0x01** or `XorValue`.

`G_LIN__ADV_LIB__ERROR_MODE__SET__STOP_AFTER_SYNC_BYTE`
The frame is stopped after the sync byte

`G_LIN__ADV_LIB__ERROR_MODE__SET__STOP_AFTER_ID_BYTE`
The frame is stopped after the ID byte

`G_LIN__ADV_LIB__ERROR_MODE__SET__STOP_AFTER_DATA_BYTE_0`
The frame is stopped after data byte **0**

`G_LIN__ADV_LIB__ERROR_MODE__SET__STOP_AFTER_DATA_BYTE_1`
The frame is stopped after data byte **1**

`G_LIN__ADV_LIB__ERROR_MODE__SET__STOP_AFTER_DATA_BYTE_2`
The frame is stopped after data byte **2**

`G_LIN__ADV_LIB__ERROR_MODE__SET__STOP_AFTER_DATA_BYTE_3`
The frame is stopped after data byte **3**

`G_LIN__ADV_LIB__ERROR_MODE__SET__STOP_AFTER_DATA_BYTE_4`
The frame is stopped after data byte **4**

`G_LIN__ADV_LIB__ERROR_MODE__SET__STOP_AFTER_DATA_BYTE_5`
The frame is stopped after data byte **5**

`G_LIN__ADV_LIB__ERROR_MODE__SET__STOP_AFTER_DATA_BYTE_6`
The frame is stopped after data byte **6**

`G_LIN__ADV_LIB__ERROR_MODE__SET__STOP_AFTER_DATA_BYTE_7`
The frame is stopped after data byte **7**

`G_LIN__ADV_LIB__ERROR_MODE__SET__INCONSISTENT_SYNC_BYTE`
The sync byte is inconsistent

`id`
Identifier (0x00..0x3F)

`numberOfErrors`
Number of times the error is generated

`extendedParameters`
Pointer to union with extended parameters

Extended parameters are needed for the following error modes:
- `G_LIN__ADV_LIB__ERROR_MODE__SET__INVALID_CHECKSUM_2`
- `G_LIN__ADV_LIB__ERROR_MODE__SET__INCONSISTENT_SYNC_BYTE`

For all other error modes, the pointer can be NULL.

The union consists of the following elements:

`InvalidChecksum2`
Structure with parameters for `mode = G_LIN__ADV_LIB__ERROR_MODE__SET__INVALID_CHECKSUM_2`

The structure consists of the following elements:

`XorValue`
User definable XOR value for generating checksum errors

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`InconsistentSyncByte`
Structure with parameters for `mode = G_LIN__ADV_LIB__ERROR_MODE__SET__INCONSISTENT_SYNC_BYTE`

The structure consists of the following elements:

`SyncByte`
User definable SyncByte value for generating SyncByte errors

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_AdvLib_ErrorMode_Reset

G_Lin_AdvLib_ErrorMode_Reset — Stop generating frame errors

Description

The command is used to stop the generation of frame errors.

To use this function, the advanced library feature must be unlocked.

Definition

```
G_Error_t
G_Lin_AdvLib_ErrorMode_Reset(
 const G_PortHandle_t portHandle,
 const G_Lin_AdvLib_ErrorMode_Reset_CmdFlags_t  cmdFlags,
 const u8_t  id
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_LIN__ADV_LIB__ERROR_MODE__SET__CMD_FLAG__ZERO`
No flag set

`G_LIN__ADV_LIB__ERROR_MODE__SET__CMD_FLAG__ALL_IDS`
Error mode is reset for all IDs

The parameter `id` is disregarded.

`id`
Identifier (0x00..0x3F)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

12 Bus Idle Detection

These functions provide control over the bus idle detection functionality of your LIN hardware.

G_Lin_BusIdleDetection_SetThreshold

G_Lin_BusIdleDetection_SetThreshold — Set BreakDetectionThreshold parameter

Description

This command sets the **BreakDetectionThreshold** parameter.

Definition

```
G_Error_t
G_Lin_BusIdleDetection_SetThreshold (
 const G_PortHandle_t portHandle,
 const G_Lin_BusIdleDetection_SetThreshold_CmdFlags_t  cmdFlags,
 const u32_t  threshold
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_LIN__BUS_IDLE_DETECTION__SET_THRESHOLD__CMD_FLAG__ZERO`
No flag is set

`G_LIN__BUS_IDLE_DETECTION__SET_THRESHOLD__CMD_FLAG__SET_DEFAULT`
Set the default threshold of **4 seconds**

The parameter `threshold` is disregarded.

`threshold`
Bus idle detection threshold in microseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_BusIdleDetection_GetState

G_Lin_BusIdleDetection_GetState — Query bus idle state

Description

This command is used to query the bus idle state.

Definition

```
G_Error_t
G_Lin_BusIdleDetection_GetState (
 const G_PortHandle_t portHandle,
 G_Lin_BusIdleDetection_GetState_RspFlags_t * const rspFlags,
 u32_t * const idleTime
);
```

Parameters

`portHandle`
Handle to the communication port

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G__LIN__BUS_IDLE_DETECTION__GET_STATE__RSP_FLAG__ZERO`
No flag is set

`G__LIN__BUS_IDLE_DETECTION__GET_STATE__RSP_FLAG__BUS_IDLE`
The bus is idle, there is **NO** bus activity

`G__LIN__BUS_IDLE_DETECTION__GET_STATE__RSP_FLAG__OVERFLOW`
An overflow of the internal idle timer occurred

The `idleTime` parameter cannot indicate the real idle time.

The bus is in idle state for more than **0xFFFFFFFF** microseconds (= 4294 seconds = 71,58 minutes).

`idleTime`
Idle time in microseconds (time elapsed since the last bus flank)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

13 Network Management

These functions provide control over the network management functionality of your LIN hardware.

G_Lin_Nm_WakeUp_SendImmediately

G_Lin_Nm_WakeUp_SendImmediately — Send wake up

Description

Send a wake up once if the bus is in **idle state**.

In contrast to command G_Lin_Node_SendWakeUp, the wake up is only sent if the bus is in **idle state**.

Definition

```
G_Error_t
G_Lin_Nm_WakeUp_SendImmediately (
 const G_PortHandle_t portHandle,
 const u32_t  wakeUpTime,
 G_Lin_Nm_WakeUp_SendImmediately_RspFlags_t * const rspFlags,
 u32_t * const usedWakeUpTime
);
```

Parameters

`portHandle`
Handle to the communication port

`wakeUpTime`
Duration of the dominant bus level in microseconds

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_LIN__NM__WAKE_UP__SEND_IMMEDIATELY__RSP_FLAG__ZERO`
No flag is set

`G_LIN__NM__WAKE_UP__SEND_IMMEDIATELY__RSP_FLAG__WAKE_UP_SENT`
The bus was in **idle state**, the wake up could be sent immediately.

`usedWakeUpTime`
Used duration of the dominant level in microseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Nm_WakeUp_SendAutomatically

G_Lin_Nm_WakeUp_SendAutomatically — Send wake up automatically

Description

Send a wake up automatically (if applicable several times) if **bus idle state** is detected.

Definition

```
G_Error_t
G_Lin_Nm_WakeUp_SendAutomatically (
 const G_PortHandle_t portHandle,
 const G_Lin_Nm_WakeUp_SendAutomatically_CmdFlags_t  cmdFlags,
 const u32_t  wakeUpTime,
 const u32_t  delayTime,
 const u32_t  numberOfWakeUps,
 G_Lin_Nm_WakeUp_SendAutomatically_RspFlags_t * const rspFlags,
 u32_t * const usedWakeUpTime,
 u32_t * const remainingWakeUps
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_LIN__NM__WAKE_UP__SEND_AUTOMATICALLY__CMD_FLAG__ZERO`
No flag is set

`G_LIN__NM__WAKE_UP__SEND_AUTOMATICALLY__CMD_FLAG__SEND_IMMEDIATELY`
If the bus is currently in **bus idle state**, the first wake up is sent immediately

`wakeUpTime`
Duration of the dominant bus level in microseconds

`delayTime`
Delay between recognizing the **bus idle state** and sending the wake up in microseconds

`numberOfWakeUps`
Number of wake ups to be transmitted

- **0** : Deactivate automatically sending of wake ups
- **0xFFFFFFFF** : Send unlimited number of wake ups

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_LIN__NM__WAKE_UP__SEND_AUTOMATICALLY__RSP_FLAG__ZERO`
No flag is set

`G_LIN__NM__WAKE_UP__SEND_AUTOMATICALLY__RSP_FLAG__BUS_WAS_IDLE`
The bus was in **idle state** during command execution

`usedWakeUpTime`
Used duration of the dominant level in microseconds

`remainingWakeUps`
Number of wake ups still to be transmitted

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

14 Trigger

These commmands are used to control the **LIN Trigger** functionality.

The LIN Trigger can be configured for several applications, including the transmission of frame responses or generating monitor events when configured trigger conditions are met.

G_Lin_Trigger_Output_Config

G_Lin_Trigger_Output_Config — Configure trigger output

Description

Configure LIN Trigger to generate a trigger signal on the trigger output according to parameter `mode`.

Definition

```
G_Error_t
G_Lin_Trigger_Output_Config(
 const G_PortHandle_t portHandle,
 const G_Lin_Trigger_Output_Config_CmdFlags_t  cmdFlags,
 const G_Lin_Trigger_Mode_t  mode,
 const u8_t  numberOfIds,
 const u8_t * const ids
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_LIN__TRIGGER__OUTPUT__CONFIG__CMD_FLAG__ZERO`
No flag is set

`G_LIN__TRIGGER__OUTPUT__CONFIG__CMD_FLAG__SELECT_ALL_IDS`
All received identifiers are considered as trigger condition

(default: identifiers defined by `ids` are considered as trigger conditions)

`mode`
Trigger mode

The following values are possible:

`G_LIN__TRIGGER__MODE__NORMAL`
Triggering after the identifier is received

`G_LIN__TRIGGER__MODE__BREAK_DELIMITER`
Triggering after the rising edge of the **break delimiter** (not ID-selective)

`G_LIN__TRIGGER__MODE__SYNC_BYTE`
Triggering after the startbit of the **sync byte** (not ID-selective)

`G_LIN__TRIGGER__MODE__ID_BYTE`
Triggering after the startbit of the **ID byte** (not ID-selective)

`G_LIN__TRIGGER__MODE__DATA_BYTE_0`
Triggering after the startbit of **data byte 0**

`G_LIN__TRIGGER__MODE__DATA_BYTE_1`
Triggering after the startbit of **data byte 1**

`G_LIN__TRIGGER__MODE__DATA_BYTE_2`
Triggering after the startbit of **data byte 2**

`G_LIN__TRIGGER__MODE__DATA_BYTE_3`
Triggering after the startbit of **data byte 3**

`G_LIN__TRIGGER__MODE__DATA_BYTE_4`
Triggering after the startbit of **data byte 4**

`G_LIN__TRIGGER__MODE__DATA_BYTE_5`
Triggering after the startbit of **data byte 5**

`G_LIN__TRIGGER__MODE__DATA_BYTE_6`
Triggering after the startbit of **data byte 6**

`G_LIN__TRIGGER__MODE__DATA_BYTE_7`
Triggering after the startbit of **data byte 7**

`G_LIN__TRIGGER__MODE__DATA_BYTE_8`
Triggering after the startbit of **data byte 8**

`G_LIN__TRIGGER__MODE__ERROR_INCONSISTENT_SYNC_FIELD`
Triggering after an **inconsistent sync field** (not ID-selective)

`G_LIN__TRIGGER__MODE__ERROR_IDENTIFIER_PARITY`
Triggering after an **ID parity error**

`G_LIN__TRIGGER__MODE__ERROR_BIT`
Triggering after a **bit error** (the received data does not equal the sent data)

`G_LIN__TRIGGER__MODE__ERROR_CHECKSUM`
Triggering after a **checksum error**

`G_LIN__TRIGGER__MODE__ERROR_SLAVE_NOT_RESPONDING`
Triggering after a **no response error**

`G_LIN__TRIGGER__MODE__RECEIVED_FRAME`
Triggering after a **frame is received completely**

`G_LIN__TRIGGER__MODE__RECEIVED_WAKE_UP`
Triggering after receiving a **wake up** (not ID-selective)

`G_LIN__TRIGGER__MODE__RX_TIMEOUT_INTERRUPT`
Triggering after a **receiving timeout interrupt** occurred (not ID-selective)

`G_LIN__TRIGGER__MODE__EXCEEDED_BREAK_DETECTION_THRESHOLD`
Triggering after an **overrun of the break detection threshold** occurred (not ID-selective)

`G_LIN__TRIGGER__MODE__BUS_IDLE_DETECTION`
Triggering after **bus idle detection** (not ID-selective)

`G_LIN__TRIGGER__MODE__BUS_TRAFFIC_DETECTION`
Triggering after **bus traffic detection** (not ID-selective)

`numberOfIds`
Number of identifiers considered as trigger conditions

In case of a **none-ID-selective** `mode`, parameters `numberOfIds` and `ids` are disregarded.

`ids`
Pointer to array with identifiers that are considered as trigger conditions

Each array element represents one identifier

Depending on the data length of a LIN message, the checksum is in byte **length + 1** , i.e., it can be in **data byte 0..8** . The ninth data byte (data byte 8) can only contain the checksum, as a LIN message can include a maximum number of eight data bytes (0..7).

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Trigger_Output_Deactivate

G_Lin_Trigger_Output_Deactivate — Deactivate LIN trigger output

Description

Use this command to deactivate the LIN trigger output.

Definition

```
G_Error_t
G_Lin_Trigger_Output_Deactivate(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Trigger_Output_Property_SetById

G_Lin_Trigger_Output_Property_SetById — Set LIN trigger output properties

Description

Use this command to set LIN trigger output properties.

Definition

```
G_Error_t
G_Lin_Trigger_Output_Property_SetById(
 const G_PortHandle_t portHandle,
 const G_Lin_Trigger_Output_PropertyId_t  id,
 const u32_t  value
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Property ID

The following values are possible:

`G_LIN__TRIGGER__OUTPUT__PROPERTY_ID__UNKNOWN`
The property ID is unknown

`G_LIN__TRIGGER__OUTPUT__PROPERTY_ID__SOFTWARE_OUT__CHANNEL`
Number of the software output that is used as trigger output (starting with **1**)

A trigger unit's source ("digital in", "trigger bus line in",...) can be routed to a trigger unit's target ("digital out", "trigger bus line out",...) with G_Io_Trigger_Source_Set .

`G_LIN__TRIGGER__OUTPUT__PROPERTY_ID__LEVEL`
Level of the trigger output (0 = low, 1 = high)

`value`
Property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Trigger_Output_Property_GetById

G_Lin_Trigger_Output_Property_GetById — Query LIN trigger output properties

Description

Use this command to query LIN trigger output properties.

Definition

```
G_Error_t
G_Lin_Trigger_Output_Property_GetById(
 const G_PortHandle_t portHandle,
 const G_Lin_Trigger_Output_PropertyId_t  id,
 u32_t * const value
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Property ID (see G_Lin_Trigger_Output_PropertyId_t)

`value`
: Returns property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Trigger_Input_Config

G_Lin_Trigger_Input_Config — Configure trigger input

Description

Use this command to configure the actions that are executed when a trigger signal is detected at the trigger input.

Definition

```
G_Error_t
G_Lin_Trigger_Input_Config(
 const G_PortHandle_t portHandle,
 const G_Lin_Trigger_Input_Mode_t  mode,
 const G_Lin_Trigger_Input_InputSelection_t  inputSelection,
 const G_Lin_Trigger_Input_EdgeSelection_t  edgeSelection,
 const G_Lin_Trigger_Input_LevelSelection_t  levelSelection
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
Trigger action

The following values are possible:

`G_LIN__TRIGGER__INPUT__MODE__RESET_MONITOR_TIMESTAMP`
Reset the monitor timestamp

The monitor timestamp is reset in the case of a corresponding flank (selected with **edgeSelection**) at the trigger input.

`G_LIN__TRIGGER__INPUT__MODE__SEND_PREPARED_FRAME_RESPONSES`
Send prepared LIN frame responses

The prepared LIN frame responses are sent in the case of a corresponding level (selected with **levelSelection**) at the trigger input.

`G_LIN__TRIGGER__INPUT__MODE__WRITE_EVENT_INTO_MONITOR`
Write event into monitor buffer

A trigger event is written into the monitor buffer in case of a corresponding flank (selected with **edgeSelection**) at the trigger input.

`inputSelection`
Trigger input selection

The following values are possible:

`G_LIN__TRIGGER__INPUT_SELECTION__FRONT`
Use trigger input of the front connector

`G_LIN__TRIGGER__INPUT_SELECTION__BACK`
Use trigger input of the backplane

`edgeSelection`
Edge selection of the trigger signal

Only relevant for `mode` = `G_LIN__TRIGGER__INPUT__MODE__RESET_MONITOR_TIMESTAMP` and `mode` = `G_LIN__TRIGGER__INPUT__MODE__WRITE_EVENT_INTO_MONITOR`

The following values are possible:

`G_LIN__TRIGGER__INPUT__EDGE_SELECTION__RISING`
Triggering at a **rising edge** at the trigger input

`G_LIN__TRIGGER__INPUT__EDGE_SELECTION__FALLING`
Triggering at a **falling edge** at the trigger input

`G_LIN__TRIGGER__INPUT__EDGE_SELECTION__BOTH`
Triggering at a **rising edge** and a **falling edge** at the trigger input

`levelSelection`
Level selection of the trigger signal

Only relevant for `mode` = `G_LIN__TRIGGER__INPUT__MODE__SEND_PREPARED_FRAME_RESPONSES`

The following values are possible:

`G_LIN__TRIGGER__INPUT__LEVEL_SELECTION__LOW`
Triggering at a **low level** at the trigger input

`G_LIN__TRIGGER__INPUT__LEVEL_SELECTION__HIGH`
Triggering at a **high level** at the trigger input

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Trigger_Input_GetState

G_Lin_Trigger_Input_GetState — Query trigger input state

Description

Use this command to query the state of the trigger input.

Definition

```
G_Error_t
G_Lin_Trigger_Input_GetState(
 const G_PortHandle_t portHandle,
 G_Lin_Trigger_Input_GetState_Level_t * const level,
 u32_t * const eventCount
);
```

Parameters

`portHandle`
Handle to the communication port

`level`
Returns trigger input level

The following values are possible:

`G_LIN__TRIGGER__INPUT__GET_STATE__LEVEL__LOW`
Low level at trigger input

`G_LIN__TRIGGER__INPUT__GET_STATE__LEVEL__HIGH`
High level at trigger input

`eventCount`
Returns event count (counts the activated flanks according to the configuration)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Trigger_Input_Deactivate

G_Lin_Trigger_Input_Deactivate — Deactivate trigger input

Description

Use this command to deactivate the trigger input and to reset its configuration.

Definition

```
G_Error_t
G_Lin_Trigger_Input_Deactivate(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Trigger_Input_Property_SetById

G_Lin_Trigger_Input_Property_SetById — Set LIN trigger input properties

Description

Use this command to set LIN trigger input properties.

Definition

```
G_Error_t
G_Lin_Trigger_Input_Property_SetById(
 const G_PortHandle_t portHandle,
 const G_Lin_Trigger_Input_PropertyId_t  id,
 const u32_t  value
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Property ID

The following values are possible:

`G_LIN__TRIGGER__INPUT__PROPERTY_ID__UNKNOWN`
The property ID is unknown

`G_LIN__TRIGGER__INPUT__PROPERTY_ID__SOFTWARE_IN__CHANNEL`
Number of the software input that is used as trigger input (starting with **1**)

A trigger unit's source ("digital in", "trigger bus line in",...) can be routed to a trigger unit's target ("digital out", "trigger bus line out",...) with G_Io_Trigger_Source_Set .

`value`
Property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_Trigger_Input_Property_GetById

G_Lin_Trigger_Input_Property_GetById — Query LIN trigger input properties

Description

Use this command to query LIN trigger input properties.

Definition

```
G_Error_t
G_Lin_Trigger_Input_Property_GetById(
 const G_PortHandle_t portHandle,
 const G_Lin_Trigger_Input_PropertyId_t  id,
 u32_t * const value
);
```

Parameters

`portHandle`
: Handle to the communication port

`id`
: Property ID (see G_Lin_Trigger_Input_PropertyId_t)

`value`
: Returns property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

15 Sporadic Frames

These functions provide control over the **Sporadic Frames** functionality of your LIN hardware.

A **Sporadic Frame** is an **Unconditional Frame** that is sharing its frame slot with other **Unconditional Frames** .

Sporadic Frames are transmitted by the LIN Master completely. Therefore, collisions on the bus are avoided.

G_Lin_SporadicFrames_Define

G_Lin_SporadicFrames_Define — Define **Sporadic Frame**

Description

Use this command to define a **Sporadic Frame** with the belonging **Unconditional Frames**.

Similar to the **Frames List** for the **Unconditional Frames** of a LIN Description File (**LDF** file), there is possibly a **Sporadic Frames List** for **Sporadic Frames**.

Definition

```
G_Error_t
G_Lin_SporadicFrames_Define(
 const G_PortHandle_t portHandle,
 const u8_t  sporadicFrameListIndex,
 const u16_t  numberOfUnconditionalFrames,
 const u8_t * const unconditionalFrameIds
);
```

Parameters

`portHandle`
Handle to the communication port

`sporadicFrameListIndex`
List index (starting with **0**)

(see also G_Lin_MasterTask_FillScheduleTable)

`numberOfUnconditionalFrames`
Number of **Unconditional Frames** belonging to a **Sporadic Frame**

`unconditionalFrameIds`
Array with IDs of **Unconditional Frames** belonging to a **Sporadic Frame** (0x00..0x3F)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_SporadicFrames_Delete

G_Lin_SporadicFrames_Delete — Delete **Sporadic Frame**

Description

Use this command to delete a **Sporadic Frame** with the belonging **Unconditional Frames .**

Definition

```
G_Error_t
G_Lin_SporadicFrames_Delete(
 const G_PortHandle_t portHandle,
 const u8_t  sporadicFrameListIndex
);
```

Parameters

`portHandle`
Handle to the communication port

`sporadicFrameListIndex`
List index (starting with **0**)

(see also G_Lin_MasterTask_FillScheduleTable)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_SporadicFrames_Activate

G_Lin_SporadicFrames_Activate — Activate the sending of a **Sporadic Frame**

Description

Use this command to activate the sending of a **Sporadic Frame**.

The header with `unconditionalFrameId` is only sent **once**.

The **Unconditional Frame** specified by `unconditionalFrameId` must be defined by G_Lin_MasterTask_FillScheduleTable_WithFrameType or G_Lin_FrameResponses_DefineFrameResponse.

Definition

```
G_Error_t
G_Lin_SporadicFrames_Activate(
 const G_PortHandle_t portHandle,
 const u8_t  sporadicFrameListIndex,
 const u8_t  unconditionalFrameId
);
```

Parameters

`portHandle`
Handle to the communication port

`sporadicFrameListIndex`
List index (starting with **0**)

(see also G_Lin_MasterTask_FillScheduleTable)

`unconditionalFrameId`
Identifier (0x00..0x3F) of the **Unconditional Frame** belonging to this **Sporadic Frame**

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_SporadicFrames_Deactivate

G_Lin_SporadicFrames_Deactivate — Deactivate the sending of a **Sporadic Frame**

Description

Use this command to deactivate the sending of a **Sporadic Frame .**

Definition

```
G_Error_t
G_Lin_SporadicFrames_Deactivate(
 const G_PortHandle_t portHandle,
 const u8_t  sporadicFrameListIndex,
 const u8_t  unconditionalFrameId
);
```

Parameters

`portHandle`
Handle to the communication port

`sporadicFrameListIndex`
List index (starting with **0**)

(see also G_Lin_MasterTask_FillScheduleTable)

`unconditionalFrameId`
Identifier (0x00..0x3F) of the **Unconditional Frame** belonging to this **Sporadic Frame**

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_SporadicFrames_GetState

G_Lin_SporadicFrames_GetState — Query the state of an **Unconditional Frame** within a **Sporadic Frame**

Description

Use this command to query the state of an **Unconditional Frame** within a **Sporadic Frame .**

Definition

```
G_Error_t
G_Lin_SporadicFrames_GetState(
 const G_PortHandle_t portHandle,
 const u8_t  sporadicFrameListIndex,
 const u8_t  unconditionalFrameId,
 G_Lin_SporadicFrames_GetState_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`sporadicFrameListIndex`
List index (starting with **0**)

(see also G_Lin_MasterTask_FillScheduleTable)

`unconditionalFrameId`
Identifier (0x00..0x3F) of the **Unconditional Frame** belonging to this **Sporadic Frame**

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_LIN__SPORADIC_FRAMES__GET_STATE__RSP_FLAG__ZERO`
No flag is set

`G_LIN__SPORADIC_FRAMES__GET_STATE__RSP_FLAG__TX_PENDING`
The transmission is pending

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

16 Event Triggered Frames

These functions provide control over the **Event Triggered Frames** functionality of your LIN hardware.

An **Event Triggered Frame** has the structure of an **Unconditional Frame** with the difference, that one frame header has several dedicated **Frame Responses** from different LIN slaves.

The **Frame Response** that is used to complete the **Event Triggered Frame Header** is selected in dependency on the data the LIN Slave has to transmit. It is indicated within the first byte by the PID of the associated **Unconditional Frame .**

In contrast to **Sporadic Frames** , collisions are possible. If a collison is detected, the LIN Master needs to transmit all **Unconditional Frames** that belong to the **Event Triggered Frame .** This is achieved by using a **Collision Resolving Schedule Table .**

G_Lin_EventTriggeredFrames_Define

G_Lin_EventTriggeredFrames_Define — Define **Event Triggered Frame**

Description

Use this command to define an **Event Triggered Frame** with the belonging **Unconditional Frames** .

With each reception of a **Frame Response** the master checks its content and starts sending the **Unconditional Frames** belonging to this **Event Triggered Frame** in the case at least one of the following conditions is met:

- A transmission error occurred (e.g. wrong checksum).
- The first data byte does not include any of the **Unconditional Frames** identifiers.
- The `G_LIN__EVENT_TRIGGERED_FRAMES__DEFINE__CMD_FLAG__IGNORE_LENGTH` flag is not set, and the data length of the received **Frame Response** does not correspond to the data length configured by `length`.

Definition

```
G_Error_t
G_Lin_EventTriggeredFrames_Define(
 const G_PortHandle_t portHandle,
 const u8_t  eventTriggeredFrameListIndex,
 const G_Lin_EventTriggeredFrames_Define_CmdFlags_t  cmdFlags,
 const u8_t  id,
 const u8_t  length,
 const u16_t  numberOfUnconditionalFrames,
 const u8_t * const unconditionalFrameIds
);
```

Parameters

`portHandle`
Handle to the communication port

`eventTriggeredFrameListIndex`
List index (starting with **0**)

(see also G_Lin_MasterTask_FillScheduleTable)

`cmdFlags`
Command Flags

The following values are possible and can be combined:

`G_LIN__EVENT_TRIGGERED_FRAMES__DEFINE__CMD_FLAG__ZERO`
No flag is set

`G_LIN__EVENT_TRIGGERED_FRAMES__DEFINE__CMD_FLAG__IGNORE_LENGTH`
Ignore the data length

`id`
Identifier of the **Event Triggered Frame** (0x00..0x3F)

`length`
Data length of the event triggered frame (0..8)

`numberOfUnconditionalFrames`
Number of **Unconditional Frames** belonging to an **Event Triggered Frame**

`unconditionalFrameIds`
Array with IDs of **Unconditional Frames** belonging to an **Event Triggered Frame** (0x00..0x3F)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_EventTriggeredFrames_Delete

G_Lin_EventTriggeredFrames_Delete — Delete **Event Triggered Frame**

Description

Use this command to delete an **Event Triggered Frame** with the belonging **Unconditional Frames .**

Definition

```
G_Error_t
G_Lin_EventTriggeredFrames_Delete(
 const G_PortHandle_t portHandle,
 const u8_t  eventTriggeredFrameListIndex
);
```

Parameters

`portHandle`
Handle to the communication port

`eventTriggeredFrameListIndex`
List index (starting with **0**)

(see also G_Lin_MasterTask_FillScheduleTable)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_EventTriggeredFrames_Activate

G_Lin_EventTriggeredFrames_Activate — Activate the sending of an **Event Triggered Frame**

Description

Activates sending in the **Event Triggered Frame Slot** as long as the **Frame Response** specified by `unconditionalFrameId` has been sent successful and fault-free on the bus (see Section 1.3, "Structure of a LIN Frame").

The **Unconditional Frame** specified by `unconditionalFrameId` must be defined by G_Lin_FrameResponses_DefineFrameResponse.

Definition

```
G_Error_t
G_Lin_EventTriggeredFrames_Activate(
 const G_PortHandle_t portHandle,
 const u8_t  eventTriggeredFrameListIndex,
 const u8_t  unconditionalFrameId
);
```

Parameters

`portHandle`
: Handle to the communication port

`eventTriggeredFrameListIndex`
: List index (starting with **0**)

(see also G_Lin_MasterTask_FillScheduleTable)

`unconditionalFrameId`
: Identifier (0x00..0x3F) of the **Unconditional Frame** belonging to this **Event Triggered Frame**

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_EventTriggeredFrames_Deactivate

G_Lin_EventTriggeredFrames_Deactivate — Deactivate the sending of an **Event Triggered Frame**

Description

Use this command to deactivate the sending of an **Event Triggered Frame .**

Definition

```
G_Error_t
G_Lin_EventTriggeredFrames_Deactivate(
 const G_PortHandle_t portHandle,
 const u8_t  eventTriggeredFrameListIndex,
 const u8_t  unconditionalFrameId
);
```

Parameters

`portHandle`
Handle to the communication port

`eventTriggeredFrameListIndex`
List index (starting with **0**)

(see also G_Lin_MasterTask_FillScheduleTable)

`unconditionalFrameId`
Identifier (0x00..0x3F) of the **Unconditional Frame** belonging to this **Event Triggered Frame**

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_EventTriggeredFrames_GetState

G_Lin_EventTriggeredFrames_GetState — Query the state of an **Unconditional Frame** within an **Event Triggered Frame**

Description

Use this command to query the state of an **Unconditional Frame** within an **Event Triggered Frame .**

Definition

```
G_Error_t
G_Lin_EventTriggeredFrames_GetState(
 const G_PortHandle_t portHandle,
 const u8_t  eventTriggeredFrameListIndex,
 const u8_t  unconditionalFrameId,
 G_Lin_EventTriggeredFrames_GetState_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`eventTriggeredFrameListIndex`
List index (starting with **0**)

(see also G_Lin_MasterTask_FillScheduleTable)

`unconditionalFrameId`
Identifier (0x00..0x3F) of the **Unconditional Frame** belonging to this **Event Triggered Frame**

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_LIN__EVENT_TRIGGERED_FRAMES__GET_STATE__RSP_FLAG__ZERO`
No flag is set

`G_LIN__EVENT_TRIGGERED_FRAMES__GET_STATE__RSP_FLAG__TX_PENDING`
The transmission is pending

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lin_EventTriggeredFrames_SetCollisionResolvingTable

G_Lin_EventTriggeredFrames_SetCollisionResolvingTable — Specify a **Collision Resolving Schedule Table**

Description

Use this command to specify a **Collision Resolving Table** for resolving collisions on the bus.

Definition

```
G_Error_t
G_Lin_EventTriggeredFrames_SetCollisionResolvingTable(
 const G_PortHandle_t portHandle,
 const u8_t  eventTriggeredFrameListIndex,
 const u8_t  scheduleTableNumber
);
```

Parameters

`portHandle`
Handle to the communication port

`eventTriggeredFrameListIndex`
List index (starting with **0**)

(see also G_Lin_MasterTask_FillScheduleTable)

`scheduleTableNumber`
Number of the schedule table used to resolve collisions

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

LVDS Functions

These **G-API** functions provide access to the LVDS-Functionality of your Hardware. If the return value of type **G_Error_t** does not correspond to G_NO_ERROR, an error has occurred. A description of this error can be retrieved by calling G_GetLastErrorDescription .

1 Common LVDS commands

These functions provide control over configuration and data communication for LVDS devices.

The VideoDragon basicCON 4121 features the following sideband communication modes depending on the available extension board:

- SPI
- I2C
- UART

The communication with SPI or I2C uses a master and slave and therefore the VideoDragon can be configured for both purposes.

Data communication needs to be enabled with command G_Lvds_Common_Data_Property_SetById and property ID `G_LVDS__COMMON__DATA__PROPERTY_ID__DATA_MODE`.

G_Lvds_Common_Property_GetById

G_Lvds_Common_Property_GetById — Query LVDS property value by ID

Description

Use this command to query LVDS property values.

Definition

```
G_Error_t
G_Lvds_Common_Property_GetById(
 const G_PortHandle_t  portHandle,
 const u16_t  propertyId,  // use 'G_Lvds_Common_PropertyId_t'
 u32_t * const length,
 u8_t * const value
);
```

Parameters

`portHandle`
Handle to the communication port

`propertyId`
ID of the property to be queried

The following values are possible:

`G_LVDS__COMMON__PROPERTY_ID__VHDL_VERSION`
VHDL version of the device

Version info is specified in whole numbers, meaning a value of **202** equals version number **2.02**

size = 4 bytes

This property is read-only.

`G_LVDS__COMMON__PROPERTY_ID__HARDWARE_VERSION`
Hardware version of the device

Version info is specified in whole numbers, meaning a value of **202** equals version number **2.02**

size = 4 bytes

This property is read-only.

`G_LVDS__COMMON__PROPERTY_ID__SERDES`
Serializer / Deserializer type

size = 1 byte

This property is read-only.

The following values are possible:

`G_LVDS__COMMON__SER_DES__MAX9247`
Maxim MAX9247

G_LVDS__COMMON__SER_DES__MAX9209_MAX9213
Maxim MAX9209 or Maxim MAX9213

G_LVDS__COMMON__SER_DES__DS90C241
National Semiconductor DS90C241

G_LVDS__COMMON__SER_DES__APIX_INAP125T24
Apix INAP125T24

G_LVDS__COMMON__SER_DES__APIX_INAP375T
Apix INAP375T

G_LVDS__COMMON__SER_DES__DS90UR905Q
National Semiconductor DS90UB905Q

G_LVDS__COMMON__SER_DES__APIX_INAP375R
Apix INAP375R

G_LVDS__COMMON__SER_DES__DS90UB926Q
National Semiconductor DS90UB926Q

G_LVDS__COMMON__SER_DES__ADV7611
Analog Devices ADV7611

G_LVDS__COMMON__SER_DES__DS90UR906Q
National Semiconductor DS90UR906Q

G_LVDS__COMMON__SER_DES__DS90UB914Q
National Semiconductor DS90UB914Q

G_LVDS__COMMON__SER_DES__MAX9260
Maxim MAX9260

G_LVDS__COMMON__SER_DES__DS90UB925Q
National Semiconductor DS90UB925Q

G_LVDS__COMMON__SER_DES__DS90UB913Q
National Semiconductor DS90UB913Q

G_LVDS__COMMON__SER_DES__MAX9259
Maxim MAX9259

G_LVDS__COMMON__SER_DES__DS90C385A
National Semiconductor DS90C385A

G_LVDS__COMMON__SER_DES__MAX9248
Maxim MAX9248

G_LVDS__COMMON__SER_DES__MAX9275
Maxim MAX9275

G_LVDS__COMMON__SER_DES__MAX9276
Maxim Max9276

G_LVDS__COMMON__SER_DES__MAX9273
Maxim MAx9273

G_LVDS__COMMON__SER_DES__MAX9272
MAxim MAX9272

G_LVDS__COMMON__SER_DES__MAX9271
Maxim MAX9271

G_LVDS__COMMON__SER_DES__DS90UB948Q
National Semiconductor DS90UB948Q

G_LVDS__COMMON__SER_DES__DS90UB947Q
National Semiconductor DS90UB947Q

G_LVDS__COMMON__SER_DES__RGB888R
Goepel electronic RGB 888 Receiver

G_LVDS__COMMON__SER_DES__RGB888 T
Goepel electronic RGB 888 Transmitter

G_LVDS__COMMON__SER_DES__MAX9296A
Maxim MAX9296A

G_LVDS__COMMON__SER_DES__MAX9295A
Maxim MAX9295A

G_LVDS__COMMON__SER_DES__DS90UB954
Texas Instruments DS90UB954

G_LVDS__COMMON__SER_DES__DS90UB953
Texas Instruments DS90UB953

G_LVDS__COMMON__SER_DES__DS90C386
Texas Instruments DS90C386

G_LVDS__COMMON__PROPERTY_ID__LOCK_STATE
Lock state of LVDS signal

0 : no lock

1 : lock

size = 1 byte

This property is read-only.

G_LVDS__COMMON__PROPERTY_ID__CAN_GRAB
Availability of Frame Grabber features

0 : no Frame Grabber features available

1 : Frame Grabber features available

size = 1 byte

This property is read-only.

G_LVDS__COMMON__PROPERTY_ID__CAN_GENERATE
Availability of Frame Generator features

0 : no Frame Generator features available

1 : Frame Generator features available

size = 1 byte

This property is read-only.

G_LVDS__COMMON__PROPERTY_ID__LVDS_CHANNEL
LVDS channel for interfaces supporting multiple LVDS channels

default = 0

size = 1 byte

length
in: size of buffer `value`, in bytes

out: property value size, in bytes

value
Returns property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_Property_SetById

G_Lvds_Common_Property_SetById — Set LVDS property value by ID

Description

Use this command to set LVDS property values.

Definition

```
G_Error_t
G_Lvds_Common_Property_SetById(
 const G_PortHandle_t  portHandle,
 const u16_t  propertyId,  // use 'G_Lvds_Common_PropertyId_t'
 const u32_t  length,
 const u8_t * const value
);
```

Parameters

`portHandle`
Handle to the communication port

`propertyId`
ID of the property to be set

The following values are possible:

`G_LVDS__COMMON__PROPERTY_ID__VHDL_VERSION`
VHDL version of the device

Version info is specified in whole numbers, meaning a value of **202** equals version number **2.02**

size = 4 bytes

This property is read-only.

`G_LVDS__COMMON__PROPERTY_ID__HARDWARE_VERSION`
Hardware version of the device

Version info is specified in whole numbers, meaning a value of **202** equals version number **2.02**

size = 4 bytes

This property is read-only.

`G_LVDS__COMMON__PROPERTY_ID__SERDES`
Serializer / Deserializer type

size = 1 byte

This property is read-only.

The following values are possible:

`G_LVDS__COMMON__SER_DES__MAX9247`
Maxim MAX9247

G_LVDS__COMMON__SER_DES__MAX9209_MAX9213
Maxim MAX9209 or Maxim MAX9213

G_LVDS__COMMON__SER_DES__DS90C241
National Semiconductor DS90C241

G_LVDS__COMMON__SER_DES__APIX_INAP125T24
Apix INAP125T24

G_LVDS__COMMON__SER_DES__APIX_INAP375T
Apix INAP375T

G_LVDS__COMMON__SER_DES__DS90UR905Q
National Semiconductor DS90UB905Q

G_LVDS__COMMON__SER_DES__APIX_INAP375R
Apix INAP375R

G_LVDS__COMMON__SER_DES__DS90UB926Q
National Semiconductor DS90UB926Q

G_LVDS__COMMON__SER_DES__ADV7611
Analog Devices ADV7611

G_LVDS__COMMON__SER_DES__DS90UR906Q
National Semiconductor DS90UR906Q

G_LVDS__COMMON__SER_DES__DS90UB914Q
National Semiconductor DS90UB914Q

G_LVDS__COMMON__SER_DES__MAX9260
Maxim MAX9260

G_LVDS__COMMON__SER_DES__DS90UB925Q
National Semiconductor DS90UB925Q

G_LVDS__COMMON__SER_DES__DS90UB913Q
National Semiconductor DS90UB913Q

G_LVDS__COMMON__SER_DES__MAX9259
Maxim MAX9259

G_LVDS__COMMON__SER_DES__DS90C385A
National Semiconductor DS90C385A

G_LVDS__COMMON__SER_DES__MAX9248
Maxim MAX9248

G_LVDS__COMMON__SER_DES__MAX9275
Maxim MAX9275

G_LVDS__COMMON__SER_DES__MAX9276
Maxim Max9276

G_LVDS__COMMON__SER_DES__MAX9273
Maxim MAx9273

G_LVDS__COMMON__SER_DES__MAX9272
MAxim MAX9272

G_LVDS__COMMON__SER_DES__MAX9271
Maxim MAX9271

G_LVDS__COMMON__SER_DES__DS90UB948Q
National Semiconductor DS90UB948Q

G_LVDS__COMMON__SER_DES__DS90UB947Q
National Semiconductor DS90UB947Q

G_LVDS__COMMON__SER_DES__RGB888R
Goepel electronic RGB 888 Receiver

G_LVDS__COMMON__SER_DES__RGB888 T
Goepel electronic RGB 888 Transmitter

G_LVDS__COMMON__SER_DES__MAX9296A
Maxim MAX9296A

G_LVDS__COMMON__SER_DES__MAX9295A
Maxim MAX9295A

G_LVDS__COMMON__SER_DES__DS90UB954
Texas Instruments DS90UB954

G_LVDS__COMMON__SER_DES__DS90UB953
Texas Instruments DS90UB953

G_LVDS__COMMON__SER_DES__DS90C386
Texas Instruments DS90C386

G_LVDS__COMMON__PROPERTY_ID__LOCK_STATE
Lock state of LVDS signal

0 : no lock

1 : lock

size = 1 byte

This property is read-only.

G_LVDS__COMMON__PROPERTY_ID__CAN_GRAB
Availability of Frame Grabber features

0 : no Frame Grabber features available

1 : Frame Grabber features available

size = 1 byte

This property is read-only.

G_LVDS__COMMON__PROPERTY_ID__CAN_GENERATE
Availability of Frame Generator features

0 : no Frame Generator features available

1 : Frame Generator features available

size = 1 byte

This property is read-only.

G_LVDS__COMMON__PROPERTY_ID__LVDS_CHANNEL
LVDS channel for interfaces supporting multiple LVDS channels

default = 0

size = 1 byte

length
Property size in bytes

value
Property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_DeserializerVector_Config

G_Lvds_Common_DeserializerVector_Config — Configure deserializer vector

Description

Use this command to configure the deserializer vector of your device.

Definition

```
G_Error_t
G_Lvds_Common_DeserializerVector_Config(
 const G_PortHandle_t  portHandle,
 const G_Lvds_Common_DeserializerVector_Config_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__COMMON__DES_VECTOR__CONFIG__CMD_FLAG__NONE`
No flag is set

`G_LVDS__COMMON__DES_VECTOR__CONFIG__CMD_FLAG__NO_RESET`
Do not reset extension board

`NumberOfRegisters`
Number of configuration registers

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Registers`
Array with configuration register parameters

The stucture contains the following elements:

`RegisterAddress`
Register address for this byte

`DeviceNumber`
Device number for this byte, starting with **0**

On some deserializer boards there is more than one device. This parameter is used to specify the device on the board that is to be configured.

`RegisterData`
Configuration data byte for the specified address

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_DeserializerVector_Get

G_Lvds_Common_DeserializerVector_Get — Query deserializer vector values

Description

Use this command to query the current deserializer vector values of your device.

Definition

```
G_Error_t
G_Lvds_Common_DeserializerVector_Get(
 const G_PortHandle_t  portHandle,
 const G_Lvds_Common_DeserializerVector_Get_Cmd_t * const cmd,
 G_Lvds_Common_DeserializerVector_Get_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__COMMON__DES_VECTOR__GET__CMD_FLAG__NONE`
No flag is set

`NumberOfRegisters`
Number of configuration registers to be queried

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Registers`
Array with configuration register parameters to be queried

The stucture contains the following elements:

`RegisterAddress`
Register address for this byte

`DeviceNumber`
Device number for this byte, starting with **0**

On some deserializer boards there is more than one device. This parameter is used to specify the device on the board that is to be configured.

`reserved`
reserved parameter (must be initialized with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_LVDS__COMMON__DES_VECTOR__GET__RSP_FLAG__NONE`
No flag is set

`NumberOfRegisters`
Number of configuration registers that are returned

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Registers`
Array with configuration register parameters

The stucture contains the following elements:

`RegisterAddress`
Register address for this byte

`DeviceNumber`
Device number for this byte, starting with **0**

On some deserializer boards there is more than one device. This parameter is used to specify the device on the board that is to be configured.

`RegisterData`
Configuration data byte for the specified address

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_Data_Property_SetById

G_Lvds_Common_Data_Property_SetById — Set data property value

Description

Use this command to set a data property value.

Definition

```
G_Error_t
G_Lvds_Common_Data_Property_SetById(
 const G_PortHandle_t  portHandle,
 const u16_t  id,  // use "G_Lvds_Common_Data_PropertyId_t"
 const u16_t  length,
 const u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Property ID

The following values are possible:

`G_LVDS__COMMON__DATA__PROPERTY_ID__DATA_MODE`
u8_t

The following values are possible:

`G_LVDS__COMMON__DATA__DATA_MODE__NONE`
no data mode (default)

`G_LVDS__COMMON__DATA__DATA_MODE__I2C_MASTER_TO_DESERIALIZER`
I2C master mode to deserializer

`G_LVDS__COMMON__DATA__DATA_MODE__I2C_MASTER_TO_SERIALIZER`
I2C master mode to serializer

`G_LVDS__COMMON__DATA__DATA_MODE__SPI_MASTER_TO_DESERIALIZER`
SPI master mode to deserializer

`G_LVDS__COMMON__DATA__DATA_MODE__SPI_MASTER_TO_SERIALIZER`
SPI master mode to serializer

`G_LVDS__COMMON__DATA__DATA_MODE__SPI_PASSTHROUGH_DUAL`
Data connection between transmitter and receiver in dual mode

`G_LVDS__COMMON__DATA__DATA_MODE__SPI_RECEIVER_DUAL`
Connection to receiver in dual mode

`G_LVDS__COMMON__DATA__DATA_MODE__UART_TO_DESERIALIZER`
UART to deserializer

`G_LVDS__COMMON__DATA__DATA_MODE__UART_TO_SERIALIZER`
UART to serializer

G_LVDS__COMMON__DATA__DATA_MODE__SPI_TRANSMITTER_DUAL
Connection to transmitter in dual mode

G_LVDS__COMMON__DATA__DATA_MODE__SPI_SLAVE_TO_DESERIALIZER
SPI slave to deserializer

G_LVDS__COMMON__DATA__DATA_MODE__SPI_SLAVE_TO_SERIALIZER
SPI slave to serializer

G_LVDS__COMMON__DATA__DATA_MODE__I2C_SLAVE_TO_DESERIALIZER
I2C slave to deserializer

G_LVDS__COMMON__DATA__DATA_MODE__I2C_SLAVE_TO_SERIALIZER
I2C slave to serializer

G_LVDS__COMMON__DATA__PROPERTY_ID__SPI_MASTER_CPHA
u8_t

SPI clock phase at valid data

0 - data is valid at first clock edge after chip select (default)

1 - data is valid at second clock edge after chip select

G_LVDS__COMMON__DATA__PROPERTY_ID__SPI_MASTER_CPOL
u8_t

SPI clock polarity in idle state

0 - clock polarity in idle state is low (default)

1 - clock polarity in idle state is high

G_LVDS__COMMON__DATA__PROPERTY_ID__SPI_MASTER_CS_MODE
u8_t

SPI chip select behavior between consecutive bytes of the same transfer

0 - chip select remains low (active) between consecutive bytes (default)

1 - chip select switches to high (inactive) between consecutive bytes

G_LVDS__COMMON__DATA__PROPERTY_ID__SPI_MASTER_CLOCK_DIVIDER
u8_t

SPI clock divider - determines SPI clock frequency based on freq. 25MHz

0 - divider is 1 ... clock frequency is 25MHz (default)

1 - divider is 2 ... clock frequency is 12.5MHz

2 - divider is 3 ... clock frequency is 8.33MHz

...

255 - divider is 256 .. clock frequency is 0.1953125MHz

G_LVDS__COMMON__DATA__PROPERTY_ID__SPI_SLAVE_CPHA
u8_t

SPI clock phase at valid data

0 - data is valid at first clock edge after chip select (default)

1 - data is valid at second clock edge after chip select

`G_LVDS__COMMON__DATA__PROPERTY_ID__SPI_SLAVE_CS_IDLE`
u8_t

SPI chip select idle minimum in 20ns steps

0 - 127 * 20ns = 2.54us chip select idle after transfer (default)

1 - 1 * 20ns = 20ns

2 - 2 * 20ns = 40ns

...

255 - 255 * 20ns = 5.1us

`G_LVDS__COMMON__DATA__PROPERTY_ID__I2C_MASTER_BAUDRATE`
u8_t

I2C master baud rate

0 - 100 kbps (default)

1 - 400 kbps

2 - 1000 kbps

3 - 3400 kbps

4 - 5000 kbps

`G_LVDS__COMMON__DATA__PROPERTY_ID__I2C_MASTER_CLOCK_STRETCHING`
u8_t

I2C master clockstretching

The I2C slave can delay the communication by keeping the clock signal low.

0 - disabled

1 - enabled (default)

`G_LVDS__COMMON__DATA__PROPERTY_ID__I2C_MASTER_STOP_NACK`
u8_t

I2C master stop no acknowledge

0 - send ack after last byte from slave

1 - send NO ack after last byte from slave (default)

`G_LVDS__COMMON__DATA__PROPERTY_ID__UART_BAUDRATE`
u32_t

The nearest possible baud rate will be set (value can be read by G_Lvds_FrameGrabber_Data_Property_SetById)

Default value: 115200 baud

Possible values:

- 9600 baud

- 19200 baud
- 38400 baud
- 57600 baud
- 115200 baud
- 230400 baud
- 460800 baud
- 500000 baud
- 1000000 baud

`G_LVDS__COMMON__DATA__PROPERTY_ID__UART_PARITY`
u8_t

0 - no parity (default)

1 - even parity

2 - odd parity

`G_LVDS__COMMON__DATA__PROPERTY_ID__I2C_SLAVE_BAUDRATE`
u8_t

Default value: 100 kHz

Possible values:

- **0** : 100 kHz (default)
- **1** : 400 kHz

`G_LVDS__COMMON__DATA__PROPERTY_ID__PASSTHROUGH_SWITCH`
u8_t

Magic switch for pass through

Bit 0 Enable, Bit 1 Cross

The following values are possible:

`0`
00b - Pass through disabled

`1`
01b - Pass through with I2C enabled

`2`
10b - Pass through disabled, crossing enabled

`3`
11b - Pass trough with UART enabled, crossing enabled

Supported SerDes: MAX9276 v1.1

`length`
Length of property data in bytes

`data`
Property data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_Data_Property_GetById

G_Lvds_Common_Data_Property_GetById — Query data property value

Description

Use this command to query a data property value.

Definition

```
G_Error_t
G_Lvds_Common_Data_Property_GetById(
 const G_PortHandle_t  portHandle,
 const u16_t  id,  // use "G_Lvds_Common_Data_PropertyId_t"
 u16_t * const length,
 u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Property ID

The following values are possible:

`G_LVDS__COMMON__DATA__PROPERTY_ID__DATA_MODE`
u8_t

The following values are possible:

`G_LVDS__COMMON__DATA__DATA_MODE__NONE`
no data mode (default)

`G_LVDS__COMMON__DATA__DATA_MODE__I2C_MASTER_TO_DESERIALIZER`
I2C master mode to deserializer

`G_LVDS__COMMON__DATA__DATA_MODE__I2C_MASTER_TO_SERIALIZER`
I2C master mode to serializer

`G_LVDS__COMMON__DATA__DATA_MODE__SPI_MASTER_TO_DESERIALIZER`
SPI master mode to deserializer

`G_LVDS__COMMON__DATA__DATA_MODE__SPI_MASTER_TO_SERIALIZER`
SPI master mode to serializer

`G_LVDS__COMMON__DATA__DATA_MODE__SPI_PASSTHROUGH_DUAL`
Data connection between transmitter and receiver in dual mode

`G_LVDS__COMMON__DATA__DATA_MODE__SPI_RECEIVER_DUAL`
Connection to receiver in dual mode

`G_LVDS__COMMON__DATA__DATA_MODE__UART_TO_DESERIALIZER`
UART to deserializer

`G_LVDS__COMMON__DATA__DATA_MODE__UART_TO_SERIALIZER`
UART to serializer

G_LVDS__COMMON__DATA__DATA_MODE__SPI_TRANSMITTER_DUAL
Connection to transmitter in dual mode

G_LVDS__COMMON__DATA__DATA_MODE__SPI_SLAVE_TO_DESERIALIZER
SPI slave to deserializer

G_LVDS__COMMON__DATA__DATA_MODE__SPI_SLAVE_TO_SERIALIZER
SPI slave to serializer

G_LVDS__COMMON__DATA__DATA_MODE__I2C_SLAVE_TO_DESERIALIZER
I2C slave to deserializer

G_LVDS__COMMON__DATA__DATA_MODE__I2C_SLAVE_TO_SERIALIZER
I2C slave to serializer

G_LVDS__COMMON__DATA__PROPERTY_ID__SPI_MASTER_CPHA
u8_t

SPI clock phase at valid data

0 - data is valid at first clock edge after chip select (default)

1 - data is valid at second clock edge after chip select

G_LVDS__COMMON__DATA__PROPERTY_ID__SPI_MASTER_CPOL
u8_t

SPI clock polarity in idle state

0 - clock polarity in idle state is low (default)

1 - clock polarity in idle state is high

G_LVDS__COMMON__DATA__PROPERTY_ID__SPI_MASTER_CS_MODE
u8_t

SPI chip select behavior between consecutive bytes of the same transfer

0 - chip select remains low (active) between consecutive bytes (default)

1 - chip select switches to high (inactive) between consecutive bytes

G_LVDS__COMMON__DATA__PROPERTY_ID__SPI_MASTER_CLOCK_DIVIDER
u8_t

SPI clock divider - determines SPI clock frequency based on freq. 25MHz

0 - divider is 1 ... clock frequency is 25MHz (default)

1 - divider is 2 ... clock frequency is 12.5MHz

2 - divider is 3 ... clock frequency is 8.33MHz

...

255 - divider is 256 .. clock frequency is 0.1953125MHz

G_LVDS__COMMON__DATA__PROPERTY_ID__SPI_SLAVE_CPHA
u8_t

SPI clock phase at valid data

0 - data is valid at first clock edge after chip select (default)

1 - data is valid at second clock edge after chip select

`G_LVDS__COMMON__DATA__PROPERTY_ID__SPI_SLAVE_CS_IDLE`
u8_t

SPI chip select idle minimum in 20ns steps

0 - 127 * 20ns = 2.54us chip select idle after transfer (default)

1 - 1 * 20ns = 20ns

2 - 2 * 20ns = 40ns

...

255 - 255 * 20ns = 5.1us

`G_LVDS__COMMON__DATA__PROPERTY_ID__I2C_MASTER_BAUDRATE`
u8_t

I2C master baud rate

0 - 100 kbps (default)

1 - 400 kbps

2 - 1000 kbps

3 - 3400 kbps

4 - 5000 kbps

`G_LVDS__COMMON__DATA__PROPERTY_ID__I2C_MASTER_CLOCK_STRETCHING`
u8_t

I2C master clockstretching

The I2C slave can delay the communication by keeping the clock signal low.

0 - disabled

1 - enabled (default)

`G_LVDS__COMMON__DATA__PROPERTY_ID__I2C_MASTER_STOP_NACK`
u8_t

I2C master stop no acknowledge

0 - send ack after last byte from slave

1 - send NO ack after last byte from slave (default)

`G_LVDS__COMMON__DATA__PROPERTY_ID__UART_BAUDRATE`
u32_t

The nearest possible baud rate will be set (value can be read by G_Lvds_FrameGrabber_Data_Property_SetById)

Default value: 115200 baud

Possible values:

- 9600 baud

- 19200 baud
- 38400 baud
- 57600 baud
- 115200 baud
- 230400 baud
- 460800 baud
- 500000 baud
- 1000000 baud

`G_LVDS__COMMON__DATA__PROPERTY_ID__UART_PARITY`
u8_t

0 - no parity (default)

1 - even parity

2 - odd parity

`G_LVDS__COMMON__DATA__PROPERTY_ID__I2C_SLAVE_BAUDRATE`
u8_t

Default value: 100 kHz

Possible values:

- **0** : 100 kHz (default)
- **1** : 400 kHz

`G_LVDS__COMMON__DATA__PROPERTY_ID__PASSTHROUGH_SWITCH`
u8_t

Magic switch for pass through

Bit 0 Enable, Bit 1 Cross

The following values are possible:

`0`
00b - Pass through disabled

`1`
01b - Pass through with I2C enabled

`2`
10b - Pass through disabled, crossing enabled

`3`
11b - Pass trough with UART enabled, crossing enabled

Supported SerDes: MAX9276 v1.1

`length`
in : size of buffer `data`

out : number of returned bytes in `data`

`data`
Returns property data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_Data_Register_Write

G_Lvds_Common_Data_Register_Write — Write register data

Description

Use this command to write data into a device register.

To use this command, the data mode needs to be enabled (command G_Lvds_Common_Data_Property_SetById , property `G_LVDS__COMMON__DATA__PROPERTY_ID__DATA_MODE`).

Definition

```
G_Error_t
G_Lvds_Common_Data_Register_Write(
 const G_PortHandle_t  portHandle,
 const G_Lvds_Common_Data_Register_Write_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__COMMON__DATA__REG__WR__CMD_FLAG__NONE`
No flag is set

`G_LVDS__COMMON__DATA__REG__WR__CMD_FLAG__REG_ADDR_WIDTH_16_BIT`
Use 16 bit for register address

`G_LVDS__COMMON__DATA__REG__WR__CMD_FLAG__DEV_ADDR_7_BIT`
Use 7 bit for device address

`G_LVDS__COMMON__DATA__REG__WR__CMD_FLAG__EXTENDED_DATA`
Use extended data bytes defined by `Data`

`RegisterValue` is ignored.

`Data` contains the consecutive register values starting from address given by `RegisterAddress`.

`G_LVDS__COMMON__DATA__REG__WR__CMD_FLAG__REGISTER_BLOCK`
0 : do not use register block write

1 : use extended data bytes for register block transfer

Parameters `RegisterAddress`, `DeviceAddress` and `RegisterValue` are ignored.

Data contains the combination of RegisterAddress, DeviceAddress and RegisterValue

NumberOfRegisters contains the complete number of bytes within Data.

RegisterAddress
Address of the register

DeviceAddress
Address of the device

RegisterValue
Value to be written

Disregarded if flag G_LVDS__COMMON__DATA__REG__WR__CMD_FLAG__EXTENDED_DATA is set.

NumberOfRegisters
Number of registers to be written

Only used if flag G_LVDS__COMMON__DATA__REG__WR__CMD_FLAG__EXTENDED_DATA is set.

reserved1
reserved parameter (must be initialized with **0**)

reserved2
reserved parameter (must be initialized with **0**)

reserved3
reserved parameter (must be initialized with **0**)

Data
Extended data bytes (0..255)

Used if flag G_LVDS__COMMON__DATA__REG__WR__CMD_FLAG__EXTENDED_DATA or G_LVDS__COMMON__DATA__REG__WR__CMD_FLAG__REGISTER_BLOCK is set.

If flag G_LVDS__COMMON__DATA__REG__WR__CMD_FLAG__EXTENDED_DATA is set, Data contains NumberOfRegisters consecutive register values starting from address RegisterAddress.

If flag G_LVDS__COMMON__DATA__REG__WR__CMD_FLAG__REGISTER_BLOCK is set, Data contains triplets of RegisterAddress, DeviceAddress and RegisterValue for writing to different registers. If flag G_LVDS__COMMON__DATA__REG__WR__CMD_FLAG__REG_ADDR_WIDTH_16_BIT a triplet consists of 4 bytes, otherwise a triplet consists of 3 bytes. NumberOfRegisters specifies the complete number of bytes used. (see example)

Example

The following example shows how to write a block of registers.

```
// write to registers 0x0A, 0x0C and 0x0E

G_Lvds_Common_Data_Register_Write_Cmd_t  cmd;

cmd.reserved1 = 0;
cmd.reserved2 = 0;
cmd.reserved3 = 0;
```

```
cmd.Flags = G_LVDS__COMMON__DATA__REG__WR__CMD_FLAG__REGISTER_BLOCK;
cmd.NumberOfRegisters = 9;

cmd.Data[0] = 0x0A;  // register address 1
cmd.Data[1] = 0x01;  // device address
cmd.Data[2] = 0x12;  // data 1

cmd.Data[3] = 0x0C;  // register address 2
cmd.Data[4] = 0x01;  // device address
cmd.Data[5] = 0x34;  // data 2

cmd.Data[3] = 0x0E;  // register address 3
cmd.Data[4] = 0x01;  // device address
cmd.Data[5] = 0x56;  // data 3
```

The following example shows how to write a block of registers with 16 bit addressing.

```
// write to registers 0x11AA, 0x22CC and 0x33EE (16 bit addressing)

G_Lvds_Common_Data_Register_Write_Cmd_t  cmd;

cmd.reserved1 = 0;
cmd.reserved2 = 0;
cmd.reserved3 = 0;

cmd.Flags =
 (G_LVDS__COMMON__DATA__REG__WR__CMD_FLAG__REG_ADDR_WIDTH_16_BIT
 | G_LVDS__COMMON__DATA__REG__WR__CMD_FLAG__REGISTER_BLOCK);

cmd.NumberOfRegisters = 12;

cmd.Data[0] = 0xAA;  // register address 1 low byte
cmd.Data[1] = 0x11;  // register address 1 high byte
cmd.Data[2] = 0x01;  // device address
cmd.Data[3] = 0x12;  // data 1

cmd.Data[5] = 0xBB;  // register address 2 low byte
cmd.Data[6] = 0x22;  // register address 2 high byte
cmd.Data[7] = 0x01;  // device address
cmd.Data[8] = 0x34;  // data 2

cmd.Data[9] = 0xCC;  // register address 3 low byte
cmd.Data[10] = 0x33;  // register address 3 high byte
cmd.Data[11] = 0x01;  // device address
cmd.Data[12] = 0x56;  // data 3
```

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_Data_Register_Read

G_Lvds_Common_Data_Register_Read — Read data register value

Description

Use this command to read the value of a data register.

To use this command, the data mode needs to be enabled (command G_Lvds_Common_Data_Property_SetById , property `G_LVDS__COMMON__DATA__PROPERTY_ID__DATA_MODE`).

Definition

```
G_Error_t
G_Lvds_Common_Data_Register_Read(
 const G_PortHandle_t  portHandle,
 const G_Lvds_Common_Data_Register_Read_Cmd_t * const cmd,
 G_Lvds_Common_Data_Register_Read_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__COMMON__DATA__REG__RD__CMD_FLAG__NONE`
No flag is set

`G_LVDS__COMMON__DATA__REG__RD__CMD_FLAG__REG_ADDR_WIDTH_16_BIT`
Use 16 bit for register address

`G_LVDS__COMMON__DATA__REG__RD__CMD_FLAG__DEV_ADDR_7_BIT`
Use 7 bit for device address

`G_LVDS__COMMON__DATA__REG__RD__CMD_FLAG__EXTENDED_DATA`
Read extended data bytes

`RegisterAddress`
Address of the register

`DeviceAddress`
Address of the device

`NumberOfRegisters`
Number of registers to be read

Used if flag `G_LVDS__COMMON__DATA__REG__RD__CMD_FLAG__EXTENDED_DATA` is set.

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_LVDS__COMMON__DATA__REG__RD__RSP_FLAG__NONE`
No flag is set

`RegisterValue`
Returns value of data register

Disregarded if command flag `G_LVDS__COMMON__DATA__REG__RD__CMD_FLAG__EXTENDED_DATA` is set.

`NumberOfRegisters`
Number of read registers

Used if command flag `G_LVDS__COMMON__DATA__REG__RD__CMD_FLAG__EXTENDED_DATA` is set.

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Data`
Returns register data bytes if command flag `G_LVDS__COMMON__DATA__REG__RD__CMD_FLAG__EXTENDED_DATA` is set.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_Data_Transfer_Cleanup

G_Lvds_Common_Data_Transfer_Cleanup — Stop and cleanup any pending data transfer

Description

Use this command to stop and cleanup any pending data transfer.

Definition

```
G_Error_t
G_Lvds_Common_Data_Transfer_Cleanup(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_Data_I2cTransfer

G_Lvds_Common_Data_I2cTransfer — Initiate I2C transfer

Description

Use this command to initiate an I2C transfer.

To use this command, the data mode needs to be enabled (command G_Lvds_Common_Data_Property_SetById , property `G_LVDS__COMMON__DATA__PROPERTY_ID__DATA_MODE`).

Depending on the used extension board, a serializer and/or a deserializer are available and the following data mode can be set:

- `G_LVDS__COMMON__DATA__DATA_MODE__I2C_MASTER_TO_DESERIALIZER`
- `G_LVDS__COMMON__DATA__DATA_MODE__I2C_MASTER_TO_SERIALIZER`

The following table shows the possible data modes depending on the used extension board.

Extension Board	Master to Deserializer	Master to Serializer
DS90UB913	-	to DS90UB913
DS90UB914	to DS90UB914	to DS90UB913
DS90UB925 (v2.0)	-	to DS90UB925
DS90UB926 (v2.0)	to DS90UB926	to DS90UB925
MAX9259	-	to MAX9259
MAX9260	to MAX9260	to MAX9259
MAX9275	-	to MAX9275
MAX9276	to MAX9276	to MAX9275
MAX9271	-	to MAX9271
MAX9272	to MAX9272	to MAX9273
DS90UB947	-	to DS90UB947
DS90UB948 (v1.0)	to DS90UB948	-
DS90UB948 (v2.0)	to DS90UB948	to DS90UB947

Table 1 Extension boards for I2C communication

Definition

```
G_Error_t
G_Lvds_Common_Data_I2cTransfer(
 const G_PortHandle_t  portHandle,
 const G_Lvds_Common_Data_I2cTransfer_Cmd_t * const cmd,
 G_Lvds_Common_Data_I2cTransfer_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__COMMON__DATA__I2C_TRANSFER__CMD_FLAG__NONE`
No flag is set

`NumberOfTxBytes`
Number of bytes to be transmitted (0..8)

`NumberOfRxBytes`
Number of bytes to be received (0..8)

`SendStartMask`
Bitmask that defines the bytes that are preceded by a start symbol

The first byte is always preceded by a start symbol, therefore the first bit of parameter `SendStartMask` must not be set.

`reserved`
reserved parameter (must be initialized with **0**)

`TxData`
Data bytes to be transmitted

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_LVDS__COMMON__DATA__I2C_TRANSFER__RSP_FLAG__NONE`
No flag is set

`AckMask`
Acknowledge mask

Bit mask that specifies the bytes that were aknowledged when transmitted.

`NumberOfRxBytes`
Number of received bytes

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`RxData`
Received data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_Data_I2cTransferEx

G_Lvds_Common_Data_I2cTransferEx — Initiate I2C transfer (extended)

Description

Use this command to initiate an I2C transfer.

To use this command, the data mode needs to be enabled (command G_Lvds_Common_Data_Property_SetById , property `G_LVDS__COMMON__DATA__PROPERTY_ID__DATA_MODE`).

Depending on the used extension board, a serializer and/or a deserializer are available and the following data mode can be set:

- `G_LVDS__COMMON__DATA__DATA_MODE__I2C_MASTER_TO_DESERIALIZER`
- `G_LVDS__COMMON__DATA__DATA_MODE__I2C_MASTER_TO_SERIALIZER`

The following table shows the possible data modes depending on the used extension board.

Extension Board	Master to Deserializer	Master to Serializer
DS90UB913	-	to DS90UB913
DS90UB914	to DS90UB914	to DS90UB913
DS90UB925 (v2.0)	-	to DS90UB925
DS90UB926 (v2.0)	to DS90UB926	to DS90UB925
MAX9259	-	to MAX9259
MAX9260	to MAX9260	to MAX9259
MAX9275	-	to MAX9275
MAX9276	to MAX9276	to MAX9275
MAX9271	-	to MAX9271
MAX9272	to MAX9272	to MAX9273
DS90UB947	-	to DS90UB947
DS90UB948 (v1.0)	to DS90UB948	-
DS90UB948 (v2.0)	to DS90UB948	to DS90UB947

Table 2 Extension boards for I2C communication

Definition

```
G_Error_t
G_Lvds_Common_Data_I2cTransferEx(
 const G_PortHandle_t  portHandle,
 const G_Lvds_Common_Data_I2cTransferEx_Cmd_t * const cmd,
 G_Lvds_Common_Data_I2cTransferEx_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__COMMON__DATA__I2C_TRANSFER_EX__CMD_FLAG__NONE`
No flag is set

`G_LVDS__COMMON__DATA__I2C_TRANSFER_EX__CMD_FLAG__NORMAL_READ_WRITE`
Perform a normal read write operation

In a normal read write operation the **start symbol** is placed automatically at the right position. Therefore parameter `SendStartMask` is disregarded.

`NumberOfTxBytes`
Number of bytes to be transmitted (0..255)

`NumberOfRxBytes`
Number of bytes to be received (0..255)

`SendStartMask`
Bitmask that defines the bytes that are preceded by a start symbol

The first byte is always preceded by a start symbol, therefore the first bit of parameter `SendStartMask` must not be set.

If command flag `G_LVDS__COMMON__DATA__I2C_TRANSFER_EX__CMD_FLAG__NORMAL_READ_WRITE` is set, the start symbol is placed at the right position automatically, therefore this parameter is disregarded.

`reserved`
reserved parameter (must be initialized with **0**)

`TxData`
Data bytes to be transmitted

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_LVDS__COMMON__DATA__I2C_TRANSFER_EX__RSP_FLAG__NONE`
No flag is set

`NumberOfAckBytes`
Number of bytes that were acknowledged

`NumberOfRxBytes`
Number of received bytes

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`RxData`
Received data (0..255)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_Data_SpiTransfer

G_Lvds_Common_Data_SpiTransfer — Initiate SPI transfer

Description

Use this command to initiate an SPI transfer.

To use this command, the data mode needs to be enabled (command G_Lvds_Common_Data_Property_SetById , property `G_LVDS__COMMON__DATA__PROPERTY_ID__DATA_MODE`).

Depending on the used extension board, a serializer and/or a deserializer are available and the following data mode can be set:

- `G_LVDS__COMMON__DATA__DATA_MODE__SPI_MASTER_TO_DESERIALIZER`
- `G_LVDS__COMMON__DATA__DATA_MODE__SPI_MASTER_TO_SERIALIZER`

The following table shows the possible data modes depending on the used extension board.

Extension Board	Master to Deserializer	Master to Serializer
INAP375R (v <3.0)	to INAP375R	-
INAP375R (v3.0)	to INAP375R	to INAP375T
INAP375T	-	to INAP375T
DS90UB948 (<v2.0)	to DS90UB948	to DS90UB947
DS90UB947	-	to DS90UB947

Table 3 Extension boards for SPI communication

When configured as an SPI slave device, this command is used for reading and writing data. Therefore parameter `TxData` specifies the response data for a request from the master and parameter `RxBytes` delivers the received data from SPI master.

Definition

```
G_Error_t
G_Lvds_Common_Data_SpiTransfer(
 const G_PortHandle_t  portHandle,
 const G_Lvds_Common_Data_SpiTransfer_Cmd_t * const cmd,
 G_Lvds_Common_Data_SpiTransfer_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

G_LVDS__COMMON__DATA__SPI_TRANSFER__CMD_FLAG__NONE
No flag is set

NumberOfTxBytes
Number of bytes to be transmitted (0..255)

NumberOfRxBytes
Number of bytes to be received (0..255)

reserved1
reserved parameter (must be initialized with **0**)

reserved2
reserved parameter (must be initialized with **0**)

TxData
Data bytes to be transmitted

rsp
Returns response parameters

The structure contains the following elements:

Flags
Response flags

The following values are possible and can be combined:

G_LVDS__COMMON__DATA__SPI_TRANSFER__RSP_FLAG__NONE
No flag is set

NumberOfRxBytes
Number of received bytes

reserved1
reserved parameter (must be initialized with **0**)

reserved2
reserved parameter (must be initialized with **0**)

reserved3
reserved parameter (must be initialized with **0**)

RxData
Received data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_Data_UartTransfer

G_Lvds_Common_Data_UartTransfer — Initiate UART transfer

Description

Use this command to initiate an UART transfer.

To use this command, the data mode needs to be enabled (command G_Lvds_Common_Data_Property_SetById , property `G_LVDS__COMMON__DATA__PROPERTY_ID__DATA_MODE`).

The following table shows the possible extension board configurations.

Extension Board	UART to Deserializer	UART to Serializer
MAX9259	-	to MAX9259
MAX9260	to MAX9260	to MAX9259
MAX9275	-	to MAX9275
MAX9276	to MAX9276	to MAX9275
MAX9271	-	to MAX9271
MAX9272	to MAX9272	to MAX9273

Table 4 Extension boards for UART communication

Definition

```
G_Error_t
G_Lvds_Common_Data_UartTransfer(
 const G_PortHandle_t  portHandle,
 const G_Lvds_Common_Data_UartTransfer_Cmd_t * const cmd,
 G_Lvds_Common_Data_UartTransfer_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__COMMON__DATA__UART_TRANSFER__CMD_FLAG__NONE`
No flag is set

`G_LVDS__COMMON__DATA__UART_TRANSFER__CMD_FLAG__READ_ALL`
Read all available data

`G_LVDS__COMMON__DATA__UART_TRANSFER__CMD_FLAG__WITH_TIMESTAMP`
Add a timestamp for each received byte

`NumberOfTxBytes`
Number of bytes to be transmitted (0..255)

`NumberOfRxBytes`
Number of bytes to be received (0..255)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`TxData`
Data bytes to be transmitted

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_LVDS__COMMON__DATA__UART_TRANSFER__RSP_FLAG__NONE`
No flag is set

`NumberOfRxBytes`
Number of received bytes

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`RxData`
Received data

If command flag `G_LVDS__COMMON__DATA__UART_TRANSFER__CMD_FLAG__WITH_TIMESTAMP` is set, the following structure for `RxData` is used:

```
struct {
 u64_t  Timestamps[NumberOfRxBytes];
 u8_t  RxBytes[NumberOfRxBytes];
}
```

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_Data_I2cSlaveDevice_Define

G_Lvds_Common_Data_I2cSlaveDevice_Define — Define I2C slave device

Description

Use this function to define a I2C slave device.

In I2C slave mode one or more devices are simulated for responding to request from an I2C master.

Definition

```
G_Error_t
G_Lvds_Common_Data_I2cSlaveDevice_Define(
 const G_PortHandle_t  portHandle,
 const G_Lvds_Common_Data_I2cSlaveDevice_Define_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__COMMON__DATA__I2C_SLAVE_DEV__DEF__CMD_FLAG__NONE`
No flag is set

`G_LVDS__COMMON__DATA__I2C_SLAVE_DEV__DEF__CMD_FLAG__REG_ADD_WIDTH_16BIT`
Register address width is 16 bit

`G_LVDS__COMMON__DATA__I2C_SLAVE_DEV__DEF__CMD_FLAG__DATA_WIDTH_16BIT`
Data address width is 16 bit

`G_LVDS__COMMON__DATA__I2C_SLAVE_DEV__DEF__CMD_FLAG__ACCESS_WO_ADDRESS`
Access without address

`G_LVDS__COMMON__DATA__I2C_SLAVE_DEV__DEF__CMD_FLAG__OVERWRITE_DEVICE`
An already existing device will be overwritten

`G_LVDS__COMMON__DATA__I2C_SLAVE_DEV__DEF__CMD_FLAG__DEVICE_ADDRESS_7BIT`
Device address is 7 bit

`DeviceAddress`
Address of slave device

`NAckByte`
NAckByte

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_Data_I2cSlaveDevice_Delete

G_Lvds_Common_Data_I2cSlaveDevice_Delete — Delete I2C slave device

Description

Use this function to delete a I2C slave device.

Definition

```
G_Error_t
G_Lvds_Common_Data_I2cSlaveDevice_Delete(
 const G_PortHandle_t  portHandle,
 const G_Lvds_Common_Data_I2cSlaveDevice_Delete_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`DeviceAddress`
Address of slave device

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_Data_I2cSlaveRegister_Define

G_Lvds_Common_Data_I2cSlaveRegister_Define — Define I2C slave register

Description

Use this function to define a I2C slave register.

In I2C slave mode one or more devices are simulated for responding to request from an I2C master.

Definition

```
G_Error_t
G_Lvds_Common_Data_I2cSlaveRegister_Define(
 const G_PortHandle_t  portHandle,
 const G_Lvds_Common_Data_I2cSlaveDevice_Register_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__COMMON__DATA__I2C_SLAVE_REG__DEF__CMD_FLAG__NONE`
No flag is set

`G_LVDS__COMMON__DATA__I2C_SLAVE_REG__DEF__CMD_FLAG__OVERWRITE_REGISTER`
An already existing register will be overwritten

`G_LVDS__COMMON__DATA__I2C_SLAVE_REG__DEF__CMD_FLAG__DEVICE_ADDRESS_7BIT`
Device address is 7 bit

`DeviceAddress`
Address of slave device

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`RegisterAddress`
Address of slave register

`RegisterValue`
Value of slave register

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_Data_I2cSlaveRegister_Delete

G_Lvds_Common_Data_I2cSlaveRegister_Delete — Delete I2C slave register

Description

Use this function to delete a I2C slave register.

Definition

```
G_Error_t
G_Lvds_Common_Data_I2cSlaveRegister_Delete(
 const G_PortHandle_t  portHandle,
 const G_Lvds_Common_Data_I2cSlaveRegister_Delete_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`DeviceAddress`
Address of slave device

`reserved`
reserved parameter (must be initialized with **0**)

`RegisterAddress`
Address of slave register

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_VideoSettings_Property_SetById

G_Lvds_Common_VideoSettings_Property_SetById — Change video settings

Description

Use this command to change video settings by property ID.

Definition

```
G_Error_t
G_Lvds_Common_VideoSettings_Property_SetById(
 const G_PortHandle_t  portHandle,
 const u16_t  id,  // use "G_Lvds_Common_VideoSettings_PropertyId_t"
 const u16_t  length,
 const u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Property ID

The following values are possible:

`G_LVDS__COMMON__VIDEO_SETTINGS__PROPERTY_ID__EDID_LOOP_THROUGH`
u8_t

0 - no loop through, EDID data of serdes extension board used (default)

1 - loop through enabled, EDID data of output device is looped to input

(suppported by ADV7611)

`G_LVDS__COMMON__VIDEO_SETTINGS__PROPERTY_ID__BUS_WIDTH_SELECT`
u8_t

0 - 24 bit bus mode (default)

1 - 32 bit bus mode

2 - High bandwidth mode

(supported by MAX9260, MAX9272, MAX9276, MAX9259, MAX9271, MAX9275)

(high bandwidth mode only supported by MAX9276 and MAX9275)

`G_LVDS__COMMON__VIDEO_SETTINGS__PROPERTY_ID__APIX_MODE`
u8_t

0 - APIX2 mode (default)

1 - APIX1 mode

(supported by INAP375R)

`G_LVDS__COMMON__VIDEO_SETTINGS__PROPERTY_ID__TRANSCVR_POWERDOWN`
u8_t

0 - transceiver is powered up (default)

1 - transceiver is powered down

(supported by MAX9260, MAX9272, MAX9276, DS90UB914)

`G_LVDS__COMMON__VIDEO_SETTINGS__PROPERTY_ID__MODE_SELECT`
u8_t

0 - base mode (default)

1 - bypass mode

(supported by MAX9260, MAX9272, MAX9276)

`G_LVDS__COMMON__VIDEO_SETTINGS__PROPERTY_ID__LOW_FREQUENCY_MODE`
u8_t

0 - normal mode (default)

1 - low frequency mode enabled

(supported by DS90UB947, DS90UB948 - low frequency is < 50MHz)

`G_LVDS__COMMON__VIDEO_SETTINGS__PROPERTY_ID__LINK_DATA_MAPPING`
u8_t

0 - mapping mode 0 (default) - SPWG mapping for DS90UB947

1 - mapping mode 1 - OpenLDI for DS90UB947

(supported by DS90UB947)

`length`
Data length in byte

`data`
Property data to be set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_VideoSettings_Property_GetById

G_Lvds_Common_VideoSettings_Property_GetById — Query video settings value

Description

Use this command to query a video settings value by property ID.

Definition

```
G_Error_t
G_Lvds_Common_VideoSettings_Property_GetById(
 const G_PortHandle_t  portHandle,
 const u16_t  id,  // use "G_Lvds_Common_VideoSettings_PropertyId_t"
 u16_t * const length,
 u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Property ID

The following values are possible:

`G_LVDS__COMMON__VIDEO_SETTINGS__PROPERTY_ID__EDID_LOOP_THROUGH`
u8_t

0 - no loop through, EDID data of serdes extension board used (default)

1 - loop through enabled, EDID data of output device is looped to input

(suppported by ADV7611)

`G_LVDS__COMMON__VIDEO_SETTINGS__PROPERTY_ID__BUS_WIDTH_SELECT`
u8_t

0 - 24 bit bus mode (default)

1 - 32 bit bus mode

2 - High bandwidth mode

(supported by MAX9260, MAX9272, MAX9276, MAX9259, MAX9271, MAX9275)

(high bandwidth mode only supported by MAX9276 and MAX9275)

`G_LVDS__COMMON__VIDEO_SETTINGS__PROPERTY_ID__APIX_MODE`
u8_t

0 - APIX2 mode (default)

1 - APIX1 mode

(supported by INAP375R)

`G_LVDS__COMMON__VIDEO_SETTINGS__PROPERTY_ID__TRANSCVR_POWERDOWN`
u8_t

0 - transceiver is powered up (default)

1 - transceiver is powered down

(supported by MAX9260, MAX9272, MAX9276, DS90UB914)

`G_LVDS__COMMON__VIDEO_SETTINGS__PROPERTY_ID__MODE_SELECT`
u8_t

0 - base mode (default)

1 - bypass mode

(supported by MAX9260, MAX9272, MAX9276)

`G_LVDS__COMMON__VIDEO_SETTINGS__PROPERTY_ID__LOW_FREQUENCY_MODE`
u8_t

0 - normal mode (default)

1 - low frequency mode enabled

(supported by DS90UB947, DS90UB948 - low frequency is < 50MHz)

`G_LVDS__COMMON__VIDEO_SETTINGS__PROPERTY_ID__LINK_DATA_MAPPING`
u8_t

0 - mapping mode 0 (default) - SPWG mapping for DS90UB947

1 - mapping mode 1 - OpenLDI for DS90UB947

(supported by DS90UB947)

`length`
in : size of buffer `data`

out : number of returned bytes in `data`

`data`
Returns property data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_VideoSettings_Routing_Set

G_Lvds_Common_VideoSettings_Routing_Set — Set video routing

Description

Use this command to set the video routing values of your device.

Definition

```
G_Error_t
G_Lvds_Common_VideoSettings_Routing_Set(
 const G_PortHandle_t  portHandle,
 const G_Lvds_Common_VideoSettings_Routing_Set_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__COMMON__VIDEO_SETTINGS__ROUTING__SET__CMD_FLAG__NONE`
No flag is set

`VideoBits`
Video bit to pin routing

The index of this array represents the pin number (starting with 0) and the value of each array member represents the video bit parameter that is routed to this pin.

The following values are possible:

`G_LVDS__VIDEO_SOURCE__COLOR_RED_0`
Color bit 'red 0'

`G_LVDS__VIDEO_SOURCE__COLOR_RED_1`
Color bit 'red 1'

`G_LVDS__VIDEO_SOURCE__COLOR_RED_2`
Color bit 'red 2'

`G_LVDS__VIDEO_SOURCE__COLOR_RED_3`
Color bit 'red 3'

`G_LVDS__VIDEO_SOURCE__COLOR_RED_4`
Color bit 'red 4'

`G_LVDS__VIDEO_SOURCE__COLOR_RED_5`
Color bit 'red 5'

G_LVDS__VIDEO_SOURCE__COLOR_RED_6
Color bit 'red 6'

G_LVDS__VIDEO_SOURCE__COLOR_RED_7
Color bit 'red 7'

G_LVDS__VIDEO_SOURCE__COLOR_RED_8
Color bit 'red 8'

G_LVDS__VIDEO_SOURCE__COLOR_RED_9
Color bit 'red 9'

G_LVDS__VIDEO_SOURCE__COLOR_GREEN_0
Color bit 'green 0'

G_LVDS__VIDEO_SOURCE__COLOR_GREEN_1
Color bit 'green 1'

G_LVDS__VIDEO_SOURCE__COLOR_GREEN_2
Color bit 'green 2'

G_LVDS__VIDEO_SOURCE__COLOR_GREEN_3
Color bit 'green 3'

G_LVDS__VIDEO_SOURCE__COLOR_GREEN_4
Color bit 'green 4'

G_LVDS__VIDEO_SOURCE__COLOR_GREEN_5
Color bit 'green 5'

G_LVDS__VIDEO_SOURCE__COLOR_GREEN_6
Color bit 'green 6'

G_LVDS__VIDEO_SOURCE__COLOR_GREEN_7
Color bit 'green 7'

G_LVDS__VIDEO_SOURCE__COLOR_GREEN_8
Color bit 'green 8'

G_LVDS__VIDEO_SOURCE__COLOR_GREEN_9
Color bit 'green 9'

G_LVDS__VIDEO_SOURCE__COLOR_BLUE_0
Color bit 'blue 0'

G_LVDS__VIDEO_SOURCE__COLOR_BLUE_1
Color bit 'blue 1'

G_LVDS__VIDEO_SOURCE__COLOR_BLUE_2
Color bit 'blue 2'

G_LVDS__VIDEO_SOURCE__COLOR_BLUE_3
Color bit 'blue 3'

G_LVDS__VIDEO_SOURCE__COLOR_BLUE_4
Color bit 'blue 4'

G_LVDS__VIDEO_SOURCE__COLOR_BLUE_5
Color bit 'blue 5'

G_LVDS__VIDEO_SOURCE__COLOR_BLUE_6
Color bit 'blue 6'

G_LVDS__VIDEO_SOURCE__COLOR_BLUE_7
Color bit 'blue 7'

G_LVDS__VIDEO_SOURCE__COLOR_BLUE_8
Color bit 'blue 8'

G_LVDS__VIDEO_SOURCE__COLOR_BLUE_9
Color bit 'blue 9'

G_LVDS__VIDEO_SOURCE__CONTROL_HSYNC
Control bit 'horizontal sync'

G_LVDS__VIDEO_SOURCE__CONTROL_VSYNC
Control bit 'vertical sync'

G_LVDS__VIDEO_SOURCE__CONTROL_DATA_ENABLE
Control bit 'data enable'

G_LVDS__VIDEO_SOURCE__NOT_USED
Unused bit

HorizontalSyncPolarity
Horizontal sync polarity

The following values are possible:

G_LVDS__POLARITY__LOW_ACTIVE
The polarity is 'low active'

G_LVDS__POLARITY__HIGH_ACTIVE
The polarity is 'high active'

VerticalSyncPolarity
Vertical sync polarity

The following values are possible:

G_LVDS__POLARITY__LOW_ACTIVE
The polarity is 'low active'

G_LVDS__POLARITY__HIGH_ACTIVE
The polarity is 'high active'

DataEnablePolarity
Data enable polarity

The following values are possible:

G_LVDS__POLARITY__LOW_ACTIVE
The polarity is 'low active'

G_LVDS__POLARITY__HIGH_ACTIVE
The polarity is 'high active'

PixelClockPolarity
Pixel clock edge polarity

The following values are possible:

G_LVDS__EDGE_POLARITY__FALLING_EDGE
The edge polarity is 'falling edge'

G_LVDS__EDGE_POLARITY__RISING_EDGE
The edge polarity is 'rising edge'

G_LVDS__EDGE_POLARITY__UNKNOWN
The edge polarity is unknown

LockOutputEnable
Enable mode of the lock output

The following values are possible:

G_LVDS__ENABLE_MODE__DISABLE
Disabled

G_LVDS__ENABLE_MODE__ENABLE
Enabled

G_LVDS__ENABLE_MODE__UNKNOWN
Unknown

LockPolarity
Lock polarity

The following values are possible:

G_LVDS__POLARITY__LOW_ACTIVE
The polarity is 'low active'

G_LVDS__POLARITY__HIGH_ACTIVE
The polarity is 'high active'

reserved1
reserved parameter (must be initialized with **0**)

reserved2
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_VideoSettings_Routing_Get

G_Lvds_Common_VideoSettings_Routing_Get — Query video routing

Description

Use this command to query the video routing values of your device.

Definition

```
G_Error_t
G_Lvds_Common_VideoSettings_Routing_Get(
 const G_PortHandle_t  portHandle,
 const G_Lvds_Common_VideoSettings_Routing_Get_CmdFlags_t  cmdFlags,
 G_Lvds_Common_VideoSettings_Routing_Get_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_LVDS__COMMON__VIDEO_SETTINGS__ROUTING__GET__CMD_FLAG__NONE`
No flag is set

`rsp`
Returns response parameters

The structure contains the following elements:

`VideoBits`
Video bit to pin routing

The index of this array represents the pin number (starting with 0) and the value of each array member represents the video bit parameter that is routed to this pin.

The following values are possible:

`G_LVDS__VIDEO_SOURCE__COLOR_RED_0`
Color bit 'red 0'

`G_LVDS__VIDEO_SOURCE__COLOR_RED_1`
Color bit 'red 1'

`G_LVDS__VIDEO_SOURCE__COLOR_RED_2`
Color bit 'red 2'

`G_LVDS__VIDEO_SOURCE__COLOR_RED_3`
Color bit 'red 3'

`G_LVDS__VIDEO_SOURCE__COLOR_RED_4`
Color bit 'red 4'

`G_LVDS__VIDEO_SOURCE__COLOR_RED_5`
Color bit 'red 5'

G_LVDS__VIDEO_SOURCE__COLOR_RED_6
Color bit 'red 6'

G_LVDS__VIDEO_SOURCE__COLOR_RED_7
Color bit 'red 7'

G_LVDS__VIDEO_SOURCE__COLOR_RED_8
Color bit 'red 8'

G_LVDS__VIDEO_SOURCE__COLOR_RED_9
Color bit 'red 9'

G_LVDS__VIDEO_SOURCE__COLOR_GREEN_0
Color bit 'green 0'

G_LVDS__VIDEO_SOURCE__COLOR_GREEN_1
Color bit 'green 1'

G_LVDS__VIDEO_SOURCE__COLOR_GREEN_2
Color bit 'green 2'

G_LVDS__VIDEO_SOURCE__COLOR_GREEN_3
Color bit 'green 3'

G_LVDS__VIDEO_SOURCE__COLOR_GREEN_4
Color bit 'green 4'

G_LVDS__VIDEO_SOURCE__COLOR_GREEN_5
Color bit 'green 5'

G_LVDS__VIDEO_SOURCE__COLOR_GREEN_6
Color bit 'green 6'

G_LVDS__VIDEO_SOURCE__COLOR_GREEN_7
Color bit 'green 7'

G_LVDS__VIDEO_SOURCE__COLOR_GREEN_8
Color bit 'green 8'

G_LVDS__VIDEO_SOURCE__COLOR_GREEN_9
Color bit 'green 9'

G_LVDS__VIDEO_SOURCE__COLOR_BLUE_0
Color bit 'blue 0'

G_LVDS__VIDEO_SOURCE__COLOR_BLUE_1
Color bit 'blue 1'

G_LVDS__VIDEO_SOURCE__COLOR_BLUE_2
Color bit 'blue 2'

G_LVDS__VIDEO_SOURCE__COLOR_BLUE_3
Color bit 'blue 3'

G_LVDS__VIDEO_SOURCE__COLOR_BLUE_4
Color bit 'blue 4'

G_LVDS__VIDEO_SOURCE__COLOR_BLUE_5
Color bit 'blue 5'

G_LVDS__VIDEO_SOURCE__COLOR_BLUE_6
Color bit 'blue 6'

G_LVDS__VIDEO_SOURCE__COLOR_BLUE_7
Color bit 'blue 7'

G_LVDS__VIDEO_SOURCE__COLOR_BLUE_8
Color bit 'blue 8'

G_LVDS__VIDEO_SOURCE__COLOR_BLUE_9
Color bit 'blue 9'

G_LVDS__VIDEO_SOURCE__CONTROL_HSYNC
Control bit 'horizontal sync'

G_LVDS__VIDEO_SOURCE__CONTROL_VSYNC
Control bit 'vertical sync'

G_LVDS__VIDEO_SOURCE__CONTROL_DATA_ENABLE
Control bit 'data enable'

G_LVDS__VIDEO_SOURCE__NOT_USED
Unused bit

HorizontalSyncPolarity
Horizontal sync polarity

The following values are possible:

G_LVDS__POLARITY__LOW_ACTIVE
The polarity is 'low active'

G_LVDS__POLARITY__HIGH_ACTIVE
The polarity is 'high active'

VerticalSyncPolarity
Vertical sync polarity

The following values are possible:

G_LVDS__POLARITY__LOW_ACTIVE
The polarity is 'low active'

G_LVDS__POLARITY__HIGH_ACTIVE
The polarity is 'high active'

DataEnablePolarity
Data enable polarity

The following values are possible:

G_LVDS__POLARITY__LOW_ACTIVE
The polarity is 'low active'

G_LVDS__POLARITY__HIGH_ACTIVE
The polarity is 'high active'

PixelClockPolarity
Pixel clock edge polarity

The following values are possible:

G_LVDS__EDGE_POLARITY__FALLING_EDGE
The edge polarity is 'falling edge'

G_LVDS__EDGE_POLARITY__RISING_EDGE
The edge polarity is 'rising edge'

G_LVDS__EDGE_POLARITY__UNKNOWN
The edge polarity is unknown

LockOutputEnable
Enable mode of the lock output

The following values are possible:

G_LVDS__ENABLE_MODE__DISABLE
Disabled

G_LVDS__ENABLE_MODE__ENABLE
Enabled

G_LVDS__ENABLE_MODE__UNKNOWN
Unknown

LockPolarity
Lock polarity

The following values are possible:

G_LVDS__POLARITY__LOW_ACTIVE
The polarity is 'low active'

G_LVDS__POLARITY__HIGH_ACTIVE
The polarity is 'high active'

reserved1
reserved parameter (must be initialized with **0**)

reserved2
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_VideoSettings_ResetExtensionBoard

G_Lvds_Common_VideoSettings_ResetExtensionBoard — Reset extension board

Description

Use this command to reset the extension board of the connected LVDS device.

This command is only available for VideoDragon 4121 devices.

Definition

```
G_Error_t
G_Lvds_Common_VideoSettings_ResetExtensionBoard(
 const G_PortHandle_t  portHandle,
 const G_Lvds_Common_VideoSettings_ResetExtensionBoard_CmdFlags_t  cmdFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_LVDS__COMMON__VIDEO_SETTINGS__RESET_EXT_BOARD__CMD_FLAG__NONE`
No flag is set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_Config_GetSerDesName

G_Lvds_Common_Config_GetSerDesName — Query Serializer / Deserializer name

Description

Use this command to query the name of the Serializer or Deserializer board as string.

Definition

```
const char *
G_Lvds_Common_Config_GetSerDesName(
 const u8_t  serDes
);
```

Parameters

`serDes`
Serializer / Deserializer type

The following values are possible:

`G_LVDS__COMMON__SER_DES__MAX9247`
Maxim MAX9247

`G_LVDS__COMMON__SER_DES__MAX9209_MAX9213`
Maxim MAX9209 or Maxim MAX9213

`G_LVDS__COMMON__SER_DES__DS90C241`
National Semiconductor DS90C241

`G_LVDS__COMMON__SER_DES__APIX_INAP125T24`
Apix INAP125T24

`G_LVDS__COMMON__SER_DES__APIX_INAP375T`
Apix INAP375T

`G_LVDS__COMMON__SER_DES__DS90UR905Q`
National Semiconductor DS90UB905Q

`G_LVDS__COMMON__SER_DES__APIX_INAP375R`
Apix INAP375R

`G_LVDS__COMMON__SER_DES__DS90UB926Q`
National Semiconductor DS90UB926Q

`G_LVDS__COMMON__SER_DES__ADV7611`
Analog Devices ADV7611

`G_LVDS__COMMON__SER_DES__DS90UR906Q`
National Semiconductor DS90UR906Q

`G_LVDS__COMMON__SER_DES__DS90UB914Q`
National Semiconductor DS90UB914Q

`G_LVDS__COMMON__SER_DES__MAX9260`
Maxim MAX9260

`G_LVDS__COMMON__SER_DES__DS90UB925Q`
National Semiconductor DS90UB925Q

G_LVDS__COMMON__SER_DES__DS90UB913Q
National Semiconductor DS90UB913Q

G_LVDS__COMMON__SER_DES__MAX9259
Maxim MAX9259

G_LVDS__COMMON__SER_DES__DS90C385A
National Semiconductor DS90C385A

G_LVDS__COMMON__SER_DES__MAX9248
Maxim MAX9248

G_LVDS__COMMON__SER_DES__MAX9275
Maxim MAX9275

G_LVDS__COMMON__SER_DES__MAX9276
Maxim Max9276

G_LVDS__COMMON__SER_DES__MAX9273
Maxim MAx9273

G_LVDS__COMMON__SER_DES__MAX9272
MAxim MAX9272

G_LVDS__COMMON__SER_DES__MAX9271
Maxim MAX9271

G_LVDS__COMMON__SER_DES__DS90UB948Q
National Semiconductor DS90UB948Q

G_LVDS__COMMON__SER_DES__DS90UB947Q
National Semiconductor DS90UB947Q

G_LVDS__COMMON__SER_DES__RGB888R
Goepel electronic RGB 888 Receiver

G_LVDS__COMMON__SER_DES__RGB888 T
Goepel electronic RGB 888 Transmitter

G_LVDS__COMMON__SER_DES__MAX9296A
Maxim MAX9296A

G_LVDS__COMMON__SER_DES__MAX9295A
Maxim MAX9295A

G_LVDS__COMMON__SER_DES__DS90UB954
Texas Instruments DS90UB954

G_LVDS__COMMON__SER_DES__DS90UB953
Texas Instruments DS90UB953

G_LVDS__COMMON__SER_DES__DS90C386
Texas Instruments DS90C386

Return Value

String with Serializer / Deserializer description

G_Lvds_Common_Config_Get

G_Lvds_Common_Config_Get — Get LVDS configuration data from device

Description

Use this command to get LVDS configuration data from the device and store it in a file.

Definition

```
G_Error_t
G_Lvds_Common_Config_Get(
 const G_PortHandle_t  portHandle,
 const G_Lvds_Common_Config_Get_CmdFlags_t  cmdFlags,
 const char * const configurationFile
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible:

`G_LVDS__COMMON__CONFIG__GET__CMD_FLAG__NONE`
No flag is set

`configurationFile`
Name of configuration file that will be build

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Common_Config_Set

G_Lvds_Common_Config_Set — Set LVDS configuration

Description

Use this command to set the LVDS configuration of a device using a configuration file.

Definition

```
G_Error_t
G_Lvds_Common_Config_Set(
 const G_PortHandle_t  portHandle,
 const G_Lvds_Common_Config_Set_CmdFlags_t  cmdFlags,
 const char * const filename
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_LVDS__COMMON__CONFIG__SET__CMD_FLAG__NONE`
No flag is set

`filename`
Path to configuration file

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

2 Frame Generator

These functions provide control over your **LVDS-FrameGenerator** hardware.

The LVDS-FrameGenerator can be set up to work with different display types by calling function G_Lvds_FrameGenerator_SetDisplayProperties .

For a brief explanation of the display properties, see Figure 1, "LVDS display properties". For a more detailed explanation, please refer to the hardware documentation of the FrameGenerator.

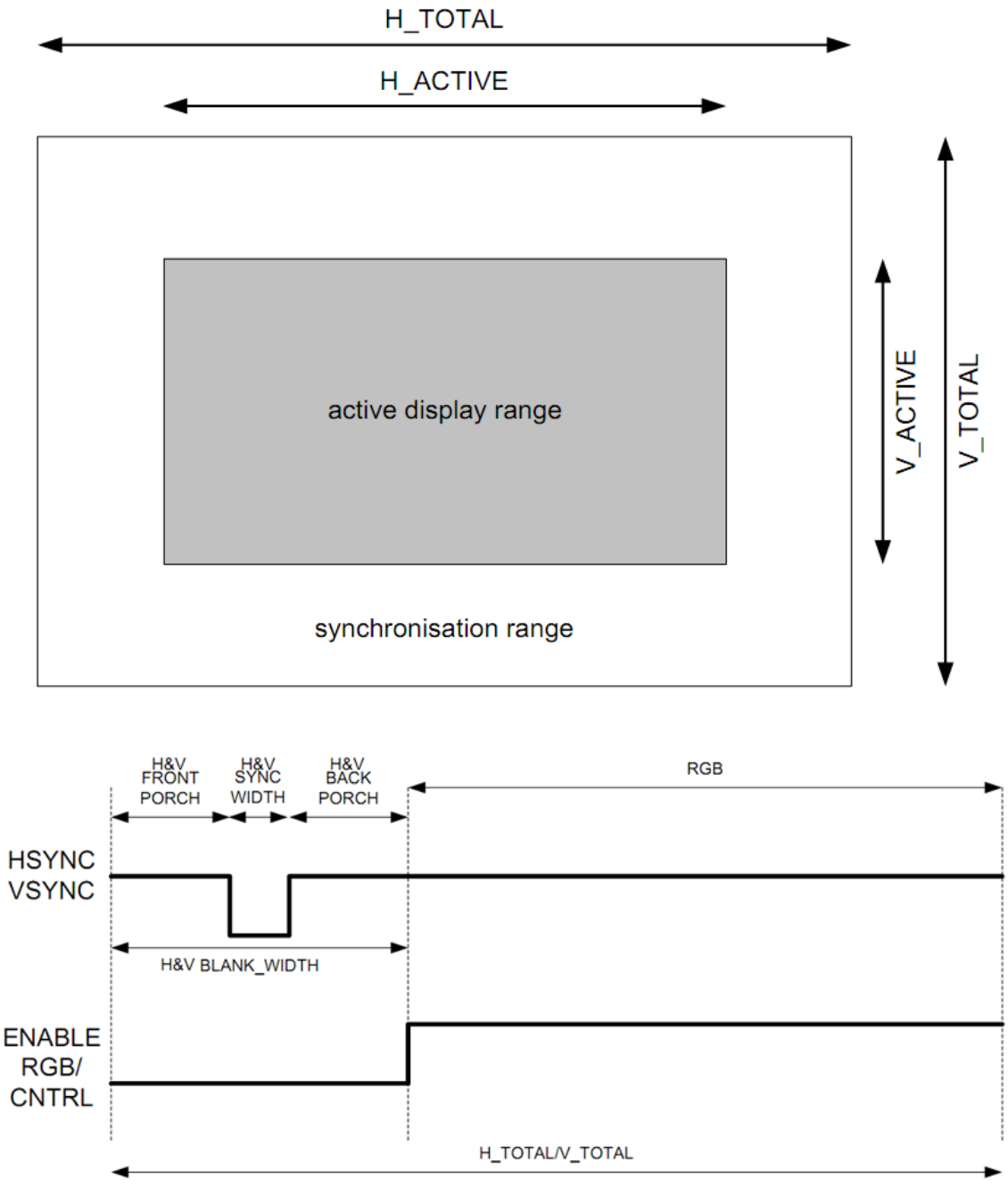

Figure 1 LVDS display properties

G_Lvds_FrameGenerator_DeleteAll

G_Lvds_FrameGenerator_DeleteAll — Delete all files from the internal FrameGenerator memory.

Description

Use this command to delete all files from the internal FrameGenerator memory and reset the internal file system to a clean state. This can be useful if the file system of the FrameGenerator is corrupt because of interrupted write actions.

Depending on the number of files that are stored in the internal FrameGenerator memory, this action may take several minutes.

After execution, all files stored in the internal FrameGenerator memory will be deleted!

Definition

```
G_Error_t
G_Lvds_FrameGenerator_DeleteAll(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGenerator_DeleteFile

G_Lvds_FrameGenerator_DeleteFile — Delete a file from the internal FrameGenerator memory

Description

Use this command to delete a file from the internal memory of the FrameGenerator.

Definition

```
G_Error_t
G_Lvds_FrameGenerator_DeleteFile(
 const G_PortHandle_t  portHandle,
 const u8_t  fileNumber
);
```

Parameters

`portHandle`
: Handle to the communication port

`fileNumber`
: Number of the file to be deleted

  Query a list of all available file numbers with G_Lvds_FrameGenerator_GetMemoryInfo .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGenerator_DisplayColour

G_Lvds_FrameGenerator_DisplayColour — Display a colour

Description

Use this command to display an image consisting of one colour.

Definition

```
G_Error_t
G_Lvds_FrameGenerator_DisplayColour(
 const G_PortHandle_t  portHandle,
 const u32_t  width,
 const u32_t  height,
 const u8_t  r,
 const u8_t  g,
 const u8_t  b
);
```

Parameters

`portHandle`
: Handle to the communication port

`width`
: Width of the image to be displayed (in pixels)

`height`
: Height of the image to be displayed (in pixels)

`r`
: Value for the **red** component of the colour

`g`
: Value for the **green** component of the colour

`b`
: Value for the **blue** component of the colour

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGenerator_DisplayFile

G_Lvds_FrameGenerator_DisplayFile — Display a file

Description

Use this command to display a file of the internal FrameGenerator memory.

Definition

```
G_Error_t
G_Lvds_FrameGenerator_DisplayFile(
 const G_PortHandle_t  portHandle,
 const u8_t  fileNumber
);
```

Parameters

`portHandle`
Handle to the communication port

`fileNumber`
Number of the file to be displayed

Query a list of all available file numbers with G_Lvds_FrameGenerator_GetMemoryInfo .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGenerator_DisplayBuffer

G_Lvds_FrameGenerator_DisplayBuffer — Display pixel data from buffer

Description

Use this command to display pixel data from a buffer.

The image needs to be a 24 bit top down image.

Definition

```
G_Error_t
G_Lvds_FrameGenerator_DisplayBuffer(
 const G_PortHandle_t  portHandle,
 const G_Lvds_FrameGenerator_DisplayBuffer_Cmd_t * const cmd,
 const u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__FRAME_GENERATOR__DISPLAY_BUFFER__CMD_FLAG__NONE`
No flag is set

`Length`
Data length in bytes

`HorizontalOffset`
Horizontal display offset in pixel

`VerticalOffset`
Vertical display offset in pixel

`HorizontalSize`
Horizontal display size in pixel

`VerticalSize`
Vertical display size in pixel

`data`
Pointer to buffer with pixel data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGenerator_GetFileInfo

G_Lvds_FrameGenerator_GetFileInfo — Query file information

Description

Use this command to query information about a specific file of the internal FrameGenerator memory.

Definition

```
G_Error_t
G_Lvds_FrameGenerator_GetFileInfo(
 const G_PortHandle_t  portHandle,
 const u8_t  fileNumber,
 G_Lvds_FrameGenerator_GetFileInfo_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`fileNumber`
Number of the file

Query a list of all available file numbers with G_Lvds_FrameGenerator_GetMemoryInfo .

`rsp`
Returns file information structure

The structure contains the following elements:

`FileType`
File type

The following values are possible:

`G_LVDS__FRAME_GENERATOR__FILE_TYPE__UNKNOWN`
The file type is unknown

`G_LVDS__FRAME_GENERATOR__FILE_TYPE__BMP`
The file is a **Bitmap File**

`reserved`
reserved parameter (must be initialized with **0**)

`BitDepth`
Bit depth of the image, in bits

`FileSize`
Size of the image, in bytes

`Width`
Width of the image, in pixels

`Height`
Height of the image, in pixels

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGenerator_GetHardwareInfo

G_Lvds_FrameGenerator_GetHardwareInfo — Query hardware information

Description

Use this command to query hardware information of the FrameGenerator.

For a brief explanation of the parameters, see Figure 1, "LVDS display properties". For a more detailed explanation, please refer to the hardware documentation of the FrameGenerator.

Definition

```
G_Error_t
G_Lvds_FrameGenerator_GetHardwareInfo(
 const G_PortHandle_t  portHandle,
 G_Lvds_FrameGenerator_GetHardwareInfo_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`rsp`
Returns structure with hardware information

The structure contains the following elements:

`PixelClock`
Pixel clock

The following values are possible:

`G_LVDS__FRAME_GRABBER__PIXEL_CLOCK__UNKNOWN`
The pixel clock is unknown

`G_LVDS__FRAME_GRABBER__PIXEL_CLOCK__33M`
The pixel clock is set to 33 MHz

`G_LVDS__FRAME_GRABBER__PIXEL_CLOCK__14M`
The pixel clock is set to 14 MHz

`G_LVDS__FRAME_GRABBER__PIXEL_CLOCK__20M`
The pixel clock is set to 20 MHz

`H_BlankWidth`
Horizontal blank width (time for control data transmission), in clocks

`H_Total`
Total horizontal width, in clocks

`H_SyncWidth`
Horizontal sync width, in clocks

`H_FrontPorch`
Horizontal front porch width, in clocks

`V_BlankWidth`
Vertical blank width (time for control data transmission), in clocks

`V_Total`
Total vertical width, in clocks

`V_SyncWidth`
Vertical sync width, in clocks

`V_FrontPorch`
Vertical front porch width, in clocks

`FpgaVersion`
VHDL design version

`Serializer`
Serializer board type

The following values are possible:

`G_LVDS__FRAME_GENERATOR__SERIALIZER__UNKNOWN`
The serializer board type is unknown

`G_LVDS__FRAME_GENERATOR__SERIALIZER__MAX9247`
Maxim MAX9247

`G_LVDS__FRAME_GENERATOR__SERIALIZER__MAX9209_MAX9213`
Maxim MAX9209 or Maxim MAX9213

`G_LVDS__FRAME_GENERATOR__SERIALIZER__DS90C241`
National Semiconductor DS90C241

`G_LVDS__FRAME_GENERATOR__SERIALIZER__INAP125T24`
Apix INAP125T24

`G_LVDS__FRAME_GENERATOR__SERIALIZER__APIX_INAP375T`
Apix INAP375T

`G_LVDS__FRAME_GENERATOR__SERIALIZER__DS90UB905Q`
National Semiconductor DS90UB905Q

`G_LVDS__FRAME_GENERATOR__SERIALIZER__DS90C385A`
National Semiconductor DS90C385A

`G_LVDS__FRAME_GENERATOR__SERIALIZER__MAX9275`
Maxim MAX9275

`G_LVDS__FRAME_GENERATOR__SERIALIZER__MAX9273`
Maxim MAX9273

`G_LVDS__FRAME_GENERATOR__SERIALIZER__MAX9271`
Maxim MAX9271

`G_LVDS__FRAME_GENERATOR__SERIALIZER__DS90UB947Q`
Texas Instruments DS90UB947Q

`G_LVDS__FRAME_GENERATOR__SERIALIZER__MAX9259`
Maxim MAX9259

`G_LVDS__FRAME_GENERATOR__SERIALIZER__DS90UB913`
Texas Instruments DS90UB913

`G_LVDS__FRAME_GENERATOR__SERIALIZER__RGB888T`
Goepel electronic RGB 888 T

`G_LVDS__FRAME_GENERATOR__SERIALIZER__MAX9295A`
Maxim MAX9295A

`G_LVDS__FRAME_GENERATOR__SERIALIZER__DS90UB953`
Texas Instruments DS90UB953

`reserved`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGenerator_GetHardwareInfo2

G_Lvds_FrameGenerator_GetHardwareInfo2 — Query hardware information

Description

Use this command to query hardware information of the FrameGenerator.

This is an advanced version of G_Lvds_FrameGenerator_GetHardwareInfo .

Definition

```
G_Error_t
G_Lvds_FrameGenerator_GetHardwareInfo2(
 const G_PortHandle_t  portHandle,
 G_Lvds_FrameGenerator_GetHardwareInfo2_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`rsp`
Returns structure with hardware information

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_LVDS__FRAME_GENERATOR__GET_HW_INFO_2__RSP_FLAG__NONE`
No flag is set

`PixelClockFrequency`
Current pixel clock frequency in Hz

`H_BlankWidth`
Horizontal blank width (time for control data transmission), in clocks

`H_Total`
Total horizontal width, in clocks

`H_SyncWidth`
Horizontal sync width, in clocks

`H_FrontPorch`
Horizontal front porch width, in clocks

`V_BlankWidth`
Vertical blank width (time for control data transmission), in clocks

`V_Total`
Total vertical width, in clocks

`V_SyncWidth`
Vertical sync width, in clocks

`V_FrontPorch`
Vertical front porch width, in clocks

`FpgaVersion`
VHDL design version

`Serializer`
Serializer board type

The following values are possible:

`G_LVDS__FRAME_GENERATOR__SERIALIZER__UNKNOWN`
The serializer board type is unknown

`G_LVDS__FRAME_GENERATOR__SERIALIZER__MAX9247`
Maxim MAX9247

`G_LVDS__FRAME_GENERATOR__SERIALIZER__MAX9209_MAX9213`
Maxim MAX9209 or Maxim MAX9213

`G_LVDS__FRAME_GENERATOR__SERIALIZER__DS90C241`
National Semiconductor DS90C241

`G_LVDS__FRAME_GENERATOR__SERIALIZER__INAP125T24`
Apix INAP125T24

`G_LVDS__FRAME_GENERATOR__SERIALIZER__APIX_INAP375T`
Apix INAP375T

`G_LVDS__FRAME_GENERATOR__SERIALIZER__DS90UB905Q`
National Semiconductor DS90UB905Q

`G_LVDS__FRAME_GENERATOR__SERIALIZER__DS90C385A`
National Semiconductor DS90C385A

`G_LVDS__FRAME_GENERATOR__SERIALIZER__MAX9275`
Maxim MAX9275

`G_LVDS__FRAME_GENERATOR__SERIALIZER__MAX9273`
Maxim MAX9273

`G_LVDS__FRAME_GENERATOR__SERIALIZER__MAX9271`
Maxim MAX9271

`G_LVDS__FRAME_GENERATOR__SERIALIZER__DS90UB947Q`
Texas Instruments DS90UB947Q

`G_LVDS__FRAME_GENERATOR__SERIALIZER__MAX9259`
Maxim MAX9259

`G_LVDS__FRAME_GENERATOR__SERIALIZER__DS90UB913`
Texas Instruments DS90UB913

`G_LVDS__FRAME_GENERATOR__SERIALIZER__RGB888T`
Goepel electronic RGB 888 T

`G_LVDS__FRAME_GENERATOR__SERIALIZER__MAX9295A`
Maxim MAX9295A

`G_LVDS__FRAME_GENERATOR__SERIALIZER__DS90UB953`
Texas Instruments DS90UB953

`reserved`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGenerator_GetPreviewImage

G_Lvds_FrameGenerator_GetPreviewImage — Save a preview image to disk

Description

Use this command to save a preview image of an image, stored in the internal FrameGenerator memory, to disk.

The preview image has a lower resolution than the original image stored in the internal FrameGenerator memory.

Definition

```
G_Error_t
G_Lvds_FrameGenerator_GetPreviewImage(
 const G_PortHandle_t  portHandle,
 const u8_t  fileNumber,
 const char * const path
);
```

Parameters

`portHandle`
: Handle to the communication port

`fileNumber`
: Number of the file

  Query a list of all available file numbers with G_Lvds_FrameGenerator_GetMemoryInfo .

`path`
: Path to the file system location where the preview image should be saved

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGenerator_GetMemoryInfo

G_Lvds_FrameGenerator_GetMemoryInfo — Query memory information

Description

Use this command to query memory information of the FrameGenerator

Definition

```
G_Error_t
G_Lvds_FrameGenerator_GetMemoryInfo(
 const G_PortHandle_t  portHandle,
 G_Lvds_FrameGenerator_GetMemoryInfo_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`rsp`
Returns structure with memory information

The structure contains the following elements:

`Flags`
Memory information flags

The following values are possible and can be combined:

`G_LVDS__FRAME_GENERATOR__GET_MEMORY_INFO__RSP_FLAG__NONE`
No flag is set

`G_LVDS__FRAME_GENERATOR__GET_MEMORY_INFO__RSP_FLAG__FILE_ERROR`
The file system is not consistent and probably some areas of the internal FrameGenerator memory can not be accessed. This can be due to an interrupted write operation.

For fixing an inconsistent file system, call G_Lvds_FrameGenerator_DeleteAll . Attention! All files will be deleted!

`MemorySize`
Size of the internal FrameGenerator memory, in kilobytes

`FreeMemory`
Size of the available internal FrameGenerator memory, in kilobytes

`NumberOfFiles`
Number of files that are stored in the internal FrameGenerator memory

`FileNumbers`
Array with file numbers of all available files of the internal FrameGenerator memory

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGenerator_InterleaveFiles

G_Lvds_FrameGenerator_InterleaveFiles — Diplay two interleaved files

Description

Use this command to display two interleaved files, e.g. for **dual view displays** .

Definition

```
G_Error_t
G_Lvds_FrameGenerator_InterleaveFiles(
 const G_PortHandle_t  portHandle,
 const u8_t  fileNumber1,
 const u8_t  fileNumber2
);
```

Parameters

`portHandle`
Handle to the communication port

`fileNumber1`
Number of the first file (displayed on the left side)

Query a list of all available file numbers with G_Lvds_FrameGenerator_GetMemoryInfo .

`fileNumber2`
Number of the second file (displayed on the right side)

Query a list of all available file numbers with G_Lvds_FrameGenerator_GetMemoryInfo .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGenerator_SaveFile

G_Lvds_FrameGenerator_SaveFile — Save file

Description

Use this command to save a file in the internal FrameGenerator memory.

The files have to be in **24 Bit Bitmap** format.

Definition

```
G_Error_t
G_Lvds_FrameGenerator_SaveFile(
 const G_PortHandle_t  portHandle,
 const u8_t  fileNumber,
 const char * const path
);
```

Parameters

`portHandle`
Handle to the communication port

`fileNumber`
Number of the file to be saved

This number is assigned to the file for identifying the file in the internal FrameGenerator memory. Each file number can only be used by one file.

Query a list of all available file numbers with G_Lvds_FrameGenerator_GetMemoryInfo .

`path`
Path to the file in the host file system

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGenerator_SetDisplayProperties

G_Lvds_FrameGenerator_SetDisplayProperties — Set display properties

Description

Use this command to set the display properties that are stored permanently in the internal memory.

For a brief explanation of the display properties, see Figure 1, "LVDS display properties". For a more detailed explanation, please refer to the hardware documentation of the FrameGenerator.

Definition

```
G_Error_t
G_Lvds_FrameGenerator_SetDisplayProperties(
 const G_PortHandle_t  portHandle,
 const G_Lvds_FrameGenerator_DisplayProperties_t * const displayProperties
);
```

Parameters

`portHandle`
Handle to the communication port

`displayProperties`
Structure with display properties

The structure contains the following elements:

`PixelClock`
See G_Lvds_FrameGenerator_PixelClock_t for details

`H_BlankWidth`
Horizontal blank width (time for control data transmission), in clocks

`H_Total`
Total horizontal width, in clocks

`H_SyncWidth`
Horizontal sync width, in clocks

`H_FrontPorch`
Horizontal front porch width, in clocks

`V_BlankWidth`
Vertical blank width (time for control data transmission), in clocks

`V_Total`
Total vertical width, in clocks

`V_SyncWidth`
Vertical sync width, in clocks

`V_FrontPorch`
Vertical front porch width, in clocks

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGenerator_SetDisplayProperties2

G_Lvds_FrameGenerator_SetDisplayProperties2 — Set display properties

Description

Use this command to set the display properties. This command is an advanced version of G_Lvds_FrameGenerator_SetDisplayProperties .

Definition

```
G_Error_t
G_Lvds_FrameGenerator_SetDisplayProperties2(
 const G_PortHandle_t  portHandle,
 const G_Lvds_FrameGenerator_SetDisplayProperties2_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Structure with command parameters

The structure contains the following elements:

`CmdFlags`
Command flags

The following values are possible and can be combined:

`G_LVDS__FRAME_GENERATOR__S_D_P_2__CMD_FLAG__NONE`
No flag is set

`G_LVDS__FRAME_GENERATOR__S_D_P_2__CMD_FLAG__DO_NOT_SAVE`
The display properties are not stored permanently. After the device has been reset, the display properties that are stored in the flash memory of the device will be recalled.

This flag is only available on USB4115 devices.

`G_LVDS__FRAME_GENERATOR__S_D_P_2__CMD_FLAG__USE_FREQUENCY`
not set : the value of `PixelClock` represents the pixel clock mode

set : the value of `PixelClock` represents the pixel clock frequency in Hz

`G_LVDS__FRAME_GENERATOR__S_D_P_2__CMD_FLAG__KEEP_FRAME_BUFFER`
not set : the currently displayed frame will be reset

set : the currently displayed frame will still be active after reconfiguration

`PixelClock`
When flag `G_LVDS__FRAME_GENERATOR__S_D_P_2__CMD_FLAG__USE_FREQUENCY` is not set: a numeric value representing the pixel clock mode.

When flag `G_LVDS__FRAME_GENERATOR__S_D_P_2__CMD_FLAG__USE_FREQUENCY` is set: the pixel clock frequency in Hz.

`H_BlankWidth`
Horizontal blank width (time for control data transmission), in clocks

`H_Total`
Total horizontal width, in clocks

`H_SyncWidth`
Horizontal sync width, in clocks

`H_FrontPorch`
Horizontal front porch width, in clocks

`V_BlankWidth`
Vertical blank width (time for control data transmission), in clocks

`V_Total`
Total vertical width, in clocks

`V_SyncWidth`
Vertical sync width, in clocks

`V_FrontPorch`
Vertical front porch width, in clocks

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGenerator_Data_Write

G_Lvds_FrameGenerator_Data_Write — Write APIX sideband data

Description

Use this command to write data via the APIX sideband channel.

Definition

```
G_Error_t
G_Lvds_FrameGenerator_Data_Write(
 const G_PortHandle_t  portHandle,
 const G_Lvds_FrameGenerator_Data_Write_CmdFlags_t  cmdFlags,
 const u32_t  numberOfBytes,
 const u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command Flags

The following values are possible and can be combined:

`G_LVDS__FRAME_GENERATOR__DATA__WRITE__CMD_FLAG__NONE`
No flag is set

`numberOfBytes`
Number of bytes to be written

`data`
Data bytes to be written

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGenerator_Data_Read

G_Lvds_FrameGenerator_Data_Read — Read APIX sideband data

Description

Use this command to read data from the APIX sideband channel.

Definition

```
G_Error_t
G_Lvds_FrameGenerator_Data_Read(
 const G_PortHandle_t  portHandle,
 const G_Lvds_FrameGenerator_Data_Read_CmdFlags_t  cmdFlags,
 u32_t * const numberOfBytes,
 u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command Flags

The following values are possible and can be combined:

`G_LVDS__FRAME_GENERATOR__DATA__READ__CMD_FLAG__NONE`
No flag is set

`numberOfBytes`
in : size of buffer `data` in bytes

out : number of returned data bytes

`data`
Buffer for data bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

3 Frame Grabber

These functions provide control over your **LVDS-Frame Grabber** hardware.

G_Lvds_FrameGrabber_GetHardwareInfo

G_Lvds_FrameGrabber_GetHardwareInfo — Get Frame Grabber hardware information

Description

This command provides information about your Frame Grabber hardware.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_GetHardwareInfo(
 const G_PortHandle_t  portHandle,
 u32_t * const vhdlVersion,
 u32_t * const hardwareVersion,
 G_Lvds_FrameGrabber_Deserializer_t * deserializer,
 G_Lvds_LockState_t * const lockState
);
```

Parameters

`portHandle`
Handle to the communication port

`vhdlVersion`
VHDL-design version

`hardwareVersion`
hardware version of Frame Grabber

`deserializer`
Type of deserializer-board

The following values are possible:

`G_LVDS__FRAME_GRABBER__DESERIALIZER__UNKNOWN`
unknown deserializer-board

`G_LVDS__FRAME_GRABBER__DESERIALIZER__DS90UR124`
Texas Instruments DS90UR124

`G_LVDS__FRAME_GRABBER__DESERIALIZER__DS90CF364`
Texas Instruments DS90CF364

`G_LVDS__FRAME_GRABBER__DESERIALIZER__INAP125R24`
Inova Semiconductors APIX INAP125R24 v.1.0

For this deserializer it is necessary to configure the deserializer vector with G_Lvds_FrameGrabber_Config_DeserializerVector .

`G_LVDS__FRAME_GRABBER__DESERIALIZER__MAX9248`
Maxim MAX9248

`G_LVDS__FRAME_GRABBER__DESERIALIZER__DS90UR906`
Texas Instruments DS90UR906

For this deserializer it is necessary to configure the deserializer vector with G_Lvds_FrameGrabber_Config_DeserializerVector .

`G_LVDS__FRAME_GRABBER__DESERIALIZER__INAP125R24_1_1`
Inova Semiconductors APIX INAP125R24 v1.1

`G_LVDS__FRAME_GRABBER__DESERIALIZER__CXB1458R`
Sony GVIF CXB1458R

`G_LVDS__FRAME_GRABBER__DESERIALIZER__MAX9260`
Maxim MAX9260

`G_LVDS__FRAME_GRABBER__DESERIALIZER__DS90UB926`
Texas Instruments DS90UB926

For this deserializer it is necessary to configure the deserializer vector with G_Lvds_FrameGrabber_Config_DeserializerVector .

`G_LVDS__FRAME_GRABBER__DESERIALIZER__APIX_INAP375R`
Inova Semiconductors APIX INAP375R

For this deserializer it is necessary to configure the deserializer vector with G_Lvds_FrameGrabber_Config_DeserializerVector2 .

`G_LVDS__FRAME_GRABBER__DESERIALIZER__ADV7611`
Analog Devices ADV7611

For this deserializer it is necessary to configure the deserializer vector with G_Lvds_FrameGrabber_Config_DeserializerVector2 .

`G_LVDS__FRAME_GRABBER__DESERIALIZER__DS90UB914`
Texas Instruments DS90UB914

For this deserializer it is necessary to configure the deserializer vector with G_Lvds_FrameGrabber_Config_DeserializerVector2 .

`G_LVDS__FRAME_GRABBER__DESERIALIZER__DS90UB925`
Texas Instruments DS90UB925

For this deserializer it is necessary to configure the deserializer vector with G_Lvds_FrameGrabber_Config_DeserializerVector2 .

`G_LVDS__FRAME_GRABBER__DESERIALIZER__DS90UB913`
Texas Instruments DS90UB913

For this deserializer it is necessary to configure the deserializer vector with G_Lvds_FrameGrabber_Config_DeserializerVector2 .

`G_LVDS__FRAME_GRABBER__DESERIALIZER__MAX9259`
Maxim MAX9259

For this deserializer it is necessary to configure the deserializer vector with G_Lvds_FrameGrabber_Config_DeserializerVector2 .

For a description of the deserializer boards, refer to the Frame Grabber hardware manual.

`lockState`
indicator for a valid LVDS signal at the input

The following values are possible:
- `G_LVDS__FRAME_GRABBER__LOCK_STATE__NO_LOCK`

no valid LVDS signal present

- `G_LVDS__FRAME_GRABBER__LOCK_STATE__LOCK`

  a valid LVDS signal is present

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_Init

G_Lvds_FrameGrabber_Init — Initialize Frame Grabber

Description

This command initializes the Frame Grabber.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_Init(
 const G_PortHandle_t  portHandle,
 const G_Lvds_FrameGrabber_Init_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
structure with initialization parameters

The structure contains the following elements:

- `Resolution`

  structure with resolution information

  The structure contains the following elements:
  - `Width`

    width of a frame in pixels
  - `Height`

    height of a frame in pixels

- `BackPorch`

  structure with synchonization information

  The structure contains the following elements:
  - `Horizontal`

    number of columns for horizontal back porch
  - `Vertical`

    number of lines for vertical back porch

- `SignalLevels`

  structure with signal level information

  For each element, the following values are possible:
  - `G_LVDS__SIGNAL_LEVEL__LOW_ACTIVE`

    low-active signal

- `G_LVDS__SIGNAL_LEVEL__HIGH_ACTIVE`

  high-active signal

The structure contains the following elements:
- `HSync`

  signal level for horizontal synchronization signal
- `VSync`

  signal level for vertical synchronization signal
- `DataEnable`

  signal level for data enable signal
- reserved

- `EdgeSelection`

  structure with edge selection information

  For each element, the following values are possible:
  - `G_LVDS__EDGE_SELECTION__FALLING_EDGE`

    falling edge is used
  - `G_LVDS__EDGE_SELECTION__RISING_EDGE`

    rising edge is used

  The structure contains the following elements:
  - `Clock`

    edge selection information of pixel clock
  - reserved1
  - reserved2
  - reserved3

- `Routing`

  structure with routing information

  With this structure, the colour-bits and control-signals are mapped to the corresponding hardware pins of the deserializer.

  The possible values for each element are **0..31** representing hardware pins **0..31**, or **0xFF** to **disable** the element.

  The structure contains the following elements:

  - `Colours`

    structure with assignment information for bits representing the colours of a frame

    The structure contains the following elements:
    - **B0..B7**

      pin numbers for bits representing the colour **blue**
    - **G0..G7**

      pin numbers for bits representing the colour **green**
    - **R0..R7**

      pin numbers for bits representing the colour **red**

- `Control`

  structure with assignment information for control data

  The structure contains the following elements:
  - `VSync`

    pin number for **vertical synchronization** signal
  - `HSync`

    pin number for **horizontal synchronization** signal
  - `DataEnable`

    pin number for **data enable** signal
  - **C3..C7**

    pin numbers for **control signals 3..7**

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Initializing the Frame Grabber

```
G_Lvds_FrameGrabber_Init_Parameters_t  parameters;
parameters.Routing.Colours.B0 = 0xFF;
parameters.Routing.Colours.B1 = 0xFF;
parameters.Routing.Colours.B2 = 15;
parameters.Routing.Colours.B3 = 16;
parameters.Routing.Colours.B4 = 17;
parameters.Routing.Colours.B5 = 18;
parameters.Routing.Colours.B6 = 19;
parameters.Routing.Colours.B7 = 20;
parameters.Routing.Colours.G0 = 0xFF;
parameters.Routing.Colours.G1 = 0xFF;
parameters.Routing.Colours.G2 = 8;
parameters.Routing.Colours.G3 = 9;
parameters.Routing.Colours.G4 = 10;
parameters.Routing.Colours.G5 = 11;
parameters.Routing.Colours.G6 = 12;
parameters.Routing.Colours.G7 = 13;
parameters.Routing.Colours.R0 = 0xFF;
parameters.Routing.Colours.R1 = 0xFF;
parameters.Routing.Colours.R2 = 1;
parameters.Routing.Colours.R3 = 2;
parameters.Routing.Colours.R4 = 3;
parameters.Routing.Colours.R5 = 4;
parameters.Routing.Colours.R6 = 5;
parameters.Routing.Colours.R7 = 6;
parameters.Routing.Control.DataEnable = 21;
parameters.Routing.Control.HSync = 22;
parameters.Routing.Control.VSync = 23;
parameters.Routing.Control.C3 = 0xFF;
parameters.Routing.Control.C4 = 0xFF;
parameters.Routing.Control.C5 = 0xFF;
parameters.Routing.Control.C6 = 0xFF;
```

```
parameters.Routing.Control.C7 = 0xFF;
parameters.Resolution.Width = 800;
parameters.Resolution.Height = 480;
parameters.SyncWidth.Horizontal = 0;
parameters.SyncWidth.Vertical = 0;
parameters.EdgeSelection.Clock = G_LVDS__EDGE_SELECTION__RISING_EDGE;
parameters.SignalLevels.DataEnable = G_LVDS__SIGNAL_LEVEL__HIGH_ACTIVE;
parameters.SignalLevels.HSync = G_LVDS__SIGNAL_LEVEL__LOW_ACTIVE;
parameters.SignalLevels.VSync = G_LVDS__SIGNAL_LEVEL__LOW_ACTIVE;
return
  G_Lvds_FrameGrabber_Init(
  PortHandle1,
  &parameters
);
```

G_Lvds_FrameGrabber_GetInitInfo

G_Lvds_FrameGrabber_GetInitInfo — Get initialization information

Description

This command provides information about the Frame Grabber initialization.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_GetInitInfo(
 const G_PortHandle_t  portHandle,
 G_Lvds_FrameGrabber_Init_Parameters_t * const rsp
);
```

Parameters

`portHandle`
: Handle to the communication port

`rsp`
: pointer to response structure (see `parameters` for more information)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_GetSyncInfo

G_Lvds_FrameGrabber_GetSyncInfo — Get synchronization information

Description

This command scans the synchronization signals and returns the gathered information.

In case a parameter is not needed, the pointer to this parameter can be `NULL`.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_GetSyncInfo(
 const G_PortHandle_t  portHandle,
 u32_t * const pixelClock,
 u16_t * const dataEnableWidth,
 u16_t * const syncWidthH,
 u16_t * const syncWidthV,
 u16_t * const syncPeriodH,
 u32_t * const syncPeriodV
);
```

Parameters

`portHandle`
Handle to the communication port

`pixelClock`
Frequency of pixel clock in kHz

`dataEnableWidth`
Number of pixel clock cycles with **data enable** active

`syncWidthH`
Number of pixel clock cycles with **horizontal synchronization** active

`syncWidthV`
Number of pixel clock cycles with **vertical synchronization** active

`syncPeriodH`
Number of pixel clock cycles between two **horizontal synchronization** edges

`syncPeriodV`
Number of pixel clock cycles between two **vertical synchronization** edges

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_GetSyncInfo2

G_Lvds_FrameGrabber_GetSyncInfo2 — Get extended synchronization information

Description

This command scans the synchronization signals and returns the gathered information.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_GetSyncInfo2(
 const G_PortHandle_t  portHandle,
 G_Lvds_FrameGrabber_SyncInfo2_t * const syncInfo2
);
```

Parameters

`portHandle`
Handle to the communication port

`syncInfo2`
Returns extended synchronization information

The structure contains the following elements:

`PixelClock`
Frequency of pixel clock in kHz

`DataEnableWidthH`
Number of horizontal pixel clock cycles with **data enable** active

`DataEnableWidthV`
Number of vertical pixel clock cycles with **data enable** active

`SyncWidthH`
Number of pixel clock cycles with **horizontal synchronization** active

`SyncWidthV`
Number of pixel clock cycles with **vertical synchronization** active

`SyncPeriodH`
Number of pixel clock cycles between two **horizontal synchronization** edges

`SyncPeriodV`
Number of pixel clock cycles between two **vertical synchronization** edges

`reserved1`
reserved

`reserved2`
reserved

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Start

G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Start — Enable permanent scan of sync values

Description

Use this command to enable the permanent scan of sync values.

Sync values can then be read with G_Lvds_FrameGrabber_SyncInfo_GetById .

To stop permanent scanning of sync values, use G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Stop .

Definition

```
G_Error_t
G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Start(
 const G_PortHandle_t  portHandle,
 const G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Start_CmdFlags_t
 cmdFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following commands are possible and can be combined:

`G_LVDS__FRAME_GRABBER__SYNC_INFO__PERMA_SCAN__START__CMD_FLAG__NONE`
No flag is set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Stop

G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Stop — Disable permanent scan of sync values

Description

Use this command to disable the permanent scan of sync values.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Stop(
 const G_PortHandle_t  portHandle,
 const G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Stop_CmdFlags_t  cmdFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following commands are possible and can be combined:

`G_LVDS__FRAME_GRABBER__SYNC_INFO__PERMA_SCAN__STOP__CMD_FLAG__NONE`
No flag is set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_SyncInfo_GetById

G_Lvds_FrameGrabber_SyncInfo_GetById — Query sync values

Description

Use this command to query sync values.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_SyncInfo_GetById(
 const G_PortHandle_t  portHandle,
 const G_Lvds_FrameGrabber_SyncInfo_GetById_CmdFlags_t  cmdFlags,
 const u16_t  id,  // use 'G_Lvds_FrameGrabber_SyncInfo_Id_t'
 u16_t * const length,
 u8_t * const value
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following commands are possible and can be combined:

`G_LVDS__FRAME_GRABBER__SYNC_INFO__GET_BY_ID__CMD_FLAG__NONE`
No flag is set

`id`
ID of sync info value to be read

The following values are possible:

`G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__PIXEL_CLOCK_FREQ_DIVIDER`
Pixel clock frequency divider

u16_t

0xFFFF invalid - clock too low or not present

`G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__VERTICAL_SYNC_PULSE_WIDTH`
Vertical sync pulse width

u16_t

0xFFF invalid - vertical sync too long

`G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__HORIZONTAL_SYNC_PULSE_WIDTH`
Horizontal sync pulse width

u16_t

0xFFF invalid - horizontal sync too long

`G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__VERTICAL_SYNC_PERIODE`
Vertical sync period

u32_t

0xFFFFFF invalid - vertical sync period too long

`G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__HORIZONTAL_SYNC_PERIODE`
Horizontal sync period

u16_t

0x1FFF invalid - horizontal sync period too long

`G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__DATA_ENABLE_PULSE_COUNT`
Data enable pulse count (vertical resolution)

u16_t

0xFFF invalid - vertical resolution too big

`G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__DATA_ENABLE_PULSE_WIDTH`
Data enable pulse width (horizontal resolution)

u16_t

0xFFF invalid - horizontal resolution too big

`G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__SCAN_COUNT`
The count of perfomed sync scans in permanent scan mode

u32_t

Permanent scan of sync values needs to be enabled with
G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Start

`G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__PIXEL_CLOCK_FREQ_DIV_MINIMUM`
Pixel clock frequency divider minimum value

100.000 / divider = frequency in MHz

u16_t

Permanent scan of sync values needs to be enabled with
G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Start

`G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__PIXEL_CLOCK_FREQ_DIV_MAXIMUM`
Pixel clock frequency divider maximum value

100.000 / divider = frequency in MHz

u16_t

Permanent scan of sync values needs to be enabled with
G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Start

`G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__V_SYNC_PULSE_WIDTH_MINIMUM`
Vertical sync pulse width minimum

u16_t

Permanent scan of sync values needs to be enabled with
G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Start

G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__V_SYNC_PULSE_WIDTH_MAXIMUM
Vertical sync pulse width maximum

u16_t

Permanent scan of sync values needs to be enabled with G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Start

G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__H_SYNC_PULSE_WIDTH_MINIMUM
Horizontal sync pulse width minimum

u16_t

Permanent scan of sync values needs to be enabled with G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Start

G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__H_SYNC_PULSE_WIDTH_MAXIMUM
Horizontal sync pulse width maximum

u16_t

Permanent scan of sync values needs to be enabled with G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Start

G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__V_SYNC_PERIODE_MINIMUM
Vertical sync period minimum

u32_t

Permanent scan of sync values needs to be enabled with G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Start

G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__V_SYNC_PERIODE_MAXIMUM
Vertical sync period maximum

u32_t

Permanent scan of sync values needs to be enabled with G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Start

G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__H_SYNC_PERIODE_MINIMUM
Horizontal sync period minimum

u32_t

Permanent scan of sync values needs to be enabled with G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Start

G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__H_SYNC_PERIODE_MAXIMUM
Horizontal sync period maximum

u32_t

Permanent scan of sync values needs to be enabled with G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Start

G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__DATA_ENABLE_COUNT_MINIMUM
Data enable count minimum (vertical resolution)

u16_t

Permanent scan of sync values needs to be enabled with G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Start

`G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__DATA_ENABLE_COUNT_MAXIMUM`
Data enable count maximum (vertical resolution)

u16_t

Permanent scan of sync values needs to be enabled with G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Start

`G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__DATA_ENABLE_WIDTH_MINIMUM`
Data enable width minimum (horizontal resolution)

u16_t

Permanent scan of sync values needs to be enabled with G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Start

`G_LVDS__FRAME_GRABBER__SYNC_INFO__ID__DATA_ENABLE_WIDTH_MAXIMUM`
Data enable width maximum (horizontal resolution)

u16_t

Permanent scan of sync values needs to be enabled with G_Lvds_FrameGrabber_SyncInfo_PermanentScan_Start

length
in : size of buffer `value` in bytes

out : size of returned value in bytes

value
Returns value for selected ID

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_StartCapturing

G_Lvds_FrameGrabber_StartCapturing — Start capture operation

Description

Use this command to start the capture operation.

This command needs to be called before any capture operation can be done.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_StartCapturing(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_StopCapturing

G_Lvds_FrameGrabber_StopCapturing — Stop capture operation

Description

Use this command to stop the capture operation.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_StopCapturing(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_Capture

G_Lvds_FrameGrabber_Capture — Capture a frame

Description

This command initiates the Frame Grabber to capture a frame.

The captured frame is stored in the internal memory and available for compare operations.

If the capture operation fails with `G_ERROR__DLL__LVDS__FRAME_ERROR`, it may be necessary to re-initialize the Frame Grabber with G_Lvds_FrameGrabber_Init, depending on the deserializer board that is used.

Definition

```
G_Error_t
 G_Lvds_FrameGrabber_Capture(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_CaptureToBuffer

G_Lvds_FrameGrabber_CaptureToBuffer — Capture a frame

Description

Use this command to capture image data to a buffer.

USB4120:

The captured frame is stored in the internal buffer and additionally transferred into the provided buffer as raw **32 bit pixel data** like in a "top down" bitmap file, starting with the top row.

basicCON4121:

The captured frame is stored in the internal buffer and additionally transferred into the provided buffer as raw **24 bit pixel data** like in a "top down" bitmap file, starting with the top row.

If the capture operation fails with `G_ERROR__DLL__LVDS__FRAME_ERROR`, it may be necessary to re-initialize the Frame Grabber with G_Lvds_FrameGrabber_Init, depending on the deserializer board that is used.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_CaptureToBuffer(
 const G_PortHandle_t  portHandle,
 u8_t * const data,
 u32_t * const length
);
```

Parameters

`portHandle`
Handle to the communication port

`data`
pointer to buffer

`length`
at function call: size of buffer in bytes

at completion: size of returned data in bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_CaptureToBuffer2

G_Lvds_FrameGrabber_CaptureToBuffer2 — Capture frame to buffer with extended parameters

Description

Use this command to capture image data to a buffer.

This command provides extended parameters over command G_Lvds_FrameGrabber_CaptureToBuffer.

The captured frame is stored in the internal buffer and additionally transferred into the provided buffer as raw **24 bit pixel data** like in a "top down" bitmap file, starting with the top row.

If the capture operation fails with `G_ERROR__DLL__LVDS__FRAME_ERROR`, it may be necessary to re-initialize the Frame Grabber with G_Lvds_FrameGrabber_Init, depending on the deserializer board that is used.

This command is not supported on basicCON4121.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_CaptureToBuffer2(
 const G_PortHandle_t  portHandle,
 const G_Lvds_FrameGrabber_CaptureToBuffer2_Cmd_t * const cmd,
 u64_t * const timestamp,
 u8_t * const data,
 u32_t * const length
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__FRAME_GRABBER__CAPTURE_TO_BUFFER_2__CMD_FLAG__NONE`
No flag is set

`G_LVDS__FRAME_GRABBER__CAPTURE_TO_BUFFER_2__CMD_FLAG__USE_TIMESTAMP`
Use timestamp option

If set, a timestamp for every frame will be provided in parameter `timestamp`.

`timestamp`
Returns timestamp for frame data, in nanoseconds

Only used if flag `G_LVDS__FRAME_GRABBER__CAPTURE_TO_BUFFER_2__CMD_FLAG__USE_TIMESTAMP` is set.

The current timestamp timer value of the device can be queried with G_Io_Trigger_Timestamp_Get .

`data`
Returns frame data

`length`
at function call: size of buffer in bytes

at completion: size of returned data in bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_CaptureToFile

G_Lvds_FrameGrabber_CaptureToFile — Capture a frame

Description

Use this command to capture image data to a bitmap file.

USB4120:

The captured frame is stored as a **32 bit** "top down" bitmap file.

basicCON4121:

The captured frame is stored as a **24 bit** "top down" bitmap file.

If another filetype is needed, use G_Lvds_FrameGrabber_CaptureToFile2!

If the capture operation fails with `G_ERROR__DLL__LVDS__FRAME_ERROR`, it may be necessary to re-initialize the Frame Grabber with G_Lvds_FrameGrabber_Init, depending on the deserializer board that is used.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_CaptureToFile(
 const G_PortHandle_t  portHandle,
 const u16_t  width,
 const u16_t  height,
 const char * const path
);
```

Parameters

`portHandle`
: Handle to the communication port

`width`
: width of the bitmap in pixels

  Has to be the same value as the width of the current capture area.

`height`
: height of bitmap in pixels

  Has to be the same value as the height of the current capture area.

`path`
: path for storing the bitmap

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Capturing a frame to the file "test.bmp"

```
return
  G_Lvds_FrameGrabber_CaptureToFile(
    PortHandle1,
    800,
    480,
    "test.bmp"
  );
```

G_Lvds_FrameGrabber_CaptureToFile2

G_Lvds_FrameGrabber_CaptureToFile2 — Capture a frame with extended parameters

Description

Use this command to capture image data to a file.

In contrast to G_Lvds_FrameGrabber_CaptureToFile this functions offers extended parameters.

If the capture operation fails with G_ERROR__DLL__LVDS__FRAME_ERROR, it may be necessary to re-initialize the Frame Grabber with G_Lvds_FrameGrabber_Init, depending on the deserializer board that is used.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_CaptureToFile2(
 const G_PortHandle_t  portHandle,
 const u8_t  mode,  // use "G_Lvds_FrameGrabber_CaptureToFile_Mode_t"
 const u16_t  width,
 const u16_t  height,
 const char * const path
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
Determines the mode of the file

The following values are possible:

`G_LVDS__FRAME_GRABBER__CAPTURE_TO_FILE__MODE__BMP__TOP_DOWN`
A bitmap file with top down row order will be saved.

The resulting file is the same as when G_Lvds_FrameGrabber_CaptureToFile is called.

USB4120:

The captured frame is stored as a **32 bit** bitmap file.

basicCON4121:

The captured frame is stored as a **24 bit** bitmap file.

`G_LVDS__FRAME_GRABBER__CAPTURE_TO_FILE__MODE__BMP__BOTTOM_UP`
A bitmap file with bottom up row order will be saved.

This mode is not as fast as the `G_LVDS__FRAME_GRABBER__CAPTURE_TO_FILE__MODE_ _BMP__TOP_DOWN` mode since the pixel data is reordered. Use this mode when your application does not support top down bitmaps.

USB4120:

The captured frame is stored as a **32 bit** bitmap file.

basicCON4121:

The captured frame is stored as a **24 bit** bitmap file.

`width`
width of the file in pixels

`height`
height of bitmap in pixels

`path`
path for storing the bitmap

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_LoadReferenceFromBuffer

G_Lvds_FrameGrabber_LoadReferenceFromBuffer — Load reference frame

Description

This command loads a reference frame into the internal memory of the Frame Grabber.

USB4120:

The source for the reference frame is a buffer with raw **32 bit pixel data** , as captured by command G_Lvds_FrameGrabber_CaptureToBuffer

basicCON4121:

The source for the reference frame is a buffer with raw **24 bit pixel data** , as captured by command G_Lvds_FrameGrabber_CaptureToBuffer

Definition

```
G_Error_t
G_Lvds_FrameGrabber_LoadReferenceFromBuffer(
 const G_PortHandle_t  portHandle,
 const u8_t * const data,
 const u32_t  length
);
```

Parameters

`portHandle`
: Handle to the communication port

`data`
: pointer to buffer with pixel data

`length`
: size of pixel data in bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_LoadReferenceFromFile

G_Lvds_FrameGrabber_LoadReferenceFromFile — Load reference frame

Description

This command loads a reference frame into the internal memory of the Frame Grabber.

USB4120:

The source for the reference frame is **32 bit** bitmap file.

basicCON4121:

The source for the reference frame is **24 bit** bitmap file.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_LoadReferenceFromFile(
 const G_PortHandle_t  portHandle,
 const char * const path
);
```

Parameters

`portHandle`
Handle to the communication port

`path`
path to reference bitmap file

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_ReadReferenceToBuffer

G_Lvds_FrameGrabber_ReadReferenceToBuffer — Read reference frame into buffer

Description

This command reads the reference frame from the internal memory of the Frame Grabber and stores it in a buffer as raw 32 bit pixel data like in a "top down" bitmap file, starting with the top row.

This command is not supported on basicCON4121.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_ReadReferenceToBuffer(
 const G_PortHandle_t  portHandle,
 u8_t * const data,
 u32_t * const length
);
```

Parameters

`portHandle`
: Handle to the communication port

`data`
: pointer to buffer

`length`
: at function call: size of buffer in bytes

: at completion: size of returned data in bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_ReadReferenceToFile

G_Lvds_FrameGrabber_ReadReferenceToFile — Read reference frame into bitmap file

Description

This command reads the reference frame from the internal memory of the Frame Grabber and stores it under the provided path as a 32 bit "top down" bitmap file.

This command is not supported on basicCON4121.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_ReadReferenceToFile(
 const G_PortHandle_t  portHandle,
 const u16_t  width,
 const u16_t  height,
 const char * const path
);
```

Parameters

`portHandle`
: Handle to the communication port

`width`
: width of bitmap in pixels

`height`
: height of bitmap in pixels

`path`
: path for storing the bitmap

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Storing the reference frame as "test.bmp"

```
return
 G_Lvds_FrameGrabber_ReadReferenceToFile(
  PortHandle1,
  800,
  480,
  "test.bmp"
 );
```

G_Lvds_FrameGrabber_ReadCapturedFrameToBuffer

G_Lvds_FrameGrabber_ReadCapturedFrameToBuffer — Read captured frame into buffer

Description

This command reads the captured frame from the internal memory of the Frame Grabber and stores it in a buffer.

USB4120:

The captured frame is provided as raw **32 bit pixel data** like in a "top down" bitmap file, starting with the top row.

basicCON4121:

The captured frame is provided as raw **24 bit pixel data** like in a "top down" bitmap file, starting with the top row.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_ReadCapturedFrameToBuffer(
 const G_PortHandle_t  portHandle,
 u8_t * const data,
 u32_t * const length
);
```

Parameters

`portHandle`
: Handle to the communication port

`data`
: pointer to buffer

`length`
: at function call: size of buffer in bytes

: at completion: size of returned data in bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_ReadCapturedFrameToFile

G_Lvds_FrameGrabber_ReadCapturedFrameToFile — Read captured frame into bitmap file

Description

This command reads the captured frame from the internal memory of the Frame Grabber and stores it under the provided path as a bitmap file.

USB4120:

The captured frame is stored as a **32 bit** "top down" bitmap file.

basicCON4121:

The captured frame is stored as a **24 bit** "top down" bitmap file.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_ReadReferenceToFile(
 const G_PortHandle_t  portHandle,
 const u16_t  width,
 const u16_t  height,
 const char * const path
);
```

Parameters

`portHandle`
: Handle to the communication port

`width`
: width of bitmap in pixels

`height`
: height of bitmap in pixels

`path`
: path for storing the bitmap

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_Compare

G_Lvds_FrameGrabber_Compare — Compare captured frame with reference frame

Description

This command compares the captured frame with the reference frame in the internal Frame Grabber memory.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_Compare(
 const G_PortHandle_t  portHandle,
 u32_t * const errorCount
);
```

Parameters

`portHandle`
: Handle to the communication port

`errorCount`
: number of mismatching pixels

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_SetBitMask

G_Lvds_FrameGrabber_SetBitMask — Set bit mask for compare operations

Description

This command sets the bit mask for compare operations.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_SetBitMask(
 const G_PortHandle_t  portHandle,
 const u32_t  bitMask
);
```

Parameters

`portHandle`
Handle to the communication port

`bitMask`
Bit mask for comparison

bits set to **0** will be disregarded.

bits	31..27	26	25	24	23..16	15..8	7..0
Content	Control (7..3)	DataEn-able	HSync	VSync	Red (7..0)	Green (7..0)	Blue (7..0)

The default value after initialization is **0x00FFFFFF** .

On **VideoDragon** devices (4121) bits **24-31** are always **disregarded** .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_ExternalTriggerMode_Start

G_Lvds_FrameGrabber_ExternalTriggerMode_Start — Start external trigger mode

Description

This command starts the external trigger mode.

In the external trigger mode, an external trigger impulse at the DIO input will trigger the Frame Grabber to capture a frame and to compare it with the reference frame in the internal memory.

If the external trigger mode is started, capture - and frame - commands sent via the USB interface are deactivated.

The advantages of the external trigger mode are trigger operations in real-time, and a host-free operation. For more details refer to the hardware manual of the Frame Grabber.

By default the following routing is applied:

Start Capture Operation: Digitial Input 1

Stop Capture Operation: Digitial Input 2

This can be changed by using in the io interface using G_Io_Trigger_Source_Set

Definition

```
G_Error_t
G_Lvds_FrameGrabber_ExternalTriggerMode_Start(
 const G_PortHandle_t  portHandle,
 const G_Lvds_SignalLevel_t  triggerLevel
);
```

Parameters

`portHandle`
Handle to the communication port

`triggerLevel`
trigger level for trigger input (see G_Lvds_SignalLevel_t for a list with possible values)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_ExternalTriggerMode_GetState

G_Lvds_FrameGrabber_ExternalTriggerMode_GetState — Get external trigger state

Description

This command provides information about the external trigger state.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_ExternalTriggerMode_GetState(
 const G_PortHandle_t  portHandle,
 G_Lvds_ExtTriggerMode_GetState_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`rspFlags`
response flags

The following values are possible and can be combined:

- `G_LVDS__EXT_TRIGGER_MODE__GET_STATE__RSP_FLAG__NONE`

  no response flag set
- `G_LVDS__EXT_TRIGGER_MODE__GET_STATE__RSP_FLAG__READY`

  ready for external trigger

  (external trigger mode started, reference frame loaded)
- `G_LVDS__EXT_TRIGGER_MODE__GET_STATE__RSP_FLAG__TRIGGERED`

  triggered a compare operation
- `G_LVDS__EXT_TRIGGER_MODE__GET_STATE__RSP_FLAG__COMPARE_FAILED`

  compare operation failed

  (frames do not match)
- `G_LVDS__EXT_TRIGGER_MODE__GET_STATE__RSP_FLAG__EXT_POWER`

  external power supply present
- `G_LVDS__EXT_TRIGGER_MODE__GET_STATE__RSP_FLAG__TRIGGER_INPUT_HIGH`

  external trigger input is high
- `G_LVDS__EXT_TRIGGER_MODE__GET_STATE__RSP_FLAG__INPUT_1_HIGH`

  external input 1 is high
- `G_LVDS__EXT_TRIGGER_MODE__GET_STATE__RSP_FLAG__INPUT_2_HIGH`

  external input 2 is high

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_ExternalTriggerMode_Stop

G_Lvds_FrameGrabber_ExternalTriggerMode_Stop — Stop external trigger mode

Description

This command stops the external trigger mode.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_ExternalTriggerMode_Stop(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_SetClockSource

G_Lvds_FrameGrabber_SetClockSource — Set source for pixel clock

Description

This command sets the source for the pixel clock.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_SetClockSource(
 const G_PortHandle_t  portHandle,
 const G_Lvds_FrameGrabber_SetClockSource_Mode_t  mode
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
pixel clock mode

The following values are possible:

- `G_LVDS__FRAME_GRABBER__SET_CLOCK_SOURCE__MODE__USE_STD_CLOCK`

  use standard pixel clock signal at pin **0**
- `G_LVDS__FRAME_GRABBER__SET_CLOCK_SOURCE__MODE__USE_PIN_19`

  use clock signal at pin **19**

The default value after initialization is

`G_LVDS__FRAME_GRABBER__SET_CLOCK_SOURCE__MODE__USE_STD_CLOCK`

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_Config_DeserializerVector

G_Lvds_FrameGrabber_Config_DeserializerVector — Configure deserializer vector

Description

This command configures the vector of the following deserializers:

`G_LVDS__FRAME_GRABBER__DESERIALIZER__INAP125R24`
Configuration addresses **0x01..0x04**

`G_LVDS__FRAME_GRABBER__DESERIALIZER__DS90UR906`
Configuration addresses **0x0..0x03**

`G_LVDS__FRAME_GRABBER__DESERIALIZER__DS90UB926`
Configuration addresses **0x0..0x03**

A call to G_Lvds_FrameGrabber_Init will reset the deserializer vector. Therefore it is necessary to configure the deserializer vector **after** G_Lvds_FrameGrabber_Init has been called.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_Config_DeserializerVector(
 const G_PortHandle_t  portHandle,
 const u8_t  byte1,
 const u8_t  byte2,
 const u8_t  byte3,
 const u8_t  byte4
);
```

Parameters

`portHandle`
Handle to the communication port

`byte1`
Byte value for 1st address

`byte2`
Byte value for 2nd address

`byte3`
Byte value for 3rd address

`byte4`
Byte value for 4th address

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

The following example shows a possible configuration for deserializer **INAP125R24** .

```
byte1 = 0xC5;  // disable dedicated upstream, remaining values are default
```

```
byte2 = 0x7F;  // pixel data width = 24, remaining values are default
byte3 = 0x28;  // reserved = 2, remaining values are default
byte4 = 0x0C;  // default value
```

Refer to the data sheet of your deserializer for more information.

G_Lvds_FrameGrabber_Config_DeserializerVector2

G_Lvds_FrameGrabber_Config_DeserializerVector2 — Configure deserializer vector (advanced)

Description

Use this command to configure the deserializer vector of the deserializer. In contrast to G_Lvds_FrameGrabber_Config_DeserializerVector , this command provides more advanced parameters that are needed for some of the deserializer types.

This command is not compatible with **USB4120** devices.

This command is deprecated, use G_Lvds_Common_DeserializerVector_Config instead!

Definition

```
G_Error_t
G_Lvds_FrameGrabber_Config_DeserializerVector2(
 const G_PortHandle_t  portHandle,
 const G_Lvds_FrameGrabber_Config_DeserializerVector2_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__FRAME_GRABBER__CONFIG__DES_VECTOR_2__CMD_FLAG__NONE`
No flag is set

`NumberOfRegisters`
Number of configuration registers

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Registers`
Array with configuration register parameters

The stucture contains the following elements:

`RegisterAddress`
Register address for this byte

`DeviceNumber`
Device number for this byte, starting with **0**

On some deserializer boards there is more than one device. This parameter is used to specify the device on the board that is to be configured.

`RegisterData`
Configuration data byte for the specified address

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_SetColourTolerance

G_Lvds_FrameGrabber_SetColourTolerance — Set colour tolerance

Description

Use this command to set the colour tolerance values for compare operations.

In contrast to G_Lvds_FrameGrabber_SetBitMask more precise comparisons can be performed because the values are not depending on bit positions.

You can use **either** G_Lvds_FrameGrabber_SetColourTolerance **or** G_Lvds_FrameGrabber_SetBitMask for specifying compare criteriaframe type.

This command is not compatible with **USB4120** devices.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_SetColourTolerance(
 const G_PortHandle_t  portHandle,
 const u8_t  red,
 const u8_t  green,
 const u8_t  blue
);
```

Parameters

`portHandle`
Handle to the communication port

`red`
Colour tolerance value for red colour

`green`
Colour tolerance value for green colour

`blue`
Colour tolerance value for blue colour

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_SetCompareArea

G_Lvds_FrameGrabber_SetCompareArea — Set compare area

Description

Use this command to specify the area of a frame that is used for the compare operation.

By default the whole frame area is used for the compare operation.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_SetCompareArea(
 const G_PortHandle_t  portHandle,
 const G_Lvds_FrameGrabber_SetCompareArea_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`VerticalStartPosition`
Vertical start position for the compare operation, in pixels (starting with 0)

`HorizontalStartPosition`
Horizontal start position for the compare operation, in pixels (starting with 0)

`VerticalEndPosition`
Vertical end position for the compare operation, in pixels (starting with 0)

`HorizontalEndPosition`
Horizontal end position for the compare operation, in pixels (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_SetCaptureArea

G_Lvds_FrameGrabber_SetCaptureArea — Set capture area

Description

Use this command to decrease the capture area of the LVDS Frame Grabber.

By default, the whole pixel area is captured.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_SetCaptureArea(
 const G_PortHandle_t  portHandle,
 const G_Lvds_FrameGrabber_SetCaptureArea_CmdFlags_t  cmdFlags,
 const u32_t  verticalStartPosition,
 const u32_t  horizontalStartPosition,
 const u32_t  verticalEndPosition,
 const u32_t  horizontalEndPosition
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_LVDS__FRAME_GRABBER__SET_CAPTURE_AREA__CMD_FLAG__NONE`
No flag is set

`verticalStartPosition`
Starting line of capture area (0..4093)

`horizontalStartPosition`
Starting column of capture area (0..4093)

`verticalEndPosition`
Ending line of capture area (0..4093)

`horizontalEndPosition`
Ending column of capture area (0..4093)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_SetGrabberMode

G_Lvds_FrameGrabber_SetGrabberMode — Set Frame Grabber mode

Description

Use this command to switch between Frame Grabber modes.

The default mode is `G_LVDS__FRAME_GRABBER__GRABBER_MODE__AUTO_EMPTY` which in most cases delivers the best performace.

If for some reason this mode leads to errors when grabbing picture data, use mode `G_LVDS__FRAME_GRABBER__GRABBER_MODE__POLLING` which is more stable in some cases.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_SetGrabberMode(
 const G_PortHandle_t  portHandle,
 const u8_t  mode
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
Grabber mode

The following values are possible:

`G_LVDS__FRAME_GRABBER__GRABBER_MODE__POLLING`
Polling mode

This is the save mode with reduced performance which delivers more stable results in some cases.

`G_LVDS__FRAME_GRABBER__GRABBER_MODE__AUTO_EMPTY`
Auto empty mode

This is the default mode which usually delivers the best performance.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_Data_Property_SetById

G_Lvds_FrameGrabber_Data_Property_SetById — Set data property value

Description

Use this command to set a data property value.

This command is deprecated, use G_Lvds_Common_Data_Property_SetById instead!

Definition

```
G_Error_t
G_Lvds_FrameGrabber_Data_Property_SetById(
 const G_PortHandle_t  portHandle,
 const u16_t  id,  // use "G_Lvds_FrameGrabber_Data_PropertyId_t"
 const u16_t  length,
 const u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Property ID

The following values are possible:

`G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__DATA_MODE`
u8_t

0 - no data mode (default)

1 - I2C master mode to deserializer

2 - I2C master mode to serializer

The data mode needs to be enabled (value **1** or **2**) for I2C operation!

`G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__SPI_MASTER_CPHA`
u8_t

SPI clock phase at valid data

0 - data is valid at first clock edge after chip select (default)

1 - data is valid at second clock edge after chip select

`G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__SPI_MASTER_CPOL`
u8_t

SPI clock polarity in idle state

0 - clock polarity in idle state is low (default)

1 - clock polarity in idle state is high

`G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__SPI_MASTER_CS_MODE`
u8_t

SPI chip select behavior between consecutive bytes of the same transfer

0 - chip select remains low (active) between consecutive bytes (default)

1 - chip select switches to high (inactive) between consecutive bytes

`G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__SPI_MASTER_CLOCK_DIVIDER`
u8_t

SPI clock divider - determines SPI clock frequency based on freq. 25 MHz

0 - divider is 1 ... clock frequency is 25 MHz (default)

1 - divider is 2 ... clock frequency is 12.5 MHz

2 - divider is 3 ... clock frequency is 8.33 MHz

...

255 - divider is 256 ... clock frequency is 0.1953125 MHz

`G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__SPI_SLAVE_CPHA`
u8_t

SPI clock phase at valid data

0 - data is valid at first clock edge after chip select (default)

1 - data is valid at second clock edge after chip select

`G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__SPI_SLAVE_CS_IDLE`
u8_t

SPI chip select idle minimum in 20ns steps

0 - 127 * 20ns = 2.54us chip select idle after transfer (default)

1 - 1 * 20ns = 20ns

2 - 2 * 20ns = 40ns

...

255 - 255 * 20ns = 5.1us

`G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__I2C_MASTER_BAUDRATE`
u8_t

I2C master baud rate

0 - 100kHz (default)

1 - 400kHz

`G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__I2C_MASTER_CLOCK_STRETCHING`
u8_t

I2C master clockstretching

The I2C slave can delay the communication by keeping the clock signal low.

0 - disabled

1 - enabled (default)

`G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__I2C_MASTER_STOP_NACK`
u8_t

I2C master stop no acknowledge

0 - send ack after last byt from slave

1 - send NO ack after last byt from slave (default)

`length`
Length of property data in bytes

`data`
Property data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_Data_Property_GetById

G_Lvds_FrameGrabber_Data_Property_GetById — Query data property value

Description

Use this command to query a data property value.

This command is deprecated, use G_Lvds_Common_Data_Property_GetById instead!

Definition

```
G_Error_t
G_Lvds_FrameGrabber_Data_Property_GetById(
 const G_PortHandle_t  portHandle,
 const u16_t  id,  // use "G_Lvds_FrameGrabber_Data_PropertyId_t"
 u16_t * const length,
 u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Property ID

The following values are possible:

`G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__DATA_MODE`
u8_t

0 - no data mode (default)

1 - I2C master mode to deserializer

2 - I2C master mode to serializer

The data mode needs to be enabled (value **1** or **2**) for I2C operation!

`G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__SPI_MASTER_CPHA`
u8_t

SPI clock phase at valid data

0 - data is valid at first clock edge after chip select (default)

1 - data is valid at second clock edge after chip select

`G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__SPI_MASTER_CPOL`
u8_t

SPI clock polarity in idle state

0 - clock polarity in idle state is low (default)

1 - clock polarity in idle state is high

`G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__SPI_MASTER_CS_MODE`
u8_t

SPI chip select behavior between consecutive bytes of the same transfer

0 - chip select remains low (active) between consecutive bytes (default)

1 - chip select switches to high (inactive) between consecutive bytes

`G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__SPI_MASTER_CLOCK_DIVIDER`
u8_t

SPI clock divider - determines SPI clock frequency based on freq. 25 MHz

0 - divider is 1 ... clock frequency is 25 MHz (default)

1 - divider is 2 ... clock frequency is 12.5 MHz

2 - divider is 3 ... clock frequency is 8.33 MHz

...

255 - divider is 256 ... clock frequency is 0.1953125 MHz

`G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__SPI_SLAVE_CPHA`
u8_t

SPI clock phase at valid data

0 - data is valid at first clock edge after chip select (default)

1 - data is valid at second clock edge after chip select

`G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__SPI_SLAVE_CS_IDLE`
u8_t

SPI chip select idle minimum in 20ns steps

0 - 127 * 20ns = 2.54us chip select idle after transfer (default)

1 - 1 * 20ns = 20ns

2 - 2 * 20ns = 40ns

...

255 - 255 * 20ns = 5.1us

`G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__I2C_MASTER_BAUDRATE`
u8_t

I2C master baud rate

0 - 100kHz (default)

1 - 400kHz

`G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__I2C_MASTER_CLOCK_STRETCHING`
u8_t

I2C master clockstretching

The I2C slave can delay the communication by keeping the clock signal low.

0 - disabled

1 - enabled (default)

`G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__I2C_MASTER_STOP_NACK`
u8_t

I2C master stop no acknowledge

0 - send ack after last byt from slave

1 - send NO ack after last byt from slave (default)

`length`
in : size of buffer `data`

out : number of returned bytes in `data`

`data`
Returns property data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_Data_Register_Write

G_Lvds_FrameGrabber_Data_Register_Write — Write register data

Description

Use this command to write data into a device register.

To use this command, the data mode needs to be enabled (command G_Lvds_FrameGrabber_Data_Property_SetById , property `G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__DATA_MODE`).

This command is deprecated, use G_Lvds_Common_Data_Register_Write instead!

Definition

```
G_Error_t
G_Lvds_FrameGrabber_Data_Register_Write(
 const G_PortHandle_t  portHandle,
 const G_Lvds_FrameGrabber_Data_Register_Write_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__FRAME_GRABBER__DATA__REG__WR__CMD_FLAG__NONE`
No flag is set

`G_LVDS__FRAME_GRABBER__DATA__REG__WR__CMD_FLAG__REG_ADDR_WIDTH_16_BIT`
Use 16 bit for register address

`G_LVDS__FRAME_GRABBER__DATA__REG__WR__CMD_FLAG__DEV_ADDR_7_BIT`
Use 7 bit for device address

`RegisterAddress`
Address of the register

`DeviceAddress`
Address of the device

`RegisterValue`
Value to be written

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_Data_Register_Read

G_Lvds_FrameGrabber_Data_Register_Read — Read data register value

Description

Use this command to read the value of a data register.

To use this command, the data mode needs to be enabled (command G_Lvds_FrameGrabber_Data_Property_SetById , property `G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__DATA_MODE`).

This command is deprecated, use G_Lvds_Common_Data_Register_Read instead!

Definition

```
G_Error_t
G_Lvds_FrameGrabber_Data_Register_Read(
 const G_PortHandle_t  portHandle,
 const G_Lvds_FrameGrabber_Data_Register_Read_Cmd_t * const cmd,
 G_Lvds_FrameGrabber_Data_Register_Read_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__FRAME_GRABBER__DATA__REG__RD__CMD_FLAG__NONE`
No flag is set

`G_LVDS__FRAME_GRABBER__DATA__REG__RD__CMD_FLAG__REG_ADDR_WIDTH_16_BIT`
Use 16 bit for register address

`G_LVDS__FRAME_GRABBER__DATA__REG__RD__CMD_FLAG__DEV_ADDR_7_BIT`
Use 7 bit for device address

`RegisterAddress`
Address of the register

`DeviceAddress`
Address of the device

`reserved`
reserved parameter (must be initialized with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_LVDS__FRAME_GRABBER__DATA__REG__RD__RSP_FLAG__NONE`
No flag is set

`RegisterValue`
Returns value of data register

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_Data_I2cTransfer

G_Lvds_FrameGrabber_Data_I2cTransfer — Initiate I2C transfer

Description

Use this command to initiate an I2C transfer.

To use this command, the data mode needs to be enabled (command G_Lvds_FrameGrabber_Data_Property_SetById , property `G_LVDS__FRAME_GRABBER__DATA__PROPERTY_ID__DATA_MODE`).

This command is deprecated, use G_Lvds_Common_Data_I2cTransfer instead!

Definition

```
G_Error_t
G_Lvds_FrameGrabber_Data_I2cTransfer(
 const G_PortHandle_t  portHandle,
 const G_Lvds_FrameGrabber_Data_I2cTransfer_Cmd_t * const cmd,
 G_Lvds_FrameGrabber_Data_I2cTransfer_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__FRAME_GRABBER__DATA__I2C_TRANSFER__RSP_FLAG__NONE`
No flag is set

`NumberOfTxBytes`
Number of bytes to be transmitted (0..8)

`NumberOfRxBytes`
Number of bytes to be received (0..8)

`SendStartMask`
Bitmask that defines the bytes that are preceded by a start symbol

The first byte is always preceded by a start symbol, therefore the first bit of parameter `SendStartMask` must not be set.

`reserved`
reserved parameter (must be initialized with **0**)

`TxData`
Data bytes to be transmitted

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_LVDS__FRAME_GRABBER__DATA__I2C_TRANSFER__RSP_FLAG__NONE`
No flag is set

`AckMask`
Acknowledge mask

Bit mask that specifies the bytes that were aknowledged when transmitted.

`NumberOfRxBytes`
Number of received bytes

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`RxData`
Received data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_VideoSettings_Property_SetById

G_Lvds_FrameGrabber_VideoSettings_Property_SetById — Change video settings

Description

Use this command to change video settings by property ID.

This command is deprecated, use G_Lvds_Common_VideoSettings_Property_SetById instead!

Definition

```
G_Error_t
G_Lvds_FrameGrabber_VideoSettings_Property_SetById(
 const G_PortHandle_t  portHandle,
 const u16_t  id,  // use "G_Lvds_FrameGrabber_VideoSettings_PropertyId_t"
 const u16_t  length,
 const u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Property ID

The following values are possible:

`G_LVDS__FRAME_GRABBER__VIDEO_SETTINGS__PROPERTY_ID__EDID_LOOP_THROUGH`
u8_t

0 - no loop through, EDID data of serdes extension board used (default)

1 - loop through enabled, EDID data of output device is looped to input

(suppported by ADV7611)

`G_LVDS__FRAME_GRABBER__VIDEO_SETTINGS__PROPERTY_ID__BUS_WIDTH_SELECT`
u8_t

0 - 24 bit bus mode

1 - 32 bit bus mode (default)

(supported by MAX9260, MAX9272, MAX9276)

`G_LVDS__FRAME_GRABBER__VIDEO_SETTINGS__PROPERTY_ID__APIX_MODE`
u8_t

0 - APIX2 mode (default)

1 - APIX1 mode

(supported by INAP375R)

`G_LVDS__FRAME_GRABBER__VIDEO_SETTINGS__PROPERTY_ID__TRANSCVR_POWER-DOWN`
u8_t

0 - transceiver is powered up (default)

1 - transceiver is powered down

(supported by MAX9260, MAX9272, MAX9276, DS90UB914, INAP375R (v3.0+))

`G_LVDS__FRAME_GRABBER__VIDEO_SETTINGS__PROPERTY_ID__MODE_SELECT`
u8_t

0 - base mode (default)

1 - bypass mode

(supported by MAX9260, MAX9272, MAX9276)d

`length`
Data length in byte

`data`
Property data to be set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_VideoSettings_Property_GetById

G_Lvds_FrameGrabber_VideoSettings_Property_GetById — Query video settings value

Description

Use this command to query a video settings value by property ID.

This command is deprecated, use G_Lvds_Common_VideoSettings_Property_GetById instead!

Definition

```
G_Error_t
G_Lvds_FrameGrabber_VideoSettings_Property_GetById(
 const G_PortHandle_t  portHandle,
 const u16_t  id,  // use "G_Lvds_FrameGrabber_VideoSettings_PropertyId_t"
 u16_t * const length,
 u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`id`
Property ID

The following values are possible:

`G_LVDS__FRAME_GRABBER__VIDEO_SETTINGS__PROPERTY_ID__EDID_LOOP_THROUGH`
u8_t

0 - no loop through, EDID data of serdes extension board used (default)

1 - loop through enabled, EDID data of output device is looped to input

(suppported by ADV7611)

`G_LVDS__FRAME_GRABBER__VIDEO_SETTINGS__PROPERTY_ID__BUS_WIDTH_SELECT`
u8_t

0 - 24 bit bus mode

1 - 32 bit bus mode (default)

(supported by MAX9260, MAX9272, MAX9276)

`G_LVDS__FRAME_GRABBER__VIDEO_SETTINGS__PROPERTY_ID__APIX_MODE`
u8_t

0 - APIX2 mode (default)

1 - APIX1 mode

(supported by INAP375R)

`G_LVDS__FRAME_GRABBER__VIDEO_SETTINGS__PROPERTY_ID__TRANSCVR_POWER-DOWN`
u8_t

0 - transceiver is powered up (default)

1 - transceiver is powered down

(supported by MAX9260, MAX9272, MAX9276, DS90UB914, INAP375R (v3.0+))

`G_LVDS__FRAME_GRABBER__VIDEO_SETTINGS__PROPERTY_ID__MODE_SELECT`
u8_t

0 - base mode (default)

1 - bypass mode

(supported by MAX9260, MAX9272, MAX9276)d

`length`
in : size of buffer `data`

out : number of returned bytes in `data`

`data`
Returns property data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_SetPixelMode

G_Lvds_FrameGrabber_SetPixelMode — Set pixel mode

Description

Use this command to set the pixel mode of the Frame Grabber.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_SetPixelMode(
 const G_PortHandle_t  portHandle,
 const u8_t  mode  // use "G_Lvds_FrameGrabber_PixelMode_t"
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
Pixel mode

The following values are possible:

`G_LVDS__FRAME_GRABBER__PIXEL_MODE__24_BIT`
when RGB888:

byte 0: B1[7..0]

byte 1: G1[7..0]

byte 2: R1[7..0]

`G_LVDS__FRAME_GRABBER__PIXEL_MODE__12_BIT`
RAW12 little endian

byte 0: P1[7..0]

byte 1: P2[3..0] P1[11..8]

byte 2: P2[11..4]

`G_LVDS__FRAME_GRABBER__PIXEL_MODE__RAW8`
byte 0: P1[7..0]

byte 1: P2[7..0]

byte 2: P3[7..0]

byte 3: P4[7..0]

`G_LVDS__FRAME_GRABBER__PIXEL_MODE__RAW12_MIPI_CSI2`
byte 0: P1[11..4]

byte 1: P2[11..4]

byte 2: P2[3..0] P1[3..0]

`G_LVDS__FRAME_GRABBER__PIXEL_MODE__RAW16_MIPI_CSI2`
byte 0: P1[15..8]

byte 1: P1[7..0]

byte 2: P2[15..8]

byte 3: P2[7..0]

`G_LVDS__FRAME_GRABBER__PIXEL_MODE__RAW12_16_SHIFTED_LE`
RAW12 in 16 bits, shifted left by 4 bits, little endian

byte 0: P1[3..0] 0b0000

byte 1: P1[11..4]

`G_LVDS__FRAME_GRABBER__PIXEL_MODE__RAW16_LE`
RAW16 little endian

byte 0: P1[7..0]

byte 1: P1[15..8]

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_GetPixelMode

G_Lvds_FrameGrabber_GetPixelMode — Query pixel mode

Description

Use this command to query the pixel mode of the Frame Grabber.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_GetPixelMode(
 const G_PortHandle_t  portHandle,
 u8_t * const mode  // use "G_Lvds_FrameGrabber_PixelMode_t"
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
Returns pixel mode

The following values are possible:

`G_LVDS__FRAME_GRABBER__PIXEL_MODE__24_BIT`
when RGB888:

byte 0: B1[7..0]

byte 1: G1[7..0]

byte 2: R1[7..0]

`G_LVDS__FRAME_GRABBER__PIXEL_MODE__12_BIT`
RAW12 little endian

byte 0: P1[7..0]

byte 1: P2[3..0] P1[11..8]

byte 2: P2[11..4]

`G_LVDS__FRAME_GRABBER__PIXEL_MODE__RAW8`
byte 0: P1[7..0]

byte 1: P2[7..0]

byte 2: P3[7..0]

byte 3: P4[7..0]

`G_LVDS__FRAME_GRABBER__PIXEL_MODE__RAW12_MIPI_CSI2`
byte 0: P1[11..4]

byte 1: P2[11..4]

byte 2: P2[3..0] P1[3..0]

G_LVDS__FRAME_GRABBER__PIXEL_MODE__RAW16_MIPI_CSI2
byte 0: P1[15..8]

byte 1: P1[7..0]

byte 2: P2[15..8]

byte 3: P2[7..0]

G_LVDS__FRAME_GRABBER__PIXEL_MODE__RAW12_16_SHIFTED_LE
RAW12 in 16 bits, shifted left by 4 bits, little endian

byte 0: P1[3..0] 0b0000

byte 1: P1[11..4]

G_LVDS__FRAME_GRABBER__PIXEL_MODE__RAW16_LE
RAW16 little endian

byte 0: P1[7..0]

byte 1: P1[15..8]

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_Property_SetById

G_Lvds_FrameGrabber_Property_SetById — Set frame grabber property values

Description

Use this command to set frame grabber property values.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_Property_SetById(
 const G_PortHandle_t  portHandle,
 const u8_t  lvdsChannel,
 const G_Lvds_FrameGrabber_PropertyId_t  propertyId,
 const u16_t  length,
 const u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`lvdsChannel`
LVDS channel

`propertyId`
Frame Grabber Property ID

The following values are possible:

`G_LVDS__FRAME_GRABBER__PROPERTY_ID__IF_TYPE`
Frame Grabber interface type

u8_t

This property is read only.

This property is global for all channels.

The following values are possible:

`G_LVDS__IF_TYPE__NONE`
No LVDS interface

`G_LVDS__IF_TYPE__PARALLEL`
Parallel LVDS interface

`G_LVDS__IF_TYPE__MIPI_CSI2`
MIPI CSI2 interface

`G_LVDS__FRAME_GRABBER__PROPERTY_ID__MIPI_CSI2_VCX`
MIPI CSI-2 virtual channel extension (default: 1)

u8_t

0 : virtual channel numbers range from **0** to **3**

1 : virtual channel numbers range from **0** to **15**

This property is available on MIPI CSI-2 deserializers only.

This property is global for all channels.

`G_LVDS__FRAME_GRABBER__PROPERTY_ID__VC_ID`
Virtual channel number

u8_t

The data range is deserializer interface type specific (see property `G_LVDS__FRAME_GRABBER__PROPERTY_ID__MIPI_CSI2_VCX`).

This property is channel specific.

`G_LVDS__FRAME_GRABBER__PROPERTY_ID__CH_EN`
Channel enable

u8_t

0 : channel disabled

1 : channel enabled

This property is channel specific.

`G_LVDS__FRAME_GRABBER__PROPERTY_ID__MIPI_CSI2_DT`
MIPI CSI-2 data type

u8_t

The following values are possible:

`G_LVDS__MIPI_CSI2_DT__FS`
Frame start

`G_LVDS__MIPI_CSI2_DT__FE`
Frame end

`G_LVDS__MIPI_CSI2_DT__LS`
Line start

`G_LVDS__MIPI_CSI2_DT__LE`
Line end

`G_LVDS__MIPI_CSI2_DT__EMBEDDED`
Embedded

`G_LVDS__MIPI_CSI2_DT__RGB888`
RGB888

`G_LVDS__MIPI_CSI2_DT__RAW8`
RAW8

`G_LVDS__MIPI_CSI2_DT__RAW10`
RAW10

`G_LVDS__MIPI_CSI2_DT__RAW12`
RAW12

`G_LVDS__MIPI_CSI2_DT__RAW16`
RAW16

Default: `G_LVDS__MIPI_CSI2_DT__RAW12`

`length`
Length of property value data, in bytes

`data`
Property value data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_Property_GetById

G_Lvds_FrameGrabber_Property_GetById — Query frame grabber property values

Description

Use this command to query frame grabber property values.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_Property_GetById(
 const G_PortHandle_t  portHandle,
 const u8_t  lvdsChannel,
 const G_Lvds_FrameGrabber_PropertyId_t  propertyId,
 u16_t * const length,
 u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`lvdsChannel`
LVDS channel

`propertyId`
Frame Grabber Property ID

The following values are possible:

`G_LVDS__FRAME_GRABBER__PROPERTY_ID__IF_TYPE`
Frame Grabber interface type

u8_t

This property is read only.

This property is global for all channels.

The following values are possible:

`G_LVDS__IF_TYPE__NONE`
No LVDS interface

`G_LVDS__IF_TYPE__PARALLEL`
Parallel LVDS interface

`G_LVDS__IF_TYPE__MIPI_CSI2`
MIPI CSI2 interface

`G_LVDS__FRAME_GRABBER__PROPERTY_ID__MIPI_CSI2_VCX`
MIPI CSI-2 virtual channel extension (default: 1)

u8_t

0 : virtual channel numbers range from **0** to **3**

1 : virtual channel numbers range from **0** to **15**

This property is available on MIPI CSI-2 deserializers only.

This property is global for all channels.

`G_LVDS__FRAME_GRABBER__PROPERTY_ID__VC_ID`
Virtual channel number

u8_t

The data range is deserializer interface type specific (see property `G_LVDS__FRAME_GRABBER__PROPERTY_ID__MIPI_CSI2_VCX`).

This property is channel specific.

`G_LVDS__FRAME_GRABBER__PROPERTY_ID__CH_EN`
Channel enable

u8_t

0 : channel disabled

1 : channel enabled

This property is channel specific.

`G_LVDS__FRAME_GRABBER__PROPERTY_ID__MIPI_CSI2_DT`
MIPI CSI-2 data type

u8_t

The following values are possible:

`G_LVDS__MIPI_CSI2_DT__FS`
Frame start

`G_LVDS__MIPI_CSI2_DT__FE`
Frame end

`G_LVDS__MIPI_CSI2_DT__LS`
Line start

`G_LVDS__MIPI_CSI2_DT__LE`
Line end

`G_LVDS__MIPI_CSI2_DT__EMBEDDED`
Embedded

`G_LVDS__MIPI_CSI2_DT__RGB888`
RGB888

`G_LVDS__MIPI_CSI2_DT__RAW8`
RAW8

`G_LVDS__MIPI_CSI2_DT__RAW10`
RAW10

`G_LVDS__MIPI_CSI2_DT__RAW12`
RAW12

`G_LVDS__MIPI_CSI2_DT__RAW16`
RAW16

Default: `G_LVDS__MIPI_CSI2_DT__RAW12`

`length`
In : length of property value data, in bytes

Out : length of returned property value data, in bytes

`data`
Returns property data value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_CaptureProperty_SetById

G_Lvds_FrameGrabber_CaptureProperty_SetById — Set frame grabber capture property values

Description

Use this command to set frame grabber capture property values.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_CaptureProperty_SetById(
 const G_PortHandle_t  portHandle,
 const u8_t  lvdsChannel,
 const G_Lvds_FrameGrabber_CapturePropertyId_t  propertyId,
 const u16_t  length,
 const u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`lvdsChannel`
LVDS channel

`propertyId`
Frame Grabber Capture Property ID

The following values are possible:

`G_LVDS__FRAME_GRABBER__CAPTURE_PROPERTY_ID__DEVICE_FB_COUNT`
u32_t

Device frame buffer count

Number of frame buffers within the device

1 .. 15 (**3** = default)

1 = capture cmd always returns a new frame but some time may be needed to finish frame capturing - use this value for single shot (one picture)

2 = capture cmd starts a new capture, a previous captured frame may be returned immediately

3 = capture cmd starts a new capture, a previous captured frame may be returned immediately - use this value for a video

`G_LVDS__FRAME_GRABBER__CAPTURE_PROPERTY_ID__FRAME_RATE_DIVISOR`
u32_t

1 .. 128

1 = every frame is captured

2 = every 2. frame is captured

3 = every 3. frame is captured

...

128 = every 128. frame is captured

`G_LVDS__FRAME_GRABBER__CAPTURE_PROPERTY_ID__DEVICE_FB_OVERWRITE`
u8_t

If device frame buffers are full and a new frame is received:

0 = the new frame is discarded, no frame buffer is modified

1 = the oldest frame buffer is overwritten (default)

`G_LVDS__FRAME_GRABBER__CAPTURE_PROPERTY_ID__FRAME_COUNT`
u32_t

Number of frames to capture

0 = no frame will be captured

1 = one frame will be captured

2 = two frames will be captured

3 .. 254 = this number of frames will be captured

0xffffffff = unlimited number of frames will be captured (default)

When number of frames to capture is reached, frame capturing is automatically disabled (see property "`CAPTURE_EN`")

`G_LVDS__FRAME_GRABBER__CAPTURE_PROPERTY_ID__CAPTURE_EN__STATE`
u8_t

This property is read only.

Capture enable - state

0 = capturing is disabled

1 = capturing is enabled

`G_LVDS__FRAME_GRABBER__CAPTURE_PROPERTY_ID__CAPTURE_EN__REQ`
u8_t

Capture enable - request

0 = capture enable is not requested

1 = capture enable is requested

`G_LVDS__FRAME_GRABBER__CAPTURE_PROPERTY_ID__FRAME_COUNTER`
u64_t

This property is read only.

Number of captured frames

Currently this is a 16 bit counter (range **0** .. **0xffff**).

`length`
Length of property value data, in bytes

`data`
Property value data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_FrameGrabber_CaptureProperty_GetById

G_Lvds_FrameGrabber_CaptureProperty_GetById — Query frame grabber capture property values

Description

Use this command to query frame grabber capture property values.

Definition

```
G_Error_t
G_Lvds_FrameGrabber_CaptureProperty_GetById(
 const G_PortHandle_t  portHandle,
 const u8_t  lvdsChannel,
 const G_Lvds_FrameGrabber_CapturePropertyId_t  propertyId,
 u16_t * const length,
 u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`lvdsChannel`
LVDS channel

`propertyId`
Frame Grabber Property ID

The following values are possible:

`G_LVDS__FRAME_GRABBER__CAPTURE_PROPERTY_ID__DEVICE_FB_COUNT`
u32_t

Device frame buffer count

Number of frame buffers within the device

1 .. 15 (**3** = default)

1 = capture cmd always returns a new frame but some time may be needed to finish frame capturing - use this value for single shot (one picture)

2 = capture cmd starts a new capture, a previous captured frame may be returned immediately

3 = capture cmd starts a new capture, a previous captured frame may be returned immediately - use this value for a video

`G_LVDS__FRAME_GRABBER__CAPTURE_PROPERTY_ID__FRAME_RATE_DIVISOR`
u32_t

1 .. 128

1 = every frame is captured

2 = every 2. frame is captured

3 = every 3. frame is captured

…

128 = every 128. frame is captured

`G_LVDS__FRAME_GRABBER__CAPTURE_PROPERTY_ID__DEVICE_FB_OVERWRITE`
u8_t

If device frame buffers are full and a new frame is received:

0 = the new frame is discarded, no frame buffer is modified

1 = the oldest frame buffer is overwritten (default)

`G_LVDS__FRAME_GRABBER__CAPTURE_PROPERTY_ID__FRAME_COUNT`
u32_t

Number of frames to capture

0 = no frame will be captured

1 = one frame will be captured

2 = two frames will be captured

3 .. 254 = this number of frames will be captured

0xffffffff = unlimited number of frames will be captured (default)

When number of frames to capture is reached, frame capturing is automatically disabled (see property "`CAPTURE_EN`")

`G_LVDS__FRAME_GRABBER__CAPTURE_PROPERTY_ID__CAPTURE_EN__STATE`
u8_t

This property is read only.

Capture enable - state

0 = capturing is disabled

1 = capturing is enabled

`G_LVDS__FRAME_GRABBER__CAPTURE_PROPERTY_ID__CAPTURE_EN__REQ`
u8_t

Capture enable - request

0 = capture enable is not requested

1 = capture enable is requested

`G_LVDS__FRAME_GRABBER__CAPTURE_PROPERTY_ID__FRAME_COUNTER`
u64_t

This property is read only.

Number of captured frames

Currently this is a 16 bit counter (range **0** .. **0xffff**).

`length`
In : length of property value data, in bytes

Out : length of returned property value data, in bytes

`data`
Returns property data value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

4 Multiplexer

These functions provide control over your **LVDS-Multiplexer** hardware.

The LVDS-Multiplexer is a tool for distributing LVDS signals. Multiple LVDS input channels can be multiplexed to one output channel.

For a detailed description of the hardware features of your LVDS-Multiplexer, please refer to the hardware documentation.

G_Lvds_Multiplexer_HostControlMode_Set

G_Lvds_Multiplexer_HostControlMode_Set — Set host control mode

Description

Set the host control mode of the multiplexer.

The host control mode determines whether the LVDS-Multiplexer is controlled by **G-API** commands or hardware switches on the circuit board.

Definition

```
G_Error_t
G_Lvds_Multiplexer_HostControlMode_Set(
 const G_PortHandle_t  portHandle,
 const G_Lvds_Multiplexer_HostControlMode_t  mode
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
Host control mode

The following values are possible:

`G_LVDS__MULTIPLEXER__HOST_CONTROL_MODE__COMMANDS`
The multiplexer is controlled by **G-API** commands.

`G_LVDS__MULTIPLEXER__HOST_CONTROL_MODE__SWITCHES`
The multiplexer is controlled by hardware switches on the circuit board (see hardware documentation for details).

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Multiplexer_PowerMode_Set

G_Lvds_Multiplexer_PowerMode_Set — Set Multiplexer power mode

Description

Use this command to enable or disable the LVDS output.

Definition

```
G_Error_t
G_Lvds_Multiplexer_PowerMode_Set(
 const G_PortHandle_t  portHandle,
 const G_Lvds_Multiplexer_PowerMode_t  mode
);
```

Parameters

`portHandle`
: Handle to the communication port

`mode`
: Power mode

  The following values are possible:

  `G_LVDS__MULTIPLEXER__POWER_MODE__OFF`
  : The LVDS output is disabled

  `G_LVDS__MULTIPLEXER__POWER_MODE__ON`
  : The LVDS output is enabled

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Multiplexer_Source_Select

G_Lvds_Multiplexer_Source_Select — Select source

Description

Use this command to select the source channel for LVDS output.

Definition

```
G_Error_t
G_Lvds_Multiplexer_Source_Select(
 const G_PortHandle_t  portHandle,
 const u8_t  source
);
```

Parameters

`portHandle`
Handle to the communication port

`source`
Selects the LVDS channel that is routed to the output.

The number of available source channels depends on your **GOEPEL electronic** LVDS multiplexer device.

0 : LVDS channel **0** is routed to the output

1 : LVDS channel **1** is routed to the output

...

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Multiplexer_Preemphasis_Enable

G_Lvds_Multiplexer_Preemphasis_Enable — Enable pre-emphasis of output signal

Description

Use this command to enable pre-emphasis of the output signal.

When pre-emphasis is enabled, high frequencies of the output signal are boosted, resulting in a higher signal-to-noise-ratio while transmitting. The receiver is responsible to compensate this frequency boost.

Definition

```
G_Error_t
G_Lvds_Multiplexer_Preemphasis_Enable(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Multiplexer_Preemphasis_Disable

G_Lvds_Multiplexer_Preemphasis_Disable — Disable pre-emphasis of output signal

Description

Use this command to disable pre-emphasis of the output signal.

When pre-emphasis is enabled, high frequencies of the output signal are boosted, resulting in a higher signal-to-noise-ratio while transmitting. The receiver is responsible to compensate this frequency boost.

Definition

```
G_Error_t
G_Lvds_Multiplexer_Preemphasis_Disable(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Multiplexer_Eq_Enable

G_Lvds_Multiplexer_Eq_Enable — Enable equalization of inputs

Description

Use this command to enable equalization of the input channels.

Equalization is used to compensate the high frequency boost of pre-emphasized signals.

Definition

```
G_Error_t
G_Lvds_Multiplexer_Eq_Enable(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Multiplexer_Eq_Disable

G_Lvds_Multiplexer_Eq_Disable — Disable equalization of inputs

Description

Use this command to disable equalization of the input channels.

Equalization is used to compensate the high frequency boost of pre-emphasized signals.

Definition

```
G_Error_t
G_Lvds_Multiplexer_Eq_Disable(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

5 Pattern Generator

These functions provide control over the **Pattern Generator** functionality of the device.

The **Pattern Generator** is used for generating LVDS image data without the need for an external source.

The required sequence to use the Pattern Generator is:

- Initialization (G_Lvds_PattGen_Init)
- Pattern Configuration (e.g. G_Lvds_PattGen_Default_Create)
- Enable Pattern Generator (G_Lvds_PattGen_Enable)
- Start Pattern Generator (G_Lvds_PattGen_Start)

G_Lvds_PattGen_Reset

G_Lvds_PattGen_Reset — Reset Pattern Generator

Description

Use this command to reset the Pattern Generator.

Definition

```
G_Error_t
G_Lvds_PattGen_Reset(
 const G_PortHandle_t  portHandle,
 const G_Lvds_PattGen_Reset_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Structure with command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_LVDS__PATT_GEN__RESET__CMD_FLAG__NONE`
No flag is set

`PattGenInstId`
Pattern generator instance ID

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_PattGen_Init

G_Lvds_PattGen_Init — Initialize Pattern Generator

Description

Use this command to initialize the Pattern Generator.

Definition

```
G_Error_t
G_Lvds_PattGen_Init(
 const G_PortHandle_t  portHandle,
 const G_Lvds_PattGen_Init_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Structure with command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_LVDS__PATT_GEN__INIT__CMD_FLAG__NONE`
No flag is set

`PattGenInstId`
Pattern generator instance ID

`PatternType`
Predefined pattern type to be generated

The following values are possible:

`G_LVDS__PATT_GEN__PATTERN_TYPE__PATTERN_1`
Pattern type 1

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_PattGen_Default_Create

G_Lvds_PattGen_Default_Create — Create default pattern for different picture formats

Description

Use this command to create a default pattern for different picture formats.

Definition

```
G_Error_t
G_Lvds_PattGen_Default_Create(
 const G_PortHandle_t  portHandle,
 const G_Lvds_PattGen_Default_Create_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Structure with command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_LVDS__PATT_GEN__DEFAULT_CREATE__CMD_FLAG__NONE`
No flag is set

`PattGenInstId`
Pattern generator instance ID

`PicFormat`
Picture format of the default pattern.

The following values are possible:

`G_LVDS__PATT_GEN__PIC_FORMAT__RGB888`
RGB888 mode

`G_LVDS__PATT_GEN__PIC_FORMAT__YUV422_8_BIT`
YUV 4:2:2 8 Bit mode

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_PattGen_Pattern1Data_Set

G_Lvds_PattGen_Pattern1Data_Set — Set pattern data

Description

Use this command to set the pattern data.

Definition

```
G_Error_t
G_Lvds_PattGen_Pattern1Data_Set(
 const G_PortHandle_t  portHandle,
 const G_Lvds_PattGen_Pattern1Data_Set_Cmd_t * const cmd,
 const u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_LVDS__PATT_GEN__PATTERN1_DATA__SET__CMD_FLAG__NONE`
No flag is set

`PattGenInstId`
Pattern generator instance ID

`PicFormat`
Picture format of the pattern

The following values are possible:

`G_LVDS__PATT_GEN__PIC_FORMAT__RGB888`
RGB888 mode

`G_LVDS__PATT_GEN__PIC_FORMAT__YUV422_8_BIT`
YUV 4:2:2 8 Bit mode

`Length`
Size of pattern data to be transfered in bytes

`data`
Pattern data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_PattGen_Pattern1Data_Get

G_Lvds_PattGen_Pattern1Data_Get — Get pattern data

Description

Use this command to query the current pattern data.

Definition

```
G_Error_t
G_Lvds_PattGen_Pattern1Data_Get(
 const G_PortHandle_t  portHandle,
 const G_Lvds_PattGen_Pattern1Data_Get_Cmd_t * const cmd,
 G_Lvds_PattGen_Pattern1Data_Get_Rsp_t * const rsp,
 u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_LVDS__PATT_GEN__PATTERN1_DATA__GET__CMD_FLAG__NONE`
No flag is set

`PattGenInstId`
Pattern generator instance ID

`reserved`
reserved parameter (must be initialized with **0**)

`Length`
Size of buffer `data` in bytes

The buffer has to be big enough to hold the pattern data, otherwise an error will be omitted.

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible:

`G_LVDS__PATT_GEN__PATTERN1_DATA__GET__RSP_FLAG__NONE`
No flag is set

`PattGenInstId`
Pattern generator instance ID

`PicFormat`
Picture format of the pattern

The following values are possible:

`G_LVDS__PATT_GEN__PIC_FORMAT__RGB888`
RGB888 mode

`G_LVDS__PATT_GEN__PIC_FORMAT__YUV422_8_BIT`
YUV 4:2:2 8 Bit mode

`Length`
Size of returned data in buffer `data`

`data`
Returns pattern data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_PattGen_Pattern1Def_Set

G_Lvds_PattGen_Pattern1Def_Set — Set pattern definition

Description

Use this command to set the pattern definition.

This includes the resolution in horizontal and vertical dimension and the number of pattern segments in each dimension.

Definition

```
G_Error_t
G_Lvds_PattGen_Pattern1Def_Set(
 const G_PortHandle_t  portHandle,
 const G_Lvds_PattGen_Pattern1Def_Set_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_LVDS__PATT_GEN__PATTERN1_DEF__SET__CMD_FLAG__NONE`
No flag is set

`PattGenInstId`
Pattern generator instance ID

`NmbrOfHseg`
Number of individual pattern segments in horizontal dimension (**1** .. **8**)

`NmbrOfVseg`
Number of individual pattern segments in vertical dimension (**1** .. **8**)

`reserved`
reserved parameter (must be initialized with **0**)

`Hres`
Horizontal resolution (has to be a multiple of **4**)

`Vres`
Vertical resolution (has to be a multiple of **4**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_PattGen_Pattern1Def_Get

G_Lvds_PattGen_Pattern1Def_Get — Get pattern definition

Description

Use this command to qery the pattern definition.

This includes the resolution in horizontal and vertical dimension and the number of pattern segments in each dimension.

Definition

```
G_Error_t
G_Lvds_PattGen_Pattern1Def_Get(
 const G_PortHandle_t  portHandle,
 const G_Lvds_PattGen_Pattern1Def_Get_Cmd_t * const cmd,
 G_Lvds_PattGen_Pattern1Def_Get_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_LVDS__PATT_GEN__PATTERN1_DEF__GET__CMD_FLAG__NONE`
No flag is set

`PattGenInstId`
Pattern generator instance ID

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Flags`
Response flags

The following values are possible:

`G_LVDS__PATT_GEN__PATTERN1_DEF__GET__RSP_FLAG__NONE`
No flag is set

`PattGenInstId`
Pattern generator instance ID

`NmbrOfHseg`
Number of individual pattern segments in horizontal dimension (**1** .. **8**)

`NmbrOfVseg`
Number of individual pattern segments in vertical dimension (**1** .. **8**)

`reserved`
reserved parameter

`Hres`
Horizontal resolution (has to be a multiple of **4**)

`Vres`
Vertical resolution (has to be a multiple of **4**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_PattGen_Timing_Set

G_Lvds_PattGen_Timing_Set — Set timing parameters

Description

Use this command to set the timing parameters of the Pattern Generator.

Definition

```
G_Error_t
G_Lvds_PattGen_Timing_Set(
 const G_PortHandle_t  portHandle,
 const G_Lvds_PattGen_Timing_Set_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_LVDS__PATT_GEN__TIMING__SET__CMD_FLAG__NONE`
No flag is set

`PattGenInstId`
Pattern generator instance ID

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`FrmTimeInUs`
Time to draw one frame, in microsends

`LineTimeInNs`
Time to draw one horizontal line, in nanoseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_PattGen_Property_SetById

G_Lvds_PattGen_Property_SetById — Set property value

Description

Use this command to set a property value.

Definition

```
G_Error_t
G_Lvds_PattGen_Property_SetById(
 const G_PortHandle_t  portHandle,
 const G_Lvds_PattGen_Property_SetById_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_LVDS__PATT_GEN__PROPERTY__SET_BY_ID__CMD_FLAG__NONE`
No flag is set

`PattGenInstId`
Pattern generator instance ID

`reserved`
reserved parameter (must be initialized with `0`)

`PropertyId`
Property ID

The following values are possible:

`G_LVDS__PATT_GEN__PROPERTY_ID__VC`
SET/GET

Virtual channel ID for MIPI CSI-2

`G_LVDS__PATT_GEN__PROPERTY_ID__DT`
SET/GET

Data type for MIPI CSI-2

`G_LVDS__PATT_GEN__PROPERTY_ID__NMBR_OF_FRMS`
SET/GET

Set the number of frames that are displayed

(**1 .. 254** and **INFINITE** (**0xFFFFFFFF**))

`G_LVDS__PATT_GEN__PROPERTY_ID__FRM_CNTR`
GET

Number of remaining frames to be displayed

Has no function for **INFINITE** (**0xFFFFFFFF**)

`G_LVDS__PATT_GEN__PROPERTY_ID__FRM_TIME_IN_US`
GET

Frame time, in microseconds

`G_LVDS__PATT_GEN__PROPERTY_ID__LINE_TIME_IN_NS`
GET

Line time, in nanoseconds

`G_LVDS__PATT_GEN__PROPERTY_ID__IS_ACTIVE`
GET

Status of frame output/generation

(**0** = inactive / **1** = active)

`Data`
Property value to be set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_PattGen_Property_GetById

G_Lvds_PattGen_Property_GetById — Get property value

Description

Use this command to query a property value.

Definition

```
G_Error_t
G_Lvds_PattGen_Property_GetById(
 const G_PortHandle_t  portHandle,
 const G_Lvds_PattGen_Property_GetById_Cmd_t * const cmd,
 G_Lvds_PattGen_Property_GetById_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_LVDS__PATT_GEN__PROPERTY__GET_BY_ID__CMD_FLAG__NONE`
No flag is set

`PattGenInstId`
Pattern generator instance ID

`reserved`
reserved parameter (must be initialized with `0`)

`PropertyId`
Property ID

The following values are possible:

`G_LVDS__PATT_GEN__PROPERTY_ID__VC`
SET/GET

Virtual channel ID for MIPI CSI-2

`G_LVDS__PATT_GEN__PROPERTY_ID__DT`
SET/GET

Data type for MIPI CSI-2

`G_LVDS__PATT_GEN__PROPERTY_ID__NMBR_OF_FRMS`
SET/GET

Set the number of frames that are displayed

(**1** .. **254** and **INFINITE** (**0xFFFFFFFF**))

G_LVDS__PATT_GEN__PROPERTY_ID__FRM_CNTR
GET

Number of remaining frames to be displayed

Has no function for **INFINITE** (**0xFFFFFFFF**)

G_LVDS__PATT_GEN__PROPERTY_ID__FRM_TIME_IN_US
GET

Frame time, in microseconds

G_LVDS__PATT_GEN__PROPERTY_ID__LINE_TIME_IN_NS
GET

Line time, in nanoseconds

G_LVDS__PATT_GEN__PROPERTY_ID__IS_ACTIVE
GET

Status of frame output/generation

(**0** = inactive / **1** = active)

Flags
Response flags

The following values are possible:

G_LVDS__PATT_GEN__PROPERTY__GET_BY_ID__RSP_FLAG__NONE
No flag is set

PattGenInstId
Pattern generator instance ID

reserved
reserved parameter (must be initialized with **0**)

PropertyId
Property ID

The following values are possible:

G_LVDS__PATT_GEN__PROPERTY_ID__VC
SET/GET

Virtual channel ID for MIPI CSI-2

G_LVDS__PATT_GEN__PROPERTY_ID__DT
SET/GET

Data type for MIPI CSI-2

G_LVDS__PATT_GEN__PROPERTY_ID__NMBR_OF_FRMS
SET/GET

Set the number of frames that are displayed

(**1** .. **254** and **INFINITE** (**0xFFFFFFFF**))

`G_LVDS__PATT_GEN__PROPERTY_ID__FRM_CNTR`
GET

Number of remaining frames to be displayed

Has no function for **INFINITE** (**0xFFFFFFFF**)

`G_LVDS__PATT_GEN__PROPERTY_ID__FRM_TIME_IN_US`
GET

Frame time, in microseconds

`G_LVDS__PATT_GEN__PROPERTY_ID__LINE_TIME_IN_NS`
GET

Line time, in nanoseconds

`G_LVDS__PATT_GEN__PROPERTY_ID__IS_ACTIVE`
GET

Status of frame output/generation

(**0** = inactive / **1** = active)

`Data`
Returns property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_PattGen_Enable

G_Lvds_PattGen_Enable — Enable Pattern Generator

Description

Use this command to enable the Pattern Generator.

Definition

```
G_Error_t
G_Lvds_PattGen_Enable(
 const G_PortHandle_t  portHandle,
 const G_Lvds_PattGen_Enable_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Structure with command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_LVDS__PATT_GEN__ENABLE__CMD_FLAG__NONE`
No flag is set

`PattGenInstId`
Pattern generator instance ID

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_PattGen_Disable

G_Lvds_PattGen_Disable — Disable Pattern Generator

Description

Use this command to disable the Pattern Generator.

Definition

```
G_Error_t
G_Lvds_PattGen_Disable(
 const G_PortHandle_t  portHandle,
 const G_Lvds_PattGen_Disable_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Structure with command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_LVDS__PATT_GEN__DISABLE__CMD_FLAG__NONE`
No flag is set

`PattGenInstId`
Pattern generator instance ID

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_PattGen_Start

G_Lvds_PattGen_Start — Start Pattern Generator

Description

Use this command to start the Pattern Generator.

Definition

```
G_Error_t
G_Lvds_PattGen_Start(
 const G_PortHandle_t  portHandle,
 const G_Lvds_PattGen_Start_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Structure with command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_LVDS__PATT_GEN__START__CMD_FLAG__NONE`
No flag is set

`PattGenInstId`
Pattern generator instance ID

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_PattGen_Stop

G_Lvds_PattGen_Stop — Stop Pattern Generator

Description

Use this command to stop the Pattern Generator.

Definition

```
G_Error_t
G_Lvds_PattGen_Stop(
 const G_PortHandle_t  portHandle,
 const G_Lvds_PattGen_Stop_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Structure with command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_LVDS__PATT_GEN__STOP__CMD_FLAG__NONE`
No flag is set

`PattGenInstId`
Pattern generator instance ID

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

6 SerDes GPIO

These functions provide control over the GPIOs of the LVDS serializer / deserializer board.

The GPIOs can be used to trigger specific actions.

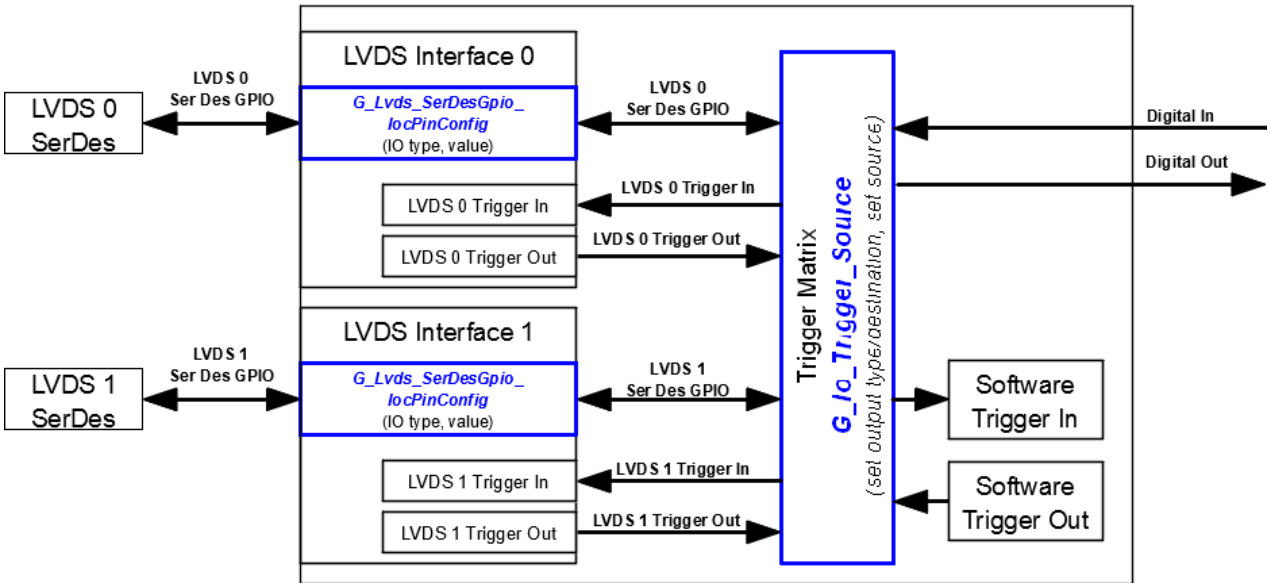

Figure 2 LVDS Trigger Matrix Overview

G_Lvds_SerDesGpio_IocPinConfig_Get

G_Lvds_SerDesGpio_IocPinConfig_Get — Get SerDes input/output controller pin configuration

Description

Use this command to query the input/output controller pin configuration of the serializer / deserializer board.

Definition

```
G_Error_t
G_Lvds_SerDesGpio_IocPinConfig_Get(
 const G_PortHandle_t  portHandle,
 const G_Lvds_SerDesGpio_IocPinConfig_Get_Cmd_t * const cmd,
 G_Lvds_SerDesGpio_IocPinConfig_Get_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`GpioIndex`
Index of the input/output controller pin (starting with **0**)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`IocOutputType`
Output type of the selected input/output controller pin

The following values are possible:

`G_LVDS__SER_DES_GPIO__IOC_OUTPUT_TYPE__DISABLED`
The selected GPIO pin is input only.

`G_LVDS__SER_DES_GPIO__IOC_OUTPUT_TYPE__PUSH_PULL`
The selected GPIO pin is in push/pull output mode.

`G_LVDS__SER_DES_GPIO__IOC_OUTPUT_TYPE__OPEN_DRAIN`
The selected GPIO pin is in open drain output mode.

`IocOutputValue`
Output value of he selected input/output controller pin (**0** or **1**)

`IocInputValue`
Input value of he selected input/output controller pin (**0** or **1**)

`reserved`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_SerDesGpio_IocPinConfig_Set

G_Lvds_SerDesGpio_IocPinConfig_Set — Set SerDes input/output controller pin configuration

Description

Use this command to set the input/output controller pin configuration of the serializer / deserializer board.

Definition

```
G_Error_t
G_Lvds_SerDesGpio_IocPinConfig_Set(
 const G_PortHandle_t  portHandle,
 const G_Lvds_SerDesGpio_IocPinConfig_Set_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__SER_DES_GPIO__IOC_PIN_CONFIG__SET__CMD_FLAG__NONE`
No flag is set

`G_LVDS__SER_DES_GPIO__IOC_PIN_CONFIG__SET__CMD_FLAG__SET_OUTPUT_TYPE`
Set output type to value of parameter `OutputType`

If not set, the value of parameter `OutputType` will be disregarded.

`GpioIndex`
SerDes GPIO number (starting with '0')

`IocOutputType`
Output type of the SerDes GPIO

The following values are possible:

`G_LVDS__SER_DES_GPIO__IOC_OUTPUT_TYPE__DISABLED`
The selected GPIO pin is input only.

`G_LVDS__SER_DES_GPIO__IOC_OUTPUT_TYPE__PUSH_PULL`
The selected GPIO pin is in push/pull output mode.

`G_LVDS__SER_DES_GPIO__IOC_OUTPUT_TYPE__OPEN_DRAIN`
The selected GPIO pin is in open drain output mode.

`IocOutputMod`
Output modification action of the selected input/output controller pin

The following values are possible:

`G_LVDS__SER_DES_GPIO__IOC_OUTPUT_MOD__NO_CHANGE`
The output state is not changed.

`G_LVDS__SER_DES_GPIO__IOC_OUTPUT_MOD__SET`
The output is set.

`G_LVDS__SER_DES_GPIO__IOC_OUTPUT_MOD__CLEAR`
The output is cleared.

`G_LVDS__SER_DES_GPIO__IOC_OUTPUT_MOD__TOGGLE`
The output is toggled.

`reserved`
reserved parameter (must be initialized with **0**)

The output modification only takes effect if the selected SerDes GPIO is not used as a GPIO Trigger Source (see G_Io_Trigger_Source_Set)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

7 Splitter

These functions provide control over your **LVDS-Splitter** hardware.

The LVDS-Splitter is a tool for distributing LVDS signals. An LVDS source can be splitted to multiple output channels.

For a detailed description of the hardware features of your LVDS-Splitter, please refer to the hardware documentation.

G_Lvds_Splitter_HostControlMode_Set

G_Lvds_Splitter_HostControlMode_Set — Set host control mode

Description

Set the host control mode of the splitter.

The host control mode determines whether the LVDS-Splitter is controlled by **G-API** commands or hardware switches on the circuit board.

Definition

```
G_Error_t
G_Lvds_Splitter_HostControlMode_Set(
 const G_PortHandle_t  portHandle,
 const G_Lvds_Splitter_HostControlMode_t  mode
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
Host control mode

The following values are possible:

`G_LVDS__SPLITTER__HOST_CONTROL_MODE__COMMANDS`
The splitter is controlled by **G-API** commands.

`G_LVDS__SPLITTER__HOST_CONTROL_MODE__SWITCHES`
The splitter is controlled by hardware switches on the circuit board (see hardware documentation for details).

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Splitter_PowerMode_Set

G_Lvds_Splitter_PowerMode_Set — Set Splitter power mode

Description

Use this command to enable or disable all outputs of the splitter.

Definition

```
G_Error_t
G_Lvds_Splitter_PowerMode_Set(
 const G_PortHandle_t  portHandle,
 const G_Lvds_Splitter_PowerMode_t  mode
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
Power mode

The following values are possible:

`G_LVDS__SPLITTER__POWER_MODE__OFF`
The outputs are disabled

`G_LVDS__SPLITTER__POWER_MODE__ON`
The outputs are enabled

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Splitter_Outputs_Enable

G_Lvds_Splitter_Outputs_Enable — Enable outputs

Description

Use this command to enable one or multiple outputs of the device.

The source signal will be routed to these outputs.

Definition

```
G_Error_t
G_Lvds_Splitter_Outputs_Enable(
 const G_PortHandle_t  portHandle,
 const u32_t  numberOfOutputs,
 const u8_t * const outputs
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfOutputs`
Number of outputs to be enabled

`outputs`
outputs (by number) to be enabled (output numbers starting with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
G_PortHandle_t  portHandle;
u32_t  numberOfOutputs;
u8_t  outputs[2];
G_Error_t  rc;

rc =
 G_Common_OpenInterface(
  "LVDS1",
  &portHandle
 );

if (rc != G_NO_ERROR) {
 return rc;
}

// enable outputs 1 and 4 (output number 0 and 3)
numberOfOutputs = 2;
outputs[0] = 0;
outputs[1] = 3;

rc =
```

```
 G_Lvds_Splitter_Outputs_Enable(
  portHandle,
  numberOfOutputs,
  outputs
 );

if (rc != G_NO_ERROR) {
 return rc;
}

return
 G_Common_CloseInterface(
  portHandle
 );
```

G_Lvds_Splitter_Outputs_Disable

G_Lvds_Splitter_Outputs_Disable — Disable outputs

Description

Use this command to disable one or multiple outputs of the device.

Definition

```
G_Error_t
G_Lvds_Splitter_Outputs_Disable(
 const G_PortHandle_t  portHandle,
 const u32_t  numberOfOutputs,
 const u8_t * const outputs
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfOutputs`
Number of outputs to be disabled

`outputs`
outputs (by number) to be disabled (output numbers starting with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Splitter_Preemphasis_Enable

G_Lvds_Splitter_Preemphasis_Enable — Enable pre-emphasis of output signals

Description

Use this command to enable pre-emphasis of the output signals.

When pre-emphasis is enabled, high frequencies of the output signals are boosted, resulting in a higher signal-to-noise-ratio while transmitting. The receiver is responsible to compensate this frequency boost.

Definition

```
G_Error_t
G_Lvds_Splitter_Preemphasis_Enable(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Splitter_Preemphasis_Disable

G_Lvds_Splitter_Preemphasis_Disable — Disable pre-emphasis of output signals

Description

Use this command to disable pre-emphasis of the output signals.

When pre-emphasis is enabled, high frequencies of the output signal are boosted, resulting in a higher signal-to-noise-ratio while transmitting. The receiver is responsible to compensate this frequency boost.

Definition

```
G_Error_t
G_Lvds_Splitter_Preemphasis_Disable(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Splitter_Eq_Enable

G_Lvds_Splitter_Eq_Enable — Enable equalization of input

Description

Use this command to enable equalization of the input channel.

Equalization is used to compensate the high frequency boost of pre-emphasized signals.

Definition

```
G_Error_t
G_Lvds_Splitter_Eq_Enable(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_Splitter_Eq_Disable

G_Lvds_Splitter_Eq_Disable — Disable equalization of input

Description

Use this command to disable equalization of the input channel.

Equalization is used to compensate the high frequency boost of pre-emphasized signals.

Definition

```
G_Error_t
G_Lvds_Splitter_Eq_Disable(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

8 Trigger Input Control

These functions provide control over the trigger input of the LVDS interface.

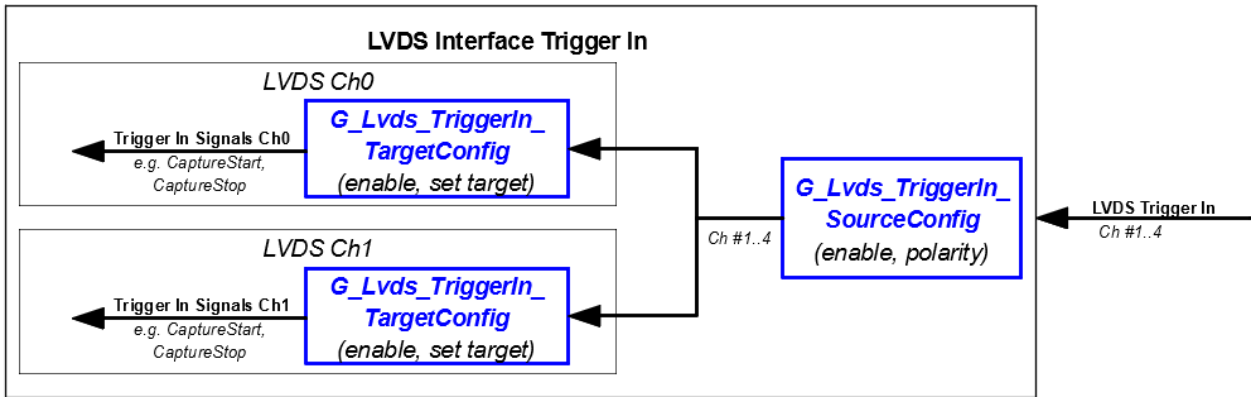

Figure 3 LVDS Trigger Input Overview

LVDS Trigger Input Example

LVDS Trigger Input Example — LVDS Trigger Input Example

Description

In this example the frame capture operation is started at **channel 1** of **LVDS Interface 0** triggered by a low active signal, which is sourced from **GPIO 5** of **LVDS 0 SerDes .**

The LVDS trigger input channel **#2** will be used internally.

Block Diagram

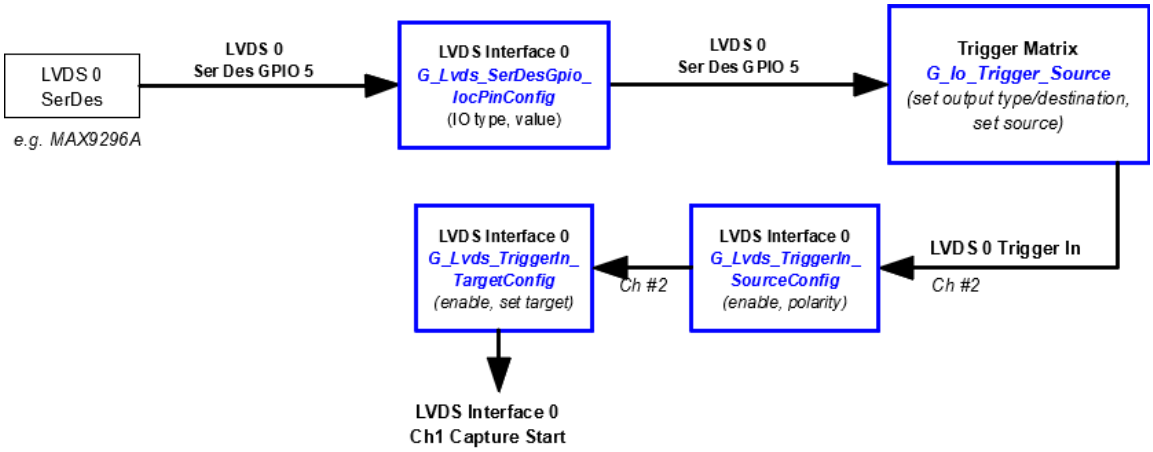

Figure 4 LVDS Trigger Input Example

Code Example:

```
G_Lvds_SerDesGpio_IocPinConfig_Set_Cmd_t  cmdIoPinConfig_Set;
G_Error_t  rc;

cmdIoPinConfig_Set.Flags =
 G_LVDS__SER_DES_GPIO__IOC_PIN_CONFIG__SET__CMD_FLAG__SET_OUTPUT_TYPE;

cmdIoPinConfig_Set.GpioIndex = 5;

cmdIoPinConfig_Set.IocOutputType = (u8_t)
 G_LVDS__SER_DES_GPIO__IOC_OUTPUT_TYPE__DISABLED;

cmdIoPinConfig_Set.IocOutputMod = (u8_t)
 G_LVDS__SER_DES_GPIO__IOC_OUTPUT_MOD__NO_CHANGE;

cmdIoPinConfig_Set.reserved = 0;

rc =
 G_Lvds_SerDesGpio_IocPinConfig_Set(
  portHandleLvds,
  &cmdIoPinConfig_Set
 );

ErrorHandler(rc);
```

```
rc =
 G_Io_Trigger_Source_Set(
  portHandleIo,
  G_IO__TRIGGER__OUTPUT_TYPE__LVDS_0_TRIGGER_IN,
  2,  // output number (starting with 1)
  G_IO__TRIGGER__SOURCE_TYPE__LVDS_0_SER_DES_GPIO,
  6  // source number (starting with 1 -> 6 equals GPIO 5)
 );

ErrorHandler(rc);

G_Lvds_TriggerIn_SourceConfig_Set_Cmd_t  cmdSourceCfgSet;

cmdSourceCfgSet.Flags =
 G_LVDS__TRIGGER_IN__SOURCE_CONFIG__SET__CMD_FLAG__ENABLE;

cmdSourceCfgSet.TriggerChannel = 2;
cmdSourceCfgSet.reserved1 = 0;
cmdSourceCfgSet.reserved2 = 0;

rc =
 G_Lvds_TriggerIn_SourceConfig_Set(
  portHandleLvds,
  &cmdSourceCfgSet
 );

ErrorHandler(rc);

G_Lvds_TriggerIn_TargetConfig_Set_Cmd_t  cmdTargetCfgSet;

cmdTargetCfgSet.Flags =
 G_LVDS__TRIGGER_IN__TARGET_CONFIG__SET__CMD_FLAG__ENABLE;

cmdTargetCfgSet.LvdsChannel = 1;
cmdTargetCfgSet.Target = (u8_t) G_LVDS__TRIGGER_IN__TARGET__CAPTURE_START;
cmdTargetCfgSet.TriggerChannel = 2;
cmdTargetCfgSet.reserved = 0;

rc =
 G_Lvds_TriggerIn_TargetConfig_Set(
  portHandleLvds,
  &cmdTargetCfgSet
 );

ErrorHandler(rc);
```

A complete code example for the LVDS Trigger Input functionality can be found in the samples folder of your **G-API** installation.

G_Lvds_TriggerIn_SourceConfig_Get

G_Lvds_TriggerIn_SourceConfig_Get — Get trigger input source configuration

Description

Use this command to query the trigger input source configuration.

Definition

```
G_Error_t
G_Lvds_TriggerIn_SourceConfig_Get(
 const G_PortHandle_t  portHandle,
 const G_Lvds_TriggerIn_SourceConfig_Get_Cmd_t * const cmd,
 G_Lvds_TriggerIn_SourceConfig_Get_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__TRIGGER_IN__SOURCE_CONFIG__GET__CMD_FLAG__NONE`
No flag is set

`TriggerChannel`
LVDS Trigger input channel (starting with **1**)

The source of the trigger input channel can be configured with G_Io_Trigger_Source_Set .

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_LVDS__TRIGGER_IN__SOURCE_CONFIG__GET__RSP_FLAG__NONE`
No flag is set

`G_LVDS__TRIGGER_IN__SOURCE_CONFIG__GET__RSP_FLAG__ENABLE`
The trigger channel is enabled

`TriggerChannel`
LVDS Trigger input channel (starting with **1**)

The source of the trigger input channel can be configured with G_Io_Trigger_Source_Set .

`Polarity`
Trigger channel polarity

The following values are possible:

`G_LVDS__POLARITY__LOW_ACTIVE`
The polarity is 'low active'

`G_LVDS__POLARITY__HIGH_ACTIVE`
The polarity is 'high active'

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_TriggerIn_SourceConfig_Set

G_Lvds_TriggerIn_SourceConfig_Set — Set trigger input source configuration

Description

Use this command to set the trigger input source configuration.

Definition

```
G_Error_t
G_Lvds_TriggerIn_SourceConfig_Set(
 const G_PortHandle_t  portHandle,
 const G_Lvds_TriggerIn_SourceConfig_Set_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__TRIGGER_IN__SOURCE_CONFIG__SET__CMD_FLAG__NONE`
No flag is set

`G_LVDS__TRIGGER_IN__SOURCE_CONFIG__SET__CMD_FLAG__ENABLE`
Enable trigger channel

`G_LVDS__TRIGGER_IN__SOURCE_CONFIG__SET__CMD_FLAG__SET_POLARITY`
Set polarity value defined by parameter `Polarity`.

If this flag is not set, parameter `Polarity` is disregarded.

`TriggerChannel`
LVDS Trigger input channel (starting with **1**)

The source of the trigger input channel can be configured with G_Io_Trigger_Source_Set .

`Polarity`
Trigger channel polarity

Used if flag `G_LVDS__TRIGGER_IN__SOURCE_CONFIG__SET__CMD_FLAG__SET_POLARITY` is set.

The following values are possible:

`G_LVDS__POLARITY__LOW_ACTIVE`
The polarity is 'low active'

`G_LVDS__POLARITY__HIGH_ACTIVE`
The polarity is 'high active'

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_TriggerIn_TargetConfig_Get

G_Lvds_TriggerIn_TargetConfig_Get — Get trigger input target configuration

Description

Use this command to query the trigger input target configuration.

Definition

```
G_Error_t
G_Lvds_TriggerIn_TargetConfig_Get(
 const G_PortHandle_t  portHandle,
 const G_Lvds_TriggerIn_TargetConfig_Get_Cmd_t * const cmd,
 G_Lvds_TriggerIn_TargetConfig_Get_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__TRIGGER_IN__TARGET_CONFIG__GET__CMD_FLAG__NONE`
No flag is set

`LvdsChannel`
LVDS interface channel (starting with **0**)

`Target`
Trigger target

This is the action that is performed if the trigger is active.

The following values are possible:

`G_LVDS__TRIGGER_IN__TARGET__CAPTURE_START`
Start capture operation

`G_LVDS__TRIGGER_IN__TARGET__CAPTURE_STOP`
Stop capture operation

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_LVDS__TRIGGER_IN__TARGET_CONFIG__GET__RSP_FLAG__NONE`
No flag is set

`G_LVDS__TRIGGER_IN__TARGET_CONFIG__GET__RSP_FLAG__ENABLE`
The trigger channel is enabled

`LvdsChannel`
LVDS interface channel (starting with **0**)

`Target`
Trigger target

This is the action that is performed if the trigger is active.

The following values are possible:

`G_LVDS__TRIGGER_IN__TARGET__CAPTURE_START`
Start capture operation

`G_LVDS__TRIGGER_IN__TARGET__CAPTURE_STOP`
Stop capture operation

`TriggerChannel`
LVDS Trigger input channel (starting with **1**)

The source of the trigger input channel can be configured with G_Io_Trigger_Source_Set .

`reserved`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_TriggerIn_TargetConfig_Set

G_Lvds_TriggerIn_TargetConfig_Set — Set trigger input target configuration

Description

Use this command to set the trigger input target configuration.

Definition

```
G_Error_t
G_Lvds_TriggerIn_TargetConfig_Set(
 const G_PortHandle_t  portHandle,
 const G_Lvds_TriggerIn_TargetConfig_Set_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__TRIGGER_IN__TARGET_CONFIG__SET__CMD_FLAG__NONE`
No flag is set

`G_LVDS__TRIGGER_IN__TARGET_CONFIG__SET__CMD_FLAG__ENABLE`
Enable trigger channel

`LvdsChannel`
LVDS channel of trigger target (starting with `0`)

`Target`
Trigger target

This is the action that is performed if the trigger is active.

The following values are possible:

`G_LVDS__TRIGGER_IN__TARGET__CAPTURE_START`
Start capture operation

`G_LVDS__TRIGGER_IN__TARGET__CAPTURE_STOP`
Stop capture operation

`TriggerChannel`
LVDS Trigger input channel (starting with `1`)

The destination of the trigger input channel can be configured with G_Io_Trigger_Source_Set .

`reserved`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

9 Trigger Output Control

These functions provide control over the trigger output of the LVDS interface.

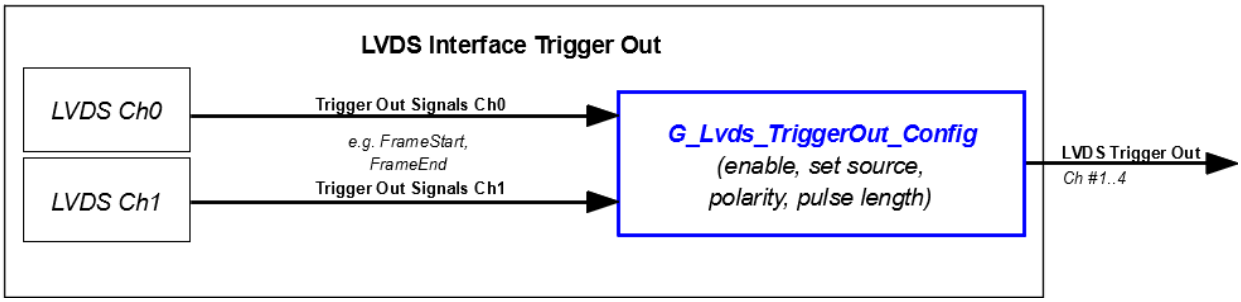

Figure 5 LVDS Trigger Output Overview

LVDS Trigger Output Example

LVDS Trigger Output Example — LVDS Trigger Output Example

Description

Frame Start Events of **LVDS Interface 1 channel 0** are configured to produce a 100 microseconds pulse on **digital output #1 .**

LVDS trigger output **#3** and LVDS trigger input channel **#2** will be used internally.

Block Diagram

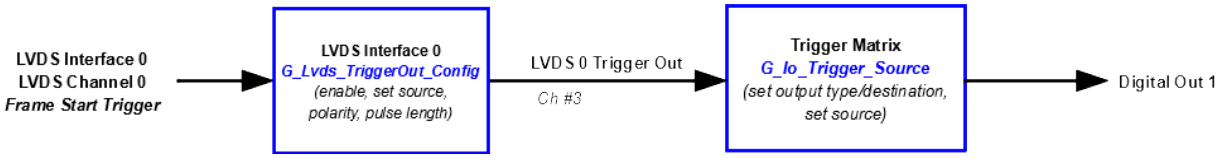

Figure 6 LVDS Trigger Output Example

Code Example:

```
G_Lvds_TriggerOut_Config_Set_Cmd_t  cmdCfgSet;
G_Error_t  rc;

cmdCfgSet.Flags =
 (G_LVDS__TRIGGER_OUT__CONFIG__SET__CMD__FLAG__ENABLE
 + G_LVDS__TRIGGER_OUT__CONFIG__SET__CMD__FLAG__SET_LVDS_CHANNEL
 + G_LVDS__TRIGGER_OUT__CONFIG__SET__CMD__FLAG__SET_SOURCE
 + G_LVDS__TRIGGER_OUT__CONFIG__SET__CMD__FLAG__SET_POLARITY
 + G_LVDS__TRIGGER_OUT__CONFIG__SET__CMD__FLAG__SET_TRIGGER_LENGTH);

cmdCfgSet.TriggerChannel = 3;
cmdCfgSet.LvdsChannel = 0;
cmdCfgSet.Source = (u8_t) G_LVDS__TRIGGER_OUT__SOURCE__FRAME_START;
cmdCfgSet.Polarity = (u8_t) G_LVDS__POLARITY__HIGH_ACTIVE;
cmdCfgSet.TriggerLength = 100;  // in microseconds

rc =
 G_Lvds_TriggerOut_Config_Set(
  portHandleLvds,
  &cmdCfgSet
 );

ErrorHandler(rc);

rc =
 G_Io_Trigger_Source_Set(
  portHandleIo,
  G_IO__TRIGGER__OUTPUT_TYPE__DIGITAL_OUT,
  2,  // output number (starting with 1)
  G_IO__TRIGGER__SOURCE_TYPE__LVDS_1_TRIGGER_OUT,
  3  // source number (starting with 1)
 );

ErrorHandler(rc);
```

A complete code example for the LVDS Trigger Output functionality can be found in the samples folder of your **G-API** installation.

G_Lvds_TriggerOut_Config_Get

G_Lvds_TriggerOut_Config_Get — Get trigger output configuration

Description

Use this command to query the trigger output configuration.

Definition

```
G_Error_t
G_Lvds_TriggerOut_Config_Get(
 const G_PortHandle_t  portHandle,
 const G_Lvds_TriggerOut_Config_Get_Cmd_t * const cmd,
 G_Lvds_TriggerOut_Config_Get_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__TRIGGER_OUT__CONFIG__GET__CMD_FLAG__NONE`
No flag is set

`TriggerChannel`
LVDS Trigger output channel (starting with **1**)

The source of the trigger output channel can be configured with G_Io_Trigger_Source_Set .

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_LVDS__TRIGGER_OUT__CONFIG__GET__RSP_FLAG__NONE`
No flag is set

`G_LVDS__TRIGGER_OUT__CONFIG__GET__RSP_FLAG__ENABLE`
The trigger channel is enabled

`TriggerChannel`
LVDS Trigger output channel (starting with **1**)

The source of the trigger output channel can be configured with G_Io_Trigger_Source_Set .

`LvdsChannel`
LVDS interface channel (starting with **0**)

`Source`
Source for trigger out

The following values are possible:

`G_LVDS__TRIGGER_OUT__SOURCE__NO_SOURCE`
No source selected

`G_LVDS__TRIGGER_OUT__SOURCE__FRAME_START`
Frame start event is trigger source

`G_LVDS__TRIGGER_OUT__SOURCE__FIRST_FRAME_START`
Only the first frame start event is the trigger source

`G_LVDS__TRIGGER_OUT__SOURCE__LAST_FRAME_END`
End of last frame is trigger source

`Polarity`
Trigger channel polarity

The following values are possible:

`G_LVDS__POLARITY__LOW_ACTIVE`
The polarity is 'low active'

`G_LVDS__POLARITY__HIGH_ACTIVE`
The polarity is 'high active'

`TriggerLength`
Length of trigger in micro seconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Lvds_TriggerOut_Config_Set

G_Lvds_TriggerOut_Config_Set — Set trigger output configuration

Description

Use this command to set the trigger output configuration.

Definition

```
G_Error_t
G_Lvds_TriggerOut_Config_Set(
 const G_PortHandle_t  portHandle,
 const G_Lvds_TriggerOut_Config_Set_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_LVDS__TRIGGER_OUT__CONFIG__SET__CMD_FLAG__NONE`
No flag is set

`G_LVDS__TRIGGER_OUT__CONFIG__SET__CMD__FLAG__ENABLE`
Enable trigger channel

`G_LVDS__TRIGGER_OUT__CONFIG__SET__CMD__FLAG__SET_LVDS_CHANNEL`
Set LVDS channel value defined by parameter `LvdsChannel`.

If this flag is not set, parameter `LvdsChannel` is disregarded.

`G_LVDS__TRIGGER_OUT__CONFIG__SET__CMD__FLAG__SET_SOURCE`
Set source value defined by parameter `Source`.

If this flag is not set, parameter `Source` is disregarded.

`G_LVDS__TRIGGER_OUT__CONFIG__SET__CMD__FLAG__SET_POLARITY`
Set polarity value defined by parameter `Polarity`.

If this flag is not set, parameter `Polarity` is disregarded.

`G_LVDS__TRIGGER_OUT__CONFIG__SET__CMD__FLAG__SET_TRIGGER_LENGTH`
Set trigger pulse length value defined by parameter `TriggerLength`.

If this flag is not set, parameter `TriggerLength` is disregarded.

`TriggerChannel`
LVDS Trigger output channel (starting with 1)

The source of the trigger output channel can be configured with G_Io_Trigger_Source_Set .

`LvdsChannel`
LVDS channel of trigger source (starting with **0**)

Used if flag `G_LVDS__TRIGGER_OUT__CONFIG__SET__CMD__FLAG__SET_LVDS_CHANNEL` is set.

`Source`
Source signal for trigger

Used if flag `G_LVDS__TRIGGER_OUT__CONFIG__SET__CMD__FLAG__SET_SOURCE` is set.

The following values are possible:

`G_LVDS__TRIGGER_OUT__SOURCE__NO_SOURCE`
No source selected

`G_LVDS__TRIGGER_OUT__SOURCE__FRAME_START`
Frame start event is trigger source

`G_LVDS__TRIGGER_OUT__SOURCE__FIRST_FRAME_START`
Only the first frame start event is the trigger source

`G_LVDS__TRIGGER_OUT__SOURCE__LAST_FRAME_END`
End of last frame is trigger source

`Polarity`
Trigger channel polarity

Used if flag `G_LVDS__TRIGGER_OUT__CONFIG__SET__CMD__FLAG__SET_POLARITY` is set.

The following values are possible:

`G_LVDS__POLARITY__LOW_ACTIVE`
The polarity is 'low active'

`G_LVDS__POLARITY__HIGH_ACTIVE`
The polarity is 'high active'

`TriggerLength`
Pulse length of trigger in micro seconds

Used if flag `G_LVDS__TRIGGER_OUT__CONFIG__SET__CMD__FLAG__SET_TRIGGER_LENGTH` is set.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

MOST Functions

These **G-API** commands provide the functionality of the MOST hardware. If the return value of type **G_Error_t** does not correspond to `G_NO_ERROR`, then an error has occurred. The error description can be called by the command G_GetLastErrorDescription .

1 Common

G_Most_InitInterface

G_Most_InitInterface — Initialize the interface

Description

This command resets the selected MOST interface without software reset into initial state.

Definition

```
G_Error_t
G_Most_InitInterface(
 const u32_t portHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
rc = G_Most_InitInterface(portHandle);
```

The hardware interface given by `portHandle` is initialized.

2 AMS

These commands are used to control the **A**pplication **M**essage **S**ervice (**AMS**).

G_Most_Ams_GetChannel

G_Most_Ams_GetChannel — Request the AMS channel

Description

This command requests an AMS channel.

This command is used for the dynamic admistration of AMS channels. An AMS channel is requested whose concrete channel number is returned in `channel`.

This AMS channel administration may be required if several applications or software threads are working with the firmware and are using the same AMS channels.

If always only one application works with the firmware, the AMS channel administration is sufficient and no administration by the firmware is required.

Definition

```
G_Error_t
G_Most_Ams_GetChannel(
 const G_PortHandle_t portHandle,
 u8_t * const channel
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Pointer to the variable `channel`

The number of the assigned AMS channel (starting with 0) is returned in this variable.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Request the AMS channel by **G_Most_Ams_GetChannel**

```
u8_t channel;

…

rc =
G_Most_Ams_GetChannel(
portHandle,
&channel
);

if (rc == G_NO_ERROR) {

…

}
```

In this example the required variable is declared first and subsequently transferred to the function **G_Most_Ams_GetChannel** additionally to the port handle. Its address has to be transferred in this context too. This is necessary, because the related return value is saved in this variable. As for `channel` it contains on function return the number of the assigned channel. If the return code `rc` with the value `G_NO_ERROR` indicates an error-free execution, the contents of the variable can be evaluated.

G_Most_Ams_GetChannel_Async

G_Most_Ams_GetChannel_Async — Request an AMS channel (by asynchronous request)

Description

This command initiates the asynchronous request of an AMS channel.

In contrast to G_Most_Ams_GetChannel , this command is returned immediately (that means without waiting for the response). As soon as the corresponding response is available, the callback function set by G_Most_Ams_GetChannel_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Most_Ams_GetChannel_Async(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Ams_GetChannel_Async_Callback

G_Most_Ams_GetChannel_Async_Callback — Callback function for the asynchronous request of an AMS channel

Description

This function will be automatically called if a response for the asynchronous request of an AMS channel with G_Most_Ams_GetChannel_Async is available. It must be entered by the user and set with G_Most_Ams_GetChannel_Async_AddCallback.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Most_Ams_GetChannel_Async_Callback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: AMS channel (starting with 0)

G_Most_Ams_GetChannel_Async_AddCallback

G_Most_Ams_GetChannel_Async_AddCallback — Set the callback function for the asynchronous request of an AMS channel

Description

This command defines a callback function for the asynchronous request of an AMS channel with G_Most_Ams_GetChannel_Async.

Definition

```
G_Error_t
G_Most_Ams_GetChannel_Async_AddCallback(
 const G_PortHandle_t portHandle,
 const G_Most_Ams_GetChannel_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`callback`
Function pointer of the callback function (see G_Most_Ams_GetChannel_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Ams_GetChannel_Async_RemoveCallback

G_Most_Ams_GetChannel_Async_RemoveCallback — Remove the callback function for the asynchronous request of an AMS channel

Description

This commmand removes a callback function for the asynchronous request of an AMS channel.

Definition

```
G_Error_t
G_Most_Ams_GetChannel_Async_RemoveCallback(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Ams_FreeChannel

G_Most_Ams_FreeChannel — Release the AMS channel

Description

This command releases an AMS channel.

This command is used for the dynamic administration of AMS channels. The AMS channel defined under `channel` is released.

This AMS channel administration may be required if several applications or software threads are working with the firmware using the same AMS channels.

If always only one application works with the firmware, an AMS channel administration in the application is sufficient and no administration by the firmware is required.

Definition

```
G_Error_t
G_Most_Ams_FreeChannel(
 const G_PortHandle_t portHandle,
 const u8_t channel
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
AMS channel (starting with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Release the AMS channel with **G_Most_Ams_FreeChannel**

```
u8_t channel;

…

channel = 0;            //AMS channel 0

rc =
G_Most_Ams_FreeChannel(
portHandle,
channel
);
```

In this example the required variable is declared, and after this the desired channel is entered. Subsequently it is transferred to the function **G_Most_Ams_FreeChannel** additionally to the port handle.

G_Most_Ams_GetState

G_Most_Ams_GetState — Query the MOST status

Description

This command queries the current MOST status.

Definition

```
G_Error_t
G_Most_Ams_GetState(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 G_Most_Status_t * const status,
 G_Most_Ams_GetState_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
AMS channel (starting with 0)

`status`
Returns MOST Node Status

The following return values are available:

- `G_MOST__STATUS__OK` - Status: OK
- `G_MOST__STATUS__UNKNOWN` - Status: Unknown
- `G_MOST__STATUS__INIC__FORMAT_FAILURE`

  INIC status: Format failure
- `G_MOST__STATUS__INIC__NETWORK_OFF`

  INIC status: Network off
- `G_MOST__STATUS__INIC__TIME_EXPIRATION`

  INIC status: Time expired
- `G_MOST__STATUS__INIC__RESET`

  INIC status: Reset
- `G_MOST__STATUS__INIC__ERROR_REPORT`

  INIC status: Error Report
- `G_MOST__STATUS__MOST__WRONG_TARGET`

  MOST status: Wrong target
- `G_MOST__STATUS__MOST__AT_LEAST_ONE_TX_SUCCESS`

  MOST status: At least one successful transmission
- `G_MOST__STATUS__MOST__BAD_CRC`

  MOST status: Wrong CRC
- `G_MOST__STATUS__MOST__BUFFER_FULL`

  MOST status: Buffer full
- `G_MOST__STATUS__MOST_HIGH__REQUEST_CONNECTION_FAILURE`

MOST-HIGH status: Failure while connection request
- G_MOST__STATUS__MOST_HIGH__CONNECTION_REJECTED_BY_DSI

MOST-HIGH status: Connection rejected by the receiver (DSI = data sink)
- G_MOST__STATUS__MOST_HIGH__UNSUPPORTED_REVISION_ID

MOST-HIGH status: Revision number unsupported
- G_MOST__STATUS__MOST_HIGH__CONNECTION_TERMINATED_BY_DSI

MOST-HIGH status: Connection terminated by the receiver (DSI = data sink)
- G_MOST__STATUS__MOST_HIGH__TRANS_TIMEOUT

MOST-HIGH status: Timeout while transmission

(see MOST-HIGH specification: Timer Trans)
- G_MOST__STATUS__MOST_HIGH__RESET

MOST-HIGH status: Reset
- G_MOST__STATUS__AMS__UNRESOLVED_TARGET__UNKNOWN

AMS status: Unknown error while target resolution
- G_MOST__STATUS__AMS__UNRESOLVED_TARGET__TX_ERROR

AMS status: Transmission error while target resolution
- G_MOST__STATUS__AMS__UNRESOLVED_TARGET__WRONG_ANSWER

AMS status: Wrong response received while target resolution
- G_MOST__STATUS__AMS__UNRESOLVED_TARGET__ERROR_ANSWER

AMS status: An error has occurred during target resolution
- G_MOST__STATUS__AMS__TIMEOUT

AMS status: Timeout
- G_MOST__STATUS__I2C__UNKNOWN

I2C status: Unknown
- G_MOST__STATUS__I2C__NO_ACK__ADDRESSING__WRITE

I2C status: No acknowledge for write instruction during addressing
- G_MOST__STATUS__I2C__NO_ACK__ADDRESSING__READ

I2C status: No acknowledge for read instruction during addressing
- G_MOST__STATUS__I2C__NO_ACK__WRITE

I2C status: No acknowledge for write instruction
- G_MOST__STATUS__I2C__TIMEOUT__ADDRESSING__WRITE

I2C status: Timeout for write instruction during addressing
- G_MOST__STATUS__I2C__TIMEOUT__ADDRESSING__READ

I2C status: Timeout for read instruction during addressing
- G_MOST__STATUS__I2C__TIMEOUT__WRITE

I2C status: Timeout for write instruction
- G_MOST__STATUS__I2C__TIMEOUT__READ

I2C status: Timeout for read instruction
- G_MOST__STATUS__INIC_BOOT_MODE__RESPONSE_TIMEOUT

Inic Boot Mode Status: Response timeout

- G_MOST__STATUS__INIC_BOOT_MODE__COMMAND_FAILED

  Inic Boot Mode Status: Command failed
- G_MOST__STATUS__FLASH__CRC_ERROR

  Flash Status: CRC Error
- G_MOST__STATUS__FLASH__FW_IMAGE_NOT_LOADED

  Flash Status: Firmware image not loaded
- G_MOST__STATUS__FLASH__FW_IMAGE_TOO_SMALL

  Flash Status: Firmware image too small
- G_MOST__STATUS__FLASH__CS_IMAGE_NOT_LOADED

  Flash Status: Configuration String image not loaded
- G_MOST__STATUS__FLASH__CS_IMAGE_TOO_SMALL

  Flash Status: Configuration String image too small
- G_MOST__STATUS__FLASH__INVALID_PROGRAMMING_IMAGE_LENGTH

  Flash Status: Invalid programming image length
- G_MOST__STATUS__FLASH__INVALID_PROGRAMMING_IMAGE

  Flash Status: Invalid programming image
- G_MOST__STATUS__FLASH__UNKNOWN_MEMORY_IMAGE

  Flash Status: Unknown memory image
- G_MOST__STATUS__FLASH__INVALID_MEMORY_IMAGE_LENGTH

  Flash Status: Invalid memory image length

`rspFlags`
Pointer to a variable of type **G_Most_Ams_GetState_RspFlags_t**

The following return flags are available:

- G_MOST__AMS__GET_STATE__RSP_FLAG__NONE

  No flag was set
- G_MOST__AMS__GET_STATE__RSP_FLAG__BUSY

  The message transmission could not be completed yet OR the response has not been received yet
- G_MOST__AMS__GET_STATE__RSP_FLAG__FREE

  AMS channel free

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Request the MOST status with **G_Most_Ams_GetState**

```
u8_t channel;
G_Most_Status_t status;
G_Most_Ams_GetState_RspFlags_t rspFlags;

…

channel = 0;            //AMS channel 0

rc =
 G_Most_Ams_GetState(
  portHandle,
  channel,
  &status,
  &rspFlags
 );

if (rc == G_NO_ERROR) {

…

}
```

In this example the required variables are declared first and then the desired values are entered. These are transferred to the function **G_Most_Ams_GetState** additionally to the port handle. In the case of certain variables it is required to transfer their addresses too. This is necessary because the related return values are saved in these variables. As for example `status` it contains on function return the current MOST status. If the return code `rc` with the value `G_NO_ERROR` indicates an error-free execution, the contents of the variables can be evaluated.

G_Most_Ams_GetState_Async

G_Most_Ams_GetState_Async — Query the MOST status (by asynchronous query)

Description

This command initiates the asynchronous query of the MOST status.

In contrast to G_Most_Ams_GetState , this command returns immediately (that means without waiting for the response). As soon as the corresponding response is available, the callback function defined by G_Most_Ams_GetState_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Most_Ams_GetState_Async(
 const G_PortHandle_t portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: AMS channel

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Ams_GetState_Async_Callback

G_Most_Ams_GetState_Async_Callback — Callback function for the asynchronous query of the MOST status

Description

This function will be automatically called, if a response for the asynchronous query of the MOST status with G_Most_Ams_GetState_Async is available. It must be entered by the user and set with G_Most_Ams_GetState_Async_AddCallback.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Most_Ams_GetState_Async_Callback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Most_Status_t  status,
 const G_Most_Ams_GetState_RspFlags_t  rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
AMS channel

`status`
MOST Node Status

For a description see MOST Node Status

`rspFlags`
Variable of type **G_Most_Ams_GetState_RspFlags_t**

The following return flags are available:

- `G_MOST__AMS__GET_STATE__RSP_FLAG__NONE`

  No flag was set
- `G_MOST__AMS__GET_STATE__RSP_FLAG__BUSY`

  The message transmission could not be completed yet OR the response has not been received yet
- `G_MOST__AMS__GET_STATE__RSP_FLAG__FREE`

  AMS channel free

G_Most_Ams_GetState_Async_AddCallback

G_Most_Ams_GetState_Async_AddCallback — Set the callback function for the asynchronous query of the MOST status

Description

This command defines a callback function the asynchronous query of the MOST status with G_Most_Ams_GetState_Async.

Definition

```
G_Error_t
G_Most_Ams_GetState_Async_AddCallback(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Most_Ams_GetState_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
AMS channel

`callback`
Function pointer of the callback function (see G_Most_Ams_GetState_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Ams_GetState_Async_RemoveCallback

G_Most_Ams_GetState_Async_RemoveCallback — Remove the callback function for the asynchronous query of the MOST status

Description

This command removes a callback function for the asynchronous query of the MOST status.

Definition

```
G_Error_t
G_Most_Ams_GetState_Async_RemoveCallback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: AMS channel

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Ams_Request

G_Most_Ams_Request — Send the AMS request

Description

This command sends an AMS request.

Definition

```
G_Error_t
G_Most_Ams_Request(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Most_Ams_Request_Parameters_t * const requestParameters,
 G_Most_Ams_GetResponse_Response_t * const responseStruct
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
AMS channel (starting with 0)

`requestParameters`
Pointer to a variable of type **G_Most_Ams_Request_Parameters_t**

The containing "reserved" bytes have to be initialized with `0`.

The structure **G_Most_Ams_Request_Parameters_t** consists of the following items:

- `Timeout`
  Timeout in microseconds
- `TargetDeviceId`
  MOST target address
- `FBlockId`
  Function block number
- `InstanceId`
  Function block instance
- `FunctionId`
  Function identifier
- `OperationType`
  Variable of type **G_Most_OperationType_t**

  The following return values are available

  (see MOST specification):
  - `G_MOST__OPERATION_TYPE__SET`
  - `G_MOST__OPERATION_TYPE__START`
  - `G_MOST__OPERATION_TYPE__GET`
  - `G_MOST__OPERATION_TYPE__ABORT`
  - `G_MOST__OPERATION_TYPE__SET_GET`
  - `G_MOST__OPERATION_TYPE__START_RESULT`
  - `G_MOST__OPERATION_TYPE__INCREMENT`
  - `G_MOST__OPERATION_TYPE__DECREMENT`
  - `G_MOST__OPERATION_TYPE__GET_INTERFACE`

- G_MOST__OPERATION_TYPE__START_RESULT_ACK
- G_MOST__OPERATION_TYPE__ABORT_ACK
- G_MOST__OPERATION_TYPE__START_ACK

- MessageType
  Variable of type **G_Most_Ams_MessageType_t**

  The following return values are available:
  - G_MOST__AMS__MESSAGE_TYPE__STANDARD

    Normal control message(s) with **TelId 0x00..0x03**
  - G_MOST__AMS__MESSAGE_TYPE__MOST_HIGH_CONTROL_CHANNEL

    MOST-HIGH protocol on the MOST control channel
  - G_MOST__AMS__MESSAGE_TYPE__MOST_HIGH_PACKET_CHANNEL

    MOST-HIGH protocol on the MOST packet data channel
- ResponseMode
  Variable of type **G_Most_Ams_ResponseMode_t**

  The following return values are available:
  - G_MOST__AMS__RESPONSE_MODE__WITHOUT_RESPONSE - Without response
  - G_MOST__AMS__RESPONSE_MODE__WITH_RESPONSE - With response
- reserved1
- reserved2
- reserved3
- DataLength
  Size of the request data
- Data
  Buffer with request data

  This buffer is set by the declaration of a variable of type

  G_Most_Ams_Request_Parameters_t

  in a standard size of G_MOST_MAX_DATA_SIZE.

responseStruct
Pointer to a variable of type **G_Most_Ams_GetResponse_Response_t**

The return values are stored in this variable.

The structure **G_Most_Ams_GetResponse_Response_t**

consists of the following items:

- Flags
  Variable of type **G_Most_Ams_GetResponse_RspFlags_t**

  The following return flags are available:
  - G_MOST__AMS__GET_RESPONSE__RSP_FLAG__NONE

    No flag was set
  - G_MOST__AMS__GET_RESPONSE__RSP_FLAG__VALID

    Valid response received
  - G_MOST__AMS__GET_RESPONSE__RSP_FLAG__BUSY

    The sending of the message could not be completed yet

    OR

    the response has not been received yet

- `status`
  MOST Node Status

  For a description see MOST Node Status
- `SourceDeviceId`
  MOST source address
- `FBlockId`
  Function block number
- `InstanceId`
  Function block instance
- `FunctionId`
  Function identifier
- `OperationType`
  Variable of type **G_Most_OperationType_t**

  The following return values are available (see MOST specification):
  - `G_MOST__OPERATION_TYPE__ERROR_ACK`
  - `G_MOST__OPERATION_TYPE__PROCESSING_ACK`
  - `G_MOST__OPERATION_TYPE__PROCESSING`
  - `G_MOST__OPERATION_TYPE__STATUS`
  - `G_MOST__OPERATION_TYPE__RESULT`
  - `G_MOST__OPERATION_TYPE__RESULT_ACK`
  - `G_MOST__OPERATION_TYPE__INTERFACE`
  - `G_MOST__OPERATION_TYPE__ERROR`
- `MessageType`
  Variable of type **G_Most_Ams_MessageType_t**

  The following return values are available:
  - `G_MOST__AMS__MESSAGE_TYPE__STANDARD`

    Normal control message(s) with **TelId 0x00..0x03**
  - `G_MOST__AMS__MESSAGE_TYPE__MOST_HIGH_CONTROL_CHANNEL`

    MOST-HIGH protocol on the MOST control channel
  - `G_MOST__AMS__MESSAGE_TYPE__MOST_HIGH_PACKET_CHANNEL`

    MOST-HIGH protocol on the MOST packet data channel
- `DataLength`

  - Out: Response data size
- `Data`
  Data buffer for saving the response data. This is set by declaration of a variable of type **G_Most_Ams_GetResponse_Response_t** with the standard size of `G_MOST_MAX_DATA_SIZE`.

  If a buffer size other than `G_MOST_MAX_DATA_SIZE` is needed, you need to define it via `G_MOST_MAX_DATA_SIZE` in your source code before including **g_api_most.h** .

Be sure to initialize the `DataLength` of the response structure before calling this function.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Send the AMS request with **G_Most_Ams_Request**

```
u8_t channel;
G_Most_Ams_Request_Parameters_t requestParameters;
G_Most_Ams_GetResponse_Response_t responseStruct;

…

channel = 0;                          //AMS channel 0
requestParameters.Timeout = 123456;
requestParameters.TargetDeviceId = 0xFFFF;
requestParameters.FBlockId = 0x1;
requestParameters.InstanceId = 0x1;
requestParameters.FunctionId = 0x0;
requestParameters.OperationType = G_MOST__OPERATION_TYPE__GET;
requestParameters.MessageType =
G_MOST__AMS__MESSAGE_TYPE__STANDARD;
requestParameters.ResponseMode =
G_MOST__AMS__RESPONSE_MODE__WITH_RESPONSE;
requestParameters.Data[0] = 0x1;
requestParameters.Data[1] = 0x2;
requestParameters.DataLength = 2;

requestParameters.reserved1 = 0;  //has to be zero
requestParameters.reserved2 = 0;  //has to be zero
requestParameters.reserved3 = 0;  //has to be zero

rc =
G_Most_Ams_Request(
portHandle,
channel,
&requestParameters,
&responseStruct
);

if (rc == G_NO_ERROR) {

…

}
```

In this example, first the required variables are declared and then the required values and parameters are entered into the corresponding variables. Only two bytes of `requestParameters.Data` are used, which is reflected in the data length `requestParameters.DataLength`. These parameters are subsequently transferred to the function **G_Most_Ams_Request** additionally to the port handle. In this context, the addresses of the variables `requestParameters` and `responseStruct` are indicated. This is required for `responseStruct` because the corresponding return values are stored in this variable. The command execution is terminated after getting a response or an error. If the return code `rc` with the value `G_NO_ERROR` indicates an error-free execution, the contents of the variables can be evaluated.

G_Most_Ams_QueueRequest

G_Most_Ams_QueueRequest — Send the AMS request

Description

This command enqueues an AMS request into the sending operation of the specified hardware.

Please consider that on function return it cannot be ensured that the corresponding request has been transmitted by the hardware. The request is queued. If you want to make sure that the request has been sent it is necessary to use G_Most_Ams_Request.

Definition

```
G_Error_t
G_Most_Ams_QueueRequest(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Most_Ams_Request_Parameters_t * const requestParameters,
 G_Most_Ams_Request_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
AMS channel (starting with 0)

`requestParameters`
Pointer to a variable of type **G_Most_Ams_Request_Parameters_t**

The containing "reserved" bytes have to be initialized with `0`.

The structure **G_Most_Ams_Request_Parameters_t** consists of the following items:

- `Timeout`
  Timeout in microseconds
- `TargetDeviceId`
  MOST target address
- `FBlockId`
  Function block number
- `InstanceId`
  Function block instance
- `FunctionId`
  Function identifier
- `OperationType`
  Variable of type **G_Most_OperationType_t**

  The following return values are available (see MOST specification):
  - `G_MOST__OPERATION_TYPE__SET`
  - `G_MOST__OPERATION_TYPE__START`
  - `G_MOST__OPERATION_TYPE__GET`
  - `G_MOST__OPERATION_TYPE__ABORT`

  - G_MOST__OPERATION_TYPE__SET_GET
  - G_MOST__OPERATION_TYPE__START_RESULT
  - G_MOST__OPERATION_TYPE__INCREMENT
  - G_MOST__OPERATION_TYPE__DECREMENT
  - G_MOST__OPERATION_TYPE__GET_INTERFACE
  - G_MOST__OPERATION_TYPE__START_RESULT_ACK
  - G_MOST__OPERATION_TYPE__ABORT_ACK
  - G_MOST__OPERATION_TYPE__START_ACK
- MessageType
  Variable of type **G_Most_Ams_MessageType_t**

  The following return values are available:
  - G_MOST__AMS__MESSAGE_TYPE__STANDARD

    Normal control message(s) with **TelId 0x00..0x03**
  - G_MOST__AMS__MESSAGE_TYPE__MOST_HIGH_CONTROL_CHANNEL

    MOST-HIGH protocol on the MOST control channel
  - G_MOST__AMS__MESSAGE_TYPE__MOST_HIGH_PACKET_CHANNEL

    MOST-HIGH protocol on the MOST packet data channel
- ResponseMode
  Variable of type **G_Most_Ams_ResponseMode_t**

  The following return values are available:
  - G_MOST__AMS__RESPONSE_MODE__WITHOUT_RESPONSE - Without response
  - G_MOST__AMS__RESPONSE_MODE__WITH_RESPONSE - With response
- reserved1
- reserved2
- reserved3
- DataLength
  Size of the request data
- Data
  Buffer with request data. This buffer is set by the declaration of a variable of type **G_Most_Ams_Request_Parameters_t** in a standard size of G_MOST_MAX_DATA_SIZE.

rspFlags
Pointer to a variable of type **G_Most_Ams_Request_RspFlags_t**

The following flags are available:
- G_MOST__AMS__REQUEST__RSP_FLAG__NONE

  No flag was set
- G_MOST__AMS__REQUEST__RSP_FLAG__CANCELED_PREVIOUS

  The sending of the previous request was canceled by this sending request

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Send the AMS request with **G_Most_Ams_QueueRequest**

```
u8_t channel;
G_Most_Ams_Request_Parameters_t requestParameters;
G_Most_Ams_Request_RspFlags_t rspFlags;

…

channel = 0;                          //AMS channel 0
requestParameters.Timeout = 123456;
requestParameters.TargetDeviceId = 0xFFFF;
requestParameters.FBlockId = 0x1;
requestParameters.InstanceId = 0x1;
requestParameters.FunctionId = 0x0;
requestParameters.OperationType = G_MOST__OPERATION_TYPE__GET;
requestParameters.MessageType =
G_MOST__AMS__MESSAGE_TYPE__STANDARD;
requestParameters.ResponseMode =
G_MOST__AMS__RESPONSE_MODE__WITH_RESPONSE;
requestParameters.Data[0] = 0x1;
requestParameters.Data[1] = 0x2;
requestParameters.DataLength = 2;

requestParameters.reserved1 = 0;  //has to be zero
requestParameters.reserved2 = 0;  //has to be zero
requestParameters.reserved3 = 0;  //has to be zero

rc =
G_Most_Ams_QueueRequest(
portHandle,
channel,
&requestParameters,
&rspFlags
);

if (rc == G_NO_ERROR) {

…

}
```

In this example, first the required variables are declared and then the required values and parameters are entered into the corresponding variables. Only two bytes of `requestParameters.Data` are used, which is reflected in the data length `requestParameters.DataLength`. These parameters are subsequently transferred to the function **G_Most_Ams_QueueRequest** additionally to the port handle. In this context, the addresses of the variables `requestParameters` and `rspFlags` are indicated. This is required for `rspFlags` because the corresponding return values are saved in this variable. If the return code `rc` with the value `G_NO_ERROR` indicates an error-free execution, the contents of the variables can be evaluated.

G_Most_Ams_GetResponse

G_Most_Ams_GetResponse — Get the AMS response

Description

This command retrieves an AMS response from the specified hardware.

Definition

```
G_Error_t
G_Most_Ams_GetResponse(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 G_Most_Ams_GetResponse_Response_t * const responseStruct
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
AMS channel (starting with 0)

`responseStruct`
Pointer to a variable of type **G_Most_Ams_GetResponse_Response_t**

The return values are stored in this variable. The structure **G_Most_Ams_GetResponse_Response_t** consists of the following items:

- `Flags`
  Variable of type **G_Most_Ams_GetResponse_RspFlags_t**

  The following return flags are available:
  - `G_MOST__AMS__GET_RESPONSE__RSP_FLAG__NONE`

    No flag was set
  - `G_MOST__AMS__GET_RESPONSE__RSP_FLAG__VALID`

    Valid response received
  - `G_MOST__AMS__GET_RESPONSE__RSP_FLAG__BUSY`

    The sending of the message could not be completed yet OR the response was not yet received
- `status`
  MOST Node Status

  For a description see MOST Node Status
- `SourceDeviceId`
  MOST source address
- `FBlockId`
  Function block number
- `InstanceId`
  Function block instance
- `FunctionId`
  Function identifier
- `OperationType`
  Variable of type **G_Most_OperationType_t**

The following return values are available (see MOST specification):
- G_MOST__OPERATION_TYPE__ERROR_ACK
- G_MOST__OPERATION_TYPE__PROCESSING_ACK
- G_MOST__OPERATION_TYPE__PROCESSING
- G_MOST__OPERATION_TYPE__STATUS
- G_MOST__OPERATION_TYPE__RESULT
- G_MOST__OPERATION_TYPE__RESULT_ACK
- G_MOST__OPERATION_TYPE__INTERFACE
- G_MOST__OPERATION_TYPE__ERROR

- MessageType
  Variable of type **G_Most_Ams_MessageType_t**

  The following return values are available:
  - G_MOST__AMS__MESSAGE_TYPE__STANDARD

    Normal control message(s) with **TelId 0x00..0x03**
  - G_MOST__AMS__MESSAGE_TYPE__MOST_HIGH_CONTROL_CHANNEL

    MOST-HIGH protocol on the MOST control channel
  - G_MOST__AMS__MESSAGE_TYPE__MOST_HIGH_PACKET_CHANNEL

    MOST-HIGH protocol on the MOST packet data channel
- DataLength
  - Content on calling: Buffer size for response data
  - Content on returning: Response data size
- Data
  Data buffer for response data storage

  This buffer is set by declaration of a variable of type **G_Most_Ams_GetResponse_Response_t** with a standard size of G_MOST_MAX_DATA_SIZE.

Be sure to initialize DataLength of the response structure before calling this function.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Retrieve the AMS response with **G_Most_Ams_GetResponse**

```
u8_t channel;
G_Most_Ams_GetResponse_Response_t responseStruct;

…

channel = 0;    //AMS channel 0

rc =
G_Most_Ams_GetResponse(
portHandle,
channel,
&responseStruct
);

if (rc == G_NO_ERROR) {

…

}
```

In this example the required variables are declared first and the desired values are entered. These values are subsequently transferred to the function **G_Most_Ams_GetResponse** additionally to the port handle. The address of the variable `responseStruct` is indicated. This is necessary, because the related return values are saved in this variable. The command is executed immediately after getting a response or an error. If the return code `rc` with the value `G_NO_ERROR` indicates an error-free execution, the contents of the variables can be evaluated.

G_Most_Ams_GetResponse_Async_Enable

G_Most_Ams_GetResponse_Async_Enable — Activate the asynchronous receiving of AMS responses

Description

This command activates the asynchronous receiving of AMS responses. As soon as a response is available, the callback function set with G_Most_Ams_GetResponse_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Most_Ams_GetResponse_Async_Enable(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: AMS channel

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Ams_GetResponse_Async_Disable

G_Most_Ams_GetResponse_Async_Disable — Deactivate the asynchronous receiving of AMS responses

Description

This command deactivates the asynchronous receiving of AMS responses.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Most_Ams_GetResponse_Async_Disable(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
    Handle to the communication port

`channel`
    AMS channel

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Ams_GetResponse_Async_Callback

G_Most_Ams_GetResponse_Async_Callback — Callback function for the asynchronous receiving of AMS responses

Description

This function will be automatically called, if during asynchronous operation an AMS response is received. It must be entered by the user and be set with G_Most_Ams_GetResponse_Async_AddCallback . Furthermore, the asynchronous receiving of AMS responses has to be activated with G_Most_Ams_GetResponse_Async_Enable.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Most_Ams_GetResponse_Async_Callback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Most_Ams_GetResponse_Response_t * const responseStruct
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
AMS channel

`responseStruct`
Pointer to a variable of type **G_Most_Ams_GetResponse_Response_t**

The return values are stored in this variable. The structure **G_Most_Ams_GetResponse_Response_t** consists of the following items:

- `Flags`
  Variable of type **G_Most_Ams_GetResponse_RspFlags_t**

  The following return flags are available:
  - `G_MOST__AMS__GET_RESPONSE__RSP_FLAG__NONE`

    No flag was set
  - `G_MOST__AMS__GET_RESPONSE__RSP_FLAG__VALID`

    Valid response received
  - `G_MOST__AMS__GET_RESPONSE__RSP_FLAG__BUSY`

    The sending of the message could not be completed yet OR the response was not yet received
- `status`
  MOST Node Status

  For a description see MOST Node Status
- `SourceDeviceId`
  MOST source address
- `FBlockId`
  Function block number

- `InstanceId`
  Function block instance
- `FunctionId`
  Function identifier
- `OperationType`
  Variable of type **G_Most_OperationType_t**

  The following return values are available (see MOST specification):
  - `G_MOST__OPERATION_TYPE__ERROR_ACK`
  - `G_MOST__OPERATION_TYPE__PROCESSING_ACK`
  - `G_MOST__OPERATION_TYPE__PROCESSING`
  - `G_MOST__OPERATION_TYPE__STATUS`
  - `G_MOST__OPERATION_TYPE__RESULT`
  - `G_MOST__OPERATION_TYPE__RESULT_ACK`
  - `G_MOST__OPERATION_TYPE__INTERFACE`
  - `G_MOST__OPERATION_TYPE__ERROR`
- `MessageType`
  Variable of type **G_Most_Ams_MessageType_t**

  The following return values are available:
  - `G_MOST__AMS__MESSAGE_TYPE__STANDARD`

    Normal control message(s) with **TelId 0x00..0x03**
  - `G_MOST__AMS__MESSAGE_TYPE__MOST_HIGH_CONTROL_CHANNEL`

    MOST-HIGH protocol on the MOST control channel
  - `G_MOST__AMS__MESSAGE_TYPE__MOST_HIGH_PACKET_CHANNEL`

    MOST-HIGH protocol on the MOST packet data channel
- `DataLength`
  - Content on calling: Buffer size for response data
  - Content on returning: Response data size
- `Data`
  Data buffer for response data storage

  This buffer is set by the declaration of a variable of type **G_Most_Ams_GetResponse_Response_t** with a standard size of `G_MOST_MAX_DATA_SIZE`.

G_Most_Ams_GetResponse_Async_AddCallback

G_Most_Ams_GetResponse_Async_AddCallback — Set the callback function for the asynchronous receiving of AMS responses

Description

This command defines a callback function for the asynchronous receiving of AMS responses.

Definition

```
G_Error_t
G_Most_Ams_GetResponse_Async_AddCallback(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Most_Ams_GetResponse_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
AMS channel

`callback`
Function pointer of the callback function (see G_Most_Ams_GetResponse_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Ams_GetResponse_Async_RemoveCallback

G_Most_Ams_GetResponse_Async_RemoveCallback — Remove the callback function for the asynchronous receiving of AMS responses

Description

This command removes a callback function for the asynchronous receiving of AMS responses.

Definition

```
G_Error_t
G_Most_Ams_GetResponse_Async_RemoveCallback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: AMS channel

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Ams_RxMsgBuffer_Config

G_Most_Ams_RxMsgBuffer_Config — Configure the message receiving buffer

Description

This command configures the AMS message receiving buffer.

Definition

```
G_Error_t
G_Most_Ams_RxMsgBuffer_Config(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Most_Ams_RxMsgBuffer_Mode_t mode,
 const u32_t bufferSize
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
AMS channel (starting with 0)

`mode`
Variable of type **G_Most_Ams_RxMsgBuffer_Mode_t** indicating the buffer mode

The following return values are available:

- `G_MOST__AMS__RX_MSG_BUFFER__MODE__DISABLE`

  Deactivate the buffering of messages received
- `G_MOST__AMS__RX_MSG_BUFFER__MODE__UNEXPECTED_MESSAGES`

  Buffering of unexpected messages
- `G_MOST__AMS__RX_MSG_BUFFER__MODE__ALL_MESSAGES`

  Buffering of all messages received

`bufferSize`
Buffer size in bytes

If `0` is transferred, the Default buffer size will be used. This allows the buffering of at least two messages with 64 kbytes of data each.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Configure the message receiving buffer with **G_Most_Ams_RxMsgBuffer_Config**

```
u8_t channel;
G_Most_Ams_RxMsgBuffer_Mode_t mode;
u32_t bufferSize;

…

channel = 0;                          //AMS channel 0
mode = G_MOST__AMS__RX_MSG_BUFFER__MODE__ALL_MESSAGES;
bufferSize = 0;

rc =
G_Most_Ams_RxMsgBuffer_Config(
portHandle,
channel,
mode,
bufferSize
);
```

In this example, first the required variables are declared and the required values and parameters are entered into the corresponding variables. These are subsequently transferred to the function **G_Most_Ams_RxMsgBuffer_Config** additionally to the port handle.

G_Most_Ams_RxMsgBuffer_GetMsgs_Async_Enable

G_Most_Ams_RxMsgBuffer_GetMsgs_Async_Enable — Activate the asynchronous receiving of AMS messages

Description

This command activates the asynchronous receiving of AMS messages. As soon as a message is available, the callback function set with G_Most_Ams_RxMsgBuffer_GetMsgs_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Most_Ams_RxMsgBuffer_GetMsgs_Async_Enable(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: AMS channel

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Ams_RxMsgBuffer_GetMsgs_Async_Disable

G_Most_Ams_RxMsgBuffer_GetMsgs_Async_Disable — Deactivate the asynchronous receiving of AMS messages

Description

This command deactivates the asynchronous receiving of AMS messages.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Most_Ams_RxMsgBuffer_GetMsgs_Async_Disable(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
    Handle to the communication port

`channel`
    AMS channel

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Ams_RxMsgBuffer_GetMsgs_Async_Callback

G_Most_Ams_RxMsgBuffer_GetMsgs_Async_Callback — Callback function for the asynchronous receiving of AMS messages

Description

This function will be automatically called, if during asynchronous operation an AMS message is received. It has to be entered by the user and be set with G_Most_Ams_RxMsgBuffer_GetMsgs_Async_AddCallback . Furthermore, the asynchronous receiving of AMS messages must be activated with G_Most_Ams_RxMsgBuffer_GetMsgs_Async_Enable.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Most_Ams_RxMsgBuffer_GetMsgs_Async_Callback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Most_Ams_RxMsgBuffer_GetMsgs_Async_RxMsg_t * const rxMsg,
 const G_Most_Ams_RxMsgBuffer_GetMsgs_Async_Flags_t  flags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
AMS channel

`rxMsg`
Pointer to a variable of type **G_Most_Ams_RxMsgBuffer_GetMsgs_Async_RxMsg_t**

The return values are stored in this variable. The structure **G_Most_Ams_RxMsgBuffer_GetMsgs_Async_RxMsg_t** consists of the following items:

- `SourceDeviceId`
  MOST source address
- `FBlockId`
  Function block number
- `InstanceId`
  Function block instance
- `FunctionId`
  Function identifier
- `OperationType`
  Variable of type **G_Most_OperationType_t**

  The following return values are available (see MOST specification):
  - `G_MOST__OPERATION_TYPE__ERROR_ACK`
  - `G_MOST__OPERATION_TYPE__PROCESSING_ACK`
  - `G_MOST__OPERATION_TYPE__PROCESSING`
  - `G_MOST__OPERATION_TYPE__STATUS`
  - `G_MOST__OPERATION_TYPE__RESULT`
  - `G_MOST__OPERATION_TYPE__RESULT_ACK`
  - `G_MOST__OPERATION_TYPE__INTERFACE`
  - `G_MOST__OPERATION_TYPE__ERROR`

- `MessageType`
  Variable of type **G_Most_Ams_MessageType_t**

  The following return values are available:
  - `G_MOST__AMS__MESSAGE_TYPE__STANDARD`

    Normal control message(s) with **TelId 0x00..0x03**
  - `G_MOST__AMS__MESSAGE_TYPE__MOST_HIGH_CONTROL_CHANNEL`

    MOST-HIGH protocol on the MOST control channel
  - `G_MOST__AMS__MESSAGE_TYPE__MOST_HIGH_PACKET_CHANNEL`

    MOST-HIGH protocol on the MOST packet data channel
- `Length`
  Response data size
- `Data`
  Response data

`flags`
Variable of type **G_Most_Ams_RxMsgBuffer_GetMsgs_Async_Flags_t**

The following return flags are available:
- `G_MOST__AMS__RX_MSG_BUFFER__GET_MSGS__ASYNC__FLAG__NONE`

  No flag was set
- `G_MOST__AMS__RX_MSG_BUFFER__GET_MSGS__ASYNC__FLAG__MSG_NOT_COMPLETE`

  The received message is uncompleted
- `G_MOST__AMS__RX_MSG_BUFFER__GET_MSGS__ASYNC__FLAG__MSG_LOST`

  The receiving of a message is failed

G_Most_Ams_RxMsgBuffer_GetMsgs_Async_AddCallback

G_Most_Ams_RxMsgBuffer_GetMsgs_Async_AddCallback — Set the callback function for the asynchronous receiving of AMS messages

Description

This command defines a callback function for the asynchronous receiving of AMS messages.

Definition

```
G_Error_t
G_Most_Ams_RxMsgBuffer_GetMsgs_Async_AddCallback(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Most_Ams_RxMsgBuffer_GetMsgs_Async_Callback_t  callback
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: AMS channel

`callback`
: Function pointer of the callback function (see G_Most_Ams_RxMsgBuffer_GetMsgs_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Ams_RxMsgBuffer_GetMsgs_Async_RemoveCallback

G_Most_Ams_RxMsgBuffer_GetMsgs_Async_RemoveCallback — Remove the callback function for the asynchronous receiving of AMS messages

Description

This command removes a callback function for the asynchronous receiving of AMS messages.

Definition

```
G_Error_t
G_Most_Ams_RxMsgBuffer_GetMsgs_Async_RemoveCallback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: AMS channel

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

3 CMS

These commands are used to control the **C**ontrol **M**essage **S**ervice (**CMS**).

G_Most_Cms_GetTxState

G_Most_Cms_GetTxState — Query the sending status

Description

This command queries the current CMS sending status.

Definition

```
G_Error_t
G_Most_Cms_GetTxState(
 const G_PortHandle_t portHandle,
 G_Most_Status_t * const status,
 G_Most_Cms_GetTxState_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`status`
Returns MOST Node Status (see MOST Node Status)

`rspFlags`
Pointer to a variable of type **G_Most_Cms_GetTxState_RspFlags_t**

The following return flags are available:

- `G_MOST__CMS__GET_TX_STATE__RSP_FLAG__NONE`

  No flag was set
- `G_MOST__CMS__GET_TX_STATE__RSP_FLAG__BUSY`

  The sending of the message could not be completed yet

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Cms_GetTxState_Async

G_Most_Cms_GetTxState_Async — Query the CMS sending status (by asynchronous query)

Description

This command initiates the asynchronous query of the CMS sending status.

In contrast to G_Most_Cms_GetTxState, this command returns immediately (that means without waiting for the response). As soon as the corresponding response is available, the callback function set by G_Most_Cms_GetTxState_Async_AddCallback_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Most_Cms_GetTxState_Async(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Cms_GetTxState_Async_Callback

G_Most_Cms_GetTxState_Async_Callback — Callback function for the asynchronous query of the CMS sending status

Description

This function will be automatically called, if a response for the asynchronous query of the CMS sending status with G_Most_Cms_GetTxState_Async is available. It has to be entered by the user and set with G_Most_Cms_GetTxState_Async_AddCallback_Async_AddCallback.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Most_Cms_GetTxState_Async_Callback(
 const G_PortHandle_t  portHandle,
 const G_Most_Status_t  status,
 const G_Most_Cms_GetTxState_RspFlags_t  rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`status`
Returns MOST Node Status (see MOST Node Status)

`rspFlags`
Variable of type **G_Most_Cms_GetTxState_RspFlags_t**

The following return flags are available:

- `G_MOST__CMS__GET_TX_STATE__RSP_FLAG__NONE`

  No flag was set
- `G_MOST__CMS__GET_TX_STATE__RSP_FLAG__BUSY`

  The sending of the message could not be completed yet

G_Most_Cms_GetTxState_Async_AddCallback_Async_AddCallback

G_Most_Cms_GetTxState_Async_AddCallback_Async_AddCallback — Set the callback function for the asynchronous query of the CMS sending status

Description

This command defines a callback function for the asynchronous query of the CMS sending status.

Definition

```
G_Error_t
G_Most_Cms_GetTxState_Async_AddCallback(
 const G_PortHandle_t portHandle,
 const G_Most_Cms_GetTxState_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`callback`
Function pointer of the callback function (see G_Most_Cms_GetTxState_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Cms_GetTxState_Async_RemoveCallback

G_Most_Cms_GetTxState_Async_RemoveCallback — Remove the callback function for the asynchronous query of the CMS sending status

Description

This command removes a callback function for the asynchronous query of the CMS sending status.

Definition

```
G_Error_t
G_Most_Cms_GetTxState_Async_RemoveCallback(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Cms_SendMessage

G_Most_Cms_SendMessage — Send a CMS message

Description

This command sends a CMS message.

Definition

```
G_Error_t
G_Most_Cms_SendMessage(
 const G_PortHandle_t portHandle,
 const u32_t timeout,
 const G_Most_Cms_Message_Parameters_t * const messageParameters,
 const u8_t * const data,
 const u32_t dataLength,
 G_Most_Status_t * const status,
 G_Most_Cms_SendMessage_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`timeout`
Sending timeout (in microseconds)

`messageParameters`
Pointer to a variable of type **G_Most_Cms_Message_Parameters_t**

The structure **G_Most_Cms_Message_Parameters_t** consists of the following items:

- `TargetDeviceId`
  MOST target address
- `FBlockId`
  Function block number
- `InstanceId`
  Function block instance
- `FunctionId`
  Function identifier
- `OperationType`
  Variable of type **G_Most_OperationType_t**

  The following return values are available (see MOST specification):
  - `G_MOST__OPERATION_TYPE__SET`
  - `G_MOST__OPERATION_TYPE__START`
  - `G_MOST__OPERATION_TYPE__GET`
  - `G_MOST__OPERATION_TYPE__ABORT`
  - `G_MOST__OPERATION_TYPE__SET_GET`
  - `G_MOST__OPERATION_TYPE__START_RESULT`
  - `G_MOST__OPERATION_TYPE__INCREMENT`
  - `G_MOST__OPERATION_TYPE__DECREMENT`
  - `G_MOST__OPERATION_TYPE__GET_INTERFACE`
  - `G_MOST__OPERATION_TYPE__START_RESULT_ACK`
  - `G_MOST__OPERATION_TYPE__ABORT_ACK`
  - `G_MOST__OPERATION_TYPE__START_ACK`

- `TelId`
  Telegram identifier (0x00..0x0F)

`data`
Data buffer for data bytes

`dataLength`
Number of data bytes

`status`
Returns MOST Node Status (see MOST Node Status)

`rspFlags`
Pointer to a variable of type **G_Most_Cms_SendMessage_RspFlags_t**

The following return flags are available:

- `G_MOST__CMS__SEND_MESSAGE__RSP_FLAG__NONE`

  No flag was set
- `G_MOST__CMS__SEND_MESSAGE__RSP_FLAG__CANCELED_PREVIOUS`

  The sending of the previous message was canceled by this sending request
- `G_MOST__CMS__SEND_MESSAGE__RSP_FLAG__BUSY`

  The sending of the message could not be completed yet

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Cms_QueueMessage

G_Most_Cms_QueueMessage — Send the CMS message

Description

This command enqueues a CMS message into the sending operation of the specified hardware.

Please consider that on function return it cannot be ensured that the corresponding message has been transmitted. The message is only queued. If you want to ensure that the message has been sent, it is necessary to use G_Most_Cms_SendMessage .

Definition

```
G_Error_t
G_Most_Cms_QueueMessage(
 const G_PortHandle_t portHandle,
 const G_Most_Cms_Message_Parameters_t * const messageParameters,
 const u8_t * const data,
 const u32_t dataLength,
 G_Most_Cms_QueueMessage_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`messageParameters`
Pointer to a variable of type **G_Most_Cms_Message_Parameters_t**

The structure **G_Most_Cms_Message_Parameters_t** consists of the following items:

- `TargetDeviceId`
  MOST target address
- `FBlockId`
  Function block number
- `InstanceId`
  Function block instance
- `FunctionId`
  Function identifier
- `OperationType`
  Variable of type **G_Most_OperationType_t**

  The following return values are available (see MOST specification):
  - `G_MOST__OPERATION_TYPE__SET`
  - `G_MOST__OPERATION_TYPE__START`
  - `G_MOST__OPERATION_TYPE__GET`
  - `G_MOST__OPERATION_TYPE__ABORT`
  - `G_MOST__OPERATION_TYPE__SET_GET`
  - `G_MOST__OPERATION_TYPE__START_RESULT`
  - `G_MOST__OPERATION_TYPE__INCREMENT`
  - `G_MOST__OPERATION_TYPE__DECREMENT`
  - `G_MOST__OPERATION_TYPE__GET_INTERFACE`
  - `G_MOST__OPERATION_TYPE__START_RESULT_ACK`
  - `G_MOST__OPERATION_TYPE__ABORT_ACK`
  - `G_MOST__OPERATION_TYPE__START_ACK`
- `TelId`
  Telegram identifier (0x00..0x0F)

`data`
Data buffer for data bytes

`dataLength`
Number of data bytes

`rspFlags`
Pointer to a variable of type **G_Most_Cms_QueueMessage_RspFlags_t**

The following return flags are available:

- `G_MOST__CMS__QUEUE_MESSAGE__RSP_FLAG__NONE`

  No flag was set
- `G_MOST__CMS__QUEUE_MESSAGE__RSP_FLAG__CANCELED_PREVIOUS`

  The sending of the previous message was canceled by this sending request

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

4 Diag

These commands are used for the diagnostics control.

G_Most_Diag_GetState

G_Most_Diag_GetState — Get the diagnostics status

Description

This command is used to query the actual diagnostics status.

Definition

```
G_Error_t
G_Most_Diag_GetState(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Most_Diag_GetState_CmdFlags_t cmdFlags,
 G_Error_t * const lastErrorCode,
 G_Most_Diag_Type_t * const type,
 G_Most_Diag_State_t * const state,
 G_Most_Diag_GetState_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`cmdFlags`
Variable of type **G_Most_Diag_GetState_CmdFlags_t**

The following flags are possible:

- `G_MOST__DIAG__GET_STATE__CMD_FLAG__NONE`

  No flag, if no other has to be transferred
- `G_MOST__DIAG__GET_STATE__CMD_FLAG__RESET_LAST_ERROR`

  Reset the error code of the last error that has occurred in the firmware

`lastErrorCode`
Pointer to a variable of type **G_Error_t**

The error code of the last error that has occurred in the firmware (or `G_NO_ERROR` = no error) is returned in this variable. For getting an error description you can call G_Common_GetFirmwareErrorDescription.

`type`
Variable of type **G_Most_Diag_Type_t**

The diagnostics type is returned in this variable.

The following return values are possible:

- `G_MOST__DIAG__TYPE__OFF` - No diagnostics
- `G_MOST__DIAG__TYPE__KW2000_TP_2_0` - KWP2000 on TP2.0
- `G_MOST__DIAG__TYPE__UDS_KEYWORD_REC` - UDS

  (Function block: Diagnosis, Function: Keyword_Rec)

`state`
Pointer to a variable of type **G_Most_Diag_State_t**

The actual diagnostics status is returned in this variable.

The following return values are possible:
- `G_MOST__DIAG__STATE__NOT_INITIALIZED`

  Not initialized
- `G_MOST__DIAG__STATE__DISCONNECTED`

  Disconnected
- `G_MOST__DIAG__STATE__CONNECT_IN_PROGRESS`

  The connecting is currently in progress.
- `G_MOST__DIAG__STATE__CONNECTED`

  Connected
- `G_MOST__DIAG__STATE__DISCONNECT_IN_PROGRESS`

  The disconnection is in progress.

`rspFlags`
Pointer to a variable of type **G_Most_Diag_GetState_RspFlags_t**

The following return flags are possible:
- `G_MOST__DIAG__GET_STATE__RSP_FLAG__NONE`

  No flag was set
- `G_MOST__DIAG__GET_STATE__RSP_FLAG__BUSY`

  A request has not been responded yet/successfully sent yet
- `G_MOST__DIAG__GET_STATE__RSP_FLAG__SYNC_RX_BUFFER_NOT_EMPTY`

  Synchronous receive buffer is not empty
- `G_MOST__DIAG__GET_STATE__RSP_FLAG__ASYNC_RX_BUFFER_NOT_EMPTY`

  Asynchronous receive buffer is not empty

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Diag_GetState_Async

G_Most_Diag_GetState_Async — Get the diagnostics status (by asynchronous operation)

Description

This command initiates the asynchronous query of the diagnostics status.

In contrast to G_Most_Diag_GetState, this command returns immediately (that means without waiting for the response). As soon as the corresponding response is available, the callback function set by G_Most_Diag_GetState_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Most_Diag_GetState_Async(
 const G_PortHandle_t portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Diag_GetState_Async_Callback

G_Most_Diag_GetState_Async_Callback — Callback function for the asynchronous query of the diagnostics status

Description

This function is automatically called if a response for the asynchronous query of the diagnostics status with G_Most_Diag_GetState_Async is available. It has to be entered by the user and set with G_Most_Diag_GetState_Async_AddCallback.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Most_Diag_GetState_Async_Callback(
 const G_PortHandle_t  portHandle,
 const u8_t channel,
 const G_Error_t  lastErrorCode,
 const G_Most_Diag_Type_t  type,
 const G_Most_Diag_State_t  state,
 const G_Most_Diag_GetState_RspFlags_t  rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`lastErrorCode`
Variable of type **G_Error_t**

It returns the error code of the last error which has occurred in the firmware (or `G_NO_ERROR` = no error). For getting an error description, you can call G_Common_GetFirmwareErrorDescription.

`type`
Variable of type **G_Most_Diag_Type_t**

It returns the diagnostics type.

The following return values are possible:

- `G_MOST__DIAG__TYPE__OFF` - No diagnostics
- `G_MOST__DIAG__TYPE__KW2000_TP_2_0` - KWP2000 on TP2.0
- `G_MOST__DIAG__TYPE__UDS_KEYWORD_REC` - UDS

  (Function block: Diagnosis, Function: Keyword_Rec)

`state`
Variable of type **G_Most_Diag_State_t**

It returns the current diagnostics status.

The following return values are possible:

- `G_MOST__DIAG__STATE__NOT_INITIALIZED`

  Not initialized
- `G_MOST__DIAG__STATE__DISCONNECTED`

  Disconnected
- `G_MOST__DIAG__STATE__CONNECT_IN_PROGRESS`

  The connecting is in progress.
- `G_MOST__DIAG__STATE__CONNECTED`

  Connected
- `G_MOST__DIAG__STATE__DISCONNECT_IN_PROGRESS`

  The disconnection is in progress.

`rspFlags`
Variable of type **G_Most_Diag_GetState_RspFlags_t**

The following return flags are possible:

- `G_MOST__DIAG__GET_STATE__RSP_FLAG__NONE`

  No flag was set
- `G_MOST__DIAG__GET_STATE__RSP_FLAG__BUSY`

  A request has not been responded yet/successfully sent yet
- `G_MOST__DIAG__GET_STATE__RSP_FLAG__SYNC_RX_BUFFER_NOT_EMPTY`

  Synchronous receive buffer is not empty
- `G_MOST__DIAG__GET_STATE__RSP_FLAG__ASYNC_RX_BUFFER_NOT_EMPTY`

  Asynchronous receive buffer is not empty

G_Most_Diag_GetState_Async_AddCallback

G_Most_Diag_GetState_Async_AddCallback — Set the callback function for the asynchronous query of the diagnostics state

Description

This command defines a callback function for the asynchronous query of the diagnostics status with G_Most_Diag_GetState_Async.

Definition

```
G_Error_t
G_Most_Diag_GetState_Async_AddCallback(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Most_Diag_GetState_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`callback`
Function pointer of the callback function (see G_Most_Diag_GetState_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Diag_GetState_Async_RemoveCallback

G_Most_Diag_GetState_Async_RemoveCallback — Remove the callback function for the asynchronous query of the MOST diagnostics status

Description

This command removes the callback function for the asynchronous query of the MOST diagnostics status.

Definition

```
G_Error_t
G_Most_Diag_GetState_Async_RemoveCallback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Diag_Config_Kw2000_Tp_2_0

G_Most_Diag_Config_Kw2000_Tp_2_0 — Configure the diagnostics function

Description

This command configures the diagnostics function for KWP2000 on TP2.0.

Definition

```
G_Error_t
G_Most_Diag_Config_Kw2000_Tp_2_0(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Most_Diag_Config_Mode_t mode,
 const u32_t globalTimeout,
 const G_Most_Diag_Config_CmdFlags_t cmdFlags,
 const u16_t p2max,
 const u16_t repetitions
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`mode`
Variable of type **G_Most_Diag_Config_Mode_t**

The following return values are possible:

- `G_MOST__DIAG__CONFIG__MODE__SET_PARAMS`

  Set parameters, with initialization
- `G_MOST__DIAG__CONFIG__MODE__SET_FLAGS_ONLY_WITHOUT_INIT`

  Set global timeout and flags only, no initialization
- `G_MOST__DIAG__CONFIG__MODE__SET_PARAMS_WITHOUT_INIT`

  Set parameters, no initialization

`globalTimeout`
Global timeout in milliseconds (starts directly before sending the request and stops after the complete reception of the response or after the successful sending of the request if no response is expected, e.g. 10000 milliseconds)

`cmdFlags`
Variable of type **G_Most_Diag_Config_CmdFlags_t**

The following flags are possible:

- `G_MOST__DIAG__CONFIG__CMD_FLAG__NONE`

  No flag, if no other has to be transferred
- `G_MOST__DIAG__CONFIG__CMD_FLAG__DISABLE_21_HANDLING`

  The negative response `BusyRepeatRequest` is not handled.
- `G_MOST__DIAG__CONFIG__CMD_FLAG__DISABLE_23_HANDLING`

  The negative response `RoutineNotComplete` is not handled.
- `G_MOST__DIAG__CONFIG__CMD_FLAG__DISABLE_78_HANDLING`

  The negative response `RequestCorrectlyReceivedResponsePending` is not handled.
- `G_MOST__DIAG__CONFIG__CMD_FLAG__ALL_RESPONSES_AS_SYNC`

  Also unexpectedly received diagnostics responses are written in the common diagnostics receive buffer.

`p2max`
Timeout-information in milliseconds >

(maximum time between end of request and start of response, e.g. 600 milliseconds)

`repetitions`
Number of repetitions of request if the control unit does not react within the timeout-time (P2max) (e.g. 2)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Diag_Config_Uds_KeywordRec

G_Most_Diag_Config_Uds_KeywordRec — Configure the diagnostics

Description

This command configures the diagnostics function for UDS with KeywordRec.

Definition

```
G_Error_t
G_Most_Diag_Config_Uds_KeywordRec(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Most_Diag_Config_Mode_t mode,
 const u32_t globalTimeout,
 const G_Most_Diag_Config_CmdFlags_t cmdFlags,
 const G_Most_Diag_Config_Uds_KeywordRec_Parameters_t \
 * const udsKeywordRecParameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`mode`
Variable of type **G_Most_Diag_Config_Mode_t**

The following return values are possible:

- `G_MOST__DIAG__CONFIG__MODE__SET_PARAMS`

  Set parameters, with initialization
- `G_MOST__DIAG__CONFIG__MODE__SET_FLAGS_ONLY_WITHOUT_INIT`

  Set only global timeout and flags, no initialization
- `G_MOST__DIAG__CONFIG__MODE__SET_PARAMS_WITHOUT_INIT`

  Set parameters, no initialization

`globalTimeout`
Global timeout in milliseconds (starts directly before sending the request and stops after the complete reception of the response or after the successful sending of the request if no response is expected, e.g. 10000 milliseconds)

`cmdFlags`
A variable of type **G_Most_Diag_Config_CmdFlags_t**

The following flags are possible:
- `G_MOST__DIAG__CONFIG__CMD_FLAG__NONE`

  No command flag set
- `G_MOST__DIAG__CONFIG__CMD_FLAG__DISABLE_21_HANDLING`

  The negative response `BusyRepeatRequest` is not treated.
- `G_MOST__DIAG__CONFIG__CMD_FLAG__DISABLE_23_HANDLING`

  The negative response `RoutineNotComplete` is not treated.
- `G_MOST__DIAG__CONFIG__CMD_FLAG__DISABLE_78_HANDLING`

  The negative response `RequestCorrectlyReceivedResponsePending` is not treated.
- `G_MOST__DIAG__CONFIG__CMD_FLAG__ALL_RESPONSES_AS_SYNC`

  Also unexpectedly received diagnostics responses are written in the common diagnostics receive buffer.

`udsKeywordRecParameters`
Pointer to a variable of type **G_Most_Diag_Config_Uds_KeywordRec_Parameters_t**

The structure **G_Most_Diag_Config_Uds_KeywordRec_Parameters_t** consists of the following items:
- `Flags`
  Variable of type **G_Most_Diag_Config_Uds_KeywordRec_CmdFlags_t**

  The following flags are possible:
  - `G_MOST__DIAG__CONFIG__UDS_KEYWORD_REC__CMD_FLAG__NONE`

    No flag set
  - `G_MOST__DIAG__CONFIG__UDS_KEYWORD_REC__CMD_FLAG__ENABLE_TESTER_PRESENT`

    Activate tester present
- `P2max`
  Timeout information in milliseconds (maximum time between end of request and start of response, e.g. 200 milliseconds)
- `P3max`
  Extended timeout in msec (after getting the negative response

  `0x78: RequestCorrectlyReceivedResponsePending`, e.g. 5100 milliseconds)
- `Repetitions`
  Number of request repetitions if the control unit does not react within the timeout-time (P2max) (e.g. **2**)
- `reserved1`
- `reserved2`
- `TesterPresent`
  Structure for tester present

  The structure consists of the following items:
  - `CycleTime`
    Cycle time in milliseconds
  - `RequestMode`
    Variable of type **G_Most_Diag_RequestMode_t** indicating the request mode

    The following return values are possible:
    - `G_MOST__DIAG__REQUEST_MODE__PHYSICAL_WITH_RESPONSE`

      Physical request with response

  - `G_MOST__DIAG__REQUEST_MODE__FUNCTIONAL_WITH_RESPONSE`

    Functional request with response
  - `G_MOST__DIAG__REQUEST_MODE__PHYSICAL_WITHOUT_RESPONSE`

    Physical request without response
  - `G_MOST__DIAG__REQUEST_MODE__FUNCTIONAL_WITHOUT_RESPONSE`

    Functional request without response
- `DataLength`
  Number of data bytes
- `Data`
  Buffer with query data

  When declaring a variable of type **G_Most_Diag_Config_Uds_KeywordRec_Parameters_t** this buffer is set with `8` bytes.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Diag_Config_Off

G_Most_Diag_Config_Off — Deactivate the diagnostics function

Description

This command deactivates the diagnostics function.

Definition

```
G_Error_t
G_Most_Diag_Config_Off(
 const G_PortHandle_t portHandle,
 const u8_t channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

5 INIC

These commands are used to control the **I**ntelligent **N**etwork **I**nterface **C**ontroller (**INIC**).

Please refer to www.smsc-ais.com/AIS to get more information regarding INIC or INIC-API.

G_Most_Inic_GetState

G_Most_Inic_GetState — Get the INIC state

Description

This command requests the current INIC status.

Definition

```
G_Error_t
G_Most_Inic_GetState(
 const G_PortHandle_t portHandle,
 G_Most_Inic_LockState_t * const lockState,
 G_Most_Inic_EhciState_t * const ehciState,
 G_Most_Inic_NiState_t * const niState,
 G_Most_Inic_NcState_t * const ncState,
 G_Most_Inic_DeviceMode_t * const deviceMode
);
```

Parameters

`portHandle`
Handle to the communication port

`lockState`
Pointer to a variable of type **G_Most_Inic_LockState_t**

The following return values are available:
- `G_MOST__INIC__LOCK_STATE__UNKNOWN` - unknown lock state
- `G_MOST__INIC__LOCK_STATE__UNLOCK` - lock state 'unlocked'
- `G_MOST__INIC__LOCK_STATE__STABLE_LOCK` - lock state 'locked'

`ehciState`
Pointer to a variable of type **G_Most_Inic_EhciState_t**

The current status of the "external host controller interface" is returned in this variable.

The following return values are available:
- `G_MOST__INIC__EHCI_STATE__UNKNOWN` - Status unknown
- `G_MOST__INIC__EHCI_STATE__PROTECTED` - protected
- `G_MOST__INIC__EHCI_STATE__SEMI_PROTECTED` - semi protected
- `G_MOST__INIC__EHCI_STATE__ATTACHED` - attached (Normal state)

`niState`
Pointer to a variable of type **G_Most_Inic_NiState_t**

The current status of the "network interface" is returned in this variable.

The following return values are available:
- `G_MOST__INIC__NI_STATE__UNKNOWN` - Status unknown
- `G_MOST__INIC__NI_STATE__OFF` - off
- `G_MOST__INIC__NI_STATE__INIT` - init
- `G_MOST__INIC__NI_STATE__RING_BREAK_DIAG` - ring break diagnosis
- `G_MOST__INIC__NI_STATE__ON` - on

`ncState`
Pointer to a variable of type **G_Most_Inic_NcState_t**

The current status of the "network configuration state machine" is returned in this variable.

The following return values are available:
- `G_MOST__INIC__NC_STATE__UNKNOWN` - Status unknown
- `G_MOST__INIC__NC_STATE__NOT_OK` - not ok
- `G_MOST__INIC__NC_STATE__OK` - ok

`deviceMode`
Pointer to a variable of type **G_Most_Inic_DeviceMode_t**

The following return values are available:
- `G_MOST__INIC__DEVICE_MODE__UNKNOWN` - Status unknown
- `G_MOST__INIC__DEVICE_MODE__SLAVE` - Timing Slave
- `G_MOST__INIC__DEVICE_MODE__MASTER` - Timing Master

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Inic_GetState_Async

G_Most_Inic_GetState_Async — Get the INIC report (by asynchronous operation)

Description

This command initiates the asynchronous query of the INIC report.

In contrast to G_Most_Inic_GetState , this command returns immediately (that means without waiting for the response). As soon as the specific response is available, the callback function set with G_Most_Inic_GetState_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Most_Inic_GetState_Async(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Inic_GetState_Async_Callback

G_Most_Inic_GetState_Async_Callback — Callback function for the asynchronous query of the INIC status

Description

This function will be automatically called if a response for the asynchronous query of the INIC status with G_Most_Inic_GetState_Async is available. It must be entered by the user and set with G_Most_Inic_GetState_Async_AddCallback.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Most_Inic_GetState_Async_Callback(
 const G_PortHandle_t  portHandle,
 const G_Most_Inic_LockState_t  lockState,
 const G_Most_Inic_EhciState_t  ehciState,
 const G_Most_Inic_NiState_t  niState,
 const G_Most_Inic_NcState_t  ncState,
 const G_Most_Inic_DeviceMode_t  deviceMode
);
```

Parameters

`portHandle`
Handle to the communication port

`lockState`
Variable of type **G_Most_Inic_LockState_t**

The following return values are available:
- `G_MOST__INIC__LOCK_STATE__UNKNOWN` - unknown lock state
- `G_MOST__INIC__LOCK_STATE__UNLOCK` - lock state 'unlocked'
- `G_MOST__INIC__LOCK_STATE__STABLE_LOCK` - lock state 'locked'

`ehciState`
Variable of type **G_Most_Inic_EhciState_t**

The current status of the "external host controller interface" is returned in this variable.

The following return values are available:
- `G_MOST__INIC__EHCI_STATE__UNKNOWN` - Status unknown
- `G_MOST__INIC__EHCI_STATE__PROTECTED` - protected
- `G_MOST__INIC__EHCI_STATE__SEMI_PROTECTED` - semi protected
- `G_MOST__INIC__EHCI_STATE__ATTACHED` - attached (Normal state)

`niState`
Variable of type **G_Most_Inic_NiState_t**

The current state of the "network interface" is returned in this variable.

The following return values are available:
- `G_MOST__INIC__NI_STATE__UNKNOWN` - Status unknown
- `G_MOST__INIC__NI_STATE__OFF` - off
- `G_MOST__INIC__NI_STATE__INIT` - init

- G_MOST__INIC__NI_STATE__RING_BREAK_DIAG - ring break diagnosis
- G_MOST__INIC__NI_STATE__ON - on

ncState
Variable of type **G_Most_Inic_NcState_t**

The current status of the "network configuration state machine" is returned in this variable.

The following return values are available:
- G_MOST__INIC__NC_STATE__UNKNOWN - Status unknown
- G_MOST__INIC__NC_STATE__NOT_OK - not ok
- G_MOST__INIC__NC_STATE__OK - ok

deviceMode
Variable of type **G_Most_Inic_DeviceMode_t**

The following return values are available:
- G_MOST__INIC__DEVICE_MODE__UNKNOWN - Status unknown
- G_MOST__INIC__DEVICE_MODE__SLAVE - Timing Slave
- G_MOST__INIC__DEVICE_MODE__MASTER - Timing Master

G_Most_Inic_GetState_Async_AddCallback

G_Most_Inic_GetState_Async_AddCallback — Set the callback function for the asynchronous INIC status query

Description

This command defines a callback function for the asynchronous query of the INIC status with G_Most_Inic_GetState_Async.

Definition

```
G_Error_t
G_Most_Inic_GetState_Async_AddCallback(
 const G_PortHandle_t portHandle,
 const G_Most_Inic_GetState_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`callback`
Function pointer of the callback function (see G_Most_Inic_GetState_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Inic_GetState_Async_RemoveCallback

G_Most_Inic_GetState_Async_RemoveCallback — Remove the callback function for the asynchronous query of the INIC status

Description

This command removes a callback function for the asynchronous query of the INIC status.

Definition

```
G_Error_t
G_Most_Inic_GetState_Async_RemoveCallback(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Inic_Request

G_Most_Inic_Request — Send the command

Description

This command sends a command to the INIC function block.

Definition

```
G_Error_t
G_Most_Inic_Request(
 const G_PortHandle_t portHandle,
 const u32_t timeout,
 const u16_t functionId,
 const G_Most_OperationType_t operationType,
 const u8_t * const data,
 const u32_t dataLength,
 G_Most_Inic_Request_Response_t * const responseStruct,
 u8_t * const responseData,
 u32_t * const responseLength
);
```

Parameters

`portHandle`
Handle to the communication port

`timeout`
Sending timeout in milliseconds

`functionId`
Function identifier for functions within the INIC function block

`operationType`
Variable of type **G_Most_OperationType_t**

The following return values are available (see MOST specification):

- `G_MOST__OPERATION_TYPE__SET`
- `G_MOST__OPERATION_TYPE__START`
- `G_MOST__OPERATION_TYPE__GET`
- `G_MOST__OPERATION_TYPE__ABORT`
- `G_MOST__OPERATION_TYPE__SET_GET`
- `G_MOST__OPERATION_TYPE__START_RESULT`
- `G_MOST__OPERATION_TYPE__INCREMENT`
- `G_MOST__OPERATION_TYPE__DECREMENT`
- `G_MOST__OPERATION_TYPE__GET_INTERFACE`
- `G_MOST__OPERATION_TYPE__START_RESULT_ACK`
- `G_MOST__OPERATION_TYPE__ABORT_ACK`
- `G_MOST__OPERATION_TYPE__START_ACK`

`data`
Pointer to the data buffer for data bytes of the INIC command

`dataLength`
Number of data bytes

`responseStruct`
Pointer to a variable of type **G_Most_Inic_Request_Response_t**

The return values are saved in this variable.

The structure **G_Most_Inic_Request_Response_t** consists of the following items:

- `Flags`
  Variable of type **G_Most_Inic_Request_RspFlags_t**

  The following return flags are available:
  - `G_MOST__INIC__REQUEST__RSP_FLAG__NONE`

    No flag was set
  - `G_MOST__INIC__REQUEST__RSP_FLAG__VALID`

    Valid report received
  - `G_MOST__INIC__REQUEST__RSP_FLAG__BUSY`

    Transmission of the INIC command is not completed yet OR the INIC report is not received yet
  - `G_MOST__INIC__REQUEST__RSP_FLAG__TX_PENDING`

    Transmission of the INIC command is not terminated yet
  - `G_MOST__INIC__REQUEST__RSP_FLAG__RX_PENDING`

    INIC report is not received yet
  - `G_MOST__INIC__REQUEST__RSP_FLAG__TX_ERROR`

    While sending, an error has occurred (see parameter status)
- `Status`
  MOST Node Status (see MOST Node Status)
- `FunctionId`
  Pointer to the variable `functionId`

  The function identifier for functions within the INIC function block is returned in this variable.
- `OperationType`
  Pointer to a variable of type **G_Most_OperationType_t**

  The following return values are available (see MOST specification):
  - `G_MOST__OPERATION_TYPE__ERROR_ACK`
  - `G_MOST__OPERATION_TYPE__PROCESSING_ACK`
  - `G_MOST__OPERATION_TYPE__PROCESSING`
  - `G_MOST__OPERATION_TYPE__STATUS`
  - `G_MOST__OPERATION_TYPE__RESULT`
  - `G_MOST__OPERATION_TYPE__RESULT_ACK`
  - `G_MOST__OPERATION_TYPE__INTERFACE`
  - `G_MOST__OPERATION_TYPE__ERROR`
- `reserved`
  reserved parameter (must be initialized with **0**)

`responseData`
Pointer to the data buffer for data bytes of the INIC report

`responseLength`
Pointer to the variable `reponseLength`
- Content on call: Size of the buffer for data bytes
- Content on return: Number of data bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Inic_Request_NwStartup_StartResult

G_Most_Inic_Request_NwStartup_StartResult — Send the INIC command

Description

This command sends an INIC command with `NwStartup` and `StartResult`.

Definition

```
G_Error_t
G_Most_Inic_Request_NwStartup_StartResult(
 const G_PortHandle_t portHandle,
 const u32_t timeout,
 G_Most_Inic_Request_Response_t * const responseStruct,
 u8_t * const responseData,
 u32_t * const responseLength
);
```

Parameters

`portHandle`
Handle to the communication port

`timeout`
Send the timeout in milliseconds

`responseStruct`
Pointer to a variable of type **G_Most_Inic_Request_Response_t**

The return values are saved in this variable.

The structure **G_Most_Inic_Request_Response_t** consists of the following items:

- `Flags`
  Variable of type **G_Most_Inic_Request_RspFlags_t**

  The following return flags are available:
  - `G_MOST__INIC__REQUEST__RSP_FLAG__NONE`

    No flag was set
  - `G_MOST__INIC__REQUEST__RSP_FLAG__VALID`

    Valid report received
  - `G_MOST__INIC__REQUEST__RSP_FLAG__BUSY`

    Transmission of the INIC command not terminated yet OR INIC report not received yet
  - `G_MOST__INIC__REQUEST__RSP_FLAG__TX_PENDING`

    Transmission of the INIC command not terminated yet
  - `G_MOST__INIC__REQUEST__RSP_FLAG__RX_PENDING`

    INIC report not received yet
  - `G_MOST__INIC__REQUEST__RSP_FLAG__TX_ERROR`

    While transmission, an error has occurred (see parameter status)
- `Status`
  MOST Node Status (see MOST Node Status)
- `FunctionId`
  Pointer to the variable `functionId`

The function identifier for functions within the INIC function block is returned in this variable.

- `OperationType`
  Pointer to a variable of type **G_Most_OperationType_t**

  The following return values are available (see MOST Specification):
  - `G_MOST__OPERATION_TYPE__ERROR_ACK`
  - `G_MOST__OPERATION_TYPE__PROCESSING_ACK`
  - `G_MOST__OPERATION_TYPE__PROCESSING`
  - `G_MOST__OPERATION_TYPE__STATUS`
  - `G_MOST__OPERATION_TYPE__RESULT`
  - `G_MOST__OPERATION_TYPE__RESULT_ACK`
  - `G_MOST__OPERATION_TYPE__INTERFACE`
  - `G_MOST__OPERATION_TYPE__ERROR`
- `reserved`
  reserved parameter (must be initialized with **0**)

`responseData`
Pointer to the data buffer for data bytes of the INIC report

`responseLength`
Pointer to the variable `reponseLength`
- Content on calling: Buffer size for data bytes
- Content on returning: Number of data bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Inic_Request_NwShutDown_StartResult

G_Most_Inic_Request_NwShutDown_StartResult — Send the INIC command

Description

This command sends an INIC command with `NwShutDown` and `StartResult`.

Definition

```
G_Error_t
G_Most_Inic_Request_NwShutdown_StartResult(
 const G_PortHandle_t portHandle,
 const u32_t timeout,
 G_Most_Inic_Request_Response_t * const responseStruct,
 u8_t * const responseData,
 u32_t * const responseLength
);
```

Parameters

`portHandle`
Handle to the communication port

`timeout`
Sending timeout in milliseconds

`responseStruct`
Pointer to a variable of type **G_Most_Inic_Request_Response_t**

The return values are saved in this variable.

The structure **G_Most_Inic_Request_Response_t** consists of the following items:

- `Flags`
  Variable of type **G_Most_Inic_Request_RspFlags_t**

  The following return flags are available:
  - `G_MOST__INIC__REQUEST__RSP_FLAG__NONE`

    No flag was set
  - `G_MOST__INIC__REQUEST__RSP_FLAG__VALID`

    Valid report received
  - `G_MOST__INIC__REQUEST__RSP_FLAG__BUSY`

    The sending of the INIC command could not been completed yet OR INIC report was not received yet
  - `G_MOST__INIC__REQUEST__RSP_FLAG__TX_PENDING`

    The sending of the INIC command could not be completed yet
  - `G_MOST__INIC__REQUEST__RSP_FLAG__RX_PENDING`

    INIC report was not received yet
  - `G_MOST__INIC__REQUEST__RSP_FLAG__TX_ERROR`

    An error has occurred while sending (see parameter status)
- `Status`
  Pointer to a variable of type **G_Most_Status_t**

  The current MOST status is returned in this variable.

The following return values are available:

- G_MOST__STATUS__OK - Status: OK
- G_MOST__STATUS__UNKNOWN - Status: Unknown
- G_MOST__STATUS__INIC__FORMAT_FAILURE

  INIC status: Format failure
- G_MOST__STATUS__INIC__NETWORK_OFF

  INIC status: Network off
- G_MOST__STATUS__INIC__TIME_EXPIRATION

  INIC status: Time expired
- G_MOST__STATUS__MOST__WRONG_TARGET

  MOST status: Wrong target
- G_MOST__STATUS__MOST__AT_LEAST_ONE_TX_SUCCESS

  MOST status: At least one successful transmission
- G_MOST__STATUS__MOST__BAD_CRC

  MOST status: Wrong CRC
- G_MOST__STATUS__MOST__BUFFER_FULL

  MOST status: Buffer full
- G_MOST__STATUS__MOST_HIGH__REQUEST_CONNECTION_FAILURE

  MOST-HIGH status: Failure while connection request
- G_MOST__STATUS__MOST_HIGH__CONNECTION_REJECTED_BY_DSI

  MOST-HIGH status: Connection to the receiver rejected (DSI = data sink)
- G_MOST__STATUS__MOST_HIGH__UNSUPPORTED_REVISION_ID

  MOST-HIGH status: Revision number unsupported
- G_MOST__STATUS__MOST_HIGH__CONNECTION_TERMINATED_BY_DSI

  MOST-HIGH status: Connection to the receiver terminated (DSI = data sink)
- G_MOST__STATUS__MOST_HIGH__TRANS_TIMEOUT

  MOST-HIGH status: Timeout while transmission

  (see MOST-HIGH specification: Timer Trans)
- G_MOST__STATUS__MOST_HIGH__RESET

  MOST-HIGH status: Reset
- G_MOST__STATUS__AMS__UNRESOLVED_TARGET__UNKNOWN

  AMS status: Unknown error while target resolution
- G_MOST__STATUS__AMS__UNRESOLVED_TARGET__TX_ERROR

  AMS status: Transmission error while target resolution
- G_MOST__STATUS__AMS__UNRESOLVED_TARGET__WRONG_ANSWER

  AMS status: Wrong response while target resolution received
- G_MOST__STATUS__AMS__UNRESOLVED_TARGET__ERROR_ANSWER

  AMS status: Response received that an error has occurred while target resolution
- G_MOST__STATUS__AMS__TIMEOUT

  AMS status: Timeout

- `FunctionId`
  Pointer to the variable `functionId`

  The function identifier for functions within the INIC function block is returned in this variable.
- `OperationType`
  Pointer to a variable of type **G_Most_OperationType_t**

  The following return values are available (see MOST specification):
  - `G_MOST__OPERATION_TYPE__ERROR_ACK`
  - `G_MOST__OPERATION_TYPE__PROCESSING_ACK`
  - `G_MOST__OPERATION_TYPE__PROCESSING`
  - `G_MOST__OPERATION_TYPE__STATUS`
  - `G_MOST__OPERATION_TYPE__RESULT`
  - `G_MOST__OPERATION_TYPE__RESULT_ACK`
  - `G_MOST__OPERATION_TYPE__INTERFACE`
  - `G_MOST__OPERATION_TYPE__ERROR`
- `reserved`
  reserved parameter (must be initialized with **0**)

`responseData`
Pointer to the data buffer for data bytes of the INIC report

`responseLength`
Pointer to the variable `reponseLength`
- Content on calling: Buffer size for data bytes
- Content on returning: Number of data bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Inic_QueueRequest

G_Most_Inic_QueueRequest — Queue INIC request

Description

This command sends a command to the INIC function block.

Please consider that on function return it cannot be ensured that the corresponding command has been transmitted. The command is only transferred to the hardware. If you want to ensure that the command has been transmitted, it is necessary to use G_Most_Inic_Request .

Definition

```
G_Error_t
G_Most_Inic_QueueRequest(
 const G_PortHandle_t portHandle,
 const u16_t functionId,
 const G_Most_OperationType_t operationType,
 const u8_t telId,
 const u8_t * const data,
 const u32_t dataLength
);
```

Parameters

`portHandle`
Handle to the communication port

`functionId`
Function identifier for functions within the INIC function block

`operationType`
Variable of type **G_Most_OperationType_t**

The following return values are available (see MOST specification):

- `G_MOST__OPERATION_TYPE__SET`
- `G_MOST__OPERATION_TYPE__START`
- `G_MOST__OPERATION_TYPE__GET`
- `G_MOST__OPERATION_TYPE__ABORT`
- `G_MOST__OPERATION_TYPE__SET_GET`
- `G_MOST__OPERATION_TYPE__START_RESULT`
- `G_MOST__OPERATION_TYPE__INCREMENT`
- `G_MOST__OPERATION_TYPE__DECREMENT`
- `G_MOST__OPERATION_TYPE__GET_INTERFACE`
- `G_MOST__OPERATION_TYPE__START_RESULT_ACK`
- `G_MOST__OPERATION_TYPE__ABORT_ACK`
- `G_MOST__OPERATION_TYPE__START_ACK`

`telId`
Telegram identifier, has always to be set to **0**

`data`
Pointer to the data buffer for data bytes of the INIC command

`dataLength`
Number of data bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Inic_GetReport

G_Most_Inic_GetReport — Report request

Description

This command requests the report from the INIC function block.

Definition

```
G_Error_t
G_Most_Inic_GetReport(
 const G_PortHandle_t portHandle,
 G_Most_Inic_GetReport_RspFlags_t * const rspFlags,
 G_Most_Status_t * const status,
 u16_t * const functionId,
 G_Most_OperationType_t * const operationType,
 u8_t * const telId,
 u8_t * const responseData,
 u32_t * const responseLength
);
```

Parameters

`portHandle`
Handle to the communication port

`rspFlags`
Pointer to the variable of type **G_Most_Inic_GetReport_RspFlags_t**

The following return flags are available:

- `G_MOST__INIC__GET_REPORT__RSP_FLAG__NONE`

  No flag was set
- `G_MOST__INIC__GET_REPORT__RSP_FLAG__VALID`

  Valid report received
- `G_MOST__INIC__GET_REPORT__RSP_FLAG__BUSY`

  The sending of the INIC command could not be completed yet OR INIC report was not received yet
- `G_MOST__INIC__GET_REPORT__RSP_FLAG__TX_PENDING`

  The sending of the INIC command could not be completed yet
- `G_MOST__INIC__GET_REPORT__RSP_FLAG__RX_PENDING`

  INIC report was not received yet
- `G_MOST__INIC__GET_REPORT__RSP_FLAG__TX_ERROR`

  An error has occurred while sending (see parameter status)

`status`
Returns MOST Node Status (see MOST Node Status)

`functionId`
Pointer to the variable `functionId`

The function identifier for functions within the INIC function block is returned in this variable.

`operationType`
Pointer to the variable of type **G_Most_OperationType_t**

The following return values are available (see MOST specification):
- `G_MOST__OPERATION_TYPE__ERROR_ACK`
- `G_MOST__OPERATION_TYPE__PROCESSING_ACK`
- `G_MOST__OPERATION_TYPE__PROCESSING`
- `G_MOST__OPERATION_TYPE__STATUS`
- `G_MOST__OPERATION_TYPE__RESULT`
- `G_MOST__OPERATION_TYPE__RESULT_ACK`
- `G_MOST__OPERATION_TYPE__INTERFACE`
- `G_MOST__OPERATION_TYPE__ERROR`

`telId`
Telegram identifier

`responseData`
Pointer to the data buffer for databytes of the INIC report

`responseLength`
Pointer to the variable `reponseLength`
- Content on calling: Buffer size for data bytes
- Content on returning: Number of data bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Inic_GetReport_Async

G_Most_Inic_GetReport_Async — Query the INIC report (by asynchronous query)

Description

This command initiates the asynchronous query of the INIC report.

In contrast to G_Most_Inic_GetReport, this command returns immediately (that means without waiting for the response). As soon as the corresponding response is available, the callback function set by G_Most_Inic_GetReport_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Most_Inic_GetReport_Async(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Inic_GetReport_Async_Callback

G_Most_Inic_GetReport_Async_Callback — Asynchronous request of the callback function for the INIC report

Description

This function will be automatically called if a response for the asynchronous request of the INIC report with G_Most_Inic_GetReport_Async is available. It must be entered by the user and be set with G_Most_Inic_GetReport_Async_AddCallback

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Most_Inic_GetReport_Async_Callback(
 const G_PortHandle_t  portHandle,
 const G_Most_Inic_GetReport_RspFlags_t  rspFlags,
 const G_Most_Status_t  status,
 const u16_t  functionId,
 const G_Most_OperationType_t  operationType,
 const u8_t  telId,
 const u8_t * responseData,
 const u32_t  responseLength
);
```

Parameters

`portHandle`
Handle to the communication port

`rspFlags`
Variable of type **G_Most_Inic_GetReport_RspFlags_t**

The following return flags are available:

- `G_MOST__INIC__GET_REPORT__RSP_FLAG__NONE`

  No flag was set
- `G_MOST__INIC__GET_REPORT__RSP_FLAG__VALID`

  Valid report received
- `G_MOST__INIC__GET_REPORT__RSP_FLAG__BUSY`

  The sending of the INIC command could not be completed yet OR INIC report was not received yet
- `G_MOST__INIC__GET_REPORT__RSP_FLAG__TX_PENDING`

  The sending of the INIC command could not be completed yet
- `G_MOST__INIC__GET_REPORT__RSP_FLAG__RX_PENDING`

  INIC report was not received yet
- `G_MOST__INIC__GET_REPORT__RSP_FLAG__TX_ERROR`

  An error has occurred while sending (see parameter status)

`status`
Returns MOST Node Status (see MOST Node Status)

`functionId`
Function identifier for functions within the INIC function block

`operationType`
Variable of type **G_Most_OperationType_t**

The following return values are available (see MOST specification):
- `G_MOST__OPERATION_TYPE__ERROR_ACK`
- `G_MOST__OPERATION_TYPE__PROCESSING_ACK`
- `G_MOST__OPERATION_TYPE__PROCESSING`
- `G_MOST__OPERATION_TYPE__STATUS`
- `G_MOST__OPERATION_TYPE__RESULT`
- `G_MOST__OPERATION_TYPE__RESULT_ACK`
- `G_MOST__OPERATION_TYPE__INTERFACE`
- `G_MOST__OPERATION_TYPE__ERROR`

`telId`
Telegram identifier

`responseData`
Pointer to the data buffer for data bytes of the INIC report

`responseLength`
Number of data bytes

G_Most_Inic_GetReport_Async_AddCallback

G_Most_Inic_GetReport_Async_AddCallback — Set the callback function for the asynchronous request of the INIC report

Description

This command set a callback function for the asynchronous request of the INIC report with G_Most_Inic_GetReport_Async.

Definition

```
G_Error_t
G_Most_Inic_GetReport_Async_AddCallback(
 const G_PortHandle_t portHandle,
 const G_Most_Inic_GetReport_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`callback`
Function pointer to the callback function (see G_Most_Inic_GetReport_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Inic_GetReport_Async_RemoveCallback

G_Most_Inic_GetReport_Async_RemoveCallback — Remove the callback function for the asynchronous request of the INIC report

Description

This command removes a callback function for the asynchronous request of the INIC report.

Definition

```
G_Error_t
G_Most_Inic_GetReport_Async_RemoveCallback(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

6 Node

These commands are used to control the MOST interface as MOST node.

G_Most_Node_SetProperties

G_Most_Node_SetProperties — Set the node properties

Description

This command defines the properties of the MOST node.

Definition

```
G_Error_t
G_Most_Node_SetProperties(
 const G_PortHandle_t portHandle,
 const G_Most_Node_ClockMode_t clockMode,
 const G_Most_Node_DeviceMode_t deviceMode,
 const G_Most_Node_NetworkMasterMode_t networkMasterMode,
 const G_Most_Node_LogicalNodeAddressMode_t logicalNodeAddressMode,
 const u16_t logicalNodeAddress,
 const u16_t groupAddress
);
```

Parameters

`portHandle`
Handle to the communication port

`clockMode`
Variable of type **G_Most_Node_ClockMode_t**

The following return values are available:

- `G_MOST__NODE__CLOCK_MODE__CRYSTAL_OSCILLATOR_44100`

  44.1 KHz Crystal Oscillator
- `G_MOST__NODE__CLOCK_MODE__CRYSTAL_OSCILLATOR_48000`

  48 KHz Crystal Oscillator

`deviceMode`
Variable of type **G_Most_Node_DeviceMode_t**

The following return values are available:

- `G_MOST__NODE__DEVICE_MODE__TIMING_MASTER` - Timing Master
- `G_MOST__NODE__DEVICE_MODE__TIMING_SLAVE` - Timing Slave
- `G_MOST__NODE__DEVICE_MODE__SPY` - Spy (Bypass)

`networkMasterMode`
Variable of type **G_Most_Node_NetworkMasterMode_t**

The following return values are available:

- `G_MOST__NODE__NETWORK_MASTER_MODE__MASTER` - Network Master
- `G_MOST__NODE__NETWORK_MASTER_MODE__SLAVE` - Network Slave

`logicalNodeAddressMode`
Variable of type **G_Most_Node_LogicalNodeAddressMode_t**

The following return values are available:

- `G_MOST__NODE__LOGICAL_NODE_ADDRESS_MODE__DERIVE_FROM_POSITION`

  Derivation of the logical node address from the position in the MOST ring (logical node address = `0x100` + NodePosition)
- `G_MOST__NODE__LOGICAL_NODE_ADDRESS_MODE__STATIC_ADDRESS`

  Static logical node address (see LogicalNodeAddress)

`logicalNodeAddress`
Logical node address (only relevant if logicalNodeAddressMode = `G_MOST__NODE__LOGICAL_NODE_ADDRESS_MODE__STATIC_ADDRESS`) (0x0010..0x02FF, 0x0500..0x0FEF)

`groupAddress`
Group address (0x0300..0x03FF)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Node_GetProperties

G_Most_Node_GetProperties — Get node properties

Description

This command queries the properties of the MOST node.

Definition

```
G_Error_t
G_Most_Node_GetProperties(
 const G_PortHandle_t portHandle,
 G_Most_Node_GetProperties_Response_t * const responseStruct
);
```

Parameters

`portHandle`
Handle to the communication port

`responseStruct`
Pointer to a variable of type **G_Most_Node_GetProperties_Response_t**

The return values are saved in this variable.

The structure **G_Most_Node_GetProperties_Response_t** consists of the following items:

- `ClockMode`
  Variable of type **G_Most_Node_ClockMode_t**

  The following return values are available:
  - `G_MOST__NODE__CLOCK_MODE__CRYSTAL_OSCILLATOR_44100`
    44.1 KHz Crystal Oscillator
  - `G_MOST__NODE__CLOCK_MODE__CRYSTAL_OSCILLATOR_48000`

    48 KHz Crystal Oscillator
- `DeviceMode`
  Variable of type **G_Most_Node_DeviceMode_t**

  The following return values are available:
  - `G_MOST__NODE__DEVICE_MODE__TIMING_MASTER` - Timing Master
  - `G_MOST__NODE__DEVICE_MODE__TIMING_SLAVE` - Timing Slave
  - `G_MOST__NODE__DEVICE_MODE__SPY` - Spy (Bypass)
- `NetworkMasterMode`
  Variable of type **G_Most_Node_NetworkMasterMode_t**

  The following return values are available:
  - `G_MOST__NODE__NETWORK_MASTER_MODE__MASTER` - Network Master
  - `G_MOST__NODE__NETWORK_MASTER_MODE__SLAVE` - Network Slave
- `LogicalNodeAddressMode`
  Variable of type **G_Most_Node_LogicalNodeAddressMode_t**

  The following return values are available:
  - `G_MOST__NODE__LOGICAL_NODE_ADDRESS_MODE__DERIVE_FROM_POSITION`

    Derivation of the logical node address from the position in the MOST ring (logical node address = `0x100` + NodePosition)
  - `G_MOST__NODE__LOGICAL_NODE_ADDRESS_MODE__STATIC_ADDRESS`

Static logical node address

- `DesiredLogicalNodeAddress`
  logical node address selected with G_Most_Node_SetProperties
- `ActualLogicalNodeAddress`
  Currently (still) valid logical node address
- `DesiredGroupAddress`
  Group address selected with G_Most_Node_SetProperties
- `ActualGroupAddress`
  Currently (still) valid group address

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Node_GetProperties_Async

G_Most_Node_GetProperties_Async — (Asynchronous) Query of the MOST node properties

Description

This command initiates the asynchronous query of the the MOST node properties.

In contrast to G_Most_Node_GetProperties , this command returns immediately (that means without waiting for the response). As soon as the specific response is available, the callback function that has been set with G_Most_Node_GetProperties_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Most_Node_GetProperties_Async(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Node_GetProperties_Async_Callback

G_Most_Node_GetProperties_Async_Callback — Callback function for the asynchronous query of the MOST node properties

Description

This function will be automatically called if a response for the asynchronous query of the MOST node properties by G_Most_Node_GetProperties_Async is available. It must be entered by the user and set with G_Most_Node_GetProperties_Async_AddCallback.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Most_Node_GetProperties_Async_Callback(
 const G_PortHandle_t  portHandle,
 const G_Most_Node_GetProperties_Response_t * const responseStruct
);
```

Parameters

`portHandle`
Handle to the communication port

`responseStruct`
Pointer to a variable of type **G_Most_Node_GetProperties_Response_t**

The return values are saved in this variable.

The structure **G_Most_Node_GetProperties_Response_t** consists of the following items:

- `ClockMode`
  Variable of type **G_Most_Node_ClockMode_t**

  The following return values are available:
  - `G_MOST__NODE__CLOCK_MODE__CRYSTAL_OSCILLATOR_44100`

    44.1 KHz Crystal Oscillator
  - `G_MOST__NODE__CLOCK_MODE__CRYSTAL_OSCILLATOR_48000`

    48 KHz Crystal Oscillator
- `DeviceMode`
  Variable of type **G_Most_Node_DeviceMode_t**

  The following return values are available:
  - `G_MOST__NODE__DEVICE_MODE__TIMING_MASTER` - Timing Master
  - `G_MOST__NODE__DEVICE_MODE__TIMING_SLAVE` - Timing Slave
  - `G_MOST__NODE__DEVICE_MODE__SPY` - Spy (Bypass)
- `NetworkMasterMode`
  Variable of type **G_Most_Node_NetworkMasterMode_t**

  The following return values are available:
  - `G_MOST__NODE__NETWORK_MASTER_MODE__MASTER` - Network Master
  - `G_MOST__NODE__NETWORK_MASTER_MODE__SLAVE` - Network Slave
- `LogicalNodeAddressMode`
  Variable of type **G_Most_Node_LogicalNodeAddressMode_t**

  The following return values are available:

  - `G_MOST__NODE__LOGICAL_NODE_ADDRESS_MODE__DERIVE_FROM_POSITION`

    Derivation of the logical node address from the position in the MOST ring (logical node address = `0x100` + NodePosition)
  - `G_MOST__NODE__LOGICAL_NODE_ADDRESS_MODE__STATIC_ADDRESS`

    Static logical node address
- `DesiredLogicalNodeAddress`
  Logical node address selected with G_Most_Node_SetProperties
- `ActualLogicalNodeAddress`
  Currently (still) valid logical node address
- `DesiredGroupAddress`
  Group address selected with G_Most_Node_SetProperties
- `ActualGroupAddress`
  Currently (still) valid group address

G_Most_Node_GetProperties_Async_AddCallback

G_Most_Node_GetProperties_Async_AddCallback — Set the callback function for the asynchronous query of the MOST node properties

Description

This command defines a callback function for the asynchronous query of the MOST node properties with G_Most_Node_GetProperties_Async.

Definition

```
G_Error_t
G_Most_Node_GetProperties_Async_AddCallback(
 const G_PortHandle_t portHandle,
 const G_Most_Node_GetProperties_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`callback`
Function pointer of the callback function (see G_Most_Node_GetProperties_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Node_GetProperties_Async_RemoveCallback

G_Most_Node_GetProperties_Async_RemoveCallback — Remove the callback function for the asynchronous query of the MOST node properties

Description

This command removes a callback function for the asynchronous query of the MOST node properties.

Definition

```
G_Error_t
G_Most_Node_GetProperties_Async_RemoveCallback(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

7 PDTS

These commands are used to control the **P**acket **D**ata **T**ransmission **S**ervice (**PDTS**).

G_Most_Pdts_GetTxState

G_Most_Pdts_GetTxState — Query the sending status

Description

This command queries the PDTS sending status.

Definition

```
G_Error_t
G_Most_Pdts_GetTxState(
 const G_PortHandle_t portHandle,
 G_Most_Status_t * const status,
 G_Most_Pdts_GetTxState_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`status`
Returns MOST Node Status (see MOST Node Status)

`rspFlags`
Pointer to a variable of type **G_Most_Pdts_GetTxState_RspFlags_t**

The following return flags are available:

- `G_MOST__PDTS__GET_TX_STATE__RSP_FLAG__NONE`

  No flag was set
- `G_MOST__PDTS__GET_TX_STATE__RSP_FLAG__BUSY`

  The sending of the message could not be completed yet

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Pdts_GetTxState_Async

G_Most_Pdts_GetTxState_Async — Query the sending status (by asynchronous query)

Description

This command initiates the asynchronous query of the PDTS sending status.

In contrast to G_Most_Pdts_GetTxState , this command returns immediately (that means without waiting for the response). As soon as the corresponding response is available, the callback function set by G_Most_Pdts_GetTxState_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Most_Pdts_GetTxState_Async(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Pdts_GetTxState_Async_Callback

G_Most_Pdts_GetTxState_Async_Callback — Callback function for the asynchronous query of the sending status

Description

This function will be automatically called if a response for the asynchronous query of the PDTS sending status with G_Most_Pdts_GetTxState_Async is available. It must be entered by the user and set with G_Most_Pdts_GetTxState_Async_AddCallback.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Most_Pdts_GetTxState_Async_Callback(
 const G_PortHandle_t  portHandle,
 const G_Most_Status_t  status,
 const G_Most_Pdts_GetTxState_RspFlags_t  rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`status`
Returns MOST Node Status (see MOST Node Status)

`rspFlags`
Variable of type **G_Most_Pdts_GetTxState_RspFlags_t**

The following return flags are available:

- `G_MOST__PDTS__GET_TX_STATE__RSP_FLAG__NONE`

  No flag was set
- `G_MOST__PDTS__GET_TX_STATE__RSP_FLAG__BUSY`

  The sending of the message could not be completed yet

G_Most_Pdts_GetTxState_Async_AddCallback

G_Most_Pdts_GetTxState_Async_AddCallback — Set the callback function for the asynchronous query of the sending status

Description

This command defines a callback function for the asynchronous query of the PDTS sending status with G_Most_Pdts_GetTxState_Async.

Definition

```
G_Error_t
G_Most_Pdts_GetTxState_Async_AddCallback(
 const G_PortHandle_t portHandle,
 const G_Most_Pdts_GetTxState_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`callback`
Function pointer of the callback function (see G_Most_Pdts_GetTxState_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Pdts_GetTxState_Async_RemoveCallback

G_Most_Pdts_GetTxState_Async_RemoveCallback — Remove the callback function for the asynchronous query of the sending status

Description

This command removes a callback function for the asynchronous query of the PDTS sending status.

Definition

```
G_Error_t
G_Most_Pdts_GetTxState_Async_RemoveCallback(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Pdts_SendMessage

G_Most_Pdts_SendMessage — Send a PDTS message

Description

This command sends a PDTS message.

Definition

```
G_Error_t
G_Most_Pdts_SendMessage(
 const G_PortHandle_t portHandle,
 const u32_t timeout,
 const u16_t targetDeviceId,
 const u8_t * const data,
 const u32_t dataLength,
 G_Most_Status_t * const status,
 G_Most_Pdts_SendMessage_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`timeout`
Sending timeout in microseconds

`targetDeviceId`
MOST target address

`data`
Pointer to the data buffer for message data bytes

`dataLength`
Number of data bytes

`status`
Returns MOST Node Status (see MOST Node Status)

`rspFlags`
Pointer to a variable of type **G_Most_Pdts_SendMessage_RspFlags_t**

The following return flags are available:

- `G_MOST__PDTS__SEND_MESSAGE__RSP_FLAG__NONE`

  No flag was set
- `G_MOST__PDTS__SEND_MESSAGE__RSP_FLAG__CANCELED_PREVIOUS`

  The sending of the previous message was canceled by this sending request
- `G_MOST__PDTS__SEND_MESSAGE__RSP_FLAG__BUSY`

  The transmission of the message could not be completed yet.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Pdts_QueueMessage

G_Most_Pdts_QueueMessage — Send the PDTS message

Description

This command enqueues a PDTS message into the hardware sending operation.

Please consider that on function return it cannot be ensured that the corresponding message has been transmitted by the hardware. The message is only transferred to the hardware. If you want to ensure that the message has been sent, it is necessary to use G_Most_Pdts_SendMessage .

Definition

```
G_Error_t
G_Most_Pdts_QueueMessage(
 const G_PortHandle_t portHandle,
 const u16_t targetDeviceId,
 const u8_t * const data,
 const u32_t dataLength,
 G_Most_Pdts_QueueMessage_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`targetDeviceId`
MOST target address

`data`
Pointer to the data buffer for message data bytes

`dataLength`
Number of data bytes

`rspFlags`
Pointer to a variable of type **G_Most_Pdts_QueueMessage_RspFlags_t**

The following return flags are available:

- `G_MOST__PDTS__QUEUE_MESSAGE__RSP_FLAG__NONE`

  No flag was set
- `G_MOST__PDTS__QUEUE_MESSAGE__RSP_FLAG__CANCELED_PREVIOUS`

  The sending of the previous message was canceled by this sending request

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Pdts_Mep_GetTxState

G_Most_Pdts_Mep_GetTxState — Query **MOST Ethernet Packet** transmission state

Description

Use this command to query the **MOST Ethernet Packet** transmission state.

Definition

```
G_Error_t
G_Most_Pdts_Mep_GetTxState(
 const G_PortHandle_t portHandle,
 G_Most_Status_t * const status,
 G_Most_Pdts_Mep_GetTxState_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`status`
Returns MOST Node Status (see MOST Node Status)

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_MOST__PDTS__MEP__GET_TX_STATE__RSP_FLAG__NONE`
No flag is set

`G_MOST__PDTS__MEP__GET_TX_STATE__RSP_FLAG__BUSY`
A **MOST Ethernet Packet** is being transmitted at the moment

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Pdts_Mep_QueueMessage

G_Most_Pdts_Mep_QueueMessage — Queue **MOST Ethernet Packet**

Description

Use this command to queue a **MOST Ethernet Packet** for transmission.

Note that the function may return before the packet is sent. If you want to make sure that the packet has been sent, use G_Most_Pdts_Mep_SendMessage .

If there already is a **MOST Ethernet Packet** in the queue, it will be deleted and its transmission will be aborted. In this case, response flag `G_MOST__PDTS__MEP__QUEUE_MESSAGE__RSP_FLAG__CANCELED_PREVIOUS` will be set.

Definition

```
G_Error_t
G_Most_Pdts_Mep_QueueMessage(
 const G_PortHandle_t portHandle,
 const u16_t  length,
 const u8_t * const data,
 G_Most_Pdts_Mep_QueueMessage_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`length`
Length of **MOST Ethernet Packet** data, in bytes

`data`
MOST Ethernet Packet data bytes

The data structure of a standard Ethernet frame can be seen in the following table:

Destination Address	Source Address	VLAN Tag (optional)	Type	Data	FCS (Frame Checksum)
6 bytes	6 bytes	4 bytes	2 bytes	0..1500 bytes	4 bytes

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_MOST__PDTS__MEP__QUEUE_MESSAGE__RSP_FLAG__NONE`
No flag is set

`G_MOST__PDTS__MEP__QUEUE_MESSAGE__RSP_FLAG__CANCELED_PREVIOUS`
An existing **MOST Ethernet Packet** was deleted from the queue and its transmission was aborted.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Pdts_Mep_SendMessage

G_Most_Pdts_Mep_SendMessage — Send **MOST Ethernet Packet**

Description

Use this command to send a **MOST Ethernet Packet** .

In contrast to G_Most_Pdts_Mep_QueueMessage , the functions will wait until the data has been transmitted or the user defined timeout was reached.

If there already is a **MOST Ethernet Packet** in the queue, it will be deleted and its transmission will be aborted. In this case, response flag `G_MOST__PDTS__MEP__SEND_MESSAGE__RSP_FLAG__CANCELED_PREVIOUS` will be set.

Definition

```
G_Error_t
G_Most_Pdts_Mep_SendMessage(
 const G_PortHandle_t portHandle,
 const u32_t  timeout,
 const u16_t  length,
 const u8_t * const data,
 G_Most_Status_t * const status,
 G_Most_Pdts_Mep_SendMessage_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`timeout`
Timeout for transmission, in microseconds

If the data could not be transmitted completely within the given timeout, response flag `G_MOST__PDTS__SEND_MESSAGE__RSP_FLAG__BUSY` will be set.

`length`
Length of **MOST Ethernet Packet** data, in bytes

`data`
MOST Ethernet Packet data bytes

The data structure of a standard Ethernet frame can be seen in MOST Ethernet Frame [1521].

`status`
Returns MOST Node Status (see MOST Node Status)

`rspFlags`
Returns response flags

The following values are possible and can be combined:

`G_MOST__PDTS__SEND_MESSAGE__RSP_FLAG__NONE`
No flag is set

`G_MOST__PDTS__SEND_MESSAGE__RSP_FLAG__CANCELED_PREVIOUS`
An existing **MOST Ethernet Packet** was deleted from the queue and its transmission was aborted.

G_MOST__PDTS__SEND_MESSAGE__RSP_FLAG__BUSY
Not all data could be transmitted within the given timeout

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

8 RBD

These commands are used to control the **R**ing **B**reak **D**iagnosis (**RBD**).

G_Most_Rbd_Initiate

G_Most_Rbd_Initiate — Start the ring break diagnosis

Description

This command initiates the start of the ring break diagnosis.

Please consider that on function return it cannot be ensured that the starting operation of the ring break diagnosis has been completed. The start request is only transferred to the hardware.

Definition

```
G_Error_t
G_Most_Rbd_Initiate(
 const G_PortHandle_t portHandle,
 const G_Most_Rbd_Initiate_CmdFlags_t cmdFlags,
 const u32_t systemStartImpulseLowTime,
 const u32_t systemStartImpulseHighTime,
 const u8_t startSequenceData
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Variable of type **G_Most_Rbd_Initiate_CmdFlags_t**

The following flags are available:

- `G_MOST__RBD__INITIATE__CMD_FLAG__NONE`

  No flag, if no other has to be transferred
- `G_MOST__RBD__INITIATE__CMD_FLAG__USE_SYSTEM_START_IMPULSE`

  Use the system start impulse

`systemStartImpulseLowTime`
Time of the "Low"-part of the start impulse in microseconds (95000..10215000)

`systemStartImpulseHighTime`
Time of the "High"-part of the start impulse in microseconds (**100000**)

`startSequenceData`
Byte for the start sequence information concerning the ring break diagnosis

(**bit 0** is **b0** , **bit 1** is **b1** , ...)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Rbd_SetSystemState

G_Most_Rbd_SetSystemState — Set the system status

Description

This command defines the system status for the ring break diagnosis.

Definition

```
G_Error_t
G_Most_Rbd_SetSystemState(
 const G_PortHandle_t portHandle,
 const G_Most_Rbd_SetSystemState_State_t state
);
```

Parameters

`portHandle`
Handle to the communication port

`state`
Variable of type **G_Most_Rbd_SetSystemState_State_t**

The desired status is specified in this variable.

The following return values are available:

- `G_MOST__RBD__SET_SYSTEM_STATE__STATE__SYSTEM_OFF` - System off
- `G_MOST__RBD__SET_SYSTEM_STATE__STATE__SYSTEM_ON` - System on

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Rbd_GetPortState

G_Most_Rbd_GetPortState — Query the port status

Description

This command queries the current port status.

Definition

```
G_Error_t
G_Most_Rbd_GetPortState(
 const G_PortHandle_t portHandle,
 G_Most_Rbd_State_t * const state,
 G_Most_Rbd_GetPortState_RspFlags_t * const rspFlags,
 u64_t * const lastFallingEdge,
 u64_t * const lastRisingEdge,
 u64_t * const currentTime
);
```

Parameters

`portHandle`
Handle to the communication port

`state`
Variable of type **G_Most_Rbd_State_t**

The following return values are available:

- `G_MOST__RBD__STATE__SYSTEM_OFF`
- `G_MOST__RBD__STATE__SYSTEM_ON`
- `G_MOST__RBD__STATE__WAIT_START_SEQUENCE`
- `G_MOST__RBD__STATE__EVALUATE_START_SEQUENCE`
- `G_MOST__RBD__STATE__DELAY`
- `G_MOST__RBD__STATE__RESPONSE_SEQUENCE`
- `G_MOST__RBD__STATE__PAUSE`

`rspFlags`
Variable of type **G_Most_Rbd_GetPortState_RspFlags_t**

The following return flags are available:

- `G_MOST__RBD__GET_PORT_STATE__RSP_FLAG__INPUT_IS_HIGH`

  Input "high"
- `G_MOST__RBD__GET_PORT_STATE__RSP_FLAG__OUTPUT_IS_HIGH`

  Output "high"
- `G_MOST__RBD__GET_PORT_STATE__RSP_FLAG__LAST_FALLING_EDGE_IS_VALID`

  Last falling edge is valid
- `G_MOST__RBD__GET_PORT_STATE__RSP_FLAG__LAST_RISING_EDGE_IS_VALID`

  Last rising edge is valid
- `G_MOST__RBD__GET_PORT_STATE__RSP_FLAG__DEBOUNCED_INPUT_IS_HIGH`

  Debounced input "high"

`lastFallingEdge`
in nanoseconds

`lastRisingEdge`
in nanoseconds

`currentTime`
Current time stamp in nanoseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Rbd_GetPortState_Async

G_Most_Rbd_GetPortState_Async — Query the port status (by asynchronous query)

Description

This command initiates the asynchronous query of the port status.

In contrast to G_Most_Rbd_GetPortState , this command returns immediately (that means without waiting for the response). As soon as the corresponding response is available, the callback function set by G_Most_Rbd_GetPortState_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Most_Rbd_GetPortState_Async(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Rbd_GetPortState_Async_Callback

G_Most_Rbd_GetPortState_Async_Callback — Callback function for the asynchronous query of the port status

Description

This function will be automatically called, if a response for the asynchronous query of the port status with G_Most_Rbd_GetPortState_Async is available. It has to be entered by the user and be set with G_Most_Rbd_GetPortState_Async_AddCallback.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Most_Rbd_GetPortState_Async_Callback(
 const G_PortHandle_t  portHandle,
 const G_Most_Rbd_State_t  state,
 const G_Most_Rbd_GetPortState_RspFlags_t  rspFlags,
 const u64_t  lastFallingEdge,
 const u64_t  lastRisingEdge,
 const u64_t  currentTime
);
```

Parameters

`portHandle`
Handle to the communication port

`state`
Variable of type **G_Most_Rbd_State_t**

The following return values are available:
- `G_MOST__RBD__STATE__SYSTEM_OFF`
- `G_MOST__RBD__STATE__SYSTEM_ON`
- `G_MOST__RBD__STATE__WAIT_START_SEQUENCE`
- `G_MOST__RBD__STATE__EVALUATE_START_SEQUENCE`
- `G_MOST__RBD__STATE__DELAY`
- `G_MOST__RBD__STATE__RESPONSE_SEQUENCE`
- `G_MOST__RBD__STATE__PAUSE`

`rspFlags`
Variable of type **G_Most_Rbd_GetPortState_RspFlags_t**

The following return flags are available:
- `G_MOST__RBD__GET_PORT_STATE__RSP_FLAG__INPUT_IS_HIGH`

  Input "high"
- `G_MOST__RBD__GET_PORT_STATE__RSP_FLAG__OUTPUT_IS_HIGH`

  Output "high"
- `G_MOST__RBD__GET_PORT_STATE__RSP_FLAG__LAST_FALLING_EDGE_IS_VALID`

  Last falling edge is valid
- `G_MOST__RBD__GET_PORT_STATE__RSP_FLAG__LAST_RISING_EDGE_IS_VALID`

  Last rising edge is valid

- G_MOST__RBD__GET_PORT_STATE__RSP_FLAG__DEBOUNCED_INPUT_IS_HIGH

  Debounced input “high”

lastFallingEdge
: in nanoseconds

lastRisingEdge
: in nanoseconds

currentTime
: Current time stamp in nanoseconds

G_Most_Rbd_GetPortState_Async_AddCallback

G_Most_Rbd_GetPortState_Async_AddCallback — Sets the callback function for the asynchronous query of the port status

Description

This command defines a callback function for the asynchronous query of the port status with G_Most_Rbd_GetPortState_Async.

Definition

```
G_Error_t
G_Most_Rbd_GetPortState_Async_AddCallback(
 const G_PortHandle_t portHandle,
 const G_Most_Rbd_GetPortState_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`callback`
Function pointer of the callback function (see G_Most_Rbd_GetPortState_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Rbd_GetPortState_Async_RemoveCallback

G_Most_Rbd_GetPortState_Async_RemoveCallback — Remove the callback function for the asynchronous query of the port status

Description

This command removes a callback function for the asynchronous query of the port status.

Definition

```
G_Error_t
G_Most_Rbd_GetPortState_Async_RemoveCallback(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Rbd_SetPortOutput

G_Most_Rbd_SetPortOutput — Set the initial level

Description

This command defines the initial port level.

Definition

```
G_Error_t
G_Most_Rbd_SetPortOutput(
 const G_PortHandle_t portHandle,
 const G_Most_Rbd_Level_t level
);
```

Parameters

`portHandle`
Handle to the communication port

`level`
Variable of type **G_Most_Rbd_Level_t** indicating the initial level

The following return values are available:
- `G_MOST__RBD__LEVEL__LOW`
- `G_MOST__RBD__LEVEL__HIGH`

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Rbd_SetResponses

G_Most_Rbd_SetResponses — Set responses

Description

This command defines responses concerning the ring break diagnosis.

This command allows to send several responses. If the `cmdFlag` `G_MOST__RBD__SET_RESPONSES__CMD_FLAG__RESULT_IN_OWN_RESPONSE_SLOT` has been set, the result of the ring break diagnosis of the own node will be sent in "ownResponseSlot" to the response sequence of the ring break diagnosis.

Definition

```
G_Error_t
G_Most_Rbd_SetResponses(
 const G_PortHandle_t portHandle,
 const G_Most_Rbd_SetResponses_CmdFlags_t cmdFlags,
 const u8_t ownResponseSlot,
 const u8_t numberOfResponses,
 const G_Most_Rbd_ResponseTypes_t responses[60]
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Variable of type **G_Most_Rbd_SetResponses_CmdFlags_t**

The following flags are available:

- `G_MOST__RBD__SET_RESPONSES__CMD_FLAG__NONE`

  No flag, if no other has to be transferred
- `G_MOST__RBD__SET_RESPONSES__CMD_FLAG__RESULT_IN_OWN_RESPONSE_SLOT`

  The result of the ring break diagnosis of the own node is sent in `ownResponseSlot` to the response sequence of the ring break diagnosis

`ownResponseSlot`
Own response slot (starting with 0)

`numberOfResponses`
Number of responses (0..60)

`responses`
Buffer for responses to be set

Each individual byte is a response of type **G_Most_Rbd_ResponseTypes_t.**

Following types are available:

- `G_MOST__RBD__RESPONSE_TYPE__NO_RESPONSE`
- `G_MOST__RBD__RESPONSE_TYPE__OK`
- `G_MOST__RBD__RESPONSE_TYPE__INVALID`
- `G_MOST__RBD__RESPONSE_TYPE__NO_LIGHT`

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Rbd_GetResponses

G_Most_Rbd_GetResponses — Request of responses

Description

This command requests responses concerning the ring break diagnosis.

Definition

```
G_Error_t
G_Most_Rbd_GetResponses(
 const G_PortHandle_t portHandle,
 G_Most_Rbd_GetResponses_RspFlags_t * const rspFlags,
 G_Most_Rbd_State_t * const state,
 G_Most_Inic_LockState_t * const lockStateOnTest,
 u8_t * const numberOfResponses,
 G_Most_Rbd_ResponseTypes_t * const responses
);
```

Parameters

`portHandle`
Handle to the communication port

`rspFlags`
Pointer to variable of type **G_Most_Rbd_GetResponses_RspFlags_t**

The following return flags are available:

- `G_MOST__RBD__GET_RESPONSES__RSP_FLAG__NONE`

  No flag was set
- `G_MOST__RBD__GET_RESPONSES__RSP_FLAG__RESPONSES_ARE_VALID`

  The responses contained in `responses` are valid

`state`
Pointer to the variable of type **G_Most_Rbd_State_t**

The following return values are available:

- `G_MOST__RBD__STATE__SYSTEM_OFF`
- `G_MOST__RBD__STATE__SYSTEM_ON`
- `G_MOST__RBD__STATE__WAIT_START_SEQUENCE`
- `G_MOST__RBD__STATE__EVALUATE_START_SEQUENCE`
- `G_MOST__RBD__STATE__DELAY`
- `G_MOST__RBD__STATE__RESPONSE_SEQUENCE`
- `G_MOST__RBD__STATE__PAUSE`

`lockStateOnTest`
Pointer to the variable of type **G_Most_Inic_LockState_t**

The following return values are available:

- `G_MOST__INIC__LOCK_STATE__UNKNOWN`
- `G_MOST__INIC__LOCK_STATE__UNLOCK`
- `G_MOST__INIC__LOCK_STATE__STABLE_LOCK`

`numberOfResponses`
Pointer to the variable `numberOfResponses`

The number of the responses contained is returned in this variable (0..60).

`responses`
Pointer to the buffer with the received responses

Each individual byte is a response of type **G_Most_Rbd_ResponseTypes_t**.

The following types are available:
- `G_MOST__RBD__RESPONSE_TYPE__NO_RESPONSE`
- `G_MOST__RBD__RESPONSE_TYPE__OK`
- `G_MOST__RBD__RESPONSE_TYPE__INVALID`
- `G_MOST__RBD__RESPONSE_TYPE__NO_LIGHT`

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Rbd_GetResponses_Async

G_Most_Rbd_GetResponses_Async — Request ring break diagnosis responses (by asynchronous request)

Description

This command initiates the asynchronous request of ring break diagnosis responses.

In contrast to G_Most_Rbd_GetResponses , this command returns immediately (that means without waiting forr the response). As soon as the corresponding response is available, the callback function set by G_Most_Rbd_GetResponses_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Most_Rbd_GetResponses_Async(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Rbd_GetResponses_Async_Callback

G_Most_Rbd_GetResponses_Async_Callback — Callback function for the asynchronous request of ring break diagnosis responses

Description

This function will be automatically called if a response for the asynchronous request of ring break diagnosis responses with G_Most_Rbd_GetResponses_Async is available. It must be entered by the user and be set with G_Most_Rbd_GetResponses_Async_AddCallback.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Most_Rbd_GetResponses_Async_Callback(
 const G_PortHandle_t  portHandle,
 const G_Most_Rbd_GetResponses_RspFlags_t  rspFlags,
 const G_Most_Rbd_State_t  state,
 const G_Most_Inic_LockState_t  lockStateOnTest,
 const u8_t  numberOfResponses,
 const G_Most_Rbd_ResponseTypes_t *  const responses
);
```

Parameters

`portHandle`
Handle to the communication port

`rspFlags`
Variable of type **G_Most_Rbd_GetResponses_RspFlags_t**

The following return flags are available:
- `G_MOST__RBD__GET_RESPONSES__RSP_FLAG__NONE`

  No flag was set
- `G_MOST__RBD__GET_RESPONSES__RSP_FLAG__RESPONSES_ARE_VALID`

  The responses contained in `responses` are valid

`state`
Variable of type **G_Most_Rbd_State_t**

The following return values are available:
- `G_MOST__RBD__STATE__SYSTEM_OFF`
- `G_MOST__RBD__STATE__SYSTEM_ON`
- `G_MOST__RBD__STATE__WAIT_START_SEQUENCE`
- `G_MOST__RBD__STATE__EVALUATE_START_SEQUENCE`
- `G_MOST__RBD__STATE__DELAY`
- `G_MOST__RBD__STATE__RESPONSE_SEQUENCE`
- `G_MOST__RBD__STATE__PAUSE`

`lockStateOnTest`
Variable of type **G_Most_Inic_LockState_t**

The following return values are available:
- `G_MOST__INIC__LOCK_STATE__UNKNOWN`

- G_MOST__INIC__LOCK_STATE__UNLOCK
- G_MOST__INIC__LOCK_STATE__STABLE_LOCK

numberOfResponses
Number of the responses contained (0..60)

responses
Pointer to the buffer with the received responses

Each individual byte is a response of type **G_Most_Rbd_ResponseTypes_t**.

The following types are available:
- G_MOST__RBD__RESPONSE_TYPE__NO_RESPONSE
- G_MOST__RBD__RESPONSE_TYPE__OK
- G_MOST__RBD__RESPONSE_TYPE__INVALID
- G_MOST__RBD__RESPONSE_TYPE__NO_LIGHT

G_Most_Rbd_GetResponses_Async_AddCallback

G_Most_Rbd_GetResponses_Async_AddCallback — Set the callback function for the asynchronous request of ring break diagnosis responses

Description

This command defines a callback function for the asynchronous request of ring break diagnosis responses with G_Most_Rbd_GetResponses_Async.

Definition

```
G_Error_t
G_Most_Rbd_GetResponses_Async_AddCallback(
 const G_PortHandle_t portHandle,
 const G_Most_Rbd_GetResponses_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`callback`
Function pointer of the callback function (see G_Most_Rbd_GetResponses_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Rbd_GetResponses_Async_RemoveCallback

G_Most_Rbd_GetResponses_Async_RemoveCallback — Remove the callback function for the asynchronous request of ring break diagnosis responses

Description

This command removes a callback function for the asynchronous request of ring break diagnosis responses.

Definition

```
G_Error_t
G_Most_Rbd_GetResponses_Async_RemoveCallback(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Rbd_Reset

G_Most_Rbd_Reset — Reset the ring break diagnosis

Description

This command resets a ring break diagnosis.

Definition

```
G_Error_t
G_Most_Rbd_Reset(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

9 TP

This command is used to control the **T**ransport **P**rotocol (**TP**).

G_Most_Tp_GetChannel

G_Most_Tp_GetChannel — Request a multisession channel

Description

This command requests a multisession channel.

This command is used for the dynamic administration of multisession channels. A multisession channel is requested whose concrete channel number is returned in `channel`.

This multisession channel administration may be necessary if several applications or software threads operate with the firmware and with the same multisession channels.

If always only one application works with the firmware, a multisession channel administration within the application is sufficient and no administration by the firmware is necessary.

Definition

```
G_Error_t
G_Most_Tp_GetChannel(
 const G_PortHandle_t portHandle,
 u8_t *  const channel
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Pointer to the variable `channel`

The number of the assigned multisession channel is returned in this variable (starting with 0).

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Tp_GetChannel_Async

G_Most_Tp_GetChannel_Async — Request a multisession channel (by asynchronous request)

Description

This command initiates the asynchronous request of a multisession channel.

This command is used for the dynamic administration of multisession channels. A multisession channel is requested whose concrete channel number is returned in `channel`.

This multisession channel administration may be necessary if several applications or software threads operate with the firmware and with the same multisession channels.

If always only one application works with the firmware, a multisession channel administration within the application is sufficient and no administration by the firmware is necessary.

In contrast to G_Most_Tp_GetChannel , this command returns immediately (that means without waiting for the response). As soon as the corresponding response is available, the callback function set by G_Most_Tp_GetChannel_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Most_Tp_GetChannel_Async(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Tp_GetChannel_Async_Callback

G_Most_Tp_GetChannel_Async_Callback — Callback function to request a multisession channel

Description

This function will be automatically called if a response for the asynchronous request of a multisession channel with G_Most_Tp_GetChannel_Async is available. It has to be entered by the user and be set with G_Most_Tp_GetChannel_Async_AddCallback.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Most_Tp_GetChannel_Async_Callback(
 const G_PortHandle_t  portHandle,
 const u8_t channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel

G_Most_Tp_GetChannel_Async_AddCallback

G_Most_Tp_GetChannel_Async_AddCallback — Set the callback function for the asynchronous request of a multisession channel

Description

This command defines a callback function for the asynchronous request of a multisession channel with G_Most_Tp_GetChannel_Async.

Definition

```
G_Error_t
G_Most_Tp_GetChannel_Async_AddCallback(
 const G_PortHandle_t portHandle,
 const G_Most_Tp_GetChannel_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`callback`
Function pointer of the callback function (see G_Most_Tp_GetChannel_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Tp_GetChannel_Async_RemoveCallback

G_Most_Tp_GetChannel_Async_RemoveCallback — Remove a callback function for the asynchronous request of a multisession channel

Description

This command removes a callback function for the asynchronous request of a multisession channel.

Definition

```
G_Error_t
G_Most_Tp_GetChannel_Async_RemoveCallback(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Tp_FreeChannel

G_Most_Tp_FreeChannel — Release the multisession channel

Description

This command releases the multisession channel specified by `channel`.

This command is used for the dynamic administration of multisession channels. The multisession channel defined under `channel` is released.

This multisession channel administration may be necessary if several applications or software threads operate with the firmware and with the same multisession channels.

If always only one application works with the firmware, a multisession channel administration within the application is sufficient and no administration by the firmware is required.

Definition

```
G_Error_t
G_Most_Tp_FreeChannel(
 const G_PortHandle_t portHandle,
 const u8_t channel
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Tp_Config_KeywordRec

G_Most_Tp_Config_KeywordRec — Configure the multisession channel

Description

This command configures the multisession channel specified by `channel`, for the transport protocol "KeywordRec".

Definition

```
G_Error_t
G_Most_Tp_Config_KeywordRec(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Most_Tp_Config_KeywordRec_Parameters_t \
 * const keywordRecParameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`keywordRecParameters`
Pointer to a variable of type **G_Most_Tp_Config_KeywordRec_Parameters_t**

The structure **G_Most_Tp_Config_KeywordRec_Parameters_t** consists of the following items:

- `Flags`
  Variable of type **G_Most_Tp_Config_KeywordRec_CmdFlags_t**

  The following flags are available:
  - `G_MOST__TP__CONFIG__KEYWORD_REC__CMD_FLAG__NONE`

    No flag, if no other was set
  - `G_MOST__TP__CONFIG__KEYWORD_REC__CMD_FLAG__USE_PACKET_CHANNEL_FUNCTION`

    Use the packet channel function
- `PhysicalTargetDeviceId`
  MOST target address for physical addressing, can be set on 0xFFFF, if unknown.
- `FunctionalTargetDeviceId`
  MOST target address for functional addressing, e.g. 0x3C8 for broadcast
- `PhysicalSourceAddress`
  Physical diagnostics address of the own node
- `PhysicalTargetAddress`
  Physical diagnostics target address
- `FunctionalSourceAddress`
  Functional diagnostics address of the own node
- `FunctionalTargetAddress`
  Functional diagnostics target address
- `PacketChannelFunctionId`
  MOST function identifier for transmitting the diagnostics requests or for receiving the diagnostics responses on the packet channel of the MOST-High protocol (0x55 or 0x51)

- `InstanceId`
  Instance identifier of the target diagnostics function block, should correspond to `PhysicalTargetAddress`
- `reserved`
  reserved parameter (must be initialized with **0**)
- `TxTimeout`
  Sending timeout in microseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Tp_Config_Tp_2_0

G_Most_Tp_Config_Tp_2_0 — Configure the multisession channel

Description

This command configures the multisession channel specified with `channel` for the transport protocol "TP2.0".

Definition

```
G_Error_t
G_Most_Tp_Config_Tp_2_0(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Most_Tp_Config_Tp_2_0_Parameters_t * const tp_2_0Parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`tp_2_0Parameters`
Pointer to a variable of type **G_Most_Tp_Config_Tp_2_0_Parameters_t**

The structure **G_Most_Tp_Config_Tp_2_0_Parameters_t** consists of the following items:

- `SourceAddress`
  Own control unit address
- `TargetAddress`
  Control unit address of the test object
- `BlockSize`
  Block size (0..15)
- `ApplicationType`
  Variable of type **G_Most_Tp_Config_Tp_2_0_ApplicationType_t** indicating the application type

  The following return values are available:
  - `G_MOST__TP__CONFIG__TP_2_0__APPLICATION_TYPE__DIAG`

    Diagnostics
  - `G_MOST__TP__CONFIG__TP_2_0__APPLICATION_TYPE__INFO`

    Infotainment communication
  - `G_MOST__TP__CONFIG__TP_2_0__APPLICATION_TYPE__APP`

    Application protocol
  - `G_MOST__TP__CONFIG__TP_2_0__APPLICATION_TYPE__WFS_WIV`

    WFS / WIV
- `SourceSetupId`
  Own identifier for channel setup (e.g. 0x200 + SourceAddress), Set the SourceSetupId to 0xFFFFFFFF in the case of static channels
- `TargetSetupId`
  Test object identifier for channel setup (e.g. 0x200 + TargetAddress), Set the TargetSetupId to 0xFFFFFFFF in the case of static channels

- `SourceChannelId`
  Own identifier for data exchange (completely stipulated by the communication partner in the case of dynamic channels)
- `TargetChannelId`
  Test object identifier for data exchange
- `TimeoutT1`
  Time T1 in microseconds (Acknowledgment timeout for data telegrams, e.g. 100000 µs)
- `TimeoutT3`
  Time T3 in microseconds (minimum time between two telegrams, e.g. 5000 µs)
- `TxTimeout`
  Sending timeout in microseconds (e.g. 100000 µs)
- `OwnInstanceId`
  Instance identifier for the own diagnostics function block
- `reserved1`
  Reserved byte, must be initialized with `0`
- `reserved2`
  Reserved byte, must be initialized with `0`
- `reserved3`
  Reserved byte, must be initialized with `0`

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Tp_Config_Off

G_Most_Tp_Config_Off — Deactivate the configuration

Description

This command deactivates the current configuration of the multisession channel specified by `channel`.

Definition

```
G_Error_t
G_Most_Tp_Config_Off(
  const G_PortHandle_t portHandle,
  const u8_t channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Tp_Reset

G_Most_Tp_Reset — Reset the multisession channel

Description

This command resets the multisession channel specified by `channel`.

Definition

```
G_Error_t
G_Most_Tp_Reset(
 const G_PortHandle_t portHandle,
 const u8_t channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Tp_BroadCast_StartTransmission

G_Most_Tp_BroadCast_StartTransmission — Start the transmission

Description

This command starts the broadcast message transmission (even multiple times).

Definition

```
G_Error_t
G_Most_Tp_Broadcast_StartTransmission(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Most_Tp_BroadCast_TransmissionMode_t mode,
 const u8_t * const data,
 const u32_t dataLength
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`mode`
Variable of type **G_Most_Tp_Broadcast_TransmissionMode_t** indicating the transmission mode

The `mode = G_MOST__TP__BROADCAST__TRANSMISSION_MODE__REQUEST_WITH_RETRIGGER` allows the repeated initiating of broadcast requests. This repeating can be deactivated by the command G_Most_Tp_Broadcast_StopTransmission .

The following return values are available:

- `G_MOST__TP__BROADCAST__TRANSMISSION_MODE__REQUEST`

  Request
- `G_MOST__TP__BROADCAST__TRANSMISSION_MODE__REQUEST_WITH_RETRIGGER`

  Request with retriggering
- `G_MOST__TP__BROADCAST__TRANSMISSION_MODE__RESPONSE`

  Response
- `G_MOST__TP__BROADCAST__TRANSMISSION_MODE__REQUEST_WITH_RESPONSE`

  Request with response

`data`
Pointer to the data buffer for data bytes of the broadcast message

`dataLength`
Number of data bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Tp_Broadcast_StopTransmission

G_Most_Tp_Broadcast_StopTransmission — Stop the transmission

Description

This command stops the repeated transmission of a broadcast message.

Definition

```
G_Error_t
G_Most_Tp_Broadcast_StopTransmission(
 const G_PortHandle_t portHandle,
 const u8_t channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Tp_BroadCast_GetData

G_Most_Tp_BroadCast_GetData — Data request of a received broadcast message

Description

This command requests data of a received broadcast message.

Definition

```
G_Error_t
G_Most_Tp_Broadcast_GetData(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 u8_t * const responseData,
 u32_t * const responseLength
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`responseData`
Pointer to the response data buffer

`responseLength`
Pointer to the variable `reponseLength`

Content when invoked: Size of response data buffer

Returned content: Size of response data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Tp_BroadCast_GetData_Async

G_Most_Tp_BroadCast_GetData_Async — Data request of a received broadcast message (asynchronous request)

Description

This command initiates the asynchronous request of the data of a received broadcast message.

In contrast to G_Most_Tp_BroadCast_GetData , this command returns immediately (that means without waiting for the response). As soon as the corresponding response is available, the callback function set by G_Most_Tp_BroadCast_GetData_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Most_Tp_BroadCast_GetData_Async(
 const G_PortHandle_t portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Tp_BroadCast_GetData_Async_Callback

G_Most_Tp_BroadCast_GetData_Async_Callback — Callback function for the data request of a received broadcast message

Description

This function will be automatically called if a response for the asynchronous data request of a received broadcast message with G_Most_Tp_BroadCast_GetData_Async is available. It has to be entered by the user and set with G_Most_Tp_BroadCast_GetData_Async_AddCallback.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Most_Tp_BroadCast_GetData_Async_Callback(
 const G_PortHandle_t  portHandle,
 const u8_t channel,
 const u8_t * const responseData,
 const u32_t  responseLength
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel

`responseData`
: Pointer to the data of the received broadcast message

`responseLength`
: Data size of the received broadcast message in byte

G_Most_Tp_BroadCast_GetData_Async_AddCallback

G_Most_Tp_BroadCast_GetData_Async_AddCallback — Set the callback function for the asynchronous data request of a received broadcast message

Description

This command defines a callback function for the asynchronous data request of a received broadcast message with G_Most_Tp_BroadCast_GetData_Async.

Definition

```
G_Error_t
G_Most_Tp_BroadCast_GetData_Async_AddCallback(
 const G_PortHandle_t portHandle,
 const u8_t  channel,
 const G_Most_Tp_Broadcast_GetData_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`callback`
Function pointer of the callback function (see G_Most_Tp_BroadCast_GetData_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Tp_BroadCast_GetData_Async_RemoveCallback

G_Most_Tp_BroadCast_GetData_Async_RemoveCallback — Remove the callback function for the asynchronous data request of a received broadcast message

Description

This command removes a callback function for the asynchronous data request of a received broadcast message.

Definition

```
G_Error_t
G_Most_Tp_BroadCast_GetData_Async_RemoveCallback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

10 TpDirect

These commands are used to control the direct access to the transport protocol.

G_Most_TpDirect_GetResponse

G_Most_TpDirect_GetResponse — Query the response

Description

This command directly queries responses from the transport protocol.

Definition

```
G_Error_t
G_Most_TpDirect_GetResponse(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 G_Error_t * const lastErrorCode,
 G_Most_TpDirect_State_t * const tpDirectState,
 u8_t * const responseData,
 u32_t * const responseLength
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`lastErrorCode`
Pointer to a variable of type **G_Error_t**. The error code of the last error that has occurred in the firmware (or `G_NO_ERROR` = no error) is returned in this variable. For getting an error description you can call G_Common_GetFirmwareErrorDescription.

`tpDirectState`
Variable of type **G_Most_TpDirect_State_t**

The following return values are available:

- `G_MOST__TP_DIRECT__STATE__NOT_INITIALIZED`

  Transport protocol was not initialized
- `G_MOST__TP_DIRECT__STATE__DISCONNECTED`

  Transport protocol is disconnected
- `G_MOST__TP_DIRECT__STATE__CONNECT_IN_PROGRESS`

  Connecting is in progress
- `G_MOST__TP_DIRECT__STATE__CONNECTED`

  Transport protocol is connected
- `G_MOST__TP_DIRECT__STATE__DISCONNECT_IN_PROGRESS`

  Disconnecting is in progress

`responseData`
Pointer to the response data buffer

`responseLength`
Pointer to the variable `reponseLength`

Content when invoked: Size of response data buffer

Returned content: Size of response data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_TpDirect_GetResponse_Async_Enable

G_Most_TpDirect_GetResponse_Async_Enable — Activate the asynchronous receiving of transport protocol messages

Description

This command activates the asynchronous receiving of transport protocol messages. As soon as a message is available, the callback function set with G_Most_TpDirect_GetResponse_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Most_TpDirect_GetResponse_Async_Enable(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_TpDirect_GetResponse_Async_Disable

G_Most_TpDirect_GetResponse_Async_Disable — Deactivate the asynchronous receiving of transport protocol messages

Description

This commmand deactivates the asynchronous receiving of transport protocol messages.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Most_TpDirect_GetResponse_Async_Disable(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_TpDirect_GetResponse_Async_Callback

G_Most_TpDirect_GetResponse_Async_Callback — Callback function for the asynchronous receiving of transport protocol messages

Description

This function will be automatically called if during asynchronous operation a transport protocol message is received. It must be entered by the user and set with G_Most_TpDirect_GetResponse_Async_AddCallback . Furthermore, the asynchronous receiving of transport protocol messages has to be activated with G_Most_TpDirect_GetResponse_Async_Enable.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Most_TpDirect_GetResponse_Async_Callback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Error_t  lastErrorCode,
 const G_Most_TpDirect_State_t  tpDirectState,
 const u8_t * const responseData,
 const u32_t  responseLength
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`lastErrorCode`
Pointer to a variable of type **G_Error_t**. The error code of the last error that has occurred in the firmware (or `G_NO_ERROR` = no error) is returned in this variable. For getting an error description, you can call G_Common_GetFirmwareErrorDescription.

`tpDirectState`
Variable of type **G_Most_TpDirect_State_t**

The following return values are available:

- `G_MOST__TP_DIRECT__STATE__NOT_INITIALIZED`

  Transport protocol was not initialized
- `G_MOST__TP_DIRECT__STATE__DISCONNECTED`

  Transport protocol is disconnected
- `G_MOST__TP_DIRECT__STATE__CONNECT_IN_PROGRESS`

  Connecting is in progress
- `G_MOST__TP_DIRECT__STATE__CONNECTED`

  Transport protocol is connected
- `G_MOST__TP_DIRECT__STATE__DISCONNECT_IN_PROGRESS`

  Disconnecting is in progress

`responseData`
Pointer to the response data buffer

`responseLength`
Pointer to the variable `reponseLength`

Content when invoked: Size of response data buffer

Returned content: Size of response data

G_Most_TpDirect_GetResponse_Async_AddCallback

G_Most_TpDirect_GetResponse_Async_AddCallback — Set the callback function for the asynchronous receiving of transport protocol messages

Description

This command defines a callback function for the asynchronous receiving of transport protocol messages.

Definition

```
G_Error_t
G_Most_TpDirect_GetResponse_Async_AddCallback(
 const G_PortHandle_t portHandle,
 const u8_t  channel,
 const G_Most_TpDirect_GetResponse_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`callback`
Function pointer of the callback function (see G_Most_TpDirect_GetResponse_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_TpDirect_GetResponse_Async_RemoveCallback

G_Most_TpDirect_GetResponse_Async_RemoveCallback — Remove callback function for the asynchronous receiving of transport protocol messages

Description

This command removes a callback function for the asynchronous receiving of transport protocol messages.

Definition

```
G_Error_t
G_Most_TpDirect_GetResponse_Async_RemoveCallback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_TpDirect_GetState

G_Most_TpDirect_GetState — Query the TpDirect status

Description

This command queries the TpDirect status.

Definition

```
G_Error_t
G_Most_TpDirect_GetState(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Most_TpDirect_GetState_CmdFlags_t cmdFlags,
 G_Error_t * const lastErrorCode,
 G_Most_Tp_Type_t * const tpType,
 G_Most_TpDirect_Type_t * const tpDirectType,
 G_Most_TpDirect_State_t * const tpDirectState,
 G_Most_TpDirect_GetState_RspFlags_t * const rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`cmdFlags`
Variable of type **G_Most_TpDirect_GetState_CmdFlags_t**

The following flags are available:

- `G_MOST__TP_DIRECT__GET_STATE__CMD_FLAG__NONE`

  No flag was set
- `G_MOST__TP_DIRECT__GET_STATE__CMD_FLAG__RESET_LAST_ERROR`

  Reset the last error that has occurred in the firmware

`lastErrorCode`
Pointer to a variable of type **G_Error_t**. The error code of the last error that has occurred in the firmware (or `G_NO_ERROR` = no error) is returned in this variable. For getting an error description, you can call G_Common_GetFirmwareErrorDescription.

`tpType`
Pointer to the variable of type **G_Most_Tp_Type_t**

The following return values are available:

- `G_MOST__TP__TYPE__OFF`

  No transport protocol
- `G_MOST__TP__TYPE__TP_2_0`

  Transport protocol TP 2.0
- `G_MOST__TP__TYPE__KEYWORD_REC`

  Transport protocol Keyword Rec

`tpDirectType`
Pointer to the variable of type **G_Most_TpDirect_Type_t**

The following return values are available:
- `G_MOST__TP_DIRECT__TYPE__OFF`

  No direct access to the transport protocol
- `G_MOST__TP_DIRECT__TYPE__NORMAL`

  Direct access to the transport protocol
- `G_MOST__TP_DIRECT__STATE__CONNECT_IN_PROGRESS`

  Connecting is in progress

`tpDirectState`
Pointer to a variable of type **G_Most_TpDirect_State_t**

The following return values are available:
- `G_MOST__TP_DIRECT__STATE__NOT_INITIALIZED`

  Transport protocol was not initialized
- `G_MOST__TP_DIRECT__STATE__DISCONNECTED`

  Transport protocol is disconnected
- `G_MOST__TP_DIRECT__STATE__CONNECT_IN_PROGRESS`

  Connecting is in progress
- `G_MOST__TP_DIRECT__STATE__CONNECTED`

  Transport protocol is connected
- `G_MOST__TP_DIRECT__STATE__DISCONNECT_IN_PROGRESS`

  Disconnecting is in progress

`rspFlags`
Pointer to a variable of type **G_Most_TpDirect_GetState_RspFlags_t**

The following return flags are available:
- `G_MOST__TP_DIRECT__GET_STATE__RSP_FLAG__NONE`

  No flag was set
- `G_MOST__TP_DIRECT__GET_STATE__RSP_FLAG__TX_BUSY`

  A request could not be successfully sent
- `G_MOST__TP_DIRECT__GET_STATE__RSP_FLAG__RX_BUFFER_NOT_EMPTY`

  Receiving buffer is not empty

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_TpDirect_GetState_Async

G_Most_TpDirect_GetState_Async — Query the TpDirect status (by asynchronous query)

Description

This command initiates the asynchronous query of the TpDirect status.

In contrast to G_Most_TpDirect_GetState , this command returns immediately (that means without waiting for the response). As soon as the corresponding response is available, the callback function set by G_Most_TpDirect_GetState_Async_AddCallback will be called.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
G_Error_t
G_Most_TpDirect_GetState_Async(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Most_TpDirect_GetState_CmdFlags_t cmdFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`cmdFlags`
Variable of type **G_Most_TpDirect_GetState_CmdFlags_t**

The following flags are available:

- `G_MOST__TP_DIRECT__GET_STATE__CMD_FLAG__NONE`

  No flag was set
- `G_MOST__TP_DIRECT__GET_STATE__CMD_FLAG__RESET_LAST_ERROR`

  Reset of the last error that has occurred in the firmware

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_TpDirect_GetState_Async_Callback

G_Most_TpDirect_GetState_Async_Callback — Callback function for the asynchronous query of the TpDirect status

Description

This function will be automatically called if a response for the asynchronous query of the TpDirect status with G_Most_TpDirect_GetState_Async is available. It must be entered by the user and be set with G_Most_TpDirect_GetState_Async_AddCallback.

See Asynchronous functionality for a detailed description of asynchronous functionality.

Definition

```
void
G_Most_TpDirect_GetState_Async_Callback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Error_t  lastErrorCode,
 const G_Most_Tp_Type_t  tpType,
 const G_Most_TpDirect_Type_t  tpDirectType,
 const G_Most_TpDirect_State_t  tpDirectState,
 const G_Most_TpDirect_GetState_RspFlags_t  rspFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`lastErrorCode`
Variable of type **G_Error_t**

The error code of the last error that has occurred in the firmware (or `G_NO_ERROR` = no error) is returned in this variable. For getting an error description, you can call G_Common_GetFirmwareErrorDescription.

`tpType`
Variable of type **G_Most_Tp_Type_t**

The following return values are available:

- `G_MOST__TP__TYPE__OFF`

  No transport protocol
- `G_MOST__TP__TYPE__TP_2_0`

  Transport protocol TP 2.0
- `G_MOST__TP__TYPE__KEYWORD_REC`

  Transport protocol Keyword Rec

`tpDirectType`
Variable of type **G_Most_TpDirect_Type_t**

The following return values are available:
- `G_MOST__TP_DIRECT__TYPE__OFF`

  No direct access to the transport protocol
- `G_MOST__TP_DIRECT__TYPE__NORMAL`

  Direct access to the transport protocol
- `G_MOST__TP_DIRECT__STATE__CONNECT_IN_PROGRESS`

  Connecting is in progress

`tpDirectState`
Variable of type **G_Most_TpDirect_State_t**

The following return values are available:
- `G_MOST__TP_DIRECT__STATE__NOT_INITIALIZED`

  Transport protocol was not initialized
- `G_MOST__TP_DIRECT__STATE__DISCONNECTED`

  Transport protocol is disconnected
- `G_MOST__TP_DIRECT__STATE__CONNECT_IN_PROGRESS`

  Connecting is in progress
- `G_MOST__TP_DIRECT__STATE__CONNECTED`

  Transport protocol is connected
- `G_MOST__TP_DIRECT__STATE__DISCONNECT_IN_PROGRESS`

  Disconnecting in progress

`rspFlags`
Variable of type **G_Most_TpDirect_GetState_RspFlags_t**

The following return flags are available:
- `G_MOST__TP_DIRECT__GET_STATE__RSP_FLAG__NONE`

  No flag was set
- `G_MOST__TP_DIRECT__GET_STATE__RSP_FLAG__TX_BUSY`

  A request could not be successfully sent
- `G_MOST__TP_DIRECT__GET_STATE__RSP_FLAG__RX_BUFFER_NOT_EMPTY`

  Receiving buffer is not empty

G_Most_TpDirect_GetState_Async_AddCallback

G_Most_TpDirect_GetState_Async_AddCallback — Set the callback function for the asynchronous query of the TpDirect status

Description

This command defines a callback function for the asynchronous query of the TpDirect status with G_Most_TpDirect_GetState_Async.

Definition

```
G_Error_t
G_Most_TpDirect_GetState_Async_AddCallback(
 const G_PortHandle_t portHandle,
 const u8_t  channel,
 const G_Most_TpDirect_GetState_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel

`callback`
Function pointer of the callback function (see G_Most_TpDirect_GetState_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_TpDirect_GetState_Async_RemoveCallback

G_Most_TpDirect_GetState_Async_RemoveCallback — Remove the callback function for the asynchronous query of the TpDirect status

Description

This command removes a callback function for the asynchronous query of the TpDirect status.

Definition

```
G_Error_t
G_Most_TpDirect_GetState_Async_RemoveCallback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_TpDirect_Init_Normal

G_Most_TpDirect_Init_Normal — Initialize the direct access to the transport protocol

Description

This command initializes the direct access to the transport protocol.

Definition

```
G_Error_t
G_Most_TpDirect_Init_Normal(
 const G_PortHandle_t portHandle,
 const u8_t channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_TpDirect_Init_Off

G_Most_TpDirect_Init_Off — Remove the direct access to the transport protocol

Description

This command removes the direct access to the transport protocol.

Definition

```
G_Error_t
G_Most_TpDirect_Init_Off(
 const G_PortHandle_t portHandle,
 const u8_t channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_TpDirect_InitiateStart

G_Most_TpDirect_InitiateStart — Initiate the direct access to the transport protocol

Description

This command initiates the start of the direct access to the transport protocol.

Please consider that on function return it cannot be ensured that the starting operation has been completed. The starting request is only transferred to the hardware.

Definition

```
G_Error_t
G_Most_TpDirect_InitiateStart(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Most_TpDirect_RequestMode_t requestMode,
 const u8_t * const data,
 const u32_t dataLength
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`requestMode`
Variable of type **G_Most_TpDirect_RequestMode_t**

The following return values are available:

- `G_MOST__TP_DIRECT__REQUEST_MODE__PHYSICAL`

  Physical message
- `G_MOST__TP_DIRECT__REQUEST_MODE__FUNCTIONAL`

  Functional message

`data`
Pointer to data to be transmitted

`dataLength`
Length of the data to be transmitted in byte

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_TpDirect_InitiateStop

G_Most_TpDirect_InitiateStop — Initiate the stop of the direct access to the transport protocol

Description

This command initiates the stop of the direct access to the transport protocol.

Please consider that on function return it cannot be ensured that the stopping operation has been completed. The stopping request is only transferred to the hardware.

Definition

```
G_Error_t
G_Most_TpDirect_InitiateStop(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Most_TpDirect_RequestMode_t requestMode,
 const u8_t * const data,
 const u32_t dataLength
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`requestMode`
Variable of type **G_Most_TpDirect_RequestMode_t**

The following return values are available:

- `G_MOST__TP_DIRECT__REQUEST_MODE__PHYSICAL`

  Physical message
- `G_MOST__TP_DIRECT__REQUEST_MODE__FUNCTIONAL`

  Functional message

`data`
Pointer to data to be transmitted

`dataLength`
Length of the data to be transmitted in byte

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_TpDirect_QueueRequest

G_Most_TpDirect_QueueRequest — Send the transport protocol message

Description

This command enqueues a transport protocol message into the sending operation of the specified hardware.

Please consider that on function return it cannot be ensured that the corresponding message has been transmitted by the hardware. The request is only transferred to the hardware. If you want to make sure that the request has been transmitted, it is required to use G_Most_TpDirect_Request .

Definition

```
G_Error_t
G_Most_TpDirect_QueueRequest(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const G_Most_TpDirect_RequestMode_t requestMode,
 const u8_t * const data,
 const u32_t dataLength
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`requestMode`
Variable of type **G_Most_TpDirect_RequestMode_t**

The following return values are available:

- `G_MOST__TP_DIRECT__REQUEST_MODE__PHYSICAL`

  Physical message
- `G_MOST__TP_DIRECT__REQUEST_MODE__FUNCTIONAL`

  Functional message

`data`
Pointer to data to be transmitted

`dataLength`
Length of the data to be transmitted in byte

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_TpDirect_Request

G_Most_TpDirect_Request — Send the transport protocol message

Description

This command sends a transport protocol message.

Definition

```
G_Error_t
G_Most_TpDirect_Request(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const u32_t txTimeout,
 const G_Most_TpDirect_RequestMode_t requestMode,
 const u8_t * const data,
 const u32_t dataLength,
 G_Error_t * const lastErrorCode
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`txTimeout`
Sending timeout in microseconds, e.g. 250000 µs

`requestMode`
Variable of type **G_Most_TpDirect_RequestMode_t**

The following return values are possible:

- `G_MOST__TP_DIRECT__REQUEST_MODE__PHYSICAL`

  Physical message
- `G_MOST__TP_DIRECT__REQUEST_MODE__FUNCTIONAL`

  Functional message

`data`
Pointer to data to be transmitted

`dataLength`
Length of data to be transmitted in byte

`lastErrorCode`
Pointer to a variable of type **G_Error_t**

The error code of last error that has occurred in the firmware (or `G_NO_ERROR` = no error) is returned in this variable. For getting an error description you can call G_Common_GetFirmwareErrorDescription.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_TpDirect_Reset

G_Most_TpDirect_Reset — Reset the multisession channel

Description

This command resets the multisession channel that is indicated by `channel`.

Definition

```
G_Error_t
G_Most_TpDirect_Reset(
 const G_PortHandle_t portHandle,
 const u8_t channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_TpDirect_Start

G_Most_TpDirect_Start — Start the direct access to the transport protocol

Description

This command starts the direct access to the transport protocol.

Definition

```
G_Error_t
G_Most_TpDirect_Start(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const u32_t timeout,
 const G_Most_TpDirect_RequestMode_t requestMode,
 const u8_t * const data,
 const u32_t dataLength,
 G_Error_t * const lastErrorCode
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`timeout`
Sending timeout, given in microseconds, e.g. 250000 µs

`requestMode`
Variable of type **G_Most_TpDirect_RequestMode_t**

The following return values are available:

- `G_MOST__TP_DIRECT__REQUEST_MODE__PHYSICAL`

  Physical message
- `G_MOST__TP_DIRECT__REQUEST_MODE__FUNCTIONAL`

  Functional message

`data`
Pointer to the data to be transmitted

`dataLength`
Length of the data to be transmitted in byte

`lastErrorCode`
Pointer to a variable of type **G_Error_t**

The error code of the last error that has occurred in the firmware (or `G_NO_ERROR` = no error) is returned in this variable. For getting an error description, you can call G_Common_GetFirmwareErrorDescription.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_TpDirect_Stop

G_Most_TpDirect_Stop — Stop the direct access to the transport protocol

Description

This command stops the direct access to the transport protocol.

Definition

```
G_Error_t
G_Most_TpDirect_Stop(
 const G_PortHandle_t portHandle,
 const u8_t channel,
 const u32_t timeout,
 const G_Most_TpDirect_RequestMode_t requestMode,
 const u8_t * const data,
 const u32_t dataLength,
 G_Error_t * const lastErrorCode
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (starting with 0)

`timeout`
Sending timeout, given in microseconds, e.g. 250000 µs

`requestMode`
Variable of type **G_Most_TpDirect_RequestMode_t**

The following return values are available:

- `G_MOST__TP_DIRECT__REQUEST_MODE__PHYSICAL`

  Physical message
- `G_MOST__TP_DIRECT__REQUEST_MODE__FUNCTIONAL`

  Functional message

`data`
Pointer to data to be transmitted

`dataLength`
Length of data to be transmitted in byte

`lastErrorCode`
Pointer to a variable of type **G_Error_t**

The error code of the last error that has occurred in the firmware (or `G_NO_ERROR` = no error) is returned in this variable. For getting an error description you can call G_Common_GetFirmwareErrorDescription.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

11 Monitor

These commands are used for controlling the monitor functionality.

An example of using the **MOST Monitor** functionality is included in the **samples directory** .

G_Most_Monitor_StreamingDataConfig_Set

G_Most_Monitor_StreamingDataConfig_Set — Set the streaming data configuration

Description

The command is used to set the configuration of monitor entries with identifier ID = `G_MOST__MONITOR__ID__STREAMING_DATA`.

Definition

```
G_Error_t
G_Most_Monitor_StreamingDataConfig_Set(
 const G_PortHandle_t  portHandle,
 const G_Most_Monitor_StreamingDataConfig_Cmd_Flags_t  cmdFlags,
 const u8_t  numberOfBytes,
 const u8_t * const bytes
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

- `G_MOST__MONITOR__STREAMING_DATA_CONFIG__FLAG__NONE`

  No flag set
- `G_MOST__MONITOR__STREAMING_DATA_CONFIG__FLAG__ENABLE_TIMESTAMP`

  Enable monitor timestamp for streaming data
- `G_MOST__MONITOR__STREAMING_DATA_CONFIG__FLAG__ENABLE_FRAME_BYTE_0`

  Enable monitoring of byte number 0 of the data frame

  (byte number 0 is normally not part of the streaming data)
- `G_MOST__MONITOR__STREAMING_DATA_CONFIG__FLAG__ENABLE_FRAME_BYTE_61`

  Enable monitoring of byte number 61 of the data frame

  (byte number 61 is normally not part of the streaming data)
- `G_MOST__MONITOR__STREAMING_DATA_CONFIG__FLAG__ENABLE_FRAME_BYTE_62`

  Enable monitoring of byte number 62 of the data frame

  (byte number 62 is normally not part of the streaming data)
- `G_MOST__MONITOR__STREAMING_DATA_CONFIG__FLAG__ENABLE_FRAME_BYTE_63`

  Enable monitoring of byte number 63 of the data frame

  (byte number 63 is normally not part of the streaming data)

`numberOfBytes`
Number of streaming data bytes that are to be monitored

`bytes`
Pointer to array with byte numbers of streaming data bytes that are to be monitored

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
cmdFlags =
  G_MOST__MONITOR__STREAMING_DATA_CONFIG__CMD_FLAG__ENABLE_TIMESTAMP;

numberOfBytes = 2;

bytes[0] = 1;
bytes[1] = 3;

rc =
  G_Most_Monitor_StreamingDataConfig_Set(
    portHandle,
    cmdFlags,
    numberOfBytes,
    bytes
);
```

The monitor is configured to monitor streaming data bytes number 1 and 3 including the timestamp.

G_Most_Monitor_StreamingDataConfig_Get

G_Most_Monitor_StreamingDataConfig_Get — Get the streaming data configuration

Description

The command is used to query the configuration of monitor entries with identifier ID = `G_MOST__MONITOR__ID__STREAMING_DATA`.

Definition

```
G_Error_t
G_Most_Monitor_StreamingDataConfig_Get(
 const G_PortHandle_t  portHandle,
 G_Most_Monitor_StreamingDataConfig_Cmd_Flags_t * const rspFlags,
 u8_t * const numberOfBytes,
 u8_t * const bytes
);
```

Parameters

`portHandle`
Handle to the communication port

`rspFlags`
Response flags

The following values are possible and can be combined:

- `G_MOST__MONITOR__STREAMING_DATA_CONFIG__FLAG__NONE`

  No flag set
- `G_MOST__MONITOR__STREAMING_DATA_CONFIG__FLAG__ENABLE_TIMESTAMP`

  Enable monitor timestamp for streaming data
- `G_MOST__MONITOR__STREAMING_DATA_CONFIG__FLAG__ENABLE_FRAME_BYTE_0`

  Enable monitoring of byte number 0 of the data frame

  (byte number 0 is normally not part of the streaming data)
- `G_MOST__MONITOR__STREAMING_DATA_CONFIG__FLAG__ENABLE_FRAME_BYTE_61`

  Enable monitoring of byte number 61 of the data frame

  (byte number 61 is normally not part of the streaming data)
- `G_MOST__MONITOR__STREAMING_DATA_CONFIG__FLAG__ENABLE_FRAME_BYTE_62`

  Enable monitoring of byte number 62 of the data frame

  (byte number 62 is normally not part of the streaming data)
- `G_MOST__MONITOR__STREAMING_DATA_CONFIG__FLAG__ENABLE_FRAME_BYTE_63`

  Enable monitoring of byte number 63 of the data frame

  (byte number 63 is normally not part of the streaming data)

`numberOfBytes`
Number of configured bytes

`bytes`
Pointer to array with byte numbers of configured streaming data bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Monitor_Start

G_Most_Monitor_Start — Start the MOST monitor

Description

This command is used to configure and start the MOST monitor.

Definition

```
G_Error_t
G_Most_Monitor_Start(
 const G_PortHandle_t  portHandle,
 const G_Most_Monitor_Start_CmdFlags_t  cmdFlags,
 const u32_t  bufferSize,
 const u32_t  numberOfIds,
 const G_Most_Monitor_Id_t * const ids
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

- G_MOST__MONITOR__START__CMD_FLAG__NONE

  No command flag set

`bufferSize`
Size of the internal monitor buffer (0 = use minimum possible size)

`numberOfIds`
Number of different IDs that are to be monitored

`ids`
MOST monitor item IDs

The following values are possible:

`G_MOST__MONITOR__ID__DATA_LOST`
The hardware was not able to capture the whole data.

`G_MOST__MONITOR__ID__COMPLETE_FRAME`
Complete MOST Frames

`G_MOST__MONITOR__ID__LIGHT_ON`
Light On Events

`G_MOST__MONITOR__ID__LIGHT_OFF`
Light Off Events

`G_MOST__MONITOR__ID__LOCKED`
Locked Events

A **Lock Event** is generated when the network controller synchronized to the input signal for the first time after an **Unlock Event** .

G_MOST__MONITOR__ID__UNLOCKED
Unlocked Events

An **Unlocked Event** is generated when the network controller can no longer synchronize to the signal at the input.

G_MOST__MONITOR__ID__RISING_EDGE_AT_TRIGGER_INPUT_1
Rising Edge Events at trigger input 1

G_MOST__MONITOR__ID__FALLING_EDGE_AT_TRIGGER_INPUT_1
Falling Edge Events at trigger input 1

G_MOST__MONITOR__ID__RISING_EDGE_AT_TRIGGER_INPUT_2
Rising Edge Events at trigger input 2

G_MOST__MONITOR__ID__FALLING_EDGE_AT_TRIGGER_INPUT_2
Falling Edge Events at trigger input 2

G_MOST__MONITOR__ID__CONTROL_MESSAGE
Control Messages

Control messages are messages for controlling the MOST system. They are transmitted on the control channel or use the asynchronous area.

G_MOST__MONITOR__ID__ALLOCATION_TABLE_MESSAGE
Allocation Table Messages

Allocation Table Messages are messages for broadcasting the CRA (Channel Resource Allocation Table) or for allocation/deallocation - requests of streaming data channels.

G_MOST__MONITOR__ID__STREAMING_DATA
Streaming Data Messages

Streaming Data Messages are transmitted in the synchronous area and contain mostly audio or video data.

Before starting to monitor **Streaming Data** , you need to tell the monitor which **Streaming Data Bytes** are to be monitored by command G_Most_Monitor_StreamingDataConfig_Set.

G_MOST__MONITOR__ID__PACKET_DATA
Packet Data Messages

G_MOST__MONITOR__ID__SPY__CONTROL_MESSAGE
Spy Control Messages

The single bytes of a Spy Control Message are described in the following table:

Byte Number	Description
0	The first byte indicates the high byte of the spy message's length excluding the length itself.
1	The second byte indicates the low byte of the spy message's length excluding the length itself.
2	Administration byte
3	Administration byte
4	Priority
5	Target device low byte

Byte Number	Description
6	Target device high byte
7	PACK: The PACK byte of the data packet is used to communicate a preemptive acknowledge from all potential packet receiver(s) to the packet transmitter.
8	Packet length high
9	Packet length low (packet length - 1 = number of following bytes up to CACK (including))
10	Packet counter: It increments when a packet transmission has been completed or if it is the last retry of a packet.
11	Source device high byte
12	Source device low byte
13	Data byte 0 of the control message
14	Data byte 1 of the control message
...	...
r-5	Data byte s-2 of the control message
r-4	Data byte s-1 of the control message
r-3	CRC high
r-2	CRC low
r-1	CACK

Number of bytes: r

Number of data bytes of the control message: s = r-16

If necessary zeros are added at the back to the spy message to align things at a quadlet boundary.

`G_MOST__MONITOR__ID__SPY__MOST_PACKET`

Spy Packet Data Messages

There are 2 types of Spy Data Messages: **MOST Data Packet** and **MOST Ethernet Packet** . They can be identified by the highest bit of byte 4.

Highest bit of byte 4 = **0** : **MOST Data Packet**

Highest bit of byte 4 = **1** : **MOST Ethernet Packet**

The single bytes of a Spy **MOST Data Packet** are described in the following table:

Byte Number	Description
0	The first byte indicates the high byte of the spy message's length excluding the length itself.
1	The second byte indicates the low byte of the spy message's length excluding the length itself.
2	Administration byte
3	Administration byte
4	Packet length high; high bit = 0
5	Packet length low (packet length - 1 = number of following bytes up to CACK (including))

Byte Number	Description
6	Target device low byte
7	Target device high byte
8	PACK: The PACK byte of the packet is used to communicate a preemptive acknowledge from all potential packet receiver(s) to the packet transmitter.
9	Packet Counter: It increments when a packet transmission has been completed or if it is the last retry of a packet.
10	Source device high byte
11	Source device low byte
12	Data byte 0 of the packet message
13	Data byte 1 of the packet message
...	...
r-5	Data byte s-2 of the packet message
r-4	Data byte s-1 of the packet message
r-3	CRC high
r-2	CRC low
r-1	CACK

Number of bytes in case a packet message is transmitted via a packet channel: r

Number of data bytes: s = r - 15

If necessary zeros are added at the back to the spy message to align things at a quadlet boundary.

The single bytes of a Spy **MOST Ethernet Packet** are described in the following table:

Byte Number	Description
0	The first byte indicates the high byte of the spy message's length excluding the length itself.
1	The second byte indicates the low byte of the spy message's length excluding the length itself.
2	Administration byte
3	Administration byte
4	Packet length high; high bit = 1
5	Packet length low (packet length - 1 = number of following bytes up to CACK (including))
6	Ethernet target address (47:40)
7	Ethernet target address (39:32)
8	Ethernet target address (31:24)
9	Ethernet target address (23:16)
10	Ethernet target address (15:8)
11	Ethernet target address (7:0)
12	PACK: The PACK byte of the packet is used to communicate a preemptive acknowledge from all potential packet receiver(s) to the packet transmitter.
13	Data byte 0 of the packet message

Byte Number	Description
14	Data byte 1 of the packet message
...	...
r-7	Data byte s-2 of the packet message
r-6	Data byte s-1 of the packet message
r-5	CRC0
r-4	CRC1
r-3	CRC2
r-2	CRC3
r-1	CACK

Number of bytes in case a packet message is transmitted via a packet channel: r

Number of data bytes: s = r - 18

If necessary zeros are added at the back to the spy message to align things at a quadlet boundary.

`G_MOST__MONITOR__ID__SPY__NETWORK_STATUS_INFORMATION`
Spy Network Status Information Messages

The single bytes of a Spy Network Status Information Message are described in the following table:

Byte Number	Description
0	Indicates the higher byte (bit 15:8) of the length of the network status information message excluding the length itself.
1	Indicates the lower byte (bit 7:0) of the length of the network status information message excluding the length itself.
2	MPR (Maximum Position Register), low byte: bit 6:0 (bit 7: reserved)
3	MPR (Maximum Position Register), high byte: bit (15:8): 0
4	MDC: MOST Data Channel, the width of the packet data channel on the MOST network; low byte
5	MDC: MOST Data Channel, the width of the packet data channel on the MOST network; high byte
6	Network States, low byte (bit 7:0): bit 0: System Lock Flag bit 1: Shutdown Flag bit 2: Reserved bit 3: Reserved bit (7:4): Free
7	Network States, high byte (bit 15:8): Reserved
8	Node Postition, low byte (bit 7:0)
9	Node Position, high byte (bit 15:8): Reserved
10	Timestamp byte 0, LSB

Byte Number	Description
11	Timestamp byte 1
12	Timestamp byte 2
13	Timestamp byte 3 (MSB)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
G_Most_Monitor_Id_t  ids;
u32_t  numberOfIds;
u32_t  bufferSize;
G_Most_Monitor_Start_CmdFlags_t  cmdFlags;
G_Error_t  rc;

cmdFlags = G_MOST__MONITOR__START__CMD_FLAG__NONE;

bufferSize = 50000;

numberOfIds = 2;
ids[0] = G_MOST__MONITOR__ID__LIGHT_ON;
ids[1] = G_MOST__MONITOR__ID__CONTROL_MESSAGE;

rc =
  G_Most_Monitor_Start(
    PortHandle,
    cmdFlags,
    bufferSize,
    numberOfIds,
    ids
  );
```

The MOST monitor is configured for monitoring **LightOn Events** and **Control Messages** . The size of the internal buffer is set to 50000 bytes.

G_Most_Monitor_Stop

G_Most_Monitor_Stop — Stop the MOST monitor

Description

This command is used to stop the MOST monitor.

Definition

```
G_Error_t
G_Most_Monitor_Stop(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Monitor_GetData

G_Most_Monitor_GetData — Get MOST monitor data

Description

This command is used to query received monitor data.

Definition

```
G_Error_t
G_Most_Monitor_GetData(
 const G_PortHandle_t  portHandle,
 u8_t * const bufferOverrun,
 u32_t * const length,
 u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`bufferOverrun`
Indicates a buffer overrun

0 = no buffer overrun occurred

1 = a buffer overrun occurred

`length`
Length of the returned data in bytes

`data`
Pointer to returned monitor data

The data consists of the received monitor items, starting at offsets that are multiples of 4.

A monitor item has the following structure:

- Byte **0..1** : Length of item in bytes
- Byte **2..3** : ID of item (see MOST Monitor Ids)
- Byte **4..11** : Timestamp (in nanoseconds)

For streaming data (ID = `G_MOST__MONITOR__ID__STREAMING_DATA`) the timestamp is deactivated by default. Therefore the data bytes will start at this byte offset. By calling G_Most_Monitor_StreamingDataConfig_Set with command flag `G_MOST__MONITOR__STREAMING_DATA_CONFIG__FLAG__ENABLE_TIMESTAMP` set, the timestamp can be activated for streaming data.

- **Byte 12..(length-1)**: Monitor item data

As the starting offset of each monitor item is at an index that is a multiple of 4, you need to round the length information of each item supplied by `length` to a multiple of 4, to get the starting index of the following item.

Example

An example of using the **MOST Monitor** functionality is included in the **samples directory** .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Monitor_EnabledIds_Get

G_Most_Monitor_EnabledIds_Get — Get configured IDs

Description

This command is used to query the IDs that are configured for monitoring.

Definition

```
G_Error_t
G_Most_Monitor_EnabledIds_Get(
 const G_PortHandle_t  portHandle,
 G_Most_Monitor_EnabledIds_Get_RspFlags_t * const rspFlags,
 u32_t * const numberOfIds,
 G_Most_Monitor_Id_t * const ids
);
```

Parameters

`portHandle`
Handle to the communication port

`rspFlags`
Response flags

The following values are possible and can be combined:

- `G_MOST__MONITOR__ENABLED_IDS__GET__RSP_FLAG__NONE`

  No response flag set

`numberOfIds`
Number of configured IDs

`ids`
Configured IDs

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Monitor_EnabledIds_Change

G_Most_Monitor_EnabledIds_Change — Change monitor configuration

Description

This command is used to change the IDs that are configured for monitoring.

Definition

```
G_Error_t
G_Most_Monitor_EnabledIds_Change(
 const G_PortHandle_t  portHandle,
 G_Most_Monitor_EnabledIds_Change_CmdFlags_t  cmdFlags,
 const u32_t  numberOfIds,
 const G_Most_Monitor_Id_t * const ids
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

- `G_MOST__MONITOR__ENABLED_IDS__GET__CMD_FLAG__NONE`

  No command flag set

`numberOfIds`
Number of IDs to change

`ids`
Ids that are to be changed

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Monitor_InputSelection_Set

G_Most_Monitor_InputSelection_Set — Set the input configuration of the monitor

Description

This command is used to set the input configuration of the monitor.

Definition

```
G_Error_t
G_Most_Monitor_InputSelection_Set(
 const G_PortHandle_t  portHandle,
 const G_Most_Monitor_Input_t  input
);
```

Parameters

`portHandle`
Handle to the communication port

`input`
Input path of the monitor

The following values are possible:

- `G_MOST__MONITOR__INPUT__RX_LINE`

  The interface captures the data at the **RX Line**.

  This is the default configuration.
- `G_MOST__MONITOR__INPUT__TX_LINE`

  The interface captures the data at the **TX Line**.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Monitor_InputSelection_Get

G_Most_Monitor_InputSelection_Get — Query the input configuration of the monitor

Description

This command is used to query the input configuration of the monitor.

Definition

```
G_Error_t
G_Most_Monitor_InputSelection_Get(
 const G_PortHandle_t  portHandle,
 G_Most_Monitor_Input_t * input
);
```

Parameters

`portHandle`
Handle to the communication port

`input`
Input path of the monitor

The following values are possible:

- `G_MOST__MONITOR__INPUT__RX_LINE`

  The interface captures the data at the **RX Line**.

  This is the default configuration.
- `G_MOST__MONITOR__INPUT__TX_LINE`

  The interface captures the data at the **TX Line**.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Monitor_DataHoldTime_Set

G_Most_Monitor_DataHoldTime_Set — Configure the data hold time of the hardware

Description

This command is used to configure the hold time of the hardware monitor.

The hardware stores received monitor data in an internal buffer and sends the stored data to the host when the buffer is full. The **Data Hold Time** tells the hardware how long to keep the data before sending it to the host when the buffer is not full.

Definition

```
G_Error_t
G_Most_Monitor_DataHoldTime_Set(
 const G_PortHandle_t  portHandle,
 const u16_t  holdTime
);
```

Parameters

`portHandle`
Handle to the communication port

`holdTime`
Data Hold Time of the monitor buffer (in microseconds)

The default **Data Hold Time** is 1000 microseconds.

The **Data Hold Time** provides the opportunity to adapt the monitor functionality to your application.

E.g. if you want to react as quick as possible to a specific monitor event, you can set the **Data Hold Time** to a low value.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Monitor_DataHoldTime_Get

G_Most_Monitor_DataHoldTime_Get — Query the data hold time of the hardware

Description

This command is used to query the **Data Hold Time** of the hardware.

The hardware stores received monitor data in an internal buffer and sends the stored data to the host when the buffer is full. The **Data Hold Time** tells the hardware how long to keep the data before sending it to the host when the buffer is not full.

Definition

```
G_Error_t
G_Most_Monitor_DataHoldTime_Get(
 const G_PortHandle_t  portHandle,
 u16_t * const holdTime
);
```

Parameters

`portHandle`
Handle to the communication port

`holdTime`
Data Hold Time of the monitor buffer (in microseconds)

The **Data Hold Time** provides the opportunity to adapt the monitor functionality to your application.

E.g. if you want to react as quick as possible to a specific monitor event, you can set the **Data Hold Time** to a low value.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Monitor_EnableEvent

G_Most_Monitor_EnableEvent — Enable monitor event

Description

This command enables the monitor to signal the event specified by `eventHandle` as soon as monitor data is received.

For general notes on **G-API** event handling, see Section 5, "Events".

Definition

```
G_Error_t
G_Most_Monitor_EnableEvent(
 const G_PortHandle_t portHandle,
 const G_Common_EventHandle_t * const eventHandle
);
```

Parameters

`portHandle`
Handle to the communication port

`eventHandle`
Handle to the event

Each time monitor data is received, the specified handle is signaled.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Most_Monitor_DisableEvent

G_Most_Monitor_DisableEvent — Disable monitor event

Description

This command disables a monitor event.

After this function has been executed, no event will be signaled when new monitor data is received.

For general notes on **G-API** event handling, see Section 5, “Events”.

Definition

```
G_Error_t
G_Most_Monitor_DisableEvent(
 const G_PortHandle_t portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Net2Run Functions

These **G-API** functions provide control over the **Net2Run** -Functionality of your hardware.

The **Net2Run** interface provides a set of functions that enable you to build signal based applications. You can set up a **Rest Bus Simulation** (RBS) and access included signals by name.

1 BroadcastTpC1

Configuration functions for the **Broadcast T**ransport **P**rotocol **C**ustom **1**.

Net2Run Broadcast Transport Protocol Custom 1 Adaptors are required for sending Daimler Broadcasts only and have to be configured manually. The module implements the Daimler Broadcast mechanism from the VSM to all other ECUs (SSA-IS-1069).

G_Net2Run_BroadcastTpC1_TxAdaptor

G_Net2Run_BroadcastTpC1_TxAdaptor — Adapts outgoing messages to the Daimler Broadcast transport protocol

Description

Use this command for configuring an adaptor for outgoing broadcast messages of the Daimler Vehicle Security Manager (VSM)

Definition

```
G_Error_t
G_Net2Run_BroadcastTpC1_TxAdaptor(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_BroadcastTpC1_TxAdaptor_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__BROADCAST__TP_C1__TX_ADAPTOR__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__BROADCAST__TP_C1__TX_ADAPTOR__CMD_FLAG__WAIT_FOR_TX_CONF`
Wait for TxConfirmation before the next message part is sent.

`TxAdaptorId`
Id of the TxAdaptor

`TxInPduId`
Id of the incoming Pdu

`TxOutPduId`
Id of the outgoing Pdu

`MessageLengthIn`
Length of the incoming Pdu

`MessageLengthOut`
Max Length of the outgoing message part

`MessagePartSendDelay`
Time delay between message parts

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_BroadcastTpC1_RxAdaptor

G_Net2Run_BroadcastTpC1_RxAdaptor — Assembles a Pdu out of the incoming messages in the Daimler Broadcast transport protocol.

Description

Use this command for configuring an adaptor for incoming broadcast messages of the Daimler Vehicle Security Manager (VSM)

Definition

```
G_Error_t
G_Net2Run_BroadcastTpC1_RxAdaptor(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_BroadcastTpC1_RxAdaptor_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__BROADCAST__TP_C1__TX_ADAPTOR__CMD_FLAG__NONE`
No flag is set

`RxAdaptorId`
Id of the RxAdaptor

`RxInPduId`
Id of the incoming Pdu

`RxOutPduId`
Id of the outgoing Pdu

`MessageLengthIn`
Max length of the incoming message part

`MessageLengthOut`
Length of the outgoing Pdu

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

2 CAN Interface

Configuration functions for the **CAN Interfaces** of **Net2Run**

Net2Run CAN interfaces are configured automatically via the **Net2Run Network Runtime Configurator** . In case the **Net2Run Network Runtime Configurator** is not used, you can use these functions to access the Net2Run CAN interfaces for manual configuration.

G_Net2Run_CanIf_BaudRate_Config

G_Net2Run_CanIf_BaudRate_Config — Baud rate configuration

Description

Use this function to configure the **baud rate** of a **Net2Run CAN interface**.

Definition

```
G_Error_t
G_Net2Run_CanIf_BaudRate_Config(
 const G_PortHandle_t  portHandle,
 const u8_t  busId,
 const u32_t  baudRate,
 const G_Net2Run_CanIf_BaudRate_Config_ExtendedParam_t * const
 extendedParam
);
```

Parameters

`portHandle`
Handle to the communication port

`busId`
CAN interface number (**0** = 1st CAN interface, **1** = 2nd CAN interface, ...)

`baudRate`
Baud rate in Baud (e.g. **500000** for **500** kBaud)

`extendedParam`
Structure with extended parameters

This structure is optional, if you don't want to specify any optional parameter, the function can be called with `extendedParam` = `NULL`.

The structure contains the following elements:

`SamplePoint_Min`
Minimum sample point, use **0** if you don't want to specify this parameter

`SamplePoint_Max`
Maximum sample point, use **0** if you don't want to specify this parameter

`NumberOfTimeQuanta_Min`
Minimum number of time quanta (1 + `TSeg1` + `TSeg2` = 8..25), use **0** if you don't want to specify this parameter

`NumberOfTimeQuanta_Max`
Maximum number of time quanta (1 + `TSeg1` + `TSeg2` = 8..25), use **0** if you don't want to specify this parameter

`TSeg1_Min`
Minimum number of time quanta before sample point minus one (3..16), use **0** if you don't want to specify this parameter

`TSeg1_Max`
Maximum number of time quanta before sample point minus one (3..16), use **0** if you don't want to specify this parameter

`TSeg2_Min`
Minimum number of time quanta after sample point (2..8), use **0** if you don't want to specify this parameter

`TSeg2_Max`
Maximum number of time quanta after sample point (2..8), use **0** if you don't want to specify this parameter

`Sjw_Min`
Minimum resynchronization jump width (1..4), use **0** if you don't want to specify this parameter

`Sjw_Max`
Maximum resynchronization jump width (1..4), use **0** if you don't want to specify this parameter

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_CanIf_Controller_Config

G_Net2Run_CanIf_Controller_Config — Configuration of Can Controller

Description

Use this command for CAN controller configuration.

Definition

```
G_Error_t
G_Net2Run_CanIf_Controller_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_CanIf_Controller_Config_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__CAN_IF__CONTROLLER__CONFIG__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__CAN_IF__CONTROLLER__CONFIG__CMD_FLAG__FD__TX_BRS__DEFAULT`
This flag is available in `FdMode == G_NET2RUN__CAN_IF__FD_MODE__FD` only

Transmission objects that are initialized as FD messages derive their BRS bit from this flag.

0 = the default value for the BRS bit is set to **0** (**Bit Rate Switch** disabled)

1 = the default value for the BRS bit is set to **1** (**Bit Rate Switch** enabled)

`G_NET2RUN__CAN_IF__CONTROLLER__CONFIG__CMD_FLAG__FD__TX_ESI__DEFAULT`
Transmission objects that are initialized as FD messages derive their ESI bit from this flag.

0 = the default value for the ESI bit is set to 0 (error active)

1 = the default value for the ESI bit is set to 1 (error passive)

`ControllerId`
CAN controller ID

0 = CAN1, **1** = CAN2, **2** = CAN3, ...

`FdMode`
CAN FD mode

The following values are possible:

`G_NET2RUN__CAN_IF__FD_MODE__NO_FD`
Receive and transmit frames in CAN format

`G_NET2RUN__CAN_IF__FD_MODE__FD`
Receive and transmit frames in CAN or CAN FD format

`CanFdPaddingValue`
AUTOSAR name: CanFdPaddingValue

Specifies the value which is used to pad unspecified data in CAN FD frames > 8 bytes for transmission. This is necessary due to the discrete possible values of the DLC if > 8 bytes.

If the length of a PDU which was requested to be transmitted does not match the allowed DLC values, the remaining bytes up to the next possible value shall be padded with this value. e.g. 0xCC

`reserved`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_CanIf_BaudRateFd_Config

G_Net2Run_CanIf_BaudRateFd_Config — Configuration of CAN FD baud rate

Description

Use this command for CAN FD baud rate configuration.

Definition

```
G_Error_t
G_Net2Run_CanIf_BaudRateFd_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_CanIf_BaudRateFd_Config_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__CAN_IF__BAUD_RATE_FD__CONFIG__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__CAN_IF__BAUD_RATE_FD__CONFIG__CMD_FLAG__CHANGE_TDC`
See flag `G_NET2RUN__CAN_IF__BAUD_RATE_FD__CONFIG__CMD_FLAG__TDC` and parameter `TdcOffset`

0 : don't change flag `G_NET2RUN__CAN_IF__BAUD_RATE_FD__CONFIG__CMD_FLAG__TDC` and parameter `TdcOffset`

1 : change flag `G_NET2RUN__CAN_IF__BAUD_RATE_FD__CONFIG__CMD_FLAG__TDC` and parameter `TdcOffset`

`G_NET2RUN__CAN_IF__BAUD_RATE_FD__CONFIG__CMD_FLAG__TDC`
0 : disable transmitter delay compensation

1 : enable transmitter delay compensation

`BaudRate`
Baud rate in baud (e.g. 2000000 for 2 MBaud)

`SamplePoint_Min`
Minimum sample point, use 0 if you don't want to specify this parameter

`SamplePoint_Max`
Maximum sample point, use 0 if you don't want to specify this parameter

`NumberOfTimeQuanta_Min`
Minimum number of time quanta (1 + TSeg1 + TSeg2), use 0 if you don't want to specify this parameter

`NumberOfTimeQuanta_Max`
Maximum number of time quanta (1 + TSeg1 + TSeg2), use 0 if you don't want to specify this parameter

`TSeg1_Min`
Minimum number of time quanta before sample point minus one, use 0 if you don't want to specify this parameter

`TSeg1_Max`
Maximum number of time quanta before sample point minus one, use 0 if you don't want to specify this parameter

`TSeg2_Min`
Minimum number of time quanta after sample point, use 0 if you don't want to specify this parameter

`TSeg2_Max`
Maximum number of time quanta after sample point, use 0 if you don't want to specify this parameter

`Sjw_Min`
Minimum resyncronization jump width (>= 1), use 0 if you don't want to specify this parameter

`Sjw_Max`
Maximum resyncronization jump width (>= 1), use 0 if you don't want to specify this parameter

`ControllerId`
0 = CAN1, 1 = CAN2, 2 = CAN3, ...

`reserved`
reserved parameter (must be initialized with **0**)

`TdcOffset`
Transmitter delay compensation offset [ns]

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_CanIf_RxPdu_MaxNumber

G_Net2Run_CanIf_RxPdu_MaxNumber — Set maximum number of RX PDUs

Description

Use this command to set the maximum number of RX PDUs that can be configured.

This command needs to be called prior to G_Net2Run_CanIf_RxPdu_Config .

Definition

```
G_Error_t
G_Net2Run_CanIf_RxPdu_MaxNumber(
 const G_PortHandle_t  portHandle,
 const u32_t  maxNumber
);
```

Parameters

`portHandle`
: Handle to the communication port

`maxNumber`
: Maximum number of RX PDUs that can be configured

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_CanIf_TxPdu_MaxNumber

G_Net2Run_CanIf_TxPdu_MaxNumber — Set maximum number of TX PDUs

Description

Use this command to set the maximum number of TX PDUs that can be configured.

This command needs to be called prior to G_Net2Run_CanIf_TxPdu_Config .

Definition

```
G_Error_t
G_Net2Run_CanIf_TxPdu_MaxNumber(
 const G_PortHandle_t  portHandle,
 const u32_t  maxNumber
);
```

Parameters

`portHandle`
Handle to the communication port

`maxNumber`
Maximum number of TX PDUs that can be configured

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_CanIf_RxPdu_Config

G_Net2Run_CanIf_RxPdu_Config — Configure RX PDUs

Description

Use this command to configure CAN RX PDUs.

Prior to this command, G_Net2Run_CanIf_RxPdu_MaxNumber needs to be called.

Definition

```
G_Error_t
G_Net2Run_CanIf_RxPdu_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_CanIf_RxPdu_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Structure with RX PDU parameters

The structure contains the following elements:

`Flags`
Command Flags

The following values are possible and can be combined:

`G_NET2RUN__CAN_IF__RX_PDU__CONFIG__FLAG__NONE`
No flag is set

`G_NET2RUN__CAN_IF__RX_PDU__CONFIG__FLAG__ENABLE_RX_INDICATION`
Enable RX Indication to other application layers

Since the RX indication is necessary for almost all applications, it is strongly recommended to configure all RX PDUs with this flag set.

`G_NET2RUN__CAN_IF__RX_PDU__CONFIG__FLAG__USE_DLC_MIN`
0 : there is no minimum DLC required

1 : received messages with a DLC less than `DlcMin` are not processed

`PduId`
Index of PDU that is configured (starting with **0**)

This index uniquely identifies the PDU for later usage.

`CanId`
CAN identifier of the PDU

`DlcMax`
Data length code max: messages with a DLC larger than `Dlc` are not processed (0..8 classic CAN, 0..15 CAN FD)

`BusId`
Specifies the CAN interface the PDU is configured for (**0** = CAN1, **1** = CAN2, ...)

`EcuId`
Identifier of the ECU the PDU is configured for (starting with **0**)

The ECU is useful for **Net2Run ComManager** functions, e.g. Network Management

`DlcMin`
Used if flag `G_NET2RUN__CAN_IF__RX_PDU__CONFIG__FLAG__USE_DLC_MIN` is set

Data length code min: messages with a DLC smaller than `DlcMin` are not processed (0..8 classic CAN, 0..15 CAN FD)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_CanIf_TxPdu_Config

G_Net2Run_CanIf_TxPdu_Config — Configure TX PDUs

Description

Use this command to configure CAN TX PDUs.

Prior to this command, G_Net2Run_CanIf_TxPdu_MaxNumber needs to be called.

Definition

```
G_Error_t
G_Net2Run_CanIf_TxPdu_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_CanIf_TxPdu_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Structure with TX PDU parameters

The structure contains the following elements:

`Flags`
Command Flags

The following values are possible and can be combined:

`G_NET2RUN__CAN_IF__TX_PDU__CONFIG__FLAG__NONE`
No flag is set

`G_NET2RUN__CAN_IF__TX_PDU__CONFIG__FLAG__ENABLE_TX_CONFIRMATION`
Enable TX Confirmation to other application layers

Since the TX confirmation is necessary for almost all applications, it is strongly recommended to configure all TX PDUs with this flag set.

`G_NET2RUN__CAN_IF__TX_PDU__CONFIG__FLAG__USE_MSG_FLAGS`
0 : do not use parameter `MsgFlags`

1 : use parameter `MsgFlags`

`G_NET2RUN__CAN_IF__TX_PDU__CONFIG__FLAG__USE_CAN_FD_PADDING_VALUE`
0 : do not use parameter `CanFdPaddingValue`

1 : use parameter `CanFdPaddingValue`

`PduId`
Index of PDU that is configured (starting with **0**)

This index uniquely identifies the PDU for later usage.

`CanId`
CAN identifier of the PDU

`DlcMax`
Data length code max (0..8 classic CAN, 0..15 CAN FD)

`BusId`
Specifies the CAN interface the PDU is configured for (**0** = CAN1, **1** = CAN2, ...)

`EcuId`
Identifier of the ECU the PDU is configured for (starting with **0**)

The ECU is useful for **Net2Run ComManager** functions, e.g. Network Management

`MsgFlags`
Message flags

Used if flag `G_NET2RUN__CAN_IF__TX_PDU__CONFIG__FLAG__USE_MSG_FLAGS` is set

The following values are possible and can be combined:

`G_NET2RUN__CAN_IF__MSG_FLAGS__ZERO`
No flag is set

`G_NET2RUN__CAN_IF__MSG_FLAG__BRS`
Bit rate switch bit for CAN FD frames

0 = **Bit Rate Switch** is disabled

1 = **Bit Rate Switch** is enabled

`G_NET2RUN__CAN_IF__MSG_FLAG__ESI`
Error state indicator bit for CAN FD frames

0 = transmitting node is error active

1 = transmitting node is error passive

`CanFdPaddingValue`
AUTOSAR name: CanFdPaddingValue

Used if flag `G_NET2RUN__CAN_IF__TX_PDU__CONFIG__FLAG__USE_CAN_FD_PADDING_VALUE` is set

Specifies the value which is used to pad unspecified data in CAN FD frames > 8 bytes for transmission. This is necessary due to the discrete possible values of the DLC if > 8 bytes.

If the length of a PDU which was requested to be transmitted does not match the allowed DLC values, the remaining bytes up to the next possible value shall be padded with this value. e.g. 0xCC

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

3 CAN Network Management Control

Functions for controlling the CAN network management of a Net2Run **Rest Bus Simulation**

G_Net2Run_CanNm_Autosar_Config

G_Net2Run_CanNm_Autosar_Config — CAN Network Management AUTOSAR configuration

Description

Use this command for CAN Network Management AUTOSAR configuration.

Definition

```
G_Error_t
G_Net2Run_CanNm_Autosar_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_CanNm_Autosar_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__CAN_NM__AUTOSAR__FLAGS__NONE`
No flag is set

`G_NET2RUN__CAN_NM__AUTOSAR__FLAG__ENABLE_PASSIVE_MODE`
In the Passive Mode the node is only receiving Network Management PDUs but not transmitting any Network Management PDUs.

`G_NET2RUN__CAN_NM__AUTOSAR__FLAG__ENABLE_BUS_LOAD_REDUCTION`
Enable bus load reduction.

`G_NET2RUN__CAN_NM__AUTOSAR__FLAG__USE_NODE_ID_IN_PDU`
Use node id in network management PDU.

`G_NET2RUN__CAN_NM__AUTOSAR__FLAG__USE_CBV_IN_PDU`
Use node control bit vector in network management PDU.

`G_NET2RUN__CAN_NM__AUTOSAR__FLAG__USE_STATE_IN_PDU`
Use state in network management PDU.

`ControllerId`
Controller ID, starting with 0 (0=CAN1, 1=CAN2, e.t.c.)

`EcuId`
ECU ID, starting with 0

`NmNodeId`
NM node identifier that is used as source node identifier (SNI)

`reserved1`
reserved parameter (must be initialized with **0**)

`CanIdBase`
CAN identifier base (e.g. 0x400), must be divisible by `NumberOfNodes`

`NumberOfNodes`
Maximum number of nodes (e.g. **64** -> results in **CanId** = **0x400** for first node (`NmNodeId`= **0x00**) and **CanId** = **0x43F** for last node (`NmNodeId`= **0x3F**) if `CanIdBase`= **0x400**), must be a power of 2

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`NmTimeoutTime`
Network Timeout [microseconds] for NM-Messages.

It denotes how long the NM shall stay in the Network Mode before transition into Prepare Bus-Sleep Mode shall take place. It shall be equal for all nodes in the cluster.

`WaitBusSleepTime`
NM state position within PDU

Timeout [microseconds] for bus calm down phase.

It denotes how long the NM shall stay in the Prepare Bus-Sleep Mode before transition into Bus-Sleep Mode shall take place. It shall be equal for all nodes in the cluster. It shall be long enough to make all Tx-buffer empty.

`RepeatMsgTime`
Timeout [microseconds] for Repeat Message State.

It defines how long the NM shall stay in the Repeat Message State.

`MsgCycleTime`
Period [microseconds] of a NM-message.

It determines the periodic rate in the "periodic transmission mode with bus load reduction" and is the basis for transmit scheduling in the "periodic transmission mode without bus load reduction".

`MsgCycleOffset`
Time offset [microseconds] in the periodic transmission mode.

It determines the start delay of the transmission.

`MsgReducedTime`
Node specific bus cycle time [microseconds] in the periodic transmission mode with bus load reduction.

This time should be greater than **0.5** * `MsgCycleTime` and less than `MsgCycleTime`.

`PduNidPosition`
Used if flag `G_NET2RUN__CAN_NM__AUTOSAR__FLAG__USE_NODE_ID_IN_PDU` is set: node id position within PDU.

The following values are possible:

`G_NET2RUN__CAN_NM__AUTOSAR__PDU_NID_POSITION__BYTE_0`
Node id is in byte 0 AUTOSAR default, NMH2

G_NET2RUN__CAN_NM__AUTOSAR__PDU_NID_POSITION__BYTE_1
Node id is in byte 1

G_NET2RUN__CAN_NM__AUTOSAR__PDU_NID_POSITION__NOT_USED
Node id is not used, NMH1.

PduCbvPosition
Used if flag G_NET2RUN__CAN_NM__AUTOSAR__FLAG__USE_CBV_IN_PDU is set: control bit vector position within PDU.

The following values are possible:

G_NET2RUN__CAN_NM__AUTOSAR__PDU_CBV_POSITION__BYTE_0
NM control bit vector is in byte 0

G_NET2RUN__CAN_NM__AUTOSAR__PDU_CBV_POSITION__BYTE_1
NM control bit vector is in byte 1 AUTOSAR default, used for NMH2

G_NET2RUN__CAN_NM__AUTOSAR__PDU_CBV_POSITION__NOT_USED
NM control bit vector is not used. Use it for NMH1.

PduStatePosition
Used if flag G_NET2RUN__CAN_NM__AUTOSAR__FLAG__USE_STATE_IN_PDU is set: state position within PDU

The following values are possible:

G_NET2RUN__CAN_NM__AUTOSAR__PDU_STATE_POSITION__BYTE_0
NM state is in byte0, used for NMH1

G_NET2RUN__CAN_NM__AUTOSAR__PDU_STATE_POSITION__BYTE_1
NM state is in byte 1

G_NET2RUN__CAN_NM__AUTOSAR__PDU_STATE_POSITION__BYTE_2
NM state is in byte 2, used for NMH2

G_NET2RUN__CAN_NM__AUTOSAR__PDU_STATE_POSITION__NOT_USED
NM state is not used, AUTOSAR default.

PduLength
Number of PDU data bytes.

PduData
PDU data bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

4 COM Layer

Functions for controlling the COM layer of a Net2Run **Rest Bus Simulation**

G_Net2Run_Com_Init

G_Net2Run_Com_Init — Init Com layer

Description

Use this command for initializing the Com layer.

Definition

```
G_Error_t
G_Net2Run_Com_Init(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_DeInit

G_Net2Run_Com_DeInit — De-Init Com layer

Description

Use this command for de-initializing the Com layer.

Definition

```
G_Error_t
G_Net2Run_Com_DeInit(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_TxPdu_MaxNumber

G_Net2Run_Com_TxPdu_MaxNumber — Set maximum number of Com TX PDUs

Description

Use this command to set the maximum number of Com TX PDUs

This command needs to be called prior to G_Net2Run_Com_TxPdu_Config .

Definition

```
G_Error_t
G_Net2Run_Com_TxPdu_MaxNumber(
 const G_PortHandle_t  portHandle,
 const u32_t  maxNumber
);
```

Parameters

`portHandle`
Handle to the communication port

`maxNumber`
Maximum number of Com TX PDUs that can be configured

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_RxPdu_MaxNumber

G_Net2Run_Com_RxPdu_MaxNumber — Set maximum number of Com RX PDUs

Description

Use this command to set the maximum number of Com RX PDUs

This command needs to be called prior to G_Net2Run_Com_RxPdu_Config .

Definition

```
G_Error_t
G_Net2Run_Com_RxPdu_MaxNumber(
 const G_PortHandle_t  portHandle,
 const u32_t  maxNumber
);
```

Parameters

`portHandle`
Handle to the communication port

`maxNumber`
Maximum number of PDUs that can be configured

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_TxPdu_Config

G_Net2Run_Com_TxPdu_Config — Config of Com TX PDU

Description

Use this command to config a Com TX PDU.

Definition

```
G_Error_t
G_Net2Run_Com_TxPdu_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_TxPdu_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__TX_PDU__CONFIG__FLAG__NONE`
No flag is set

`TxPduId`
TX PDU ID

0..max number defined by G_Net2Run_Com_TxPdu_MaxNumber -1

`PduLength`
Length of PDU in bytes

`PduType`
PDU type

The following values are possible:

`G_NET2RUN__COM__PDU_TYPE__NORMAL`
Normal PDU

`G_NET2RUN__COM__PDU_TYPE__TP`
PDU of transport protocol

`UnusedAreasDefault`
Fills not used areas of an I-PDU with this bit-pattern.

This byte-pattern will be repeated throughout the I-PDU.

(0..255)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`MinimumDelayTime`
Minimum delay time in microseconds

Minimum delay between successive transmissions of this I-PDU, independent of the transmission mode. There is only one minimum delay time parameter for the I-PDU. This minimum delay time does not change with mode changes. Neither is the timer reset. This means that mode changes are not allowed to violate the minimum delay time.

`NumberOfSignals`
Number of signals of the PDU

If neither G_Net2Run_Com_TxPdu_TxMode_True nor G_Net2Run_Com_TxPdu_TxMode_False is configured for this TX PDU, only direct transmission (without repetition) is possible.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_RxPdu_Config

G_Net2Run_Com_RxPdu_Config — Config of Com RX PDU

Description

Use this command to config a Com RX PDU.

Definition

```
G_Error_t
G_Net2Run_Com_RxPdu_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_RxPdu_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following values:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__RX_PDU__CONFIG__FLAG__NONE`
No flag is set

`RxPduId`
RX PDU ID

0..max number defined by G_Net2Run_Com_RxPdu_MaxNumber -1

`PduLength`
Length of PDU in bytes

`PduType`
PDU type

The following values are possible:

`G_NET2RUN__COM__PDU_TYPE__NORMAL`
Normal PDU

`G_NET2RUN__COM__PDU_TYPE__TP`
PDU of transport protocol

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`NumberOfSignals`
Number of signals of the PDU

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_TxPdu_TxMode_True

G_Net2Run_Com_TxPdu_TxMode_True — Configuration of PDU transmission mode

Description

Use this command for configuration of transmission mode that is used when the filtering state for this I-PDU evaluates to true (if none of the TX PDU's signals has an filter, the filtering state always evaluates to true)

Definition

```
G_Error_t
G_Net2Run_Com_TxPdu_TxMode_True(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_TxPdu_TxMode_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__TX_PDU__TX_MODE__FLAG__NONE`
No flag is set

`TxPduId`
TX PDU ID

0..max number defined by G_Net2Run_Com_TxPdu_MaxNumber -1

`TxMode`
TX mode

The following parameters are possible:

`G_NET2RUN__COM__TX_MODE__NONE`
None

`G_NET2RUN__COM__TX_MODE__DIRECT`
Direct

`G_NET2RUN__COM__TX_MODE__PERIODIC`
Periodic

`G_NET2RUN__COM__TX_MODE__MIXED`
Mixed

`N`
Number of confirmed transmissions for transmission mode DIRECT and the event driven part of transmission mode MIXED.

If N is zero, transmission is initiated only once without waiting for the transmission confirmation.

If N is non zero, transmission is initiated periodically with a period of "RepetitionPeriod" until "N" transmission confirmations are received.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`TimeOffset`
Time offset in microseconds

Time until first transmission of this I-PDU. TimeOffset defines the time between the start of the I-PDU and the first transmission of the cyclic part of the transmission request of this I-PDU.

`TimePeriod`
Time period in microseconds

Period of the repetition of cyclic transmissions.

`RepetitionPeriod`
Repetition period in microseconds

Period of the repetition of the n transmission for the Direct/NTimes transmission mode and the event driven part of the Mixed transmission mode.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_TxPdu_TxMode_False

G_Net2Run_Com_TxPdu_TxMode_False — Configuration of PDU transmission mode

Description

Use this command for configuration of transmission mode that is used when the filtering state for this I-PDU evaluates to false.

Definition

```
G_Error_t
G_Net2Run_Com_TxPdu_TxMode_False(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_TxPdu_TxMode_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__TX_PDU__TX_MODE__FLAG__NONE`
No flag is set

`TxPduId`
TX PDU ID

0..max number defined by G_Net2Run_Com_TxPdu_MaxNumber -1

`TxMode`
TX mode

The following parameters are possible:

`G_NET2RUN__COM__TX_MODE__NONE`
None

`G_NET2RUN__COM__TX_MODE__DIRECT`
Direct

`G_NET2RUN__COM__TX_MODE__PERIODIC`
Periodic

`G_NET2RUN__COM__TX_MODE__MIXED`
Mixed

`N`
Number of confirmed transmissions for transmission mode DIRECT and the event driven part of transmission mode MIXED.

If `N` is zero, transmission is initiated only once without waiting for the transmission confirmation.

If `N` is non zero, transmission is initiated periodically with a period of "RepetitionPeriod" until "N" transmission confirmations are received.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`TimeOffset`
Time offset in microseconds

Time until first transmission of this I-PDU. TimeOffset defines the time between the start of the I-PDU and the first transmission of the cyclic part of the transmission request of this I-PDU.

`TimePeriod`
Time period in microseconds

Period of the repetition of cyclic transmissions.

`RepetitionPeriod`
Repetition period in microseconds

Period of the repetition of the n transmission for the Direct/NTimes transmission mode and the event driven part of the Mixed transmission mode.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_TxPdu_PduCounter

G_Net2Run_Com_TxPdu_PduCounter — PDU counter configuration

Description

Use this command for configuration of the PDU counter.

Definition

```
G_Error_t
G_Net2Run_Com_TxPdu_PduCounter(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_TxPdu_PduCounter_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__TX_PDU__PDU_COUNTER__FLAG__NONE`
No flag is set

`TxPduId`
TX PDU ID

0..max number defined by G_Net2Run_Com_TxPdu_MaxNumber -1

`TxPduSignalId`
TX PDU signal ID

0..(`NumberOfSignals`-1) within "TxPdu"

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_TxPdu_Signal

G_Net2Run_Com_TxPdu_Signal — Configuration of TX PDU signal

Description

Use this command for configuration of a TX PDU signal.

Definition

```
G_Error_t
G_Net2Run_Com_TxPdu_Signal(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_TxPdu_Signal_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__TX_PDU__SIGNAL__FLAG__NONE`
No flag is set

`G_NET2RUN__COM__TX_PDU__SIGNAL__FLAG__USE_UPDATE_BIT`
0 : the signal has no update bit

1 : the signal has an update bit at `UpdateBitPosition`

`G_NET2RUN__COM__TX_PDU__SIGNAL__FLAG__IS_GROUP_SIGNAL`
0 : the signal doesn't belong to a signal group

1 : the signal is a group signal (belongs to signal group identified by `TxPduSignal-GroupId`)

`TxPduId`
TX PDU ID

0..max number defined by G_Net2Run_Com_TxPdu_MaxNumber -1

`TxPduSignalId`
TX PDU signal ID

0..(`NumberOfSignals`-1) within "TxPdu"

`BitPosition`
Starting position within the I-PDU. If the endianness conversion is configured to Opaque this parameter shall define the bit0 of the first byte like in little endian byte order

`BitSize`
Size in bits

`DataInvalidValue[8]`
Data invalid value

`InitValue[8]`
Initial value

`Endianness`
Endianness

`G_NET2RUN__ENDIANNESS__LITTLE_ENDIAN`
Little endian

`G_NET2RUN__ENDIANNESS__BIG_ENDIAN`
Big endian

`G_NET2RUN__ENDIANNESS__OPAQUE`
Opaque

`SignalType`
Signal type

`G_NET2RUN__SIGNAL__TYPE__BOOLEAN`
Boolean signal (value size = 1 byte)

`G_NET2RUN__SIGNAL__TYPE__FLOAT32`
32 bit floating point signal (value size = 4 bytes)

`G_NET2RUN__SIGNAL__TYPE__FLOAT64`
64 bit floating point signal (value size = 8 bytes)

`G_NET2RUN__SIGNAL__TYPE__S8`
Signed decimal 8 bit signal (value size = 1 byte)

`G_NET2RUN__SIGNAL__TYPE__S16`
Signed decimal 16 bit signal (value size = 2 bytes)

`G_NET2RUN__SIGNAL__TYPE__S32`
Signed decimal 32 bit signal (value size = 4 bytes)

`G_NET2RUN__SIGNAL__TYPE__U8`
Unsigned decimal 8 bit signal (value size = 1 byte)

`G_NET2RUN__SIGNAL__TYPE__U16`
Unsigned decimal 16 bit signal (value size = 2 bytes)

`G_NET2RUN__SIGNAL__TYPE__U32`
Unsigned decimal 32 bit signal (value size = 4 bytes)

`G_NET2RUN__SIGNAL__TYPE__U8_N`
Array of unsigned decimal 8 bit values (value size varies)

`G_NET2RUN__SIGNAL__TYPE__U8_DYN`
Array of unsigned decimal 8 bit value with dynamic length (value size = 0 bytes)

`TransferProperty`
Transfer property

`G_NET2RUN__COM__TRANSFER_PROPERTY__TRIGGERED`
When a TX signal with this transfer property is changed, the COM layer initiates an immediate transmission of the TX PDU, except if the defined transmission mode (see "G_Net2Run_Com_TxMode_t") for the TX PDU is configured to PERIODIC or NONE

`G_NET2RUN__COM__TRANSFER_PROPERTY__PENDING`
When a TX signal with this transfer property is changed, the COM layer doesn't initiate an immediate transmission of the TX PDU. The transmission keeps pending until an other TX signal within the same TX PDU with transfer property TRIGGERED is changed or the timer for periodic transmission expires in case of the TX PDU's transmission mode PERIODIC or MIXED (see "G_Net2Run_Com_TxMode_t").

`G_NET2RUN__COM__TRANSFER_PROPERTY__TRIGGERED_ON_CHANGE`
Depending on the transmission mode, a write access to a signal with this transfer property can trigger the transmission of the corresponding tx PDU, but only in case the written value is different to the locally stored (last sent or initial value) in length or value.

AUTOSAR name: ComTransferProperty::TRIGGERED_ON_CHANGE

`G_NET2RUN__COM__TRANSFER_PROPERTY__TRIGGERED_ON_CHANGE_WITHOUT_REP`
Depending on the transmission mode, a write access to a signal with this transfer property can trigger the transmission of the corresponding I-PDU just once without a repetition, but only in case the written value is different to the locally stored (last sent or initial value) in length or value.

AUTOSAR name:
ComTransferProperty::TRIGGERED_ON_CHANGE_WITHOUT_REPETITION

`G_NET2RUN__COM__TRANSFER_PROPERTY__TRIGGERED_WITHOUT_REP`
Depending on the transmission mode, a write access to a signal with this transfer property can trigger the transmission of the corresponding I-PDU just once without a repetition.

AUTOSAR name: ComTransferProperty::TRIGGERED_WITHOUT_REPETITION

`reserved1`
reserved parameter (must be initialized with **0**)

`UpdateBitPosition`
Bit position of update-bit inside I-PDU (only relevant if flag `G_NET2RUN__COM__TX_PDU__SIGNAL__FLAG__USE_UPDATE_BIT` is set)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Timeout`
Timeout in microseconds

Defines the timeout period for the deadline monitoring.

`TxPduSignalGroupId`
TX PDU signal group ID

Optional parameter, used if flag `G_NET2RUN__COM__TX_PDU__SIGNAL__FLAG__IS_GROUP_SIGNAL` is set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_TxPdu_Signal_Filter_Config

G_Net2Run_Com_TxPdu_Signal_Filter_Config — Configuration of TX PDU signal filter

Description

Use this command for configuration of a TX PDU signal filter.

Definition

```
G_Error_t
G_Net2Run_Com_TxPdu_Signal_Filter_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_TxPdu_Signal_Filter_Config_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__TX_PDU__SIGNAL__CONFIG__FILTER__FLAG__NONE`
No flag is set

`TxPduId`
TX PDU ID

0..max number defined by G_Net2Run_Com_TxPdu_MaxNumber -1

`TxPduSignalId`
TX PDU signal ID

0..(`NumberOfSignals`-1) within "TxPdu"

`Algorithm`
AUTOSAR name: ComFilterAlgorithm AUTOSAR 4.4.0 ECUC_Com_00146

`G_NET2RUN__COM__FLT__ALG__ALWAYS`
No filtering is performed so that the message always passes

`G_NET2RUN__COM__FLT__ALG__NEVER`
The filter removes all messages

`G_NET2RUN__COM__FLT__ALG__MASKED_NEW_EQUALS_X`
Pass messages whose masked value is equal to a specific value

`G_NET2RUN__COM__FLT__ALG__MASKED_NEW_DIFFERS_X`
Pass messages whose masked value is not equal to a specific value

G_NET2RUN__COM__FLT__ALG__MASKED_NEW_DIFFERS_MASKED_OLD
Pass messages where the masked value has changed

G_NET2RUN__COM__FLT__ALG__NEW_IS_WITHIN
Pass a message if its value is within a predefined boundary

G_NET2RUN__COM__FLT__ALG__NEW_IS_OUTSIDE
Pass a message if its value is outside a predefined boundary

G_NET2RUN__COM__FLT__ALG__ONE_EVERY_N
Pass a message once every N message occurrences. Start: Occurrence = 0. Each time the message is received or transmitted, Occurrence is incremented by 1 after filtering.

G_NET2RUN__COM__FLT__ALG__UNKNOWN
This is just a placeholder

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Mask[8]`
Autosar 4.4.0 ECUC_Com_00235 Defines the mask value for the "MASKED_NEW_EQUALS_X", "MASKED_NEW_DIFFERS_X" and "MASKED_NEW_DIFFERS_MASKED_OLD" filter algorithms -2147483648...4294967295

`X[8]`
Autosar 4.4.0 ECUC_Com_00147 Defines the compare value for the "MASKED_NEW_EQUALS_X" and "MASKED_NEW_DIFFERS_X" filter algorithms -2147483648...4294967295

`Max[8]`
Autosar 4.4.0 ECUC_Com_00317 Defines the maximum value for the "NEW_IS_WITHIN" and "NEW_IS_OUTSIDE" filter algorithms -2147483648...4294967295

`Min[8]`
Autosar 4.4.0 ECUC_Com_00318 Defines the minimum value for the "NEW_IS_WITHIN" and "NEW_IS_OUTSIDE" filter algorithms -2147483648...4294967295

`Offset`
Autosar 4.4.0 ECUC_Com_00313 Defines the offset in the period for the "ONCE_EVERY_N" filter algorithm 0...(Period - 1)

`Period`
Autosar 4.4.0 ECUC_Com_00312 Defines the period for the "ONCE_EVERY_N" filter algorithm 2...4294967294

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_TxPdu_Signal_Write

G_Net2Run_Com_TxPdu_Signal_Write — Write Com TX PDU signal

Description

Use this command to write a Com TX PDU signal.

Definition

```
G_Error_t
G_Net2Run_Com_TxPdu_Signal_Write(
 const G_PortHandle_t  portHandle,
 const u32_t  txPduId,
 const u32_t  txPduSignalId,
 const u16_t  length,
 const u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`txPduId`
TX PDU ID

0..max number defined by G_Net2Run_Com_TxPdu_MaxNumber -1

`txPduSignalId`
TX PDU signal ID

0..(`NumberOfSignals`-1) within "TxPdu"

`length`
Signal length in bytes

`data`
Signal data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_RxPdu_Signal

G_Net2Run_Com_RxPdu_Signal — Configuration of RX PDU signal

Description

Use this command for configuration of an RX PDU signal.

Definition

```
G_Error_t
G_Net2Run_Com_RxPdu_Signal(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_RxPdu_Signal_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__RX_PDU__SIGNAL__FLAG__NONE`
No flag is set

`G_NET2RUN__COM__RX_PDU__SIGNAL__FLAG__USE_UPDATE_BIT`
0 : the signal has no update bit

1 : the signal has an update bit at `UpdateBitPosition`

`G_NET2RUN__COM__RX_PDU__SIGNAL__FLAG__IS_GROUP_SIGNAL`
0 : the signal doesn't belong to a signal group

1 : the signal is a group signal (belongs to signal group identified by `RxPduSignal-GroupId`)

`RxPduId`
RX PDU ID

0..max number defined by G_Net2Run_Com_RxPdu_MaxNumber -1

`RxPduSignalId`
RX PDU signal ID

0..(`NumberOfSignals`-1) within "RxPdu"

`BitPosition`
Starting position within the I-PDU. If the endianness conversion is configured to Opaque this parameter shall define the bit0 of the first byte like in little endian byte order

`BitSize`
Size in bits

`DataInvalidValue[8]`
Data invalid value

`InitValue[8]`
Initial value

`Endianness`
Endianness

`G_NET2RUN__ENDIANNESS__LITTLE_ENDIAN`
Little endian

`G_NET2RUN__ENDIANNESS__BIG_ENDIAN`
Big endian

`G_NET2RUN__ENDIANNESS__OPAQUE`
Opaque

`SignalType`
Signal type

`G_NET2RUN__SIGNAL__TYPE__BOOLEAN`
Boolean signal (value size = 1 byte)

`G_NET2RUN__SIGNAL__TYPE__FLOAT32`
32 bit floating point signal (value size = 4 bytes)

`G_NET2RUN__SIGNAL__TYPE__FLOAT64`
64 bit floating point signal (value size = 8 bytes)

`G_NET2RUN__SIGNAL__TYPE__S8`
Signed decimal 8 bit signal (value size = 1 byte)

`G_NET2RUN__SIGNAL__TYPE__S16`
Signed decimal 16 bit signal (value size = 2 bytes)

`G_NET2RUN__SIGNAL__TYPE__S32`
Signed decimal 32 bit signal (value size = 4 bytes)

`G_NET2RUN__SIGNAL__TYPE__U8`
Unsigned decimal 8 bit signal (value size = 1 byte)

`G_NET2RUN__SIGNAL__TYPE__U16`
Unsigned decimal 16 bit signal (value size = 2 bytes)

`G_NET2RUN__SIGNAL__TYPE__U32`
Unsigned decimal 32 bit signal (value size = 4 bytes)

`G_NET2RUN__SIGNAL__TYPE__U8_N`
Array of unsigned decimal 8 bit values (value size varies)

`G_NET2RUN__SIGNAL__TYPE__U8_DYN`
Array of unsigned decimal 8 bit value with dynamic length (value size = 0 bytes)

`UpdateBitPosition`
Bit position of update-bit inside I-PDU (only relevant if flag `G_NET2RUN__COM__RX_PDU__SIGNAL__FLAG__USE_UPDATE_BIT` is set)

`Timeout`
Timeout in microseconds

Defines the timeout period for the deadline monitoring.

`FirstTimeout`
First timeout in microseconds

Defines the first timeout for the deadline monitoring.

The timer for the first timeout is started on start of the corresponding RX PDU and stopped on the first reception of this signal.

`RxPduSignalGroupId`
RX PDU signal group ID

Optional parameter, used if flag `G_NET2RUN__COM__RX_PDU__SIGNAL__FLAG__IS_GROUP_SIGNAL` is set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_TxPdu_PduCounter_Increment

G_Net2Run_Com_TxPdu_PduCounter_Increment — Manipulate PDU Counter

Description

Use this command to manipulate the PDU Counter signal of a TX PDU.

Definition

```
G_Error_t
G_Net2Run_Com_TxPdu_PduCounter_Increment(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_TxPdu_PduCounter_Increment_Parameters_t * const
  parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command Flags

The following values are possible and can be combined:

`G_NET2RUN__COM__TX_PDU__PDU_COUNTER__INCREMENT__CMD_FLAG__NONE`
No flag is set

`TxPduId`
`ComId` of the PDU that can be queried with G_Net2Run_Pdu_PduInfo_Get

`TxPduSignalId`
Pdu signal ID that can be queried with G_Net2Run_Pdu_IncludedSignals_Get

`Increment`
First Increment value of the PDU Counter

`NumberOfCycles`
Number of cycles the PDU Counter manipulation will be active

`Increment2`
Second Increment value of the PDU Counter

If `NumberOfCycles` is zero, the PDU Counter's increment is set to `Increment`.

If `NumberOfCycles` is not zero, the PDU Counter's increment is set to `Increment` for `NumberOfCycles` cycles and then set to `Increment2`.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_TxPdu_Checksum_Config

G_Net2Run_Com_TxPdu_Checksum_Config — TX PDU checksum configuration

Description

Use this command to configure a TX PDU checksum.

Definition

```
G_Error_t
G_Net2Run_Com_TxPdu_Checksum_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_TxPdu_Checksum_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command Flags

The following values are possible and can be combined:

`G_NET2RUN__COM__TX_PDU__CHECKSUM__CONFIG__FLAG__NONE`
No flag is set

`G_NET2RUN__COM__TX_PDU__CHECKSUM__CONFIG__FLAG__NO_CALC_ON_TRANSMIT`
0 = calculation of checksum on "transmit"

1 = no calculation of checksum on "transmit"

If this flag is cleared, flag `G_NET2RUN__COM__TX_PDU__CHECKSUM__CONFIG__FLAG__CALC_ON_TRIGGER_TRANSMIT` should be cleared too.

`G_NET2RUN__COM__TX_PDU__CHECKSUM__CONFIG__FLAG__CALC_ON_TRIGGER_TRANSMIT`
0 = no calculation of checksum on "trigger transmit"

1 = calculation of checksum on "trigger transmit"

If this flag is set, flag `G_NET2RUN__COM__TX_PDU__CHECKSUM__CONFIG__FLAG__CALC_ON_TRIGGER_TRANSMIT` should be set too.

`TxPduId`
ID of the TX PDU containing the checksum in COM layer

`TxPduSignalId`
ID of the signal within the TX PDU containing the checksum

`ChecksumMode`
Checksum mode

The following parameters are possible:

`G_NET2RUN__CHECKSUM_MODE__OFF`
No checksum calculation

`G_NET2RUN__CHECKSUM_MODE__XOR`
XOR checksum

`G_NET2RUN__CHECKSUM_MODE__CRC8_J1850`
CRC J1850 checksum

`G_NET2RUN__CHECKSUM_MODE__ADD_AND_COMPLEMENT_TO_ONE`
"Add and complement to 1" checksum

`G_NET2RUN__CHECKSUM_MODE__XOR_WITH_USER_BYTES`
XOR checksum with user bytes

`G_NET2RUN__CHECKSUM_MODE__CRC_WITH_USER_BYTES`
CRC checksum with user bytes

`G_NET2RUN__CHECKSUM_MODE__CUSTOM_12`
CUSTOM 12 checksum

Calculate CRC over data bytes and one byte of `UserBytes`. The byte is selected (`UserBytes` is indexed) by the value of the second PDU data byte bits 0..3.

`G_NET2RUN__CHECKSUM_MODE__ADD_NIBBLES`
Add Nibbles checksum

The checksum uses the lower nibble in the last byte of a message, the checksum is located in bits 0 till 3 in the 8th payload byte of a CAN frame.

The checksum is defined as a value which results in 0 when all data nibbles are added up.

The checksum is the sum of all data nibbles of the message including CAN ID, whereas Standard ID shall be extended by loginal "1" in 12th bit and consist of 3 nibbles. Extended CAN ID shall be extened by 3 additional logical "1" in bits 29,30,31 and result in 8 nibbles.

`G_NET2RUN__CHECKSUM_MODE__CUSTOM_14`
CUSTOM 14 checksum

`G_NET2RUN__CHECKSUM_MODE__ADD_BYTES__XOR_NIBBLES`
Add bytes XOR nibbles

`Priority`
When a TX PDU has multiple checksums, this parameter defines the execution order. Checksums with higher `Priority` value are calculated after checksums with lower `Priority` value (the checksum with the lowest `Priority` value is calculated first).

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`u`
Union with configuration parameters for different checksum modes

The union contains the following parameters:

`Custom12`
Checksum parameters for `ChecksumMode = G_NET2RUN__CHECKSUM_MODE__CUSTOM_12`

The structure contains the following elements:

`Order`
Checksum order, e.g. 8

`NumberOfUserBytes`
Number of user bytes

Has to be 16!

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Polynomial`
Truncated polynomial (the MSB is omitted), e.g. 0x2F

`InitialValue`
Initial value, e.g. 0xFF

`FinalXorValue`
Final XOR value, e.g. 0xFF

`reserved`
reserved parameter (must be initialized with **0**)

`UserBytes`
User bytes

`AddNibbles`
Checksum parameters for `ChecksumMode = G_NET2RUN__CHECKSUM_MODE__ADD_NIBBLES`

The structure contains the following elements:

`ExtraBits_BitPosition`
Bit position for extra bits, e.g. 0

`ExtraBits_BitSize`
Bit size for extra bits, e.g. 4

If member `ExtraBits_BitSize` is not zero, the extra bits are also included into checksum calculation.

`ExtraBits_Endianness`
Endianess for extra bits

The following parameters are possible:

`G_NET2RUN__ENDIANNESS__LITTLE_ENDIAN`
Little endian

`G_NET2RUN__ENDIANNESS__BIG_ENDIAN`
Big endian

`G_NET2RUN__ENDIANNESS__OPAQUE`
Opaque

`reserved1`
reserved parameter (must be initialized with **0**)

`InitialValue`
Initial value (optional)

(0 if not present, e.g. 0x09)

`FinalXorValue`
Final XOR value (optional)

(0 if not present, e.g. 0xFF)

`FinalAddAfterXor`
Final Add after XOR (optional)

(0 if not present, e.g. 0x01)

`NumberOfUserBytes`
Number of user bytes in `UserBytes`

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`reserved4`
reserved parameter (must be initialized with **0**)

`UserBytes`
User bytes (maximum number of user bytes = 20)

`XorWithUserBytes`
Checksum parameters for `ChecksumMode` = `G_NET2RUN__CHECKSUM_MODE__XOR_WITH_USER_BYTES`

The structure contains the following elements:

`NumberOfUserBytes`
Number of user bytes

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`UserBytes`
User bytes (maximum number of user bytes = 20)

`CrcWithUserBytes`
Checksum parameters for `ChecksumMode` = `G_NET2RUN__CHECKSUM_MODE__CRC_WITH_USER_BYTES`

The structure contains the following elements:

`Order`
Checksum order, e.g. 8, 16

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Polynomial`
Truncated polynomial (the MSB is omitted), e.g. 0x1D for CRC8-J1850 with "x8 + x4 + x3 + x2 + 1" ("x8" is omitted)

`InitialValue`
Initial value, e.g. 0xFF for CRC8-J1850

`FinalXorValue`
Final XOR value, e.g. 0xFF for CRC8-J1850

`NumberOfUserBytesBefore`
Number of user bytes before checksum calculation over data bytes

`NumberOfUserBytesAfter`
Number of user bytes after checksum calculation over data bytes

`reserved4`
reserved parameter (must be initialized with **0**)

`reserved5`
reserved parameter (must be initialized with **0**)

`UserBytes`
Contains "user bytes before calculation over data bytes" immediately followed by the "user bytes after calculation over data bytes"

`Custom12`
Checksum parameters for `ChecksumMode = G_NET2RUN__CHECKSUM_MODE__CUSTOM_14`

The structure contains the following elements:

`MessageId`
Message ID

`MessageCounter_BitPosition`
Bit position of message counter

e.g. 56

`MessageCounter_BitSize`
Bit size of message counter

e.g. 4

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`AddBytes_XorNibbles`
Checksum parameters for `ChecksumMode = G_NET2RUN__CHECKSUM_MODE__ADD_BYTES__XOR_NIBBLES`

The structure contains the following elements:

`InitialValue`
Initial value

e.g. 0x131

`FinalXorValue`
Final XOR value

e.g. 0

`ExtraBits_BitPosition`
Bit position of extra bits

e.g. 0

`ExtraBits_BitSize`
Bit size of extra bits

e.g. 4

`ExtraBits_Endianness`
Endianess of extra bits

0 = little endian

1 = big endian

`reserved`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_TxPdu_Checksum_IncludedBytes

G_Net2Run_Com_TxPdu_Checksum_IncludedBytes — Restrict checksum calculation to specific bytes of a TX PDU

Description

Use this command to restrict checksum calculation to specific bytes of a TX PDU.

When a checksum is configured with G_Net2Run_Com_TxPdu_Checksum_Config it is calculated dependent on all bytes of the TX PDU. With this command you can restrict the checksum calculation to specific data bytes of the TX PDU.

Definition

```
G_Error_t
G_Net2Run_Com_TxPdu_Checksum_IncludedBytes(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_TxPdu_Checksum_IncludedBytes_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__TX_PDU__CHECKSUM__INCLUDED_BYTES__CMD_FLAG__NONE`
No flag is set

`TxPduId`
ID of the TX PDU containing the checksum in COM layer

`TxPduSignalId`
ID of the signal within the TX PDU containing the checksum

`NumberOfIncludedBytes`
Number of included bytes that are used for checksum calculation

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`IncludedBytes`
Byte positions of the included bytes (starting with 0)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
NumberOfIncludedBytes = 3;
IncludedBytes[0] = 0;
IncludedBytes[1] = 2;
IncludedBytes[2] = 3;
```

The checksum is calculated over `data[0]`, `data[2]` and `data[3]`.

```
NumberOfIncludedBytes = 3;
IncludedBytes[0] = 7;
IncludedBytes[1] = 6;
IncludedBytes[2] = 5;
```

The checksum is calculated over `data[7]`, `data[6]` and `data[5]`.

G_Net2Run_Com_TxPdu_Checksum_Wrapper

G_Net2Run_Com_TxPdu_Checksum_Wrapper — Restrict checksum calculation to specific signals of a TX PDU

Description

Use this command to restrict checksum calculation to specific signals of a TX PDU.

When a checksum is configured with G_Net2Run_Com_TxPdu_Checksum_Config it is calculated dependent on all bytes of the TX PDU. With this command you can restrict the checksum calculation to specific signals of the TX PDU.

Definition

```
G_Error_t
G_Net2Run_Com_TxPdu_Checksum_Wrapper(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_TxPdu_Checksum_Wrapper_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__TX_PDU__CHECKSUM__WRAPPER__CMD_FLAG__NONE`
No flag is set

`TxPduId`
ID of the TX PDU containing the checksum in COM layer

`TxPduSignalId`
ID of the signal within the TX PDU containing the checksum

`Mode`
Checksum wrapper mode

The following values are possible:

`G_NET2RUN__COM__TX_PDU__CHECKSUM__WRAPPER__MODE__OFF`
The checksum is not calculated.

Parameters: none

`G_NET2RUN__COM__TX_PDU__CHECKSUM__WRAPPER__MODE__DEFAULT`
The checksum is calculated over all TX PDU bytes except the checksum's signal byte(s) itself.

Parameters: none

G_NET2RUN__COM__TX_PDU__CHECKSUM__WRAPPER__MODE__CUSTOM_1
The checksum is calculated over an extra data array that is initialized with signal values before each checksum calculation.

Each signal's size is rounded to the next byte boundary (e.g. 4->8, 7->8, 14->16).

Unused bits (bytes) are filled with UnusedBitPattern;

If a signal uses multiple bytes (signal size > 8), little endian byte ordering is used.

Parameters: **G_Net2Run_Com_TxPdu_Checksum_Wrapper_Custom1_t**

reserved1
reserved parameter (must be initialized with **0**)

reserved2
reserved parameter (must be initialized with **0**)

reserved3
reserved parameter (must be initialized with **0**)

Parameters
Extended parameters

The union contains the following elements:

Custom1
If Mode = G_NET2RUN__COM__TX_PDU__CHECKSUM__WRAPPER__MODE__CUSTOM_1

The structure contains the following elements:

Flags
Checksum flags

The following values are possible and can be combined:

G_NET2RUN__COM__TX_PDU__CHECKSUM__WRAPPER__CUSTOM_1__CMD_FLAG_
_NONE
No flag is set

NumberOfTxPduSignals
Number of signals used for checksum calculation

UnusedBitPattern
Value for bits that are not used

reserved1
reserved parameter (must be initialized with **0**)

reserved2
reserved parameter (must be initialized with **0**)

TxPduSignalIds
IDs of the TX PDU signals that are used for checksum calculation

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_TxPdu_Checksum_Error

G_Net2Run_Com_TxPdu_Checksum_Error — Manipulate Checksum of PDU

Description

Use this command to manipulate the checksum of a TX PDU.

Definition

```
G_Error_t
G_Net2Run_Com_TxPdu_Checksum_Error(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_TxPdu_Checksum_Error_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command Flags

The following values are possible and can be combined:

`G_NET2RUN__COM__TX_PDU__CHECKSUM__ERROR__CMD_FLAG__NONE`

`TxPduId`
`ComId` of the PDU that can be queried with G_Net2Run_Pdu_PduInfo_Get

`TxPduSignalId`
Pdu signal ID that can be queried with G_Net2Run_Pdu_IncludedSignals_Get

`ErrorMode`
Error mode that specifies how the checksum is manipulated

The following values are possible:

`G_NET2RUN__CHECKSUM__ERROR_MODE__NO_ERROR`
The checksum is not manipulated

`G_NET2RUN__CHECKSUM__ERROR_MODE__ADD_ONE`
After the checksum has been calculated, the value `1` is added

`reserved1`
reserved parameter (must be initialized with `0`)

`reserved2`
reserved parameter (must be initialized with `0`)

`reserved3`
reserved parameter (must be initialized with `0`)

`NumberOfCycles`
Number of cycles the checksum is manipulated

0 = set error mode `ErrorMode` permanently, else error mode is set to `ErrorMode` for `NumberOfCycles` cycles and then it is set to `G_NET2RUN__CHECKSUM__ERROR_MODE__NO_ERROR`

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_TxPdu_Transmit_Error

G_Net2Run_Com_TxPdu_Transmit_Error — Manipulate transmission of a TX PDU

Description

Use this command to manipulate the transmission of a TX PDU.

Definition

```
G_Error_t
G_Net2Run_Com_TxPdu_Transmit_Error(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_TxPdu_Transmit_Error_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command Parameters

The structure contains the following elements:

`Flags`
Command Flags

The following values are possible and can be combined:

`G_NET2RUN__COM__TX_PDU__TRANSMIT__ERROR__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__COM__TX_PDU__TRANSMIT__ERROR__CMD_FLAG__USE_TIME`
use parameter `Time`

`TxPduId`
`ComId` of the PDU that can be queried with G_Net2Run_Pdu_PduInfo_Get

`ErrorMode`
Error mode that specifies how the transmission is manipulated

The following values are possible:

`G_NET2RUN__TRANSMIT__ERROR_MODE__NO_ERROR`
The transmission is not manipulated

`G_NET2RUN__TRANSMIT__ERROR_MODE__DISABLE_TRANSMIT`
The transmission is disabled for `NumberOfCycles` cycles

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`NumberOfErrors`
Number of cycles the transmission is manipulated

`Time`
[microseconds] relevant if flag `G_NET2RUN__COM__TX_PDU__TRANSMIT__ERROR__CMD_FLAG__USE_TIME` is set

0 : error is not disabled by time

else : error is disabled after `Time` mmicroseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_TxPdu_Ramp_Config

G_Net2Run_Com_TxPdu_Ramp_Config — TX PDU ramp configuration

Description

Use this command to configure a TX PDU ramp.

Definition

```
G_Error_t
G_Net2Run_Com_TxPdu_Ramp_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_TxPdu_Ramp_Config_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__TX_PDU__RAMP__CONFIG__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__COM__TX_PDU__RAMP__CONFIG__CMD_FLAG__START`
0 = disable ramp calculation

1 = enable ramp calculation

`G_NET2RUN__COM__TX_PDU__RAMP__CONFIG__CMD_FLAG__SET_DIRECTION_ON_START`
The ramp direction is always set on ramp configuration. When the ramp is (re-) started by an other command (not by this configuration command), this flag has the following meaning:

0 = actual ramp direction is not changed

1 = actual ramp direction is set to initial ramp direction `RampDirection`

The ramp direction changes for ramp type "triangle" from "UP" to "DOWN" when `Max` value is reached, or from "DOWN" to "UP" when `Min` value is reached.

`G_NET2RUN__COM__TX_PDU__RAMP__CONFIG__CMD_FLAG__STOP_BY_SUM_OF_INCREMENTS`
0 = ramp is not stopped by sum of increments

1 = ramp is stopped when sum of increments has reached value `SumOfIncrements`

`G_NET2RUN__COM__TX_PDU__RAMP__CONFIG__CMD_FLAG__STOP_BY_EDGE_COUNT`
0 = ramp is not stopped by edge counter

1 =ramp is stopped after `EdgeCount` edges

`G_NET2RUN__COM__TX_PDU__RAMP__CONFIG__CMD_FLAG__DELETE`
0 = do not delete ramp

1 = delete ramp

`G_NET2RUN__COM__TX_PDU__RAMP__CONFIG__CMD_FLAG__NO_CALC_ON_TRANSMIT`
0 = ramp is calculated on "transmit"

1 = ramp is not calculated on "transmit"

`G_NET2RUN__COM__TX_PDU__RAMP__CONFIG__CMD_FLAG__CALC_ON_TX_CONFIRMATION`
0 = ramp is not calculated on "TX confirmation"

1 = ramp is calculated on "TX confirmation"

If this flag is set, flag `G_NET2RUN__COM__TX_PDU__RAMP__CONFIG__CMD_FLAG__NO_CALC_ON_TRANSMIT` should be set too.

`G_NET2RUN__COM__TX_PDU__RAMP__CONFIG__CMD_FLAG__CALC_ON_TRIGGER_TRANSMIT`
0 = ramp is not calculated on "trigger transmit"

1 = ramp is calculated on "trigger transmit"

If this flag is set, flag `G_NET2RUN__COM__TX_PDU__RAMP__CONFIG__CMD_FLAG__NO_CALC_ON_TRANSMIT` should be set too.

`TxPduId`
COM ID of the TX PDU

`TxPduSignalId`
Signal ID within TX PDU

`RampType`
Ramp type

The following values are possible:

`G_NET2RUN__RAMP_TYPE__TRIANGLE`
Triangle ramp

`G_NET2RUN__RAMP_TYPE__SAWTOOTH`
Sawtooth ramp

`RampDirection`
Ramp direction

The following values are possible:

`G_NET2RUN__RAMP_DIRECTION__UP`
Ramp direction up

`G_NET2RUN__RAMP_DIRECTION__DOWN`
Ramp direction down

`UpdateRate`
0, 1 : ramp is always calculated

else : ramp is calculated every `UpdateRate`-th call

`Min`
Minimum ramp value

`Max`
Maximum ramp value

`Increment`
Ramp increment

`SumOfIncrements`
Used if flag `G_NET2RUN__COM__TX_PDU__RAMP__CONFIG__CMD_FLAG__STOP_BY_SUM_OF_INCREMENTS` is set

`EdgeCount`
Used if flag `G_NET2RUN__COM__TX_PDU__RAMP__CONFIG__CMD_FLAG__STOP_BY_EDGE_COUNT` is set

`HoldStates_OnMin`
Number of hold states when ramp has reached value `Min`

`HoldStates_OnMax`
Number of hold states when ramp has reached value `Max`

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`reserved4`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_TxPdu_Ramp_Control

G_Net2Run_Com_TxPdu_Ramp_Control — Control ramp

Description

Use this command to control a TX PDU ramp.

Definition

```
G_Error_t
G_Net2Run_Com_TxPdu_Ramp_Control(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_TxPdu_Ramp_Control_CmdFlags_t  cmdFlags,
 const u8_t  numberOfRamps,
 const G_Net2Run_Com_TxPdu_Ramp_Control_Ramp_t * const ramps
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__TX_PDU__RAMP__CONTROL__CMD_FLAG__NONE`
No flag is set

`numberOfRamps`
Number of ramps to be controlled

`ramps`
Array with ramps to be controlled

The structure contains the following elements:

`TxPduId`
COM ID of the TX PDU containing the ramp

`TxPduSignalId`
Signal ID within of the ramp signal within the TX PDU

`Mode`
Ramp control mode

The following values are possible:

`G_NET2RUN__COM__TX_PDU__RAMP__CONTROL__MODE__START`
Start the ramp

`G_NET2RUN__COM__TX_PDU__RAMP__CONTROL__MODE__STOP`
Stop the ramp

`reserved1`
reserved parameter (must be initialized with `0`)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_TxPdu_Write

G_Net2Run_Com_TxPdu_Write — Write TX PDU data

Description

Use this command to write TX PDU data.

Definition

```
G_Error_t
G_Net2Run_Com_TxPdu_Write(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_TxPdu_Write_CmdFlags_t  cmdFlags,
 const u32_t  txPduId,
 const u32_t  length,
 const u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__TX_PDU__WRITE__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__COM__TX_PDU__WRITE__CMD_FLAG__SET_PDU_LENGTH`
0 : PDU's length is not modified

1 : PDU's current length is set to `length`

`txPduId`
COM ID of the TX PDU

`length`
Data length in bytes

`data`
TX PDU data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_TxPdu_Read

G_Net2Run_Com_TxPdu_Read — Read TX PDU data

Description

Use this command to read TX PDU data.

Definition

```
G_Error_t
G_Net2Run_Com_TxPdu_Read(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_TxPdu_Read_CmdFlags_t  cmdFlags,
 const u32_t  txPduId,
 u32_t * const length,
 u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__TX_PDU__READ__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__COM__TX_PDU__READ__CMD_FLAG__USE_TRIG_TX`
0 : read directly from COM TX PDU data

1 : read by "trigger transmit" (see AUTOSAR "Com_TriggerTransmit" function)

`txPduId`
COM ID of the TX PDU

`length`
in : size of buffer `data` in bytes

out : size of returned data in bytes

`data`
Returns TX PDU data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_TxPdu_Send

G_Net2Run_Com_TxPdu_Send — Send TX PDU

Description

Use this command to send a TX PDU.

Definition

```
G_Error_t
G_Net2Run_Com_TxPdu_Send(
 const G_PortHandle_t  portHandle,
 const u32_t  txPduId
);
```

Parameters

`portHandle`
    Handle to the communication port

`txPduId`
    COM ID of the TX PDU

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_PduGroup_MaxNumber

G_Net2Run_Com_PduGroup_MaxNumber — Set maximum number of Com PDU groups

Description

Use this command to set the maximum number of Com PDU groups.

This command needs to be called prior to G_Net2Run_Com_PduGroup_Config .

Definition

```
G_Error_t
G_Net2Run_Com_PduGroup_MaxNumber(
 const G_PortHandle_t  portHandle,
 const u32_t  maxNumber
);
```

Parameters

`portHandle`
Handle to the communication port

`maxNumber`
Maximum number of PDU groups that can be configured

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_PduGroup_Config

G_Net2Run_Com_PduGroup_Config — Config Com PDU group

Description

Use this command to config a Com PDU group.

Definition

```
G_Error_t
G_Net2Run_Com_PduGroup_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_PduGroup_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__PDU_GROUP__CONFIG__FLAG__NONE`
No flag is set

`PduGroupId`
PDU group ID

0..max number defined by G_Net2Run_Com_PduGroup_MaxNumber -1

`NumberOfPduGroups`
Number of PDU groups this PDU group contains (a PDU group can include other PDU groups)

`NumberOfTxPdus`
Number of TX PDUs this PDU group contains

`NumberOfRxPdus`
Number of RX PDUs this PDU group contains

`Ids`
Array of "PDU group IDs" and/or "TX PDU IDs" and/or "RX PDU IDs"

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_Signal_MaxNumber

G_Net2Run_Com_Signal_MaxNumber — Configuration of maximum number of com signals

Description

Use this command for Configuration of maximum number of com signals.

Definition

```
G_Error_t
G_Net2Run_Com_Signal_MaxNumber(
 const G_PortHandle_t  portHandle,
 const u32_t  maxNumber
);
```

Parameters

`portHandle`
: Handle to the communication port

`maxNumber`
: Maximum number of signals

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_Signal_Config

G_Net2Run_Com_Signal_Config — Com signal configuration

Description

Use this command for Com signal configuration

Definition

```
G_Error_t
G_Net2Run_Com_Signal_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_Signal_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__SIGNAL__CONFIG__FLAG__NONE`
No flag is set

`SignalId`
Signal ID

0..max number defined by G_Net2Run_Com_Signal_MaxNumber -1

`SignalType`
Signal type

`G_NET2RUN__SIGNAL__TYPE__BOOLEAN`
Boolean signal (value size = 1 byte)

`G_NET2RUN__SIGNAL__TYPE__FLOAT32`
32 bit floating point signal (value size = 4 bytes)

`G_NET2RUN__SIGNAL__TYPE__FLOAT64`
64 bit floating point signal (value size = 8 bytes)

`G_NET2RUN__SIGNAL__TYPE__S8`
Signed decimal 8 bit signal (value size = 1 byte)

`G_NET2RUN__SIGNAL__TYPE__S16`
Signed decimal 16 bit signal (value size = 2 bytes)

`G_NET2RUN__SIGNAL__TYPE__S32`
Signed decimal 32 bit signal (value size = 4 bytes)

`G_NET2RUN__SIGNAL__TYPE__U8`
Unsigned decimal 8 bit signal (value size = 1 byte)

`G_NET2RUN__SIGNAL__TYPE__U16`
Unsigned decimal 16 bit signal (value size = 2 bytes)

`G_NET2RUN__SIGNAL__TYPE__U32`
Unsigned decimal 32 bit signal (value size = 4 bytes)

`G_NET2RUN__SIGNAL__TYPE__U8_N`
Array of unsigned decimal 8 bit values (value size varies)

`G_NET2RUN__SIGNAL__TYPE__U8_DYN`
Array of unsigned decimal 8 bit value with dynamic length (value size = 0 bytes)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`NumberOfMappings`
Number of mappings for this signal

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_Signal_WriteSignal

G_Net2Run_Com_Signal_WriteSignal — Write Com signal

Description

Use this command to write a Com signal

Definition

```
G_Error_t
G_Net2Run_Com_Signal_WriteSignal(
 const G_PortHandle_t  portHandle,
 const u16_t  signalId,
 const u16_t  signalLength,
 const u8_t * const signalData
);
```

Parameters

`portHandle`
Handle to the communication port

`signalId`
Signal ID

0..max number defined by G_Net2Run_Com_Signal_MaxNumber -1

`signalLength`
Length of signal data in bytes

`signalData`
Signal data to be written

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_TxPduSignal_Mapping

G_Net2Run_Com_TxPduSignal_Mapping — Configuration of Com TX PDU signal mapping

Description

Use this command for Configuration of Com TX PDU signal mapping.

Definition

```
G_Error_t
G_API
G_Net2Run_Com_Signal_TxPduSignal_Mapping(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_Signal_TxPduSignal_Mapping_Parameters_t * const
  parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__SIGNAL__TX_PDU_SIGNAL__MAPPING__FLAG__NONE`
No flag is set

`G_NET2RUN__COM__SIGNAL__TX_PDU_SIGNAL__MAPPING__FLAG__READ_SIGNAL`
0 : read from PDU

1 : read from signal

`G_NET2RUN__COM__SIGNAL__TX_PDU_SIGNAL__MAPPING__FLAG__DISABLE_UPD_NOT`
0 : enable signal update notification:

Depending on flag `G_NET2RUN__COM__SIGNAL__TX_PDU_SIGNAL__MAPPING__FLAG_READ_SIGNAL`, the signal is notified as updated when the signal mapping is written to the Tx PDU's signal (`G_NET2RUN__COM__SIGNAL__TX_PDU_SIGNAL__MAPPING__FLAG_READ_SIGNAL == 1`) or when the signal is written to the Tx PDU (`G_NET2RUN__COM__SIGNAL__TX_PDU_SIGNAL__MAPPING__FLAG_READ_SIGNAL == 0`).

1 : disable signal update notification

`SignalId`
Signal ID

0..max number defined by G_Net2Run_Com_Signal_MaxNumber -1

`SignalMappingId`
Signal mapping ID

0..(`NumberOfMappings-1`) within "Signal"

`TxPduId`
TX PDU ID

0..max number defined by G_Net2Run_Com_TxPdu_MaxNumber -1

`TxPduSignalId`
TX PDU signal ID

0..(`NumberOfSignals-1`) within "TxPdu"

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_RxPduSignal_Mapping

G_Net2Run_Com_RxPduSignal_Mapping — Configuration of Com RX PDU signal mapping

Description

Use this command for Configuration of Com RX PDU signal mapping.

Definition

```
G_Error_t
G_API
G_Net2Run_Com_Signal_RxPduSignal_Mapping(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_Signal_RxPduSignal_Mapping_Parameters_t * const
  parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__SIGNAL__RX_PDU_SIGNAL__MAPPING__FLAG__NONE`
No flag is set

`G_NET2RUN__COM__SIGNAL__RX_PDU_SIGNAL__MAPPING__FLAG__READ_SHADOW_BUFFER`
0 : read from signal

1 : read from shadow buffer (for group signals only)

`G_NET2RUN__COM__SIGNAL__RX_PDU_SIGNAL__MAPPING__FLAG__DISABLE_UPD_NOT`
0 : enable signal update notification:

Depending on flag `G_NET2RUN__COM__SIGNAL__RX_PDU_SIGNAL__MAPPING__FLAG__READ_SHADOW_BUFFER`, the signal is notified as updated when the signal was received (`G_NET2RUN__COM__SIGNAL__RX_PDU_SIGNAL__MAPPING__FLAG__READ_SHADOW_BUFFER == 0`) or when the signal value was copied to the shadow buffer (`G_NET2RUN__COM__SIGNAL__RX_PDU_SIGNAL__MAPPING__FLAG__READ_SHADOW_BUFFER == 1`).

1 : disable signal update notification

`SignalId`
Signal ID

0..max number defined by G_Net2Run_Com_Signal_MaxNumber -1

`SignalMappingId`
Signal mapping ID

0..(`NumberOfMappings-1`) within "Signal"

`RxPduId`
RX PDU ID

0..max number defined by G_Net2Run_Com_RxPdu_MaxNumber -1

`RxPduSignalId`
RX PDU signal ID

0..(`NumberOfSignals-1`) within "RxPdu"

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_PduGateway_Mapping_Config

G_Net2Run_Com_PduGateway_Mapping_Config — Configuration of Com PDU gateway mapping

Description

Use this command for configuration of Com PDU gateway mapping.

Definition

```
G_Error_t
G_Net2Run_Com_PduGateway_Mapping_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_PduGateway_Mapping_Config_Parameters_t * const
  parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__PDU_GATEWAY__MAPPING__CONFIG__FLAG__NONE`
No flag is set

`RxPduId`
RX PDU ID

0..max number defined by G_Net2Run_Com_RxPdu_MaxNumber -1

`TxPduId`
TX PDU ID

0..max number defined by G_Net2Run_Com_TxPdu_MaxNumber -1

`TransferProperty`
Transfer property

The following values are possible:

`G_NET2RUN__COM__TRANSFER_PROPERTY__TRIGGERED`
When a TX signal with this transfer property is changed, the COM layer initiates an immediate transmission of the TX PDU, except if the defined transmission mode (see "G_Net2Run_Com_TxMode_t") for the TX PDU is configured to PERIODIC or NONE

`G_NET2RUN__COM__TRANSFER_PROPERTY__PENDING`
When a TX signal with this transfer property is changed, the COM layer doesn't initiate an immediate transmission of the TX PDU. The transmission keeps pending until an other TX signal within the same TX PDU with transfer property TRIGGERED is changed or the timer

for periodic transmission expires in case of the TX PDU's transmission mode PERIODIC or MIXED (see "G_Net2Run_Com_TxMode_t").

`G_NET2RUN__COM__TRANSFER_PROPERTY__TRIGGERED_ON_CHANGE`
Depending on the transmission mode, a write access to a signal with this transfer property can trigger the transmission of the corresponding tx PDU, but only in case the written value is different to the locally stored (last sent or initial value) in length or value.

AUTOSAR name: ComTransferProperty::TRIGGERED_ON_CHANGE

`G_NET2RUN__COM__TRANSFER_PROPERTY__TRIGGERED_ON_CHANGE_WITHOUT_REP`
Depending on the transmission mode, a write access to a signal with this transfer property can trigger the transmission of the corresponding I-PDU just once without a repetition, but only in case the written value is different to the locally stored (last sent or initial value) in length or value.

AUTOSAR name:
ComTransferProperty::TRIGGERED_ON_CHANGE_WITHOUT_REPETITION

`G_NET2RUN__COM__TRANSFER_PROPERTY__TRIGGERED_WITHOUT_REP`
Depending on the transmission mode, a write access to a signal with this transfer property can trigger the transmission of the corresponding I-PDU just once without a repetition.

AUTOSAR name: ComTransferProperty::TRIGGERED_WITHOUT_REPETITION

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_SignalGateway_Mapping_Config

G_Net2Run_Com_SignalGateway_Mapping_Config — Configuration of Com signal gateway mapping

Description

Use this command for configuration of Com signal gateway mapping.

Definition

```
G_Error_t
G_Net2Run_Com_SignalGateway_Mapping_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_SignalGateway_Mapping_Config_Parameters_t * const
  parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__SIGNAL_GATEWAY__MAPPING__CONFIG__FLAG__NONE`
No flag is set

`G_NET2RUN__COM__SIGNAL_GATEWAY__MAPPING__CONFIG__FLAG__OFFSET_FAC-TOR`
Use parameters `Offset` and `Factor` for calculation.

`G_NET2RUN__COM__SIGNAL_GATEWAY__MAPPING__CONFIG__FLAG__PHYS_VALUES`
Use physical signal value

`RxPduId`
RX PDU ID

0..max number defined by G_Net2Run_Com_RxPdu_MaxNumber -1

`RxPduSignalId`
RX PDU signal ID

0..(`NumberOfSignals`-1) within "RxPdu"

`TxPduId`
TX PDU ID

0..max number defined by G_Net2Run_Com_TxPdu_MaxNumber -1

`TxPduSignalId`
TX PDU signal ID

0..(NumberOfSignals-1) within "TxPdu"

`Offset`
Offset for signal calculation

Used if flag `G_NET2RUN__COM__SIGNAL_GATEWAY__MAPPING__CONFIG__FLAG__OFFSET_FACTOR` is set.

`Factor`
Factor for signal calculation

Used if flag `G_NET2RUN__COM__SIGNAL_GATEWAY__MAPPING__CONFIG__FLAG__OFFSET_FACTOR` is set.

```
the TX signal is calculated by multiple methods:

 UsePhysicalValues == 0:
  UseOffsetAndFactor == 0:
   txSignalValue = rxSignalValue

  UseOffsetAndFactor == 1:
   txSignalValue = (rxSignalValue * Factor) + Offset

 UsePhysicalValues == 1
  UseOffsetAndFactor == 0:
   rxPhysicalValue = SignalValueToPhysicalValue(rxSignalValue)
   txSignalValue = PhysicalValueToSignalValue(rxPhysicalValue)

  UseOffsetAndFactor == 1:
   rxPhysicalValue = SignalValueToPhysicalValue(rxSignalValue)
   txPhysicalValue = (rxPhysicalValue * Factor) + Offset
   txSignalValue = PhysicalValueToSignalValue(txPhysicalValue)
```

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_TxPdu_Property_GetById

G_Net2Run_Com_TxPdu_Property_GetById — Get property value of TxPdu

Description

Use this command to query the value of TxPdu.

Definition

```
G_Error_t
G_Net2Run_Com_TxPdu_Property_GetById(
 const G_PortHandle_t  portHandle,
 const u32_t  txPduId,
 const u32_t  id,
 u32_t * const value
);
```

Parameters

`portHandle`
Handle to the communication port

`txPduId`
COM ID of the TX PDU

`id`
Property ID of the TX PDU

The following values are possible:

`G_NET2RUN__COM__TX_PDU__PROPERTY_ID__TX_MODE__TRUE__TIME_PERIOD`
Period [microseconds] of the repetition of cyclic transmissions in case the COM filter evaluates to true.

AUTOSAR name: ComTxModeTimePeriod

`G_NET2RUN__COM__TX_PDU__PROPERTY_ID__TX_MODE__FALSE__TIME_PERIOD`
Period [microseconds] of the repetition of cyclic transmissions in case the COM filter evaluates to false.

AUTOSAR name: ComTxModeTimePeriod

`G_NET2RUN__COM__TX_PDU__PROPERTY_ID__PDU_LENGTH__CURRENT`
actual PDU length in bytes (0..PduLength, see "Net2Run_Param_Com_TxPdu_Config_t")

`G_NET2RUN_PARAM__COM__TX_PDU__PROPERTY_ID__UNKNOWN`
The TX PDU Property is unknown

`value`
Returns property value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_TxPdu_Property_SetById

G_Net2Run_Com_TxPdu_Property_SetById — Set property value of TxPdu

Description

Use this command to set the value of TxPdu.

Definition

```
G_Error_t
G_Net2Run_Com_TxPdu_Property_SetById(
 const G_PortHandle_t  portHandle,
 const u32_t  txPduId,
 const u32_t  id,
 const u32_t  value
);
```

Parameters

`portHandle`
Handle to the communication port

`txPduId`
COM ID of the TX PDU

`id`
Property ID of the TX PDU (see G_Net2Run_Param_Com_TxPdu_PropertyId_t)

The following values are possible:

`G_NET2RUN__COM__TX_PDU__PROPERTY_ID__TX_MODE__TRUE__TIME_PERIOD`
Period [microseconds] of the repetition of cyclic transmissions in case the COM filter evaluates to true.

AUTOSAR name: ComTxModeTimePeriod

`G_NET2RUN__COM__TX_PDU__PROPERTY_ID__TX_MODE__FALSE__TIME_PERIOD`
Period [microseconds] of the repetition of cyclic transmissions in case the COM filter evaluates to false.

AUTOSAR name: ComTxModeTimePeriod

`G_NET2RUN__COM__TX_PDU__PROPERTY_ID__PDU_LENGTH__CURRENT`
actual PDU length in bytes (0..PduLength, see "Net2Run_Param_Com_TxPdu_Config_t")

`G_NET2RUN_PARAM__COM__TX_PDU__PROPERTY_ID__UNKNOWN`
The TX PDU Property is unknown

`value`
Property value to be set.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

5 COM Signal Monitor

Functions for controlling the **COM Signal Monitor**

The **COM Signal Monitor** is used for monitoring RBS signals. Every time a signal written or has been received, it is updated in the signal monitor.

As a prerequisite for the **COM Signal Monitor** , a RBS needs to be configured with G_Net2Run_Config and G_Net2Run_Signal_Config and started with G_Net2Run_Pdu_PduGroupControl .

G_Net2Run_Com_SignalMonitor_Config

G_Net2Run_Com_SignalMonitor_Config — Configuration of **COM Signal Monitor**

Description

Use this command for configuration of the **COM Signal Monitor** .

Definition

```
G_Error_t
G_Net2Run_Com_SignalMonitor_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_SignalMonitor_Config_Cmd_t * const cmd,
 const G_Net2Run_Com_SignalMonitor_Config_Signal_t * const monitorSignals,
 G_Net2Run_Com_SignalMonitor_Config_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__SIGNAL_MONITOR__CONFIG__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__COM__SIGNAL_MONITOR__CONFIG__CMD_FLAG__GENERATE_ID`
0 : the value of `SignalMonitorId` is used for the signal monitor ID

1 : the signal monitor ID is generated automatically

`SignalMonitorId`
An ID that identifies the signal monitor (used if flag `G_NET2RUN__COM__SIGNAL_MONITOR_ _CONFIG__CMD_FLAG__GENERATE_ID` is not set)

`MaxEntryDataSize`
Maximum size of the data part of an entry in the signal monitor buffer

`NumberOfEntries`
Number of entries the signal monitor buffer can contain

`NumberOfSignals`
Number of signals in `monitorSignals`

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`monitorSignals`
Array with signal definitions for the signal monitor

The structure contains the following elements:

`SignalHandle`
Signal handle of the RBS signal

A signal handle can of a configured RBS signal be queried with G_Net2Run_Signal_GetSignalHandle .

`Flags`
Signal flags for signal monitor

The following values are possible and can be combined:

`G_NET2RUN__COM__SIGNAL_MONITOR__CONFIG__SIGNAL__FLAG__NONE`
No flag is set

`G_NET2RUN__COM__SIGNAL_MONITOR__CONFIG__SIGNAL__FLAG__USE_TIMESTAMP`
0 : The signal's timestamp is not included in monitor response

1 : the signal's timestamp is included in monitor response

`G_NET2RUN__COM__SIGNAL_MONITOR__CONFIG__SIGNAL__FLAG__PHYSICAL`
0 : The signal's internal value will be read

1 : The signal's physical value will be read

`rsp`
Returns response parameters

The structure contains the following elements:

`SignalMonitorId`
Returns signal monitor ID

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_SignalMonitor_Delete

G_Net2Run_Com_SignalMonitor_Delete — Delete **COM Signal Monitor**

Description

Use this command to delete a **COM signal monitor** and free all of its resources.

Definition

```
G_Error_t
G_Net2Run_Com_SignalMonitor_Delete(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_SignalMonitor_Delete_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM__SIGNAL_MONITOR__DELETE__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__COM__SIGNAL_MONITOR__DELETE__CMD_FLAG__DELETE_ALL`
Delete all configured **COM Signal Monitors**

`SignalMonitorId`
ID of the COM Signal Monitor to be deleted (used if flags `G_NET2RUN__COM__SIGNAL_MONITOR__DELETE__CMD_FLAG__DELETE_ALL` is not set)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_SignalMonitor_Start

G_Net2Run_Com_SignalMonitor_Start — Start **COM Signal Monitor**

Description

Use this command to start a **COM Signal Monitor .**

Definition

```
G_Error_t
G_Net2Run_Com_SignalMonitor_Start(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_SignalMonitor_Start_Cmd_t * const cmd,
 G_Net2Run_Com_SignalMonitor_Start_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__COM__SIGNAL_MONITOR__START__CMD_FLAG__NONE`
No flag is set

`SignalMonitorId`
ID of the **COM Signal Monitor** to be started

`rsp`
Returns response parameters

The structure contains the following elements:

`SignalMonitorId`
ID of the **COM Signal Monitor**

`Timestamp`
Start time of monitor, in nanoseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_SignalMonitor_Stop

G_Net2Run_Com_SignalMonitor_Stop — Stop **COM Signal Monitor**

Description

Use this command to stop a **COM Signal Monitor .**

Definition

```
G_Error_t
G_Net2Run_Com_SignalMonitor_Stop(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_SignalMonitor_Stop_Cmd_t * const cmd,
 G_Net2Run_Com_SignalMonitor_Stop_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__COM__SIGNAL_MONITOR__STOP__CMD_FLAG__NONE`
No flag is set

`SignalMonitorId`
ID of the **COM Signal Monitor** to be stopped

`rsp`
Returns response parameters

The structure contains the following elements:

`SignalMonitorId`
ID of the **COM Signal Monitor**

`Timestamp`
Stop time of monitor, in nanoseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_SignalMonitor_GetSignalInfo

G_Net2Run_Com_SignalMonitor_GetSignalInfo — Get **COM Signal Monitor** signal info

Description

Use this command to query information about the current signal in the **COM Signal Monitor** buffer.

This command returns only the signal handle and the data size of the monitor signal. It can be used to prepare the buffer for the subsequent call of G_Net2Run_Com_SignalMonitor_GetSignalData that returns all available signal data.

The signal read index of the monitor buffer is not changed by this command. If you do not want to read all signal data, you can switch the read index to the next signal with command G_Net2Run_Com_SignalMonitor_SwitchToNextSignal , otherwise the read index will be switched by calling G_Net2Run_Com_SignalMonitor_GetSignalData .

Definition

```
G_Error_t
G_Net2Run_Com_SignalMonitor_GetSignalInfo(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_SignalMonitor_GetSignalInfo_Cmd_t * const cmd,
 G_Net2Run_Com_SignalMonitor_GetSignalInfo_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__COM__SIGNAL_MONITOR__GET_SIGNAL_INFO__CMD_FLAG__NONE`
No flag is set

`SignalMonitorId`
ID of the **COM Signal Monitor**

`rsp`
Returns response parameters

The structure contains the following elements:

`SignalMonitorId`
ID of the **COM Signal Monitor**

`SignalHandle`
The signal handle that identifies the current signal in the buffer

`DataLength`
Data length of the current signal in the buffer, in bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_SignalMonitor_GetSignalData

G_Net2Run_Com_SignalMonitor_GetSignalData — Get signal data of **COM Signal Monitor** signal

Description

Use this command to query the signal data of the current **COM Signal Monitor** signal.

The read index of the **COM Signal Monitor** is switched to the next signal in the buffer.

Definition

```
G_Error_t
G_Net2Run_Com_SignalMonitor_GetSignalData(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_SignalMonitor_GetSignalData_Cmd_t * const cmd,
 G_Net2Run_Com_SignalMonitor_GetSignalData_Rsp_t * const rsp,
 u32_t * const length,
 u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__COM__SIGNAL_MONITOR__GET_SIGNAL_DATA__CMD_FLAG__NONE`
No flag is set

`SignalMonitorId`
ID of the **COM Signal Monitor**

`rsp`
Returns response parameters

The structure contains the following elements:

`SignalMonitorId`
ID of the **COM Signal Monitor**

`SignalHandle`
The signal handle that identifies the current signal in the buffer

`SignalFlags`
Signal flags

The following values are possible and can be combined:

`G_NET2RUN__COM__SIGNAL_MONITOR__SIGNAL__FLAG__NONE`
No flag is set

G_NET2RUN__COM__SIGNAL_MONITOR__SIGNAL__FLAG__BUFFER_OVERFLOW
0 : There was no buffer overflow

1 : There was a buffer overflow before this entry was captured

G_NET2RUN__COM__SIGNAL_MONITOR__SIGNAL__FLAG__ENTRY_SIZE_EXCEEDED
0 : No entry was captured whose size exceeded the maximum entry size

1 : tried to capture an entry whose size exceeded the maximum entry size before this entry was captured

G_NET2RUN__COM__SIGNAL_MONITOR__SIGNAL__FLAG__PHYSICAL
0 : The signal's internal value is read

1 : The signal's physical value is read

G_NET2RUN__COM__SIGNAL_MONITOR__SIGNAL__FLAG__TIMESTAMP
0 : the signal's timestamp is not returned, response parameter `Timestamp` is **0**

1 : The signal's timestamp is returned in response parameter `Timestamp`

`Timestamp`
Returns timestamp of signal, in nanoseconds

If flag `G_NET2RUN__COM__SIGNAL_MONITOR__SIGNAL__FLAG__TIMESTAMP` is not set, this value will be **0** .

`length`
in : Size of buffer `data`, in bytes

out : Number of returned data bytes

`data`
Returns signal data bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Com_SignalMonitor_SwitchToNextSignal

G_Net2Run_Com_SignalMonitor_SwitchToNextSignal — Switch to next **COM Signal Monitor** signal in buffer

Description

Use this command to switch to the next **COM Signal Monitor** signal in the buffer.

When using command G_Net2Run_Com_SignalMonitor_GetSignalData the read index will switch automatically to the next signal and therefore a call to G_Net2Run_Com_SignalMonitor_SwitchToNextSignal will not be necessary.

Definition

```
G_Error_t
G_Net2Run_Com_SignalMonitor_SwitchToNextSignal(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Com_SignalMonitor_SwitchToNextSignal_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__COM__SIGNAL_MONITOR__SWITCH_TO_NEXT_SIGNAL__CMD_FLAG__NONE`
No flag is set

`SignalMonitorId`
ID of the **COM Signal Monitor**

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

6 Communication Manager

The Net2Run Communication Manager (Com Manager) provides control over the communication mode of members of the **Rest Bus Simulation** .

By changing the communication mode, it is possible to actively influence the network management behavior of an ECU.

Each ECU of the rest bus simulation is a **Com Manager User** and has a unique user identifier.

The Com Manager is automatically configured when G_Net2Run_Signal_Config is called.

G_Net2Run_ComManager_User_Config

G_Net2Run_ComManager_User_Config — Com Manager User Configuration

Description

Use this function for Net2Run Com Manager User configuration.

Com Manager Users can be used for controlling Network Management functionality of the restbus simulation.

Definition

```
G_Error_t
G_Net2Run_ComManager_User_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_ComManager_User_Config_CmdFlags_t  cmdFlags,
 const char * const  userName,
 const u32_t  numberOfMappings,
 const G_Net2Run_ComManager_User_Config_Mappings_t * const mappings,
 u32_t * const userId
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command Flags

The following values are possible and can be combined:

`G_ERROR__DLL__NET2RUN__COM_MANAGER__USER__CONFIG__CMD_FLAG__NONE`
No flag is set

`userName`
Name for the new Com Manager User

`numberOfMappings`
Number of mappings for the new Com Manager User

`mappings`
Mappings for the new Com Manager User

The structure contains the following elements:

`BusType`
Bus type of the User mapping

The following values are possible:

`G_NET2RUN__BUS_TYPE__CAN`
CAN bus

`G_NET2RUN__BUS_TYPE__LIN`
LIN bus

`G_NET2RUN__BUS_TYPE__FLEXRAY`
FlexRay bus

`ControllerId`
Controller ID of the user mapping that specifies the interface that is used (e.g. 0=CAN1, 1=CAN2, e.t.c.)

`EcuId`
ECU ID of the user mapping

`reserved`
reserved parameter (must be initialized with **0**)

`userId`
Returns user ID

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_ComManager_NumberOfUsers_Get

G_Net2Run_ComManager_NumberOfUsers_Get — Return total number of **Com Manager Users**

Description

Use this function to query the total number of **Com Manager Users** .

Each ECU of the rest bus simulation is a **Com Manager User** and has a unique user identifier.

For an example, see Net2Run ComManager Example.

Definition

```
G_Error_t
G_Net2Run_ComManager_NumberOfUsers_Get(
 const G_PortHandle_t  portHandle,
 u32_t * const numberOfUsers
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfUsers`
Returns total number of **Com Manager Users**

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_ComManager_UserList_Get

G_Net2Run_ComManager_UserList_Get — Return list with **Com Manager User** information

Description

Use this command to query a list with information about all **Com Manager Users** , including name and user identifier.

For an example, see Net2Run ComManager Example.

Definition

```
G_Error_t
G_Net2Run_ComManager_UserList_Get(
 const G_PortHandle_t  portHandle,
 u32_t * const numberOfUsers,
 G_Net2Run_ComManager_User_t * const users
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfUsers`
in : size of user information the buffer provided by `users`, in number of users

out : number of user information in buffer `users`

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_ComManager_UserId_Get

G_Net2Run_ComManager_UserId_Get — Return user identifier

Description

Use this command to query the user identifier of a **Com Manager User**

Each ECU of a **Rest Bus Simulation** is a **Com Manager User** and has a unique user identifier.

Definition

```
G_Error_t
G_Net2Run_ComManager_UserId_Get(
 const G_PortHandle_t  portHandle,
 const char * const name,
 u32_t * const userId
);
```

Parameters

`portHandle`
Handle to the communication port

`name`
Name of the Com Manager User (ECU)

`userId`
Returns Com Manager user identifier

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_ComManager_ResetUserBuffer

G_Net2Run_ComManager_ResetUserBuffer — Reset user list

Description

Use this command to reset the user list.

Definition

```
G_Error_t
G_Net2Run_ComManager_ResetUserBuffer(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_ComManager_ComMode_Request

G_Net2Run_ComManager_ComMode_Request — Request Communication Mode

Description

Use this command to request a communication mode.

By changing the communication mode, it is possible to actively influence the network management behavior of an ECU.

Depending on the current state of the **Rest Bus Simulation** , an ECU may not change its communication mode immediately. Therefore you can query the current mode with G_Net2Run_ComManager_ComMode_GetCurrent and the requested mode with G_Net2Run_ComManager_ComMode_GetRequested .

For an example, see Net2Run ComManager Example.

Definition

```
G_Error_t
G_Net2Run_ComManager_ComMode_Request(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_ComManager_ComMode_Request_CmdFlags_t  cmdFlags,
 const u32_t  userId,
 const G_Net2Run_ComManager_ComMode_t  comMode
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__COM_MANAGER__COM_MODE__REQUEST__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__COM_MANAGER__COM_MODE__REQUEST__CMD_FLAG__ALL_USERS`
Request this mode for all Com Manager Users (ECUs)

`userId`
Com Manager user identifier (not regarded when flag `G_NET2RUN__COM_MANAGER__COM_MODE__REQUEST__CMD_FLAG__ALL_USERS` is set)

`comMode`
Communication mode that is requested

The following values are possible:

`G_NET2RUN__COM_MANAGER__COM_MODE__UNKNOWN`
The communication mode is unknown.

Only used as a return value.

`G_NET2RUN__COM_MANAGER__COM_MODE__NO_COMMUNICATION`
Request 'No Communication' mode

The ECU shall have no transmission or reception capability.

`G_NET2RUN__COM_MANAGER__COM_MODE__SILENT_COMMUNICATION`
Request 'Silent' mode

The ECU shall have only reception capability, no transmission capability.

This mode is not a valid request for a **Com Manager User**, it is used for synchronization at shut-down.

`G_NET2RUN__COM_MANAGER__COM_MODE__FULL_COMMUNICATION`
Request 'Full' mode

The ECU shall have both transmission and reception capability.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

The following example configures a **Rest Bus Simulation**, requests a communication mode and prints the current and requested communication mode of every Com Manager User to the screen.

```
#include <stdio.h>
#include <windows.h>
#include "g_api_common.h"
#include "g_api_net2run.h"


//-----------------------------------------------------------------------------
// function:    ErrorHandler
// description: evaluates error code and closes application if an error
 occurred
// parameters:  portHandle - port handle
//              errorCode - error code
//-----------------------------------------------------------------------------
static void
ErrorHandler(
 const G_PortHandle_t  portHandle,
 const G_Error_t  errorCode
)
{
 if (errorCode != G_NO_ERROR) {
  printf("Error: %s\n", G_GetLastErrorDescription());
  (void) G_Common_CloseInterface(portHandle);

  ExitProcess(1);
 }
}


//-----------------------------------------------------------------------------
// function:    main
// description: main function
// parameters:  none
//-----------------------------------------------------------------------------
void
main(
 void
)
```

```
{
 G_PortHandle_t  portHandle;
 // The name of the RBS File, built with the Net2Run GUI.
 const char * const configFilename = "TestProject.rbs";
 G_Net2Run_Pdu_PduGroupControl_Group_t  pduGroup;
 u32_t  numberOfUsers = 5;
 G_Net2Run_ComManager_User_t  users[5];
 u32_t  i;
 G_Net2Run_ComManager_ComMode_t  comMode;
 G_Error_t  rc;

 // Open a port to the Net2Run interface with name "NET2RUN1".
 rc =
  G_Common_OpenInterface(
   "NET2RUN1",
   &portHandle
  );

 ErrorHandler(
  portHandle,
  rc
 );

 // Init all interface parameters and restore default interface state.
 rc =
  G_Common_InitInterface(
   portHandle
  );

 ErrorHandler(
  portHandle,
  rc
 );

 // Configure Rest Bus Simulation from rbs file "TestProject.rbs",
 // built with the Net2Run GUI.
 rc =
  G_Net2Run_Config(
   portHandle,
   G_NET2RUN__CONFIG__CMD_FLAG__NONE,
   configFilename
  );

 ErrorHandler(
  portHandle,
  rc
 );

 // Configure Rest Bus Simulation signals from rbs file "TestProject.rbs",
 // built with the Net2Run GUI.
 // This step is necessary for reading and writing signals.
 rc =
  G_Net2Run_Signal_Config(
   portHandle,
   configFilename
  );

 ErrorHandler(
```

```
 portHandle,
 rc
);

// Enable communication for all ECUs
rc =
 G_Net2Run_ComManager_ComMode_Request(
  portHandle,
  G_NET2RUN__COM_MANAGER__COM_MODE__REQUEST__CMD_FLAG__ALL_USERS,
  0,
  G_NET2RUN__COM_MANAGER__COM_MODE__FULL_COMMUNICATION
 );

ErrorHandler(
 portHandle,
 rc
);

// Start PDU Groups.
// The group ID "0xFFFFFFFF" represents all available PDU Groups.
pduGroup.Flags = G_NET2RUN__PDU__PDU_GROUP_CONTROL__GROUP__FLAG__START;
pduGroup.PduGroupId = 0xFFFFFFFF;

rc =
 G_Net2Run_Pdu_PduGroupControl(
  portHandle,
  1,
  &pduGroup
 );

ErrorHandler(
 portHandle,
 rc
);

// Get list with all Com Manager Users
rc =
 G_Net2Run_ComManager_UserList_Get(
  portHandle,
  &numberOfUsers,
  users
 );

ErrorHandler(
 portHandle,
 rc
);

// Print communication modes for all users
for (i = 0; i < numberOfUsers; i++) {
 printf("%s - ", users[i].Name);

 // Get current communication mode
 rc =
  G_Net2Run_ComManager_ComMode_GetCurrent(
   portHandle,
   users[i].UserId,
   &comMode
```

```
   );

  ErrorHandler(
   portHandle,
   rc
  );

  printf("current: %u ", comMode);

  // Get requested communication mode
  rc =
   G_Net2Run_ComManager_ComMode_GetRequested(
    portHandle,
    users[i].UserId,
    &comMode
   );

  ErrorHandler(
   portHandle,
   rc
  );

  printf("requested: %u\n", comMode);
 }

 // Close the port to the interface.
 rc = G_Common_CloseInterface(portHandle);

 ErrorHandler(
  portHandle,
  rc
 );
}
```

G_Net2Run_ComManager_ComMode_GetCurrent

G_Net2Run_ComManager_ComMode_GetCurrent — Return current communication mode

Description

Use this command to query the current communication for a Com Manager User.

Depending on the current state of the **Rest Bus Simulation** , an ECU may not change its communication mode immediately. Therefore you can query the current mode with G_Net2Run_ComManager_ComMode_GetCurrent and the requested mode with G_Net2Run_ComManager_ComMode_GetRequested .

For an example, see Net2Run ComManager Example.

Definition

```
G_Error_t
G_Net2Run_ComManager_ComMode_GetCurrent(
 const G_PortHandle_t  portHandle,
 const u32_t  userId,
 G_Net2Run_ComManager_ComMode_t * const comMode
);
```

Parameters

`portHandle`
: Handle to the communication port

`userId`
: Com Manager user identifier

`comMode`
: Returns current communication mode for the specified Com Manager User (ECU) (see G_Net2Run_ComManager_ComMode_t)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_ComManager_ComMode_GetRequested

G_Net2Run_ComManager_ComMode_GetRequested — Return requested communication mode

Description

Use this command to query the requested communication for a Com Manager User.

Depending on the current state of the **Rest Bus Simulation** , an ECU may not change its communication mode immediately. Therefore you can query the current mode with G_Net2Run_ComManager_ComMode_GetCurrent and the requested mode with G_Net2Run_ComManager_ComMode_GetRequested .

For an example, see Net2Run ComManager Example.

Definition

```
G_Error_t
G_Net2Run_ComManager_ComMode_GetRequested(
 const G_PortHandle_t  portHandle,
 const u32_t  userId,
 G_Net2Run_ComManager_ComMode_t * const comMode
);
```

Parameters

`portHandle`
: Handle to the communication port

`userId`
: Com Manager user identifier

`comMode`
: Returns requested communication mode for the specified Com Manager User (ECU) (see G_Net2Run_ComManager_ComMode_t)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

7 Configuration

Configuration functions for the Net2Run interface

These functions need to be called before the **Rest Bus Simulation** can be started.

G_Net2Run_Config

G_Net2Run_Config — Configure device with **Rest Bus Simulation** parameters

Description

Use this function to configure the device with **Rest Bus Simulation** parameters.

The **Rest Bus Simulation** parameters used for the configuration are taken from the **.rbs** file that is created with the Net2Run GUI.

This command is mandatory for setting up a **Rest Bus Simulation** , therefore it should be called before any other Net2Run command is called.

Definition

```
G_Error_t
G_Net2Run_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Config_CmdFlags_t  cmdFlags,
 const char * const filename
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__CONFIG__CMD_FLAG__NONE`
No flag is set

`filename`
Name of the **Rest Bus Simulation** file

This file is created by the Net2Run GUI and has usually a **.rbs** file ending.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

8 Crypto Service Manager (CSM)

Configuration functions for the **C**rypto **S**ervice **M**anager module of **Net2Run**

The CSM shall provide synchronous or asynchronous services to enable a unique access to basic cryptographic functionalities for all software modules.

For many parameters in this section, references are made to the corresponding AUTOSAR specification **Specification of Crypto Service Manager 4.3.0** that can be downloaded at http://www.autosar.org.

G_Net2Run_Csm_Job

G_Net2Run_Csm_Job — Crypto Service Manager job configuration

Description

Use this command for Crypto Service Manager job configuration.

A job is a configured Object with refers to a key and a cryptographic primitive.

Definition

```
G_Error_t
G_Net2Run_Csm_Job(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Csm_Job_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__CSM__JOB__CMD_FLAG__NONE`
No flag is set

`JobId`
Job ID

ECUC_Csm_00119

`KeyId`
Key ID

ECUC_Csm_00126

`PrimitiveType`
This parameter describes which cryptographic primitive will be used in the cryptographic algorithms.

A crypto primitive is an instance of a configured cryptographic algorithm realized in a Crypto Driver Object.

The following values are possible:

`G_NET2RUN__CSM__PRIMITIVE_TYPE__NOT_USED`
Pimitive type not used

`G_NET2RUN__CSM__PRIMITIVE_TYPE__CMAC_AES_128`
CMAC AES 128

`G_NET2RUN__CSM__PRIMITIVE_TYPE__SIP_HASH_2_4`
SIP HASH 2.4

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Csm_Key_128Bit

G_Net2Run_Csm_Key_128Bit — Configuration of a 128 bit key

Description

Use this command for Configuration of a 128 bit key.

A Key can be referenced by a job in the CSM.

Definition

```
G_Error_t
G_Net2Run_Csm_Key_128Bit(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Csm_Key_128Bit_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__CSM__KEY__128_BIT__CMD_FLAG__NONE`
No flag is set

`KeyId`
Key ID

ECUC_Csm_00015

`Key`
Key bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

9 ETH Interface

Configuration functions for the **Ethernet (ETH) Interfaces** of **Net2Run**

Net2Run Ethernet interfaces are configured automatically via the **Net2Run Network Runtime Configurator**. In case the **Net2Run Network Runtime Configurator** is not used, you can use these functions to access the Net2Run Ethernet interfaces for manual configuration.

G_Net2Run_EthIf_Controller_Config

G_Net2Run_EthIf_Controller_Config — Configuration of ETH Controller

Description

Use this command for ETH controller configuration.

Definition

```
G_Error_t
G_API
G_Net2Run_EthIf_Controller_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_EthIf_Controller_Config_Cmd_t * const cmd
)
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__ETH_IF__CONTROLLER__CONFIG__FLAG_NONE`
No flag is set

`G_NET2RUN__ETH_IF__CONTROLLER__CONFIG__FLAG__SET_OP_MODE`
Sets the given `OperationMode` for this interface.

`G_NET2RUN__ETH_IF__CONTROLLER__CONFIG__FLAG__SET_DATARATE`
Sets the given `DataRate` for this interface.

`G_NET2RUN__ETH_IF__CONTROLLER__CONFIG__FLAG__SET_MTU_SIZE`
Sets the given `MtuSize` for this interface.

`G_NET2RUN__ETH_IF__CONTROLLER__CONFIG__FLAG__SET_TX_AMPLITUDE`
Sets the given `TxAmplitude` for this interface.

`ControllerId`
ETH controller ID

0 = ETH1, **1** = ETH2, **2** = ETH3, ...

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`OperationMode`
ETH Operation mode

The following values are possible:

`G_ETHERNET__PHY__MASTER_SLAVE_MODE__MASTER`
The operation mode is set to **master**.

`G_ETHERNET__PHY__MASTER_SLAVE_MODE__SLAVE`
The operation mode is set to **slave**.

`G_ETHERNET__PHY__MASTER_SLAVE_MODE__UNKNOWN`
The operation mode is set to an unknown state.

`DataRate`
data rate in Mbps (for example 10, 100, 1000)

`MtuSize`
maximum transmission unit in bytes

`TxAmplitude`
tx amplitude in mV

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_EthIf_IpAddr_Add

G_Net2Run_EthIf_IpAddr_Add — Configuration of ETH Controller

Description

Use this command for ETH controller configuration.

Definition

```
G_Error_t
G_API
G_Net2Run_EthIf_IpAddr_Add(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_EthIf_IpAddr_Add_Cmd_t * const cmd
)
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__ETH_IF__IP_ADDR__ADD__CMD__NONE`
No flag is set

`G_NET2RUN__ETH_IF__IP_ADDR__ADD__CMD__USE_VLAN_ID`
Use VLAN ID

A **V**irtual **L**ocal **A**rea **N**etwork can be configured with G_Net2Run_EthIf_Vlan_Config .

`ControllerId`
ETH controller ID

0 = ETH1, **1** = ETH2, **2** = ETH3, ...

`EcuId`
Ecu ID

ECU ID, starting with 0

`IpAddressType`
IP-Address Type

The following values are possible:

`ETHERNET_IP_VERSION__4`
IP Version **4**.

`ETHERNET_IP_VERSION__6`
IP Version **6**.

`ETHERNET_IP_VERSION__UNKNOWN`
IP Version is unknown / invalid

`Prefix`
IP address prefix "e.g. /64"

`IpDestination`
contains the destination IP address ("`u8_t Data[16]`")

`VlanId`
VLAN ID

Only used if flag `G_NET2RUN__ETH_IF__IP_ADDR__ADD__CMD__USE_VLAN_ID` is set.

A **V**irtual **L**ocal **A**rea **N**etwork can be configured with G_Net2Run_EthIf_Vlan_Config .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_EthIf_Vlan_Config

G_Net2Run_EthIf_Vlan_Config — Configuration of ETH Controller

Description

Use this command for ETH controller configuration.

Definition

```
G_Error_t
G_API
G_Net2Run_EthIf_Vlan_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_EthIf_Vlan_Config_Cmd_t * const cmd
)
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__ETH_IF__VLAN__CONFIG_CMD__NONE`
No flag is set

`ControllerId`
ETH controller ID

0 = ETH1, **1** = ETH2, **2** = ETH3, ...

`EcuId`
Ecu ID

ECU ID, starting with 0

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`VlanId`
A **V**irtual **L**ocal **A**rea **N**etwork to be set. In the range 0..4095

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

10 FlexRay Interface

Configuration functions for the **FlexRay Interfaces** of **Net2Run**

Net2Run FlexRay interfaces are configured automatically via the **Net2Run Network Runtime Configurator** . In case the **Net2Run Network Runtime Configurator** is not used, you can use these functions to access the Net2Run FlexRay interfaces for manual configuration.

G_Net2Run_FlexRayIf_Cluster_Config

G_Net2Run_FlexRayIf_Cluster_Config — FlexRay cluster configuration

Description

Use this command for FlexRay cluster configuration.

Definition

```
G_Error_t
G_Net2Run_FlexRayIf_Cluster_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_FlexRayIf_Cluster_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`gColdStartAttempts`
Maximum number of times that a node in this cluster is permitted to attempt to start the cluster by initiating schedule synchronization

`gListenNoise`
Upper limit for the start up and wake up listen timeout in the presence of noise. Expressed as a multiple of the cluster constant `pdListenTimeout`

`gMacroPerCycle`
Duration of a FlexRay communication cycle in macroticks

`gNumberOfMinislots`
Number of minislots in the dynamic segment

`gNumberOfStaticSlots`
Number of static slots in the static segment

`gOffsetCorrectionStart`
Start of the offset correction phase within the NIT, expressed as the number of macroticks from the start of cycle.

Unit: macroticks

`gMaxWithoutClockCorrectionFatal`
Threshold used for testing the `vClockCorrectionFailed` counter

Defines the number of consecutive even/odd cycle pairs with missing clock correction terms that will cause the protocol to transition from the POC:normal active or POC:normal passive state into the POC:halt state.

Unit: Even/Odd Cycle Pairs

`gMaxWithoutClockCorrectionPassive`
Threshold used for testing the `vClockCorrectionFailed` counter

Defines the number of consecutive even/odd cycle pairs with missing clock correction terms that will cause the protocol to transition from the POC:normal active state to the POC:normal passive state.
Note that `gMaxWithoutClockCorrectonPassive` = `gMaxWithoutClockCorrection-Fatal` = 15.

`gNetworkManagementVectorLength`
Length of the Network Management vector in a cluster

Default: 0

Unit: byte

`gPayloadLengthStatic`
Payload length of a static frame (All static frames in a cluster have the same payload length)

Unit: 16-bit WORDS

`gSyncNodeMax`
Maximum number of nodes that may send frames with the sync frame indicator bit set

`gdActionPointOffset`
Number of macroticks the action point is offset from the beginning of a static slots or symbol window

`gdBit`
Nominal bit time (= 1 / fx:SPEED)

`gdBit` = `cSamplesPerBit` * `gdSampleClockPeriod`

Unit: us

`gdCasRxLowMax`
Upper limit of the CAS acceptance window

`gdCASRxLowMax[gdBit]` = ceil(2 * (`gdTSSTransmitter[gdBit]` + cdCAS[`gdBit`]) * (1 + 2 * `cClockDeviationMax`))

`gdDynamicSlotIdlePhase`
Duration of the dynamic slot idle phase in minislots

`gdMinislotActionPointOffset`
Number of macroticks the minislot action point is offset from the beginning of a minislot

Unit: macroticks

`gdMinislot`
Duration of a minislot in macroticks

`gdStaticSlot`
Duration of a slot in the static segment

Unit: macroticks

`gdSymbolWindow`
Duration of the symbol window

Unit: macroticks

`gdTSSTransmitter`
Number of bits in the Transmission Start Sequence

Unit: bitDuration

`gdWakeupSymbolRxIdle`
Number of bits used by the node to test the duration of the "idle" portion of a received wake-up symbol

Equal to 6 us minus a safe part. (Collisions, clock differences, and other effects can deform the TX-wakeup pattern.)

Unit: bitDuration

`gdWakeupSymbolRxLow`
Number of bits used by the node to test the LOW portion of a received wakeup symbol

Equal to 6 us minus a safe part. (Active stars, clock differences, and other effects can deform the TX-wakeup pattern).

Unit: bitDuration

`gdWakeupSymbolTxIdle`
Duration of the idle phase between two low phases inside a wakeup pattern

Unit: bitDuration

`gdWakeupSymbolTxLow`
Duration of low phase of a transmitted wakeup symbol

Unit: bitDuration

`gdWakeupSymbolRxWindow`
Number of bits used by a node to test the overall duration of a received wakeup symbol

Equal to 30 us plus a safe part. (Clock differences and other effects can deform the TX-wake-up pattern).

Unit: bitDuration

`gClusterDriftDamping`
Cluster drift damping factor, expressed in multiples of the smallest microtick used in the cluster

Used to compute the local cluster drift damping factor `pClusterDriftDamping`.

`gChannel`
FlexRay channel

The following values are possible:

`G_NET2RUN__FLEXRAY_IF__CHANNEL__CHA`
Channel A

`G_NET2RUN__FLEXRAY_IF__CHANNEL__CHB`
Channel B

`G_NET2RUN__FLEXRAY_IF__CHANNEL__CHAB`
Channel A and B

`ClusterId`
FlexRay cluster ID

`reserved`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_FlexRayIf_Controller_Config

G_Net2Run_FlexRayIf_Controller_Config — FlexRay controller configuration

Description

Use this command for FlexRay controller configuration.

Definition

```
G_Error_t
G_Net2Run_FlexRayIf_Controller_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_FlexRayIf_Controller_Config_Parameters_t * const
 parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`pAllowHaltDueToClock`
Boolean flag that controls the transition to the **POC:halt** state due to a clock synchronization errors

If set to **true**, the CC is allowed to transition to **POC:halt**. If set to **false**, the CC will not transition to the **POC:halt** state but will enter or remain in the **POC:normal** passive state (self healing would still be possible).

`pAllowPassiveToActive`
Number of consecutive even/odd cycle pairs that must have valid clock correction terms before the CC will be allowed to transition from the **POC:normal** passive state to **POC:normal** active state

If set to **0**, the CC is not allowed to transition from **POC:normal** passive to **POC:normal** active.

`pDecodingCorrection`
Value used by the receiver to calculate the difference between primary time reference point and secondary time reference point

`pSingleSlotEnabled`
Flag indicating whether or not the node shall enter single slot mode following startup

`pDelayCompensationA`
Compensation for reception delays on channel A in microticks

Unit: uT

`pDelayCompensationB`
Compensation for reception delays on channel B in microticks

Unit: uT

`pExternOffsetCorrection`
Number of microticks added or subtracted to the NIT to carry out a host-requested external offset correction

`pExternRateCorrection`
Number of microticks added or subtracted to the cycle to carry out a host-requested external rate correction

`pLatestTx`
Number of the last minislot in which a frame transmission can start in the dynamic segment

Unit: minislot

`pKeySlotId`
Key Slot, which is used to transmit a sync frame

The Element holds the slot-ID.

`pKeySlotUsedForStartup`
Flag indicating the Key Slot that is used to transmit a startup frame

`pKeySlotUsedForSync`
Flag indicating the Key Slot is used to transmit a sync frame

`pMicroPerMacroNom`
Number of microticks per nominal macrotick that all implementations must support

`pPayloadLengthDynMax`
Maximum payload length of a dynamic frame in 16bit WORDS

`pMicroPerCycle`
Nominal number of microticks in the communication cycle of the local node

If nodes have different microtick durations this number will differ from node to node.

`pMacroInitialOffsetA`
Integer number of macroticks between the static slot boundary and the closest macrotick boundary of the secondary time reference point based on the nominal macrotick duration

`pMacroInitialOffsetB`
Integer number of macroticks between the static slot boundary and the closest macrotick boundary of the secondary time reference point based on the nominal macrotick duration

`pMicroInitialOffsetA`
Number of microticks between the closest macrotick boundary described by `gMacroInitialOffset` and the secondary time reference point

The parameter depends on `pDelayCompensation[x]` and therefore it has to be set independently for each channel.

`pMicroInitialOffsetB`
Number of microticks between the closest macrotick boundary described by `gMacroInitialOffset` and the secondary time reference point

The parameter depends on `pDelayCompensation[x]` and therefore it has to be set independently for each channel.

`pOffsetCorrectionOut`
Magnitude of the maximum permissible offset correction value

Unit: microtick

`pRateCorrectionOut`
Magnitude of the maximum permissible rate correction value

Unit: uT

`pdAcceptedStartupRange`
Expanded range of measured clock deviation allowed for startup frames during integration

Unit: uT

`pdMaxDrift`
Maximum drift offset between two nodes that operate with unsynchronized clocks over one communication cycle

Unit: uT

`pdListenTimeout`
Upper limit for the start up listen timeout and wake up listen timeout

`pdListenTimeout[uT] = 2 * (pMicroPerCycle[uT] + pdMaxDrift[uT])`

`pWakeupChannel`
FlexRay channel used by the node to send a wakeup pattern

`pWakeupPattern`
Number of repetitions of the wakeup symbol that are combined to form a wakeup pattern when the node enters the **POC:wakeup** send state

`IsLeadingColdstarter`
Flag that defines if controller is a leading cold starter

`ControllerId`
Controller ID

0 = FLEXRAY1, 1 = FLEXRAY2, ...

`ClusterId`
FlexRay cluster ID

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__FLEXRAY_IF__CONTROLLER__CONFIG__FLAG__NONE`
No flag is set

`G_NET2RUN__FLEXRAY_IF__CONTROLLER__CONFIG__FLAG__P_CHANNELS`
Use parameter `pChannels`

`G_NET2RUN__FLEXRAY_IF__CONTROLLER__CONFIG__FLAG__P_CLUSTER_DRIFT_DAMPING`
Use parameter `pClusterDriftDamping`

`pChannels`
FlexRay channel

The following values are possible:

`G_NET2RUN__FLEXRAY_IF__CHANNEL__CHA`
Channel A

`G_NET2RUN__FLEXRAY_IF__CHANNEL__CHB`
Channel B

`G_NET2RUN__FLEXRAY_IF__CHANNEL__CHAB`
Channel A and B

This parameter is relevant if flag `G_NET2RUN__FLEXRAY_IF__CONTROLLER__CONFIG__FLAG__P_CHANNELS` within `Flags` is set.

If the flag is not set, `pChannels` is derived from cluster parameter `gChannels`: `pChannels = gChannels`

`pClusterDriftDamping`
This parameter is relevant if flag `G_NET2RUN__FLEXRAY_IF__CONTROLLER__CONFIG__FLAG__P_CLUSTER_DRIFT_DAMPING` within `Flags` is set.

If the flag is not set, `pClusterDriftDamping` is derived from cluster parameter `gClusterDriftDamping`: `pClusterDriftDamping = gClusterDriftDamping`

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_FlexRayIf_Pdu_Config

G_Net2Run_FlexRayIf_Pdu_Config — FlexRay PDU configuration

Description

Use this command for FlexRay PDU configuration.

Definition

```
G_Error_t
G_Net2Run_FlexRayIf_Pdu_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_FlexRayIf_Pdu_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__FLEXRAY_IF__PDU__CONFIG__FLAG__NONE`
No flag is set

`PduId`
PDU ID

`ControllerId`
Controller ID

0 = FLEXRAY1, 1 = FLEXRAY2

`ByteLengthMax`
Maximum byte length

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_FlexRayIf_LPdu_Config

G_Net2Run_FlexRayIf_LPdu_Config — FlexRay L-PDU configuration

Description

Use this command for FlexRay L-PDU configuration.

Definition

```
G_Error_t
G_Net2Run_FlexRayIf_LPdu_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_FlexRayIf_LPdu_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__FLEXRAY_IF__LPDU__CONFIG__FLAG__NONE`
No flag is set

`G_NET2RUN__FLEXRAY_IF__LPDU__CONFIG__FLAG__SET_UNUSED_BITS`
Set unused bits to value defined by `UnusedBitValue`

`G_NET2RUN__FLEXRAY_IF__LPDU__CONFIG__FLAG__SET_UNUSED_BYTES`
Set unused bytes to value defined by `UnusedByteValue`

`LPduId`
L-PDU ID

`ControllerId`
Controller ID

0 = FLEXRAY1, 1 = FLEXRAY2

`ByteLengthMax`
Maximum byte length

`UnusedBitValue`
Relevant if flag `G_NET2RUN__FLEXRAY_IF__LPDU__CONFIG__FLAG__SET_UNUSED_BITS` is set

`UnusedByteValue`
Relevant if flag `G_NET2RUN__FLEXRAY_IF__LPDU__CONFIG__FLAG__SET_UNUSED_BYTES` is set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_FlexRayIf_PduInLPdu_Config

G_Net2Run_FlexRayIf_PduInLPdu_Config — Configuration of PDU in L-PDU

Description

Use this command for configuration of PDU in L-PDU.

Definition

```
G_Error_t
G_Net2Run_FlexRayIf_PduInLPdu_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_FlexRayIf_PduInLPdu_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__FLEXRAY_IF__PDU_IN_LPDU__CONFIG__FLAG__NONE`
No flag is set

`G_NET2RUN__FLEXRAY_IF__PDU_IN_LPDU__CONFIG__FLAG__USE_UPDATE_BIT`
Use parameter `UpdateBitPosition`

`PduId`
PDU ID

`LPduId`
L-PDU ID

`UpdateBitPosition`
Bit position of update-bit inside PDU

Only relevant if flag `G_NET2RUN__FLEXRAY_IF__pDU_IN_LPDU__CONFIG__FLAG__USE_UPDATE_BIT` is set

`ControllerId`
Controller ID

0 = FLEXRAY1, 1 = FLEXRAY2

`BytePosition`
byte position of PDU within L-PDU

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_FlexRayIf_FrameTrigger_Config

G_Net2Run_FlexRayIf_FrameTrigger_Config — FlexRay frame trigger configuration

Description

Use this command for FlexRay frame trigger configuration.

Definition

```
G_Error_t
G_Net2Run_FlexRayIf_FrameTrigger_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_FlexRayIf_FrameTrigger_Config_Parameters_t * const
  parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__FLEXRAY_IF__FRAME_TRIGGER__CONFIG__FLAG__NONE`
No flag is set

`G_NET2RUN__FLEXRAY_IF__FRAME_TRIGGER__CONFIG__FLAG__PAYLOAD_PREAMBLE`
Enable payload preamble

`G_NET2RUN__FLEXRAY_IF__FRAME_TRIGGER__CONFIG__FLAG__ALLOW_DYN_LENGTH`
Allow dynamic length

`G_NET2RUN__FLEXRAY_IF__FRAME_TRIGGER__CONFIG__FLAG__ALWAYS_TRANSMIT`
Enable 'always transmit'

`G_NET2RUN__FLEXRAY_IF__FRAME_TRIGGER__CONFIG__FLAG__USE_IMMEDIATE_PDU`
Use immediate PDU

`G_NET2RUN__FLEXRAY_IF__FRAME_TRIGGER__CONFIG__FLAG__SEND_NULL_FRAMES`
Send NULL Frames

`LPduId_ImmediatePduId`
If flag `G_NET2RUN__FLEXRAY_IF__FRAME_TRIGGER__CONFIG__FLAG__USE_IMMEDIATE_PDU` is set, this parameter contains the immediate PDU ID.

If flag G_NET2RUN__FLEXRAY_IF__FRAME_TRIGGER__CONFIG__FLAG__USE_IMMEDIATE_PDU is not set, this parameter contains the L-PDU ID.

SlotId
Slot ID

BaseCycle
Base cycle

CycleRepetition
Cycle repetition

Channel
FlexRay channel

The following values are possible:

G_NET2RUN__FLEXRAY_IF__CHANNEL__CHA
Channel A

G_NET2RUN__FLEXRAY_IF__CHANNEL__CHB
Channel B

G_NET2RUN__FLEXRAY_IF__CHANNEL__CHAB
Channel A and B

ControllerId
Controller ID

0 = FLEXRAY1, 1 = FLEXRAY2

FillByteValue
Fill byte value

reserved2
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_FlexRayIf_TxPdu_MaxNumber

G_Net2Run_FlexRayIf_TxPdu_MaxNumber — Configure maximum number of FlexRay TX PDUs

Description

Use this command to configure the maximum number of FlexRay TX PDUs.

This command needs to be called prior to G_Net2Run_FlexRayIf_TxPdu_Config .

Definition

```
G_Error_t
G_Net2Run_FlexRayIf_TxPdu_MaxNumber(
 const G_PortHandle_t  portHandle,
 const u32_t  maxNumber
);
```

Parameters

`portHandle`
Handle to the communication port

`maxNumber`
Maximum number of FlexRay TX PDUs

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_FlexRayIf_RxPdu_MaxNumber

G_Net2Run_FlexRayIf_RxPdu_MaxNumber — Configure maximum number of FlexRay RX PDUs

Description

Use this command to configure the maximum number of FlexRay RX PDUs.

This command needs to be called prior to G_Net2Run_FlexRayIf_RxPdu_Config .

Definition

```
G_Error_t
G_Net2Run_FlexRayIf_RxPdu_MaxNumber(
 const G_PortHandle_t  portHandle,
 const u32_t  maxNumber
);
```

Parameters

`portHandle`
Handle to the communication port

`maxNumber`
Maximum number of FlexRay RX PDUs

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_FlexRayIf_TxPdu_Config

G_Net2Run_FlexRayIf_TxPdu_Config — FlexRay TX PDU configuration

Description

Use this command for FlexRay TX PDU configuration.

Prior to this command, G_Net2Run_FlexRayIf_TxPdu_MaxNumber needs to be called.

Definition

```
G_Error_t
G_Net2Run_FlexRayIf_TxPdu_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_FlexRayIf_TxPdu_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__FLEXRAY_IF__TX_PDU__CONFIG__FLAG__NONE`
No flag is set

`G_NET2RUN__FLEXRAY_IF__TX_PDU__CONFIG__FLAG__ENABLE_TX_CONFIRMATION`
Enable TX confirmation

`G_NET2RUN__FLEXRAY_IF__TX_PDU__CONFIG__NO_TRIGGER_TRANSMIT`
No triggered transmission for this PDU

`G_NET2RUN__FLEXRAY_IF__TX_PDU__CONFIG__FLAG__NONE_MODE`
The PDU has no transmission mode (`G_Net2Run_Com_TxMode_t` is set to `G_NET2RUN__COM__TX_MODE__NONE`)

AUTOSAR FlexRay Interface: "NoneMode" means that there is no API FrIf_Transmit call of the upper layer for this PDU.

`FrPduId`
PDU ID in FlexRay interface, configured with G_Net2Run_FlexRayIf_Pdu_Config

`TxPduId`
TX PDU ID

0..max number defined by G_Net2Run_FlexRayIf_TxPdu_MaxNumber -1

`EcuId`
ECU ID, starting with 0

`ControllerId`
Controller ID

0 = FLEXRAY1, 1 = FLEXRAY2

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_FlexRayIf_RxPdu_Config

G_Net2Run_FlexRayIf_RxPdu_Config — FlexRay RX PDU configuration

Description

Use this command for FlexRay RX PDU configuration.

Prior to this command, G_Net2Run_FlexRayIf_RxPdu_MaxNumber needs to be called.

Definition

```
G_Error_t
G_Net2Run_FlexRayIf_RxPdu_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_FlexRayIf_RxPdu_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__FLEXRAY_IF__RX_PDU__CONFIG__FLAG__NONE`
No flag is set

`G_NET2RUN__FLEXRAY_IF__RX_PDU__CONFIG__FLAG__ENABLE_RX_INDICATION`
Enable RX indication

`FrPduId`
PDU ID in FlexRay interface, configured with G_Net2Run_FlexRayIf_Pdu_Config

`RxPduId`
RX PDU ID

0..max number defined by G_Net2Run_FlexRayIf_RxPdu_MaxNumber -1

`EcuId`
ECU ID, starting with 0

`ControllerId`
Controller ID

0 = FLEXRAY1, 1 = FLEXRAY2

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

11 FlexRay Network Management Control

Functions for controlling the FlexRay network management of a Net2Run **Rest Bus Simulation**

G_Net2Run_FlexRayNm_Autosar_Ecu

G_Net2Run_FlexRayNm_Autosar_Ecu — Flexray Network Management autosar ecu configuration

Description

Use this command for Flexray Network Management autosar ecu configuration.

Definition

```
G_Error_t
G_Net2Run_FlexRayNm_Autosar_Ecu(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_FlexRayNm_Autosar_Ecu_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAGS__NONE`
No flag is set

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__PASSIVE_MODE_ENABLED`
In the Passive Mode the node is only receiving Network Management PDUs but not transmitting any Network Management PDUs.

AUTOSAR name: FrNmPassiveModeEnabled {FRNM_PASSIVE_MODE_ENABLED}

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__HW_VOTE_ENABLED`
0 = no use of NM hardware support

1 = use FlexRay NM hardware support (see chapter 9.3.3.4 Network Management Service of FlexRay Communications System Protocol Specification Version 2.1 Revision D)

AUTOSAR name: FrNmHwVoteEnable {FRNM_HW_VOTE_ENABLE}

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__CONTROL_BIT_VECTOR_ENABLED`
Use control bit vector in network mananagment PDU.

AUTOSAR name: FrNmControlBitVectorEnabled {FRNM_CONTROL_BIT_VECTOR_ENABLED}

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__SOURCE_NODE_IDENTIFIER_ENABLED`
Use source node identifier in network management PDU.

AUTOSAR name: FrNmSourceNodeIdentifierEnabled {FRNM_SOURCE_NODE_INDENTIFIER_ENABLED}

G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__USE_NMS_IN_PDU
Use network management state in NM data PDU

This is an extension to AUTOSAR.

G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__USER_DATA_ENABLED
0 = sending of NM-Data PDU is disabled

1 = sending of NM-Data PDU is enabled

AUTOSAR name: FrNmUserDataEnabled {FRNM_USER_DATA_ENABLED}

G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__REMOTE_SLEEP_IND_ENABLED
0 = detection of remote sleep indication is disabled

1 = detection of remote sleep indication is enabled

AUTOSAR name: FrNmRemoteSleepIndicationEnabled
{FRNM_REMOTE_SLEEP_INDICATION_ENABLED}

G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__SYNCHRONIZATION_POINT_ENABLED
This flag defines if synchronization point indication to the NM interface is enabled (1) or disabled (0).

AUTOSAR name: FrNmSynchronizationPointEnabled
{FRNM_SYNCHRONIZATIONPOINT_ENABLED}

G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__NODE_DETECTION_ENABLED
0 = repeat message requests are ignored

1 = repeat message requests are processed

Repeat message requests are detected if bit0 in control bit vector of a received message is set or function Nm_RepeatMessageRequest is called.

AUTOSAR name: FrNmNodeDetectionEnabled {FRNM_NODE_DETECTION_ENABLED}

G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__COM_USER_DATA_SUPPORT
0 = user data is not collected from an I-PDU by calling "PduRouter_FlexRayNm_TriggerTransmit"

1 = user data is collected from an I-PDU by calling "PduRouter_FlexRayNm_TriggerTransmit"

AUTOSAR name: FrNmComUserDataSupport {FRNM_COM_USER_DATA_SUPPORT}

G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__PDU_RX_INDICATION_ENABLED
0 = disable NM PDU Reception indication to NM

1 = enable NM PDU Reception indication to NM

AUTOSAR name: FrNmPduRxIndicationEnabled {FRNM_PDU_RX_INDICATION_ENABLED}

G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__STATE_CHANGE_INDICATION_ENABLED
0 = disable state change indication to NM

1 = enable state change indication to NM

AUTOSAR name: FrNmStateChangeIndicationEnabled {FRNM_STATE_CHANGE_INDICATION_ENABLED}

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__CAR_WAKE_UP_RX_ENABLED`

0 = disable CarWakeUp bit evaluation in received NM messages

1 = enable CarWakeUp bit evaluation in received NM messages

AUTOSAR name: FrNmCarWakeUpRxEnabled {FRNM_CAR_WAKE_UP_RX_ENABLED}

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__CAR_WAKE_UP_FILTER_ENABLED`

0 = CarWakeUp filtering is not supported

1 = CarWakeUp filtering is supported

If CarWakeUp filtering is supported, only the CarWakeUp bit within the NM message with source node identifier "CarWakeUpFilterNodeId" is considered as CarWakeUp request.

AUTOSAR name: FrNmCarWakeUpFilterEnabled {FRNM_CAR_WAKE_UP_FILTER_ENABLED}

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__COORDINATOR_SYNC_SUPPORT`

0 = disable coordinator synchronisation support

1 = enable coordinator synchronisation support

AUTOSAR name: FrNmCoordinatorSyncSupport {FRNM_COORDINATOR_SYNC_SUPPORT}

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__PN_ENABLED`

0 = partial networking is not supported

1 = partial networking is not supported

AUTOSAR name: FrNmPnEnabled {FRNM_PN_ENABLED}

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__PN_EIRA_CALC_ENABLED`

Specifies if FrNm calculates the PN request information for internal and external requests. (EIRA)

0 = PN requests are not calculated

1 = PN requests are calculated

AUTOSAR name: FrNmPnEiraCalcEnabled {FRNM_PN_EIRA_CALC_ENABLED}

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__PN_ERA_CALC_ENABLED`

Specifies if FrNm calculates the PN request information for external requests. (ERA)

0 = PN requests are not calculated

1 = PN requests are calculated

AUTOSAR name: FrNmPnEraCalcEnabled {FRNM_PN_ERA_CALC_ENABLED}

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__USE_ACTIVE_WAKE_UP_BIT_IN_CBV`

0 = NM doesn't change the wake up bit in CBV

1 = NM changes the wake up bit in CBV (if the node actively wakes, the active wake up bit in the CBV is set)

See AUTOSAR requirements FRNM297 and FRNM298.

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__BUS_SYNCHRONIZATION_ENABLED`
Switch for enabling the bus synchronisation

AUTOSAR name: FrNmBusSynchronizationEnabled
{FRNM_BUS_SYNCHRONIZATION_ENABLED}

`ControllerId`
Controller ID, starting with 0 (0=FLEXRAY1, 1=FLEXRAY2, e.t.c.)

`EcuId`
ECU ID, starting with 0

`NmNodeId`
NM node idenfifier that is used as source node identifier (SNI)

AUTOSAR name: FrNmNodeId {FRNM_NODE_ID}

`PduCbvPosition`
Control bit vector position within PDU

Used if flag `G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__CONTROL_BIT_VECTOR_ENABLED` is set.

The following values are possible:

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__PDU_CBV_POSITION__BYTE_0`
Control bit vector is in PDU byte #0 (AUTOSAR default)

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__PDU_CBV_POSITION__NOT_USED`
The control bit vector is not used.

`PduSniPosition`
Source node identifier position within PDU

Used if flag `G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__SOURCE_NODE_IDENTIFIER_ENABLED` is set.

The following values are possible:

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__PDU_SNI_POSITION__BYTE_0`
Source node identifier is in PDU byte #0

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__PDU_SNI_POSITION__BYTE_1`
Source node identifier is in PDU byte #1 (AUTOSAR default)

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__PDU_SNI_POSITION__NOT_USED`
Source node identifier is not used.

`PduNmsPosition`
NM state position within PDU

Used if flag `G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__USE_NMS_IN_PDU` is set.

The following values are possible:

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__PDU_NMS_POSITION__BYTE_0`
NM state position is in PDU byte #0

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__PDU_NMS_POSITION__BYTE_1`
NM state position is in PDU byte #1

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__PDU_NMS_POSITION__BYTE_2`
NM state position is in PDU byte #2 (NMH)

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__PDU_NMS_POSITION__NOT_USED`
NM state position is not used (AUTOSAR default)

The state has following PDU coding:

0x01 : NETWORK__REPEAT_MESSAGE coming from SYNCHRONIZE (BUS_SLEEP)

0x04 : NETWORK__NORMAL_OPERATION coming from NETWORK__REPEAT_MESSAGE

0x08 : NETWORK__NORMAL_OPERATION coming from NETWORK__READY_SLEEP

0x10 : NETWORK__REPEAT_MSG coming from NETWORK__NORMAL_OPERATION

0x20 : NETWORK__REPEAT_MSG coming from NETWORK__READY_SLEEP

`PduLength`
Number of NM-Data PDU data bytes

AUTOSAR name: FrNmPduLength

`UserDataOffset`
Offset within NM-Data PDU where user data begins

`PduData[8]`
NM-Data PDU data bytes

`UserDataMask[8]`
Bit mask that specifies which bits of user data are copied to NM-Data PDU (mask bit set = copy, mask bit cleared = do not copy)

User data is copied to NmDataPdu in the following way:

```
for (i = 0; i < numberOfUserDataBytes; i++) {
 NmDataPdu[UserDataOffset + i] &= ~UserDataMask[i];
 NmDataPdu[UserDataOffset + i] |= UserData[i] & UserDataMask[i];
}
```

`ReadySleepCnt`
Numbers of repetitions in the ready sleep state before NM switches to bus sleep mode.

On a value of " **1** ", the NM-State Machine will leave the Ready Sleep State after one NM Repetition Cycle with no " **keep awake** " votes.

AUTOSAR name: FrNmReadySleepCnt {FRNM_READY_SLEEP_CNT}

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`MsgTimeoutTime`
Timeout of a NM-message [microseconds]

It determines how long the NM shall wait with notification of transmission failure (Nm_TxTimeoutException) while communication errors occur on the bus.

Nm_TxTimeoutException is available, if the value of `MsgTimeoutTime` is configured as greater than 0.

AUTOSAR name: FrNmMsgTimeoutTime {FRNM_MSG_TIMEOUT_TIME}

`RepeatMessageTime`
Timeout for Repeat Message State [microseconds]

Defines the time how long the NM shall stay in the Repeat Message State. The value "**0**" denotes that no Repeat Message State is configured, which means that Repeat Message State is transient and implies that it is left immediately after entry and consequently no startup stability is guaranteed and no node detection procedure is possible.

AUTOSAR name: FrNmRepeatMessageTime {FRNM_REPEAT_MESSAGE_TIME}

`RemoteSleepIndTime`
Timeout for Remote Sleep Indication [microseconds]

It defines the time how long it shall take to recognize that all other nodes are ready to sleep. The value "**0**" denotes that no Remote Sleep Indication functionality is configured.

AUTOSAR name: FrNmRemoteSleepIndTime {FRNM_REMOTE_SLEEP_IND_TIME}

`TxUserDataPduId`
Relevant if flag `G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__COM_USER_DATA_SUPPORT` is set

The FrNm collects the NM User Data from the referenced NM I-PDU by calling PduRouter_FlexRayNm_TriggerTransmit and combines the user data with the further NM bytes each time before it requests the transmission of the corresponding NM message.

AUTOSAR name: FrNmTxUserDataPduId {FRNM_TX_USER_DATA_PDU_ID}

`VotingCycle`
Number of FlexRay Schedule Cycles needed to transmit the Nm vote of all ECUs on the FlexRay bus.

1, 2, 4, 8, 16, 32 or 64

AUTOSAR name: FrNmVotingCycle {FRNM_VOTING_CYCLE}

`DataCycle`
Number of FlexRay Schedule Cycles needed to transmit the NM Data of all ECUs on the FlexRay bus.

1, 2, 4, 8, 16, 32 or 64

AUTOSAR name: FrNmDataCycle {FRNM_DATA_CYCLE}

`RepetitionCycle`
Number of Flexray Schedule Cycles used to repeat the transmission of the Nm vote of all ECUs on the Flexray bus.

1, 2, 4, 8, 16, 32 or 64

AUTOSAR name: FrNmRepetitionCycle {FRNM_REPETITION_CYCLE}

`CarWakeUpBitPosition`
Specifies the Bit position of the CWU bit within the NM-Message.

This parameter is only relevant if flag `G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__CAR_WAKE_UP_RX_ENABLED` is set.

AUTOSAR name: FrNmCarWakeUpBitPosition {FRNM_CAR_WAKE_UP_BIT_POSITION}

`CarWakeUpBytePosition`
Specifies the Byte position of the CWU bit within the NM-Message.

This parameter is only relevant if flag `G_NET2RUN__FLEXRAY_NM__AUTOSAR__ECU__FLAG__CAR_WAKE_UP_RX_ENABLED` is set.

AUTOSAR name: FrNmCarWakeUpBytePosition {FRNM_CAR_WAKE_UP_BYTE_POSITION}

`CarWakeUpFilterNodeId`
Source node identifier for CarWakeUp filtering. If CarWakeUp filtering is supported, only the CarWakeUp bit within the NM message with source node identifier `CarWakeUpFilterNodeId` is considered as CarWakeUp request.

`PnInfoLength`
Specifies the length of the PN request information in the NM message.

AUTOSAR name: FrNmPnInfoLength {FRNM_PN_INFO_LENGTH}

`PnInfoOffset`
Specifies the offset of the PN request information in the NM message.

AUTOSAR name: FrNmPnInfoOffset {FRNM_PN_INFO_OFFSET}

`PnFilterMask[8]`
PN filter mask with a size of `PnInfoLength` bytes

This parameter defines which PN request information is relevant for the ECU and which not. Each bit has the following meaning:

0 -> The PN request information is irrelevant for the ECU.

1 -> The PN request information is relevant for the ECU.

see AUTOSAR parameter "FrNmPnFilterMaskByte"

`PnResetTime`
Specifies the runtime of the reset timer in microseconds. This reset time is valid for the reset of PN requests in the EIRA and in the ERA.

AUTOSAR name: FrNmPnResetTime {FRNM_PN_RESET_TIME}

`PnEiraRxPduId`
If the content of EIRA changes (any bit changes from 1 to 0 or from 0 to 1) because of a received or transmitted NM PDU or an EIRA reset timer expiration, FrNm informs the upper layers by calling PduRouter_FlexRayNm_RxIndication(). By means of the RX Indication function the EIRA data shall be provided to the COM module.

See AUTOSAR parameter FrNmPnEiraRxNSduRef {FRNM_PN_EIRA_RX_NSDU_REF} and AUTOSAR requirement [FRNM429]

`PnEraRxPduId`
If the content of ERA changes (any bit changes from 1 to 0 or from 0 to 1) because of a received NM PDU or an ERA reset timer expiration, FrNm informs the upper layers by calling

PduRouter_FlexRayNm_RxIndication(). By means of the RX Indication function the ERA data shall be provided to the COM module.

See AUTOSAR parameter FrNmPnEraRxNSduRef {FRNM_PN_ERA_RX_NSDU_REF} and AUTOSAR requirement [FRNM438]

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_FlexRayNm_Autosar_RxPdu

G_Net2Run_FlexRayNm_Autosar_RxPdu — FlexRay NM RX PDU configuration

Description

Use this command for FlexRay NM RX PDU configuration.

Definition

```
G_Error_t
G_Net2Run_FlexRayNm_Autosar_RxPdu(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_FlexRayNm_Autosar_RxPdu_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__RX_PDU__FLAG__NONE`
No flag is set

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__RX_PDU__FLAG__PDU_CONTAINS_DATA`
0 = PDU doesn't contain NM Data

1 = PDU contains NM Data

AUTOSAR name: FrNmRxPduContainsData {FRNM_RX_PDU_CONTAINS_DATA}

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__RX_PDU__FLAG__PDU_CONTAINS_VOTE`
0 = PDU doesn't contain NM Vote

1 = PDU contains NM Vote

AUTOSAR name: FrNmRxPduContainsVote {FRNM_RX_PDU_CONTAINS_VOTE}

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__RX_PDU__FLAG__PDU_CONTAINS_CBV`
If flag `G_NET2RUN__FLEXRAY_NM__AUTOSAR__RX_PDU__FLAG__PDU_CONTAINS_DATA` is set, this flag is ignored

If flag `G_NET2RUN__FLEXRAY_NM__AUTOSAR__RX_PDU__FLAG__PDU_CONTAINS_DATA` is not set, this flag has the the following meaning:

0 = PDU doesn't contain a control bit vector

1 = PDU contains a control bit vector

G_NET2RUN__FLEXRAY_NM__AUTOSAR__RX_PDU__FLAG__VOTE_BY_PRESENCE_OF_PDU
0 = the vote bit defines negative (0) or positive (1) vote

1 = reception of PDU is a positive vote (vote bit is ignored)

`ControllerId`
Controller ID, starting with 0 (0=FLEXRAY1, 1=FLEXRAY2, e.t.c.)

`EcuId`
ECU ID, starting with 0

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`FrPduId`
PDU ID configured with G_Net2Run_FlexRayIf_Pdu_Config

AUTOSAR name: FrNmRxPduId {FRNM_RX_PDU_ID}

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_FlexRayNm_Autosar_TxPdu

G_Net2Run_FlexRayNm_Autosar_TxPdu — FlexRay NM TX PDU configuration

Description

Use this command for FlexRay NM TX PDU configuration.

Definition

```
G_Error_t
G_Net2Run_FlexRayNm_Autosar_TxPdu(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_FlexRayNm_Autosar_TxPdu_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__TX_PDU__FLAG__NONE`
No flag is set

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__TX_PDU__FLAG__PDU_CONTAINS_DATA`
0 = PDU doesn't contain NM Data

1 = PDU contains NM Data

AUTOSAR name: FrNmTxPduContainsData {FRNM_TX_PDU_CONTAINS_DATA}

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__TX_PDU__FLAG__PDU_CONTAINS_VOTE`
0 = PDU doesn't contain NM Vote

1 = PDU contains NM Vote (positive vote = vote bit is set, negative vote = vote bit is cleared)

AUTOSAR name: FrNmTxPduContainsVote {FRNM_TX_PDU_CONTAINS_VOTE}

`G_NET2RUN__FLEXRAY_NM__AUTOSAR__TX_PDU__FLAG__PDU_CONTAINS_CBV`
If flag `G_NET2RUN__FLEXRAY_NM__AUTOSAR__TX_PDU__FLAG__PDU_CONTAINS_DATA` is set, this flag is ignored

If flag `G_NET2RUN__FLEXRAY_NM__AUTOSAR__TX_PDU__FLAG__PDU_CONTAINS_DATA` is not set, this flag has the the following meaning:

0 = PDU doesn't contain a control bit vector

1 = PDU contains a control bit vector

G_NET2RUN__FLEXRAY_NM__AUTOSAR__TX_PDU__FLAG__VOTE_BY_PRESENCE_OF_PDU

0 = PDU is always transmitted

1 = PDU is transmitted only on positive vote

`ControllerId`
Controller ID, starting with 0 (0=FLEXRAY1, 1=FLEXRAY2, e.t.c.)

`EcuId`
ECU ID, starting with 0

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`FrPduId`
PDU ID configured with G_Net2Run_FlexRayIf_Pdu_Config

AUTOSAR name: FrNmTxPduId {FRNM_TX_PDU_ID}

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

12 Freshness Value Manager (FVM)

Configuration functions for the **F**reshness **V**alue **M**anager module of **Net2Run**.

The Software Component Freshness Value Manager (FVM) shall provide the Freshness Value (FV) to Secure Onboard Communication (SecOC).

G_Net2Run_Fvm_FreshnessValue_Custom1

G_Net2Run_Fvm_FreshnessValue_Custom1 — Configure freshness value as a timestamp and broadcast it periodically and authenticated over two PDUs.

Description

Obsolete since June 2018. Replaced by `G_Net2Run_Fvm_FreshnessValue_Custom3...` commands.

Use this command for configuration of a freshness value as a timestamp and broadcast it periodically and authenticated over two PDUs.

[DataId | StatusByte | Timestamp] and [Authenticator]

Definition

```
G_Error_t
G_Net2Run_Fvm_FreshnessValue_Custom1(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Fvm_FreshnessValue_Custom1_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__FVM__FRESHNESS_VALUE__CUSTOM_1__CMD_FLAG__NONE`
No flag is set

`FreshnessValueId`
Freshness Value ID

`CsmJobId`
Crypto Service Manager Job ID

This refers to a CSM job and is used for authentication.

`TimeIncrementPeriod`
Time increment period, in us

Time after which the Freshness Value is incremented (timestamp resolution).

`MasterTickBroadcastPeriod`
Time after which master tick is broadcasted, in us

`MasterTickBroadcastPeriodOnStartup`
Master tick broadcast time on startup, in us

`MasterTickBroadcastCountOnStartup`
Count of broadcasts with `MasterTickBroadcastPeriodOnStartup` as period

`DataId`
This is used for calculating the authenticator

`StatusByte`
This is the 1st byte of the master tick broadcast

`MasterTickLength`
Length of master tick broadcasted, in byte (**0** .. **7**)

`AuthenticatorLength`
Length of authenticator broadcasted with master tick, in byte (**0** .. **8**)

`reserved1`
reserved parameter (must be initialized with **0**)

`TxPduId_MasterTick`
IPDU that contains `StatusByte` and `MasterTick`

`BroadcastTpMessagePartDelay`
Time delay between message parts for broadcast transport protocol

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Fvm_Custom2_TimeServerEcu

G_Net2Run_Fvm_Custom2_TimeServerEcu — Use this for configuring the Custom2_TimeServer_Ecu.

Description

The time server periodically sends the current freshness unauthenticated. To receive challenges from Custom2_Participant_Ecus the time server ECU requires a Custom2_TimeServer_Response linked to it for each individual Custom2_Participant_Ecu. Additionally Custom2_AuthBroadcast Fvms can be linked to the time server and provide freshness as a time stamp for SecOC. Notice that one Custom2_AuthBroadcast Fvm is required for every individual connection. The Custom2_TimeServer_Ecu must not provide freshness for secOC.

Definition

```
G_Error_t
G_Net2Run_Fvm_Custom2_TimeServerEcu(
  const G_PortHandle_t  portHandle,
  const G_Net2Run_Fvm_Custom2_TimeServerEcu_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__FVM__CUSTOM_2__TIME_SERVER_ECU__FLAG__ZERO`
No flag is set

`FreshnessValueId`
Freshness Value ID

`TimeIncrementPeriod`
Time increment period, in us

Time after which the Freshness Value is incremented (timestamp resolution).

`TimeSendPeriod`
Amount of ime increments, after which the current timestamp if broadcasted unauthenticated.

`TxPduId_UnauthenticatedTime`
This PDU contains the unauthenticated timestamp and is broadcasted after `TimeSendPeriod` time increments.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Fvm_Custom2_PartEcu

G_Net2Run_Fvm_Custom2_PartEcu — Use this for configuring a Custom2_Participant_Ecu.

Description

The Custom2_Participant_Ecu receives the unauthenticated time sent by the Custom2_TimeServer_Ecu and calculates the jitter to its own time. If the Custom2_Participant_Ecu des not have a valid time or the calculated jitter exceeds timeJitterMax a challenge is sent via a connected Custom2_P_Challenge Fvm. The response from the Custom2_Ts_Reponse Fvm will be verified by SecOC and the authenticated time accepted as the new time of the Custom2_Participant_Ecu. Additionally Custom2_AuthBroadcast Fvms can be linked to the Custom2_Participant_Ecu and provide freshness as a time stamp for SecOC. Notice that one Custom2_AuthBroadcast Fvm is required for every individual connection. The Custom2_Participant_Ecu must not provide freshness for secOC.

Definition

```
G_Error_t
G_Net2Run_Fvm_Custom2_PartEcu(
  const G_PortHandle_t  portHandle,
  const G_Net2Run_Fvm_Custom2_PartEcu_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__FVM__CUSTOM_2__PART_ECU__FLAG__ZERO`
No flag is set

`FreshnessValueId`
Freshness Value ID

`TimeIncrementPeriod`
Time increment period, in us

Time after which the Freshness Value is incremented (timestamp resolution).

`TimeJitterMax`
Maximum allowed time jitter. If exceeded a new authenticated timestamp is requested.

`TimeRequestTimeout`
Time the participant ecu will wait for an authenticated time after sending a challenge to the time server.

`RxPduId_AuthenticatedTime`
This PDU contains the authenticated time from the time server. It must be verified by SecOC beforehand. The freshness of this participant ecu is set to the received value.

`RxPduId_UnauthenticatedTime`
: This PDU contains the unauthenticated time from the time server. On reception the participant ecu compares its own time with the received time. If the jitter exceeds `TimeJitterMax`, a new authenticated time is requested.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Fvm_Custom2_AuthBroadcast

G_Net2Run_Fvm_Custom2_AuthBroadcast — This provides freshness as a time stamp.

Description

The Custom2_AuthBroadcast Freshness Value Manager provides freshness as a timestamp. It requires a corresponding Custom2_TimeServer_Ecu or a Custom2_Participant_Ecu to provide the correct and up to date timestamp. If no time is available the Custom2_AuthBroadcast Freshness Value Manager returns a fixed freshness. The AuthBroadcast Fvm maintains a ringcounter to protect the receiver from replay attacks. Therefor on reception the GetRxFreshnessAuthData function must be called.

Definition

```
G_Error_t
G_Net2Run_Fvm_Custom2_AuthBroadcast(
  const G_PortHandle_t  portHandle,
  const G_Net2Run_Fvm_Custom2_AuthBroadcast_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__FVM__CUSTOM_2__AUTH_BROADCAST__FLAG__ZERO`
No flag is set

`FreshnessValueId`
Freshness Value ID

`StartUpTime`
Startup Time , in us

Time period in which a fixed timestamp is used for mac verification of this communication channel.

`ECU_FreshnessValueId`
`Freshness Value ID` of the corresponding ECU Freshness Value Manager.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Fvm_Custom2_TimeServerEcuResponse

G_Net2Run_Fvm_Custom2_TimeServerEcuResponse — This provides freshness as a challenge, for TxFreshness.

Description

The Custom2_Ts_Response Fvm receives a PDU from a Custom2_P_Challenge and stores the received challenge, contained in the PDU. On a following request from SecOC for GetTxFrehness it provides the received challenge for authentication. The Custom2_Ts_Response requires a Custom2_TimeServer_Ecu. For each Custom2_Participant_Ecu a Custom2_Ts_Response must be configured at the Custom2_TimeServer_Ecu.

Definition

```
G_Error_t
G_Net2Run_Fvm_Custom2_TimeServerEcuResponse(
  const G_PortHandle_t  portHandle,
  const G_Net2Run_Fvm_Custom2_TimeServerEcuResponse_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__FVM__CUSTOM_2__TIME_SERVER_ECU_RESPONSE__FLAG__ZERO`
No flag is set

`FreshnessValueId`
Freshness Value ID

`ECU_FreshnessValueId`
`Freshness Value ID` of the corresponding Custom2_TimeServer_Ecu Freshness Value Manager.

`RxPduId_Challenge`
This PDU contains the incoming Challenge, sent by a Custom2_PartEcuChallenge Freshness Value Manager.

`TxPduId_Response`
This PDU contains the outgoing authenticated timestamp provided by the Custom2_TimeServer_Ecu. It must be forwareded via SecOC to become a secured authenticated time message.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Fvm_Custom2_PartEcuChallenge

G_Net2Run_Fvm_Custom2_PartEcuChallenge — This provides freshness as a challenge. It is triggered by a Custom2_Participant_Ecu to request an authenticated time.

Description

To configure a Custom2_P_Challenge a Custom2_Participant_Ecu is necessary. Each Custom2_Participant_Ecu requires one Custom2_P_Challenge Fvm. A freshness value is just available for reception and only if the Custom2_Participant_Ecu requested to send a challenge beforehand. On reception of an authenticated time message, the sent challenge is used in SecOC for verification of the message.

Definition

```
G_Error_t
G_Net2Run_Fvm_Custom2_PartEcuChallenge(
  const G_PortHandle_t  portHandle,
  const G_Net2Run_Fvm_Custom2_PartEcuChallenge_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__FVM__CUSTOM_2__PART_ECU_CHALLENGE__FLAG__ZERO`
No flag is set

`FreshnessValueId`
Freshness Value ID

`ECU_FreshnessValueId`
`Freshness Value ID` of the corresponding Custom2_Participant_Ecu Freshness Value Manager.

`CsmJobId`
Reference to a crypto service manager that is capable of providing random numbers in a size up to 64bits.

`TxPduId_Challenge`
This PDU contains the generated random number that is used as the challenge in the challenge-response-protocol to provide a secured authenticated time for the participant ECU. On reception the SecOC-Module requests this challenge via the GetRxFrehness-Function to verify the authenticity of the received time message.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Fvm_FreshnessValue_Custom3_Master

G_Net2Run_Fvm_FreshnessValue_Custom3_Master — Configure the freshness value manager master.

Description

Use this command for configuration of a freshness value manager master of the type custom 3. It provides an increasing monotonic counter that is transmitted periodically.

Definition

```
G_Error_t
G_Net2Run_Fvm_FreshnessValue_Custom3_Master(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Fvm_FreshnessValue_Custom3_Master_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__FVM__FRESHNESS_VALUE__CUSTOM_3__MASTER_CMD_FLAG__NONE`
No flag is set

`FreshnessValueId`
Freshness Value ID

`TimeIncrementPeriod`
Time increment period, in us

Time after which the Freshness Value is incremented (timestamp resolution).

`MasterTickBroadcastPeriod`
Time after which master tick is broadcasted, in us

`MasterTickBroadcastPeriodOnStartup`
Master tick broadcast time on startup, in us

`MasterTickBroadcastCountOnStartup`
Count of broadcasts with `MasterTickBroadcastPeriodOnStartup` as period

`StatusByte`
Describes the synchronization status of this master to the Vehicle security master.

`MasterTickLength`
Length of master tick broadcasted, in bytes (`0` .. `8`)

`TxPduId_MasterTick`
IPDU that contains `StatusByte` and `MasterTick`

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Fvm_FreshnessValue_Custom3_Slave

G_Net2Run_Fvm_FreshnessValue_Custom3_Slave — Configure the freshness value manager slave.

Description

Use this command for configuration of a freshness value manager slave of the type custom 3. It provides an increasing monotonic counter and receives the master tick from a **G_Net2Run_Fvm_FreshnessValue_Custom3_Master** .

Definition

```
G_Error_t
G_Net2Run_Fvm_FreshnessValue_Custom3_Slave(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Fvm_FreshnessValue_Custom3_Slave_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__FVM__FRESHNESS_VALUE__CUSTOM_3__SLAVE_CMD_FLAG__NONE`
No flag is set

`FreshnessValueId`
Freshness Value ID

`TimeIncrementPeriod`
Time increment period, in us

Time after which the Freshness Value is incremented (timestamp resolution).

`MasterTickLength`
Length of master tick received, in bytes (**0** .. **8**)

`DecreaseAcceptanceTolerance`
Decrease Acceptance Tolerance in percent

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`RxPduId_MasterTick`
IPDU contains StatusByte and MasterTick. Received from a **G_Net2Run_Fvm_FreshnessValue_Custom3_Master** .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Fvm_FreshnessValue_Custom4_Master

G_Net2Run_Fvm_FreshnessValue_Custom4_Master — Configure the freshness value manager master.

Description

Use this command for configuration of a freshness value manager master of the type custom 4. It provides an increasing monotonic counter that is transmitted periodically.

Definition

```
G_Error_t
G_Net2Run_Fvm_FreshnessValue_Custom4_Master(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Fvm_FreshnessValue_Custom4_Master_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__FVM__FRESHNESS_VALUE__CUSTOM_4__MASTER_CMD_FLAG__NONE`
No flag is set

`FreshnessValueId`
Freshness Value ID

`ResetCycleIncrementPeriod`
Time increment period, in us

Time period to increment the resetCounter and reset the message counter.

`ResetCycleSendPeriod`
Time period to send the TripCounter and ResetCounter, in us

`TripCounterLength`
Length of the Trip Counter in bits

`ResetCounterLength`
Length of the Reset Counter in bits.

`MessageCounterLength`
Length of the Message Counter in bits.

`ResetFlagLength`
Length of the Reset Flag in bits

`reserved`
reserved parameter (must be initialized with `0`)

`TxPduId_TripResetSyncMsg`
IPDU that contains `TripCounter` and `ResetCounter`

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Fvm_FreshnessValue_Custom4_Slave

G_Net2Run_Fvm_FreshnessValue_Custom4_Slave — Configure the freshness value manager slave.

Description

Use this command for configuration of a freshness value manager slave of the type custom 4. It provides an increasing monotonic counter and receives the master tick from a **G_Net2Run_Fvm_FreshnessValue_Custom4_Master** .

Definition

```
G_Error_t
G_Net2Run_Fvm_FreshnessValue_Custom4_Slave(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Fvm_FreshnessValue_Custom_Slave_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__FVM__FRESHNESS_VALUE__CUSTOM_4__SLAVE_CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__FVM__FRESHNESS_VALUE__CUSTOM_4__SLAVE_CMD_FLAG__SENDER`
Set this flag to configure the **G_Net2Run_Fvm_FreshnessValue_Custom4_Slave** as a sender.

If this flag is not set, the **G_Net2Run_Fvm_FreshnessValue_Custom4_Slave** is configured as a receiver.

On each side of the communication channel a sender and a receiver is required for bi-directional communication.

`G_NET2RUN__FVM__FRESHNESS_VALUE__CUSTOM_4__SLAVE_CMD_FLAG__INTERN_SYNC_MSG`
Set this flag if the **G_Net2Run_Fvm_FreshnessValue_Custom4_Master** is running on the same Net2Run interface.

If set, field `Internal_FvmMasterId` is required.

The Fvm Master must be configured beforehand.

This setting is necessary to provide synchronization between the local master and the local slave.

`FreshnessValueId`
Freshness Value ID

`TripCounterLength`
Length of the Trip Counter in bits.

`ResetCounterLength`
Length of the Reset Counter in bits.

`MessageCounterLength`
Length of the Message Counter in bits.

`ResetFlagLength`
Length of the Reset Flag in bits.

`reserved`
reserved parameter (must be initialized with **0**)

`RxPduId_TripResetSyncMsg`
IPDU contains TripCounter and ResetCounter. Received from a **G_Net2Run_Fvm_FreshnessValue_Custom4_Master** .

`ClearAcceptanceWindow`
Size of the ClearAcceptanceWindow.

`Internal_FvmMasterId`
ID of internal freshness value manager master just considered if flag `SyncMsgFromInternalMaster` is set.

Required if the Fvm Custom 4 Master is implemented on the same interface.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Fvm_StartCommunication

G_Net2Run_Fvm_StartCommunication — Start Freshness Value Manager communication

Description

Use this command to start Freshness Value Manager communication.

Definition

```
G_Error_t
G_Net2Run_Fvm_StartCommunication(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Fvm_StartCommunication_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__FVM__START_COMMUNICATION__CMD_FLAG__NONE`
No flag is set

`FreshnessValueId`
Freshness value ID

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Fvm_StopCommunication

G_Net2Run_Fvm_StopCommunication — Stop Freshness Value Manager communication

Description

Use this command to stop Freshness Value Manager communication.

Definition

```
G_Error_t
G_Net2Run_Fvm_StopCommunication(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Fvm_StopCommunication_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__FVM__STOP_COMMUNICATION__CMD_FLAG__NONE`
No flag is set

`FreshnessValueId`
Freshness value ID

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Fvm_GetFreshness

G_Net2Run_Fvm_GetFreshness — Read freshness value

Description

Use this command to read the freshness value.

Definition

```
G_Error_t
G_Net2Run_Fvm_GetFreshness(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Fvm_GetFreshness_Cmd_t * const cmd,
 u64_t * const value
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__FVM__GET_FRESHNESS__CMD_FLAG__NONE`
No flag is set

`FreshnessValueId`
ID of the freshness value to be read

`value`
Returns freshness value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Fvm_SetFreshness

G_Net2Run_Fvm_SetFreshness — Set freshness value

Description

Use this command to set the freshness value.

Definition

```
G_Error_t
G_Net2Run_Fvm_SetFreshness(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Fvm_SetFreshness_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__FVM__SET_FRESHNESS__CMD_FLAG__NONE`
No flag is set

`FreshnessValueId`
ID of the freshness value to be set

`Value`
Freshness value to be set

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

13 Network Management Control

Functions for controlling the network management of a Net2Run **Rest Bus Simulation**

G_Net2Run_Nm_State_Get

G_Net2Run_Nm_State_Get — Get network management state

Description

Use this command to query the network management state.

Definition

```
G_Error_t
G_Net2Run_Nm_State_Get(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Nm_State_Get_Cmd_t * const cmd,
 G_Net2Run_Nm_State_Get_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__NM__STATE__GET__CMD_FLAG__NONE`
No flag is set

`BusType`
Bus type

The following values are possible:

`G_NET2RUN__BUS_TYPE__CAN`
CAN bus

`G_NET2RUN__BUS_TYPE__LIN`
LIN bus

`G_NET2RUN__BUS_TYPE__FLEXRAY`
FlexRay bus

`ControllerId`
Controller ID, starting with 0 (e.g. 0=CAN1, 1=CAN2, e.t.c.)

`EcuId`
ECU ID, starting with 0

`reserved`
reserved parameter (must be initialized with **0**)

`rsp`
Returns response parameters

The structure contains the following elements:

`State`
Network management state

The following values are possible:

`G_NET2RUN__NM__STATE__NOT_INITIALIZED`
Network management not initialized

`G_NET2RUN__NM__STATE__BUS_SLEEP`
Network management in "Bus Sleep" mode

`G_NET2RUN__NM__STATE__PREPARE_BUS_SLEEP`
Network management in "Prepare Bus Sleep" mode

`G_NET2RUN__NM__STATE__NETWORK__READY_SLEEP`
Network management in "Ready Sleep" mode

`G_NET2RUN__NM__STATE__NETWORK__NORMAL_OPERATION`
Network management in "Normal Operation" mode

`G_NET2RUN__NM__STATE__NETWORK__REPEAT_MESSAGE`
Network management in "Repeat Message" mode

`G_NET2RUN__NM__STATE__SYNCHRONIZE`
Network management in "Synchronize" mode

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Nm_RepeatMsgRequest

G_Net2Run_Nm_RepeatMsgRequest — Set Repeat Message Request Bit for NM messages transmitted next on the bus

Description

Use this command to set the Repeat Message Request Bit for NM messages transmitted next on the bus.

Definition

```
G_Error_t
G_Net2Run_Nm_RepeatMsgRequest(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Nm_RepeatMsgRequest_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__NM__REPEAT_MSG_REQUEST__CMD_FLAG__NONE`
No flag is set

`BusType`
Bus type

The following values are possible:

`G_NET2RUN__BUS_TYPE__CAN`
CAN bus

`G_NET2RUN__BUS_TYPE__LIN`
LIN bus

`G_NET2RUN__BUS_TYPE__FLEXRAY`
FlexRay bus

`ControllerId`
Controller ID, starting with 0 (e.g. 0=CAN1, 1=CAN2, e.t.c.)

`EcuId`
ECU ID, starting with 0

`reserved`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Nm_UserData_Local_Set

G_Net2Run_Nm_UserData_Local_Set — Set user data for NM messages transmitted next on the bus

Description

Use this command to set user data for NM messages transmitted next on the bus.

Definition

```
G_Error_t
G_Net2Run_Nm_UserData_Local_Set(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Nm_UserData_Local_Set_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__NM__USER_DATA__LOCAL__SET__CMD_FLAG__NONE`
No flag is set

`BusType`
Bus type

The following values are possible:

`G_NET2RUN__BUS_TYPE__CAN`
CAN bus

`G_NET2RUN__BUS_TYPE__LIN`
LIN bus

`G_NET2RUN__BUS_TYPE__FLEXRAY`
FlexRay bus

`ControllerId`
Controller ID, starting with 0 (e.g. 0=CAN1, 1=CAN2, e.t.c.)

`EcuId`
ECU ID, starting with 0

`reserved1`
reserved parameter (must be initialized with **0**)

`Length`
Length of user data in bytes

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`Data`
User data bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Nm_UserData_Remote_Get

G_Net2Run_Nm_UserData_Remote_Get — Get user data out of the last successfully received NM message

Description

Use this command to get user data out of the last successfully received NM message.

Definition

```
G_Error_t
G_Net2Run_Nm_UserData_Remote_Get(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Nm_UserData_Remote_Get_Cmd_t * const cmd,
 G_Net2Run_Nm_UserData_Remote_Get_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__NM__USER_DATA__REMOTE__GET__CMD_FLAG__NONE`
No flag is set

`BusType`
Bus type

The following values are possible:

`G_NET2RUN__BUS_TYPE__CAN`
CAN bus

`G_NET2RUN__BUS_TYPE__LIN`
LIN bus

`G_NET2RUN__BUS_TYPE__FLEXRAY`
FlexRay bus

`ControllerId`
Controller ID, starting with 0 (e.g. 0=CAN1, 1=CAN2, e.t.c.)

`EcuId`
ECU ID, starting with 0

`reserved`
reserved parameter (must be initialized with `0`)

`rsp`
Returns response parameters

The structure contains the following elements:

`Length`
Data length in bytes

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Data`
Returns user data bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Nm_Communication_Disable

G_Net2Run_Nm_Communication_Disable — Disables the NM PDU transmission ability

Description

Use this command to disable the NM PDU transmission ability.

Definition

```
G_Error_t
G_Net2Run_Nm_Communication_Disable(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Nm_Communication_Disable_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__NM__COMMUNICATION__DISABLE__CMD_FLAG__NONE`
No flag is set

`BusType`
Bus type

The following values are possible:

`G_NET2RUN__BUS_TYPE__CAN`
CAN bus

`G_NET2RUN__BUS_TYPE__LIN`
LIN bus

`G_NET2RUN__BUS_TYPE__FLEXRAY`
FlexRay bus

`ControllerId`
Controller ID, starting with 0 (e.g. 0=CAN1, 1=CAN2, e.t.c.)

`EcuId`
ECU ID, starting with 0

`reserved`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Nm_Communication_Enable

G_Net2Run_Nm_Communication_Enable — Enables the NM PDU transmission ability

Description

Use this command to enable the NM PDU transmission ability.

Definition

```
G_Error_t
G_Net2Run_Nm_Communication_Disable(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Nm_Communication_Disable_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__NM__COMMUNICATION__ENABLE__CMD_FLAG__NONE`
No flag is set

`BusType`
Bus type

The following values are possible:

`G_NET2RUN__BUS_TYPE__CAN`
CAN bus

`G_NET2RUN__BUS_TYPE__LIN`
LIN bus

`G_NET2RUN__BUS_TYPE__FLEXRAY`
FlexRay bus

`ControllerId`
Controller ID, starting with 0 (e.g. 0=CAN1, 1=CAN2, e.t.c.)

`EcuId`
ECU ID, starting with 0

`reserved`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

14 PDU

Functions for controlling PDUs of a Net2Run **Rest Bus Simulation**

G_Net2Run_Pdu_PduGroupControl

G_Net2Run_Pdu_PduGroupControl — Control PDU groups

Description

Use this function to control PDU groups of a Net2Run **Rest Bus Simulation .**

For an example, see Read / Write Example.

Definition

```
G_Error_t
G_Net2Run_Pdu_PduGroupControl(
 const G_PortHandle_t  portHandle,
 const u32_t  numberOfGroups,
 const G_Net2Run_Pdu_PduGroupControl_Group_t * const groups
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfGroups`
Number of groups in buffer `groups`

`groups`
Array with groups to be controlled

The group structure consists of the following elements:

`PduGroupId`
Group identifier of the PDU group

All PDU Groups: `0xFFFFFFFF`

`Flags`
PDU group flags

The following values are possible and can be combined:

`G_NET2RUN__PDU__PDU_GROUP_CONTROL__GROUP__FLAG__NONE`
No flag is set

`G_NET2RUN__PDU__PDU_GROUP_CONTROL__GROUP__FLAG__START`
All PDUs in this group will be started

`G_NET2RUN__PDU__PDU_GROUP_CONTROL__GROUP__FLAG__STOP`
All PDUs in this group will be stopped

`G_NET2RUN__PDU__PDU_GROUP_CONTROL__GROUP__FLAG__ENABLE_RX_DEADLINE`
Enable RX Deadline Monitoring

This flag is not supported, yet.

`G_NET2RUN__PDU__PDU_GROUP_CONTROL__GROUP__FLAG__DISABLE_RX_DEAD-LINE`
Disable RX Deadline Monitoring

This flag is not supported, yet.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Pdu_NumberOfPdus_Get

G_Net2Run_Pdu_NumberOfPdus_Get — Return number of configured PDUs

Description

Use this command to query the number of configured PDUs.

Definition

```
G_Error_t
G_Net2Run_Pdu_NumberOfPdus_Get(
 const G_PortHandle_t  portHandle,
 u32_t * const numberOfPdus
);
```

Parameters

`portHandle`
: Handle to the communication port

`numberOfPdus`
: Returns the number of configured PDUs

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Pdu_PduList_Get

G_Net2Run_Pdu_PduList_Get — Return array with PDU info for all PDUs

Description

Use this command to return an array with PDU information for all configured PDUs.

Definition

```
G_Error_t
G_Net2Run_Pdu_PduList_Get(
 const G_PortHandle_t  portHandle,
 u32_t * const numberOfPdus,
 G_Net2Run_PduInfo_t * const pdus
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfPdus`
in : size of the buffer `pdus` in number of PDUs

out : number of returned PDU info in `pdus`

Use G_Net2Run_Pdu_NumberOfPdus_Get to query the number of available PDUs.

`pdus`
Returns array with PDU information

The structure contains the following elements:

`PduName`
Name of the PDU

`EcuName`
Name of the ECU this PDU instance is configured for

`ControllerId`
ID of the ECU controller this PDU instance is configured for

`ComId`
Communication layer ID of this PDU instance

This ID can be used with `G_Net2Run_Com_TxPdu_...` and `G_Net2Run_Com_RxPdu_...` commands.

`Flags`
PDU flags

The following values are possible and can be combined:

`G_NET2RUN__PDU_INFO__FLAG__NONE`
No flag is set

`G_NET2RUN__PDU_INFO__FLAG__TX`
If set, the PDU is a TX PDU. Otherwise the PDU is an RX PDU.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`reserved4`
reserved parameter (must be initialized with **0**)

`reserved5`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Pdu_PduInfo_Get

G_Net2Run_Pdu_PduInfo_Get — Return PDU information for specific PDU

Description

Use this command to query PDU information of a specific PDU.

The necessary information for specifying a PDU can be taken from the list of all available PDUs that can be queried with G_Net2Run_Pdu_PduList_Get .

Definition

```
G_Error_t
G_Net2Run_Pdu_PduInfo_Get(
 const G_PortHandle_t  portHandle,
 const char * const pduName,
 const char * const ecuName,
 const u32_t  controllerId,
 G_Net2Run_PduInfo_t * const pduInfo
);
```

Parameters

`portHandle`
Handle to the communication port

`pduName`
Name of the PDU

`ecuName`
Name of the ECU this PDU instance is configured for

`controllerId`
ID of the ECU controller this PDU instance is configured for

`pduInfo`
Returns structure with PDU information

The structure contains the following elements:

`PduName`
Name of the PDU

`EcuName`
Name of the ECU this PDU instance is configured for

`ControllerId`
ID of the ECU controller this PDU instance is configured for

`ComId`
Communication layer ID of this PDU instance

This ID can be used with `G_Net2Run_Com_TxPdu_...` and `G_Net2Run_Com_RxPdu_...` commands.

`Flags`
PDU flags

The following values are possible and can be combined:

`G_NET2RUN__PDU_INFO__FLAG__NONE`
No flag is set

`G_NET2RUN__PDU_INFO__FLAG__TX`
If set, the PDU is a TX PDU. Otherwise the PDU is an RX PDU.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`reserved4`
reserved parameter (must be initialized with **0**)

`reserved5`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Pdu_NumberOfIncludedSignals_Get

G_Net2Run_Pdu_NumberOfIncludedSignals_Get — Return number of signals in specific PDU

Description

Use this command to query the number of signals of a specific PDU.

The necessary information for specifying a PDU can be taken from the list of all available PDUs that can be queried with G_Net2Run_Pdu_PduList_Get .

Definition

```
G_Error_t
G_Net2Run_Pdu_NumberOfIncludedSignals_Get(
 const G_PortHandle_t  portHandle,
 const char * const pduName,
 const char * const ecuName,
 const u32_t  controllerId,
 u32_t * const numberOfSignals
);
```

Parameters

`portHandle`
Handle to the communication port

`pduName`
Name of the PDU

`ecuName`
Name of the ECU this PDU instance is configured for

`controllerId`
ID of the ECU controller this PDU instance is configured for

`numberOfSignals`
Returns number of signals of the PDU

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Pdu_IncludedSignals_Get

G_Net2Run_Pdu_IncludedSignals_Get — Return included signals of PDU

Description

Use this command to query information about included signals of a specific PDU.

The necessary information for specifying a PDU can be taken from the list of all available PDUs that can be queried with G_Net2Run_Pdu_PduList_Get .

Definition

```
G_Error_t
G_Net2Run_Pdu_IncludedSignals_Get(
 const G_PortHandle_t  portHandle,
 const char * const pduName,
 const char * const ecuName,
 const u32_t  controllerId,
 u32_t * const numberOfSignals,
 G_Net2Run_Pdu_IncludedSignal_t * const signals
);
```

Parameters

`portHandle`
Handle to the communication port

`pduName`
Name of the PDU

`ecuName`
Name of the ECU this PDU instance is configured for

`controllerId`
ID of the ECU controller this PDU instance is configured for

`numberOfSignals`
in : size of buffer `signals` in number of signals

out : number of returned signals in `signals`

`signals`
Returns array with signal information of the included signals

The structure contains the following elements:

`Name`
Signal name (represents the value **ShortName** from the **Fibex** definition of the signal)

`PduSignalId`
Index of the signal in the PDUs signal list

`BitPosition`
Bit position of the signal inside the PDU

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`reserved4`
reserved parameter (must be initialized with **0**)

`reserved5`
reserved parameter (must be initialized with **0**)

`reserved6`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

15 PDU Multiplexer

Functions for controlling the PDU multiplexer of a Net2Run **Rest Bus Simulation**

G_Net2Run_PduMux_MuxedTxPdu_MaxNumber

G_Net2Run_PduMux_MuxedTxPdu_MaxNumber — Set maximum number of multiplexed TX PDUS

Description

Use this command to set the maximum number of multiplexed TX PDUS.

This command needs to be called prior to G_Net2Run_PduMux_MuxedTxPdu_Config .

Definition

```
G_Error_t
G_Net2Run_PduMux_MuxedTxPdu_MaxNumber(
 const G_PortHandle_t  portHandle,
 const u32_t  maxNumber
);
```

Parameters

`portHandle`
Handle to the communication port

`maxNumber`
Maximum number of multiplexed TX PDUs that can be configured

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_PduMux_MuxedTxPdu_Config

G_Net2Run_PduMux_MuxedTxPdu_Config — Configuration of multiplexed TX PDU

Description

Use this command for configuration of a multiplexed TX PDU.

Definition

```
G_Error_t
G_Net2Run_PduMux_MuxedTxPdu_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_PduMux_MuxedTxPdu_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__PDU_MUX__MUXED_TX_PDU__CONFIG__FLAG__NONE`
No flag is set

`MuxedTxPduId`
PDU identifier for a multiplexed transmission PDU

(0..max number defined by G_Net2Run_PduMux_MuxedTxPdu_MaxNumber -1)

`PduLength`
Length of multiplexed TX PDU in bytes

`SelectorBitSize`
Size of the selector field in bits

`SelectorEndianness`
Endianness of the selector field

The following values are possible:

`G_NET2RUN__ENDIANNESS__LITTLE_ENDIAN`
Little endian

`G_NET2RUN__ENDIANNESS__BIG_ENDIAN`
Big endian

`G_NET2RUN__ENDIANNESS__OPAQUE`
Opaque

`SelectorBitPosition`
Selector field bit position within the multiplexed TX PDU

`InitialSelectorValue`
Selects the initial dynamic part

`UnusedAreasDefault`
Fill pattern for unused areas of multiplexed TX PDU

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`TxConfirmationTimeout`
As long as the timeout has not elapsed and as long as no transmission confirmation for this multiplexed TX PDU is received, a new transmission request with a demultiplexed TX PDU that belongs to this multiplexed TX PDU is blocked.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_PduMux_DemuxedTxPdu_MaxNumber

G_Net2Run_PduMux_DemuxedTxPdu_MaxNumber — Set maximum number of demultiplexed TX PDUS

Description

Use this command to set the maximum number of demultiplexed TX PDUS.

This command needs to be called prior to G_Net2Run_PduMux_DemuxedTxPdu_Config .

Definition

```
G_Error_t
G_Net2Run_PduMux_DemuxedTxPdu_MaxNumber(
 const G_PortHandle_t  portHandle,
 const u32_t  maxNumber
);
```

Parameters

`portHandle`
Handle to the communication port

`maxNumber`
Maximum number of demultiplexed TX PDUs that can be configured

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_PduMux_DemuxedTxPdu_Config

G_Net2Run_PduMux_DemuxedTxPdu_Config — Configuration of demultiplexed TX PDU

Description

Use this command for configuration of a demultiplexed TX PDU.

Definition

```
G_Error_t
G_Net2Run_PduMux_DemuxedTxPdu_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_PduMux_DemuxedTxPdu_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__PDU_MUX__DEMUXED_TX_PDU__CONFIG__FLAG__NONE`
No flag is set

`G_NET2RUN__PDU_MUX__DEMUXED_TX_PDU__CONFIG__FLAG__IS_STATIC_PART`
If this flag is set, this demultiplexed TX PDU is the static part of the corresponding multiplexed TX PDU.

If this flag is cleared, this demultiplexed TX PDU is the dynamic part of the corresponding multiplexed TX PDU.

`G_NET2RUN__PDU_MUX__DEMUXED_TX_PDU__CONFIG__FLAG__ENABLE_TRANSMIT`
If this flag is set and a transmission request of this demultiplexed TX PDU is received, the corresponding multiplexed TX PDU is updated and transmitted.

If this flag is cleared and a transmission request of this demultiplexed TX PDU is received, the corresponding multiplexed TX PDU is updated only.

`G_NET2RUN__PDU_MUX__DEMUXED_TX_PDU__CONFIG__FLAG__JIT_UPDATE`
If this flag is set, the data of this part is fetched just in time via PduR_IpduMTriggerTransmit.

If this flag is not set, the locally stored data is used.

AUTOSAR name: IpduMJitUpdate

`MuxedTxPduId`
PDU identifier for a multiplexed transmission PDU

(0..max number defined by G_Net2Run_PduMux_MuxedTxPdu_MaxNumber -1)

`DemuxedTxPduId`
PDU identifier for a not multiplexed transmission PDU

(0..max number defined by G_Net2Run_PduMux_DemuxedTxPdu_MaxNumber -1)

`NumberOfSegments`
Number of segments of the demultiplexed TX PDU

`SelectorValue`
Selector value

Not used if flag `G_NET2RUN__PDU_MUX__DEMUXED_TX_PDU__CONFIG__FLAG__IS_STATIC_PART` is set

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Segments`
Array with Segments of the demultiplexed TX PDU

The structure contains the following elements:

`BitPosition`
Bit position within multiplexed PDU

`BitSize`
Size in bits

`Endianness`
Endianness

The following values are possible:

`G_NET2RUN__ENDIANNESS__LITTLE_ENDIAN`
Little endian

`G_NET2RUN__ENDIANNESS__BIG_ENDIAN`
Big endian

`G_NET2RUN__ENDIANNESS__OPAQUE`
Opaque

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_PduMux_MuxedRxPdu_MaxNumber

G_Net2Run_PduMux_MuxedRxPdu_MaxNumber — Set maximum number of multiplexed RX PDUS

Description

Use this command to set the maximum number of multiplexed RX PDUS.

This command needs to be called prior to G_Net2Run_PduMux_MuxedRxPdu_Config .

Definition

```
G_Error_t
G_Net2Run_PduMux_MuxedRxPdu_MaxNumber(
 const G_PortHandle_t  portHandle,
 const u32_t  maxNumber
);
```

Parameters

`portHandle`
Handle to the communication port

`maxNumber`
Maximum number of multiplexed RX PDUs that can be configured

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_PduMux_MuxedRxPdu_Config

G_Net2Run_PduMux_MuxedRxPdu_Config — Configuration of multiplexed RX PDU

Description

Use this command for configuration of a multiplexed RX PDU.

Definition

```
G_Error_t
G_Net2Run_PduMux_MuxedRxPdu_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_PduMux_MuxedRxPdu_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__PDU_MUX__MUXED_RX_PDU__CONFIG__FLAG__NONE`
No flag is set

`MuxedRxPduId`
PDU identifier for a multiplexed PDU

(0..max number defined by G_Net2Run_PduMux_MuxedRxPdu_MaxNumber -1)

`SelectorBitSize`
Size of the selector field in bits

`SelectorEndianness`
Endianness of the selector field

The following values are possible:

`G_NET2RUN__ENDIANNESS__LITTLE_ENDIAN`
Little endian

`G_NET2RUN__ENDIANNESS__BIG_ENDIAN`
Big endian

`G_NET2RUN__ENDIANNESS__OPAQUE`
Opaque

`SelectorBitPosition`
Selector field bit position within the multiplexed RX PDU

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_PduMux_DemuxedRxPdu_MaxNumber

G_Net2Run_PduMux_DemuxedRxPdu_MaxNumber — Set maximum number of demultiplexed RX PDUS

Description

Use this command to set the maximum number of demultiplexed RX PDUS.

This command needs to be called prior to G_Net2Run_PduMux_DemuxedRxPdu_Config .

Definition

```
G_Error_t
G_Net2Run_PduMux_DemuxedRxPdu_MaxNumber(
 const G_PortHandle_t  portHandle,
 const u32_t  maxNumber
);
```

Parameters

`portHandle`
Handle to the communication port

`maxNumber`
Maximum number of demultiplexed RXPDUs that can be configured

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_PduMux_DemuxedRxPdu_Config

G_Net2Run_PduMux_DemuxedRxPdu_Config — Configuration of demultiplexed RX PDU

Description

Use this command for configuration of a demultiplexed RX PDU.

Definition

```
G_Error_t
G_Net2Run_PduMux_DemuxedRxPdu_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_PduMux_DemuxedRxPdu_Config_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__PDU_MUX__DEMUXED_RX_PDU__CONFIG__FLAG__NONE`
No flag is set

`G_NET2RUN__PDU_MUX__DEMUXED_RX_PDU__CONFIG__FLAG__IS_STATIC_PART`
If this flag is set, this demultiplexed RX PDU is the static part of the corresponding multiplexed RX PDU.

If this flag is cleared, this demultiplexed RX PDU is the dynamic part of the corresponding multiplexed RX PDU.

`MuxedRxPduId`
PDU identifier for a multiplexed RX PDU

(0..max number defined by G_Net2Run_PduMux_MuxedRxPdu_MaxNumber -1)

`DemuxedRxPduId`
PDU identifier for a demultiplexed RX PDU

(0..max number defined by G_Net2Run_PduMux_DemuxedRxPdu_MaxNumber -1)

`SelectorValue`
Selector value

Not used if flag `G_NET2RUN__PDU_MUX__DEMUXED_TX_PDU__CONFIG__FLAG__IS_STATIC_PART` is set

`reserved1`
reserved parameter (must be initialized with `0`)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

16 PDU Multiplexer 2

PDU multiplexer for mapping multiple contained PDUs to a container PDU (see AUTOSAR Release 4.2.2 Chapter 7.3 Multiple-PDU-to-Container handling)

For "I-PDU Multiplexing" handling use Section 15, “ PDU Multiplexer ”.

G_Net2Run_PduMux2_ContainerTxPdu_Config

G_Net2Run_PduMux2_ContainerTxPdu_Config — Configuration of a transmitted container PDU

Description

Use this command for configuration of a transmitted container PDU.

Definition

```
G_Error_t
G_Net2Run_PduMux2_ContainerTxPdu_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_PduMux2_ContainerTxPdu_Config_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__PDU_MUX2__CONTAINER_TX_PDU__CONFIG__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__PDU_MUX2__CONTAINER_TX_PDU__CONFIG__CMD_FLAG__FIRST_PDU_TRIGGER`
Defines if the transmission of this ContainerTxPdu shall be requested right after the first **ContainedTxPdu** was put into it.

AUTOSAR name: IpduMContainerTxFirstContainedPduTrigger

`G_NET2RUN__PDU_MUX2__CONTAINER_TX_PDU__CONFIG__CMD_FLAG__USE_SEND_TIMEOUT`
Defines if parameter `SendTimeout` is used.

AUTOSAR: see IpduMContainerTxSendTimeout

`G_NET2RUN__PDU_MUX2__CONTAINER_TX_PDU__CONFIG__CMD_FLAG__USE_SIZE_THRESHOLD`
Defines if parameter `SizeThreshold` is used.

AUTOSAR: see IpduMContainerTxSizeThreshold

`ContainerTxPduId`
PDU identifier for a container TX PDU

AUTOSAR: see IpduMContainerTxHandleId

`HeaderType`
Header type

The following values are possible:

`G_NET2RUN__PDU_MUX2__HEADER_TYPE__SHORT`
Header size is 32 bit: 24 bit HeaderId + 8 bit DLC

AUTOSAR name: IPDUM_HEADERTYPE_SHORT

`G_NET2RUN__PDU_MUX2__HEADER_TYPE__LONG`
Header size is 64 bit: 32 bit HeaderId + 32 bit DLC

AUTOSAR name: IPDUM_HEADERTYPE_LONG

AUTOSAR name: IpduMContainerHeaderSize

`HeaderByteOrder`
Header byte order

The following values are possible:

`G_NET2RUN__PDU_MUX2__HEADER_BYTE_ORDER__LITTLE_ENDIAN`
Headers inside a Container I-PDU shall be ordered little endian

AUTOSAR name: IPDUM_LITTLE_ENDIAN

`G_NET2RUN__PDU_MUX2__HEADER_BYTE_ORDER__BIG_ENDIAN`
Headers inside a Container I-PDU shall be ordered big endian

AUTOSAR name: IPDUM_BIG_ENDIAN

AUTOSAR name: IpduMHeaderByteOrder

`QueueSize`
Defines a local queue for handling of each **ContainerPdu** . Defined in number of instances of this **ContainerPdu** .

In case more than one instance of a ContainerTxPdu has to be kept by PduMux2, up to `QueueSize` instances can be stored in addition to the current instance. The current instance is one instance of the ContainerTxPdu PDU that currently instances of ContainedTxPdus are being added to.

AUTOSAR name: IpduMContainerQueueSize

`ContainerTxTriggerMode`
Container PDU TX trigger mode

The following values are possible:

`G_NET2RUN__PDU_MUX2__CONTAINER_TX_TRIGGER_MODE__DIRECT`
The PduMux2 sends this **ContainerTxPdu** when it is triggered

AUTOSAR name: IPDUM_DIRECT

`G_NET2RUN__PDU_MUX2__CONTAINER_TX_TRIGGER_MODE__TRIGGER_TRANSMIT`
The **ContainerTxPdu** is stored in the PduMux2 and fetched via trigger transmit

AUTOSAR name: IPDUM_TRIGGERTRANSMIT

AUTOSAR name: IpduMContainerTxTriggerMode

`PduLengthMax`
Maximum PDU size in bytes

`SendTimeout`
Used if flag `G_NET2RUN__PDU_MUX2__CONTAINER_TX_PDU__CONFIG__CMD_FLAG__USE_SEND_TIMEOUT` is set

When this timeout expires the ContainerTxPdu is triggered for sending. The respective timer is started when the first **ContainedTxPdu** is put into the ContainerTxPdu.

[microseconds]

AUTOSAR: see IpduMContainerTxSendTimeout

`SizeThreshold`
Used if flag `G_NET2RUN__PDU_MUX2__CONTAINER_TX_PDU__CONFIG__CMD_FLAG__USE_SIZE_THRESHOLD` is set

Defines the size threshold in bytes which, when exceeded, triggers the sending of the **ContainerPdu** although the maximum **ContainerTxPdu** size has not been reached yet.

AUTOSAR name: IpduMContainerTxSizeThreshold

`TxConfirmationTimeout`
This timeout defines the timeout period for monitoring the reception of the TxConfirmation. It is not used when an I-PDU is requested using the trigger transmit API.

In case the `TxConfirmationTimeout` is configured to a value greater than 0, as long as the corresponding transmission confirmation timeout timer has not elapsed, and no transmission confirmation for that ContainerTxPdu was received, PduMux2 waits for the TxConfirmation before invoking PduRouter_PduMux2_Transmit for the next instance of that ContainerTxPdu.

[microseconds]

AUTOSAR: see IpduMContainerTxConfirmationTimeout

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_PduMux2_ContainedTxPdu_Config

G_Net2Run_PduMux2_ContainedTxPdu_Config — Configuration of a transmitted contained PDU

Description

Use this command for configuration of a transmitted contained PDU.

Definition

```
G_Error_t
G_Net2Run_PduMux2_ContainedTxPdu_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_PduMux2_ContainedTxPdu_Config_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__PDU_MUX2__CONTAINED_TX_PDU__CONFIG__CMD_FLAG__ZERO`
No flag is set

`G_NET2RUN__PDU_MUX2__CONTAINED_TX_PDU__CONFIG__CMD_FLAG__TX_CONF`
This Flag determines whether for this **ContainedTxPdu** a **TxConfirmation** shall be provided.

AUTOSAR name: IpduMContainedTxPduConfirmation

`G_NET2RUN__PDU_MUX2__CONTAINED_TX_PDU__CONFIG__CMD_FLAG__USE_SEND_TIMEOUT`
Determines if parameter SendTimeout is used.

AUTOSAR: see IpduMContainedTxPduSendTimeout

`ContainerTxPduId`
PDU identifier for a container TX PDU

AUTOSAR: see IpduMContainedTxInContainerPduRef

`ContainedTxPduId`
PDU identifier for a contained TX PDU

AUTOSAR: see IpduMContainedTxPduHandleId

`HeaderId`
Header ID which is part of the **ContainerPdu** when this **ContainedPdu** is inside.

Range: **1..0xffffffff**

AUTOSAR name: IpduMContainedPduHeaderId

`CollectionSemantics`
Collection semantics

The following values are possible:

`G_NET2RUN__PDU_MUX2__COLLECTION_SEMANTICS__LAST_IS_BEST`
The **IpduMContainedTxPdu** data will be fetched via **TriggerTransmit** just before the transmission executes.

AUTOSAR name: IPDUM_COLLECT_LAST_IS_BEST

`G_NET2RUN__PDU_MUX2__COLLECTION_SEMANTICS__QUEUED`
The **IpduMContainedTxPdu** data will instantly be stored to the **IpduMContainerTxPdu** in the context of the Transmit API.

AUTOSAR name: IPDUM_COLLECT_QUEUED

AUTOSAR name: IpduMContainedTxPduCollectionSemantics

`Trigger`
Contained TX PDU trigger

The following values are possible:

`G_NET2RUN__PDU_MUX2__CONTAINED_TX_PDU_TRIGGER__ALWAYS`
This **ContainedTxPdu** directly triggers the sending of the ContainerTxPdu.

AUTOSAR name: IPDUM_TRIGGER_ALWAYS

`G_NET2RUN__PDU_MUX2__CONTAINED_TX_PDU_TRIGGER__NEVER`
This **ContainedTxPdu** does not trigger the sending of the **ContainerTxPdu** (other trigger criteria might still trigger sending of the **ContainerTxPdu**).

AUTOSAR name: IPDUM_TRIGGER_NEVER

AUTOSAR name: IpduMContainedTxPduTrigger

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`SendTimeout`
Used if flag `G_NET2RUN__PDU_MUX2__CONTAINED_TX_PDU__CONFIG__CMD_FLAG__USE_SEND_TIMEOUT` is set

Defines a **ContainedTxPdu** specific sender timeout which can reduce the **ContainerPdu** timer when this **ContainedTxPdu** is put inside the **ContainerTxPdu**.

[microseconds]

AUTOSAR: see IpduMContainedTxPduSendTimeout

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_PduMux2_ContainedRxPduPool_Config

G_Net2Run_PduMux2_ContainedRxPduPool_Config — Configuration of a pool of contained RX PDUs

Description

Use this command for configuration of a pool of contained RX PDUs.

With the help of a **ContainerRxPduPool** it is possible that a **ContainerRxPdu** can accept all **ContainedRxPdus** of a **ContainedRxPduPool** while an other **ContainerRxPdu** can accept the same **ContainedRxPdus** (`ContainerRxAcceptionMode = G_NET2RUN__PDU_MUX2__CONTAINER_RX_ACCEPTION_MODE__ALL`) or a subset of the same **ContainedRxPdus** (`ContainerRxAcceptionMode = G_NET2RUN__PDU_MUX2__CONTAINER_RX_ACCEPTION_MODE__CONFIGURED`).

A **ContainedRxPduPool** is not specified in AUTOSAR, but is required to simulate multiple ECUs. Each simulated ECU can have its own **ContainedRxPduPool** with its own **ContainedRxPdus**. E.g. a **ContainedRxPdu1** for **Ecu1** has **HeaderId=0x123** and **ContainedRxPdu2** for **Ecu2** has also **HeaderId=0x123**.

Definition

```
G_Error_t
G_Net2Run_PduMux2_ContainedRxPduPool_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_PduMux2_ContainedRxPduPool_Config_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__PDU_MUX2__CONTAINED_RX_PDU_POOL__CONFIG__CMD_FLAG__ZERO`
No flag is set

`ContainedRxPduPoolId`
Contained RX PDU pool identifier

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_PduMux2_ContainerRxPdu_Config

G_Net2Run_PduMux2_ContainerRxPdu_Config — Configuration of a pool of container RX PDUs

Description

Use this command for onfiguration of a pool of container RX PDUs.

AUTOSAR name: IpduMContainerRxPdu

Definition

```
G_Error_t
G_Net2Run_PduMux2_ContainerRxPdu_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_PduMux2_ContainerRxPdu_Config_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__PDU_MUX2__CONTAINER_RX_PDU__CONFIG__CMD_FLAG__NONE`
No flag is set

`ContainerRxPduId`
PDU identifier for a container RX PDU

AUTOSAR: see IpduMContainerRxHandleId

`HeaderType`
Header type

The following values are possible:

`G_NET2RUN__PDU_MUX2__HEADER_TYPE__SHORT`
Header size is 32 bit: 24 bit HeaderId + 8 bit DLC

AUTOSAR name: IPDUM_HEADERTYPE_SHORT

`G_NET2RUN__PDU_MUX2__HEADER_TYPE__LONG`
Header size is 64 bit: 32 bit HeaderId + 32 bit DLC

AUTOSAR name: IPDUM_HEADERTYPE_LONG

AUTOSAR name: IpduMContainerHeaderSize

`HeaderByteOrder`
Header byte order

The following values are possible:

`G_NET2RUN__PDU_MUX2__HEADER_BYTE_ORDER__LITTLE_ENDIAN`
Headers inside a Container I-PDU shall be ordered little endian

AUTOSAR name: IPDUM_LITTLE_ENDIAN

`G_NET2RUN__PDU_MUX2__HEADER_BYTE_ORDER__BIG_ENDIAN`
Headers inside a Container I-PDU shall be ordered big endian

AUTOSAR name: IPDUM_BIG_ENDIAN

AUTOSAR name: IpduMHeaderByteOrder

`ContainerRxAcceptionMode`
Container RX acception mode

The following values are possible:

`G_NET2RUN__PDU_MUX2__CONTAINER_RX_ACCEPTION_MODE__ALL`
The **ContainedRxPdus** which are referencing this **ContainerRxPdu** are expected inside this **ContainerRxPdu** , but there may also occur all Pdus from a **ContainerRxPduPool** inside this **IpduMRxContainerPdu** as well. This also supports the case where no **ContainedRxPdu** references the **ContainerRxPdu** .

AUTOSAR name: IPDUM_ACCEPT_ALL

`G_NET2RUN__PDU_MUX2__CONTAINER_RX_ACCEPTION_MODE__CONFIGURED`
Only the **ContainedRxPdus** which are referencing this **ContainerRxPdu** are expected inside this **ContainerRxPdu** .

AUTOSAR name: IPDUM_ACCEPT_CONFIGURED

`G_NET2RUN__PDU_MUX2__CONTAINER_RX_ACCEPTION_MODE__UNKNOWN`
Container RX acception mode is unknown

AUTOSAR name: IpduMContainerRxAcceptContainedPdu

`reserved`
reserved parameter (must be initialized with **0**)

`ContainedRxPduPoolId`
Contained RX PDU pool ID (used if `ContainerRxAcceptionMode` = `G_NET2RUN__PDU_MUX2__CONTAINER_RX_ACCEPTION_MODE__ALL`)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_PduMux2_ContainedRxPdu_Config

G_Net2Run_PduMux2_ContainedRxPdu_Config — Configuration of a received contained PDU

Description

Use this command for configuration of a received contained PDU.

AUTOSAR name: IpduMContainedRxPdu

Definition

```
G_Error_t
G_Net2Run_PduMux2_ContainedRxPdu_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_PduMux2_ContainedRxPdu_Config_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__PDU_MUX2__CONTAINED_RX_PDU__CONFIG__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__PDU_MUX2__CONTAINED_TX_PDU__CONFIG__CMD_FLAG__HAS_CONTAINER_REF`
Determines if the **ContainedRxPdu** has a reference to a **ContainerRxPdu**

If an **ContainerRxPdu** is configured with `ContainerRxAcceptionMode` = `G_NET2RUN__PDU_MUX2__CONTAINER_RX_ACCEPTION_MODE__CONFIGURED` this flag must be set to allow reception of this **ContainedRxPdu** within the **ContainerRxPdu** .

`ContainedRxPduPoolId`
Identifier for a **ContainedRxPduPool** (see G_Net2Run_PduMux2_ContainedRxPduPool_Config)

A **ContainedRxPdu** always belongs to a **ContainedRxPduPool** this parameter is not specified in AUTOSAR.

`ContainerRxPduId`
Identifier for a **ContainerRxPdu**

A **ContainedRxPdu** optionally belongs to a **ContainerRxPdu** .

Used if flag `G_NET2RUN__PDU_MUX2__CONTAINED_TX_PDU__CONFIG__CMD_FLAG__HAS_CONTAINER_REF` is set.

AUTOSAR: see IpduMContainedRxInContainerPduRef

`ContainedRxPduId`
Identifier for a **ContainedRxPdu**

AUTOSAR: see IpduMContainedRxPduRef

`HeaderId`
Header ID which is part of the **ContainerRxPdu** when this **ContainedRxPdu** is inside.

Range: **1..0xffffffff**

AUTOSAR name: IpduMContainedPduHeaderId

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

17 PDU Router

Configuration functions for the **PDU Router** of **Net2Run**

The Net2Run PDU Router is configured automatically via the **Net2Run Network Runtime Configurator** . In case the **Net2Run Network Runtime Configurator** is not used, you can use these functions to access the Net2Run PDU Router for manual configuration.

G_Net2Run_PduRouter_SourcePdu_MaxNumber

G_Net2Run_PduRouter_SourcePdu_MaxNumber — Set maximum number of source PDUs

Description

Use this command to set the maximum number of PDUs for a specific source.

This command needs to be called prior to G_Net2Run_PduRouter_RoutingPath_Config .

Definition

```
G_Error_t
G_Net2Run_PduRouter_SourcePdu_MaxNumber(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_PduRouter_RoutingPath_Source_t  source,
 const u32_t  maxNumber
);
```

Parameters

`portHandle`
Handle to the communication port

`source`
PDU source type

The following values are possible:

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__COM__TX`
TX PDU in Net2Run COM layer

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__CAN_IF__RX`
CAN RX PDU

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__CAN_TP__RX`
CAN Transport Protocol RX PDU

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__PDU_MUX__RX`
Demultiplexed RX PDU

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__PDU_MUX__TX`
Multiplexed TX PDU

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__LIN_IF__RX`
LIN RX PDU

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__FLEXRAY_IF__RX`
FrIf_RxIndication (FlexRay RX PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__FLEXRAY_NM__RX`
PduRouter_FrNm_RxIndication (FlexRay NM RX PDU)

Used to provide EIRA or ERA data to COM module.

EIRA=External Internal Requests Aggregated (aggregated state of internal and external requested partial networks)

ERA=External Requests Aggregated (aggregated state of external requested partial networks)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__PDU_MUX2__RX`
PduRouter_PduMux2_RxIndication (contained RX PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__PDU_MUX2__TX`
PduRouter_PduMux2_Transmit (container TX PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__SO_AD__RX`
PduRouter_SoAd_RxIndication (Socket Adaptor rx PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__UDP_NM__RX`
PduRouter_UdpNm_RxIndication (UDP NM rx PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__SEC_OC__RX`
PduRouter_SecOc_RxIndication (rx authentic PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__SEC_OC__TX`
PduRouter_SecOc_Transmit (tx secured PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__FVM__TX`
PduRouter_Fvm_Transmit (Fvm tx PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__BCST_TP_C1__RX`
PduRouter_BroadcastTpC1_RxIndication (BROADCAST TP C1 rx PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__BCST_TP_C1__TX`
PduRouter_BroadcastTpC1_Transmit (BROADCAST TP C1 tx PDU)

`maxNumber`
Maximum number of PDUs that can be configured for the selected source type

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_PduRouter_TargetPdu_MaxNumber

G_Net2Run_PduRouter_TargetPdu_MaxNumber — Set maximum number of target PDUs

Description

Use this command to set the maximum number of PDUs for a specific target.

This command needs to be called prior to G_Net2Run_PduRouter_RoutingPath_Config .

Definition

```
G_Error_t
G_Net2Run_PduRouter_TargetPdu_MaxNumber(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_PduRouter_RoutingPath_Target_t  target,
 const u32_t  maxNumber
);
```

Parameters

`portHandle`
Handle to the communication port

`target`
PDU target type

The following values are possible:

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__COM__RX`
RX PDU in Net2Run COM layer

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__CAN_IF__TX`
CAN TX PDU

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__CAN_TP__TX`
CAN Transport Protocol TX PDU

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__PDU_MUX__RX`
Multiplexed RX PDU

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__PDU_MUX__TX`
Demultiplexed TX PDU

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__LIN_IF__TX`
LIN TX PDU

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__FLEXRAY_IF__TX`
FlexRayIf_Transmit (FLEXRAY TX PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__FLEXRAY_NM__TX`
FlexRayNm_Transmit (FlexRay NM user data TX PDU, usually the router source is a COM TX PDU)

Used to transmit a COM TX PDU as part of the NM user data message

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__PDU_MUX2__RX`
PduMux2_RxIndication (container RX PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__PDU_MUX2__TX`
PduMux2_Transmit (contained TX PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__SO_AD__TX`
SoAd_Transmit (Socket Adaptor TX PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__SEC_OC__RX`
SecOc_RxIndication (RX secured PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__SEC_OC__TX`
SecOc_Transmit (TX authentic PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__FVM__RX`
Fvm_RxIndication (Fvm rx PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__BCST_TP_C1__RX`
PduRouter_BroadcastTpC1_RxIndication (BROADCAST TP C1 rx PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__BCST_TP_C1__TX`
PduRouter_BroadcastTpC1_Transmit (BROADCAST TP C1 rx PDU)

`maxNumber`
Maximum number of PDUs that can be configured for the selected target type

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_PduRouter_RoutingPath_Config

G_Net2Run_PduRouter_RoutingPath_Config — Configure routing path

Description

Use this command to configure the routing path of PDUs in **Net2Run Pdu Router .**

Definition

```
G_Error_t
G_Net2Run_PduRouter_RoutingPath_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_PduRouter_RoutingPath_Config_Parameters_t * const
 parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`parameters`
Structure with routing path parameters

The structure contains the following values:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__CONFIG__FLAG__NONE`
No flag is set

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__CONFIG__FLAG__ADD`
If this flag is set, the new built route from source to target is added to possibly existing routes beginning from the same source. This is used, when a source PDU is routed to multiple target PDUs. If this flag is cleared, before building a route from source to target, all existing routes from the same source are deleted.

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__CONFIG__FLAG__DELETE`
If this flag is set, a specific routing path is deleted. The routing path must exist, else an error is generated. This flag is not combinable with flags `G_NET2RUN__PDU_ROUTER__ROUTING_PATH__CONFIG__FLAG__ADD` and `G_NET2RUN__PDU_ROUTER__ROUTING_PATH__CONFIG__FLAG__DELETE_ALL`.

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__CONFIG__FLAG__DELETE_ALL`
If this flag is set, all routing paths beginning from `Source` are deleted. This flag is not combinable with flags `G_NET2RUN__PDU_ROUTER__ROUTING_PATH__CONFIG__FLAG__ADD` and `G_NET2RUN__PDU_ROUTER__ROUTING_PATH__CONFIG__FLAG__DELETE`.

`Source`

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__COM__TX`
TX PDU in Net2Run COM layer

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__CAN_IF__RX`
CAN RX PDU

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__CAN_TP__RX
CAN Transport Protocol RX PDU

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__PDU_MUX__RX
Demultiplexed RX PDU

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__PDU_MUX__TX
Multiplexed TX PDU

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__LIN_IF__RX
LIN RX PDU

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__FLEXRAY_IF__RX
FrIf_RxIndication (FlexRay RX PDU)

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__FLEXRAY_NM__RX
PduRouter_FrNm_RxIndication (FlexRay NM RX PDU)

Used to provide EIRA or ERA data to COM module.

EIRA=External Internal Requests Aggregated (aggregated state of internal and external requested partial networks)

ERA=External Requests Aggregated (aggregated state of external requested partial networks)

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__PDU_MUX2__RX
PduRouter_PduMux2_RxIndication (contained RX PDU)

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__PDU_MUX2__TX
PduRouter_PduMux2_Transmit (container TX PDU)

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__SO_AD__RX
PduRouter_SoAd_RxIndication (Socket Adaptor rx PDU)

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__UDP_NM__RX
PduRouter_UdpNm_RxIndication (UDP NM rx PDU)

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__SEC_OC__RX
PduRouter_SecOc_RxIndication (rx authentic PDU)

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__SEC_OC__TX
PduRouter_SecOc_Transmit (tx secured PDU)

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__FVM__TX
PduRouter_Fvm_Transmit (Fvm tx PDU)

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__BCST_TP_C1__RX
PduRouter_BroadcastTpC1_RxIndication (BROADCAST TP C1 rx PDU)

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__BCST_TP_C1__TX
PduRouter_BroadcastTpC1_Transmit (BROADCAST TP C1 tx PDU)

Target

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__COM__RX
RX PDU in Net2Run COM layer

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__CAN_IF__TX
CAN TX PDU

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__CAN_TP__TX
CAN Transport Protocol TX PDU

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__PDU_MUX__RX
Multiplexed RX PDU

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__PDU_MUX__TX
Demultiplexed TX PDU

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__LIN_IF__TX
LIN TX PDU

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__FLEXRAY_IF__TX
FlexRayIf_Transmit (FLEXRAY TX PDU)

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__FLEXRAY_NM__TX
FlexRayNm_Transmit (FlexRay NM user data TX PDU, usually the router source is a COM TX PDU)

Used to transmit a COM TX PDU as part of the NM user data message

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__PDU_MUX2__RX
PduMux2_RxIndication (container RX PDU)

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__PDU_MUX2__TX
PduMux2_Transmit (contained TX PDU)

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__SO_AD__TX
SoAd_Transmit (Socket Adaptor TX PDU)

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__SEC_OC__RX
SecOc_RxIndication (RX secured PDU)

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__SEC_OC__TX
SecOc_Transmit (TX authentic PDU)

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__FVM__RX
Fvm_RxIndication (Fvm rx PDU)

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__BCST_TP_C1__RX
PduRouter_BroadcastTpC1_RxIndication (BROADCAST TP C1 rx PDU)

G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__BCST_TP_C1__TX
PduRouter_BroadcastTpC1_Transmit (BROADCAST TP C1 rx PDU)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`SourcePduId`
Index of PDU that is configured (starting with **0**)

This index uniquely identifies the PDU.

The PDU needs to be configured in the specific source layer (e.g. G_Net2Run_CanIf_RxPdu_Config)

`TargetPduId`
Index of PDU that is configured (starting with **0**)

This index uniquely identifies the PDU.

The PDU needs to be configured in the specific target layer (e.g. G_Net2Run_CanIf_TxPdu_Config)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

In the following example, a Net2Run CAN Interface and the Net2Run PDU Router are configured as a gateway. An incoming CAN PDU with identifier **0x123** is routed to an outgoing CAN PDU with identifier **0x124** .

```
#include <stdio.h>
#include "g_api_common.h"
#include "g_api_net2run.h"


void
main(
 void
)
{
 G_PortHandle_t  portHandle;
 G_Error_t  rc;

 // open Net2Run interface
 rc =
  G_Common_OpenInterface(
   "NET2RUN1",
   &portHandle
  );

 if (rc == G_NO_ERROR) {
  // set max number of CAN interface RX PDUs to 1
  rc =
   G_Net2Run_CanIf_RxPdu_MaxNumber(
    portHandle,
    1
   );
 }

 if (rc == G_NO_ERROR) {
  // set max number of CAN interface TX PDUs to 1
  rc =
   G_Net2Run_CanIf_TxPdu_MaxNumber(
    portHandle,
    1
   );
 }

 if (rc == G_NO_ERROR) {
  // configure CAN interface RX PDU
  G_Net2Run_CanIf_RxPdu_Config_Parameters_t  param;

  param.BusId = 0;  // use 1st CAN interface for RX
  param.CanId = 0x123;
  param.Dlc = 8;
  param.EcuId = 0;

  param.Flags =
```

```
   G_NET2RUN__CAN_IF__RX_PDU__CONFIG__FLAG__ENABLE_RX_INDICATION;

 param.PduId = 0;
 param.reserved = 0;

 rc =
  G_Net2Run_CanIf_RxPdu_Config(
   portHandle,
   &param
  );
}

if (rc == G_NO_ERROR) {
 // configure CAN interface TX PDU
 G_Net2Run_CanIf_TxPdu_Config_Parameters_t  param;

 param.BusId = 0;  // use 1st CAN interface for TX
 param.CanId = 0x124;
 param.Dlc = 8;
 param.EcuId = 0;

 param.Flags =
  G_NET2RUN__CAN_IF__TX_PDU__CONFIG__FLAG__ENABLE_TX_CONFIRMATION;

 param.PduId = 0;
 param.reserved = 0;

 rc =
  G_Net2Run_CanIf_TxPdu_Config(
   portHandle,
   &param
  );
}

if (rc == G_NO_ERROR) {
 // set max number of PDU Router CAN RX source PDUs to 1
 rc =
  G_Net2Run_PduRouter_SourcePdu_MaxNumber(
   portHandle,
   G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__CAN_IF__RX,
   1
  );
}

if (rc == G_NO_ERROR) {
 // set max number of PDU Router CAN TX target PDUs to 1
 rc =
  G_Net2Run_PduRouter_SourcePdu_MaxNumber(
   portHandle,
   G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__CAN_IF__TX,
   1
  );
}

if (rc == G_NO_ERROR) {
 // config routing path to route RX PDU 0 (ID = 0x123) on the 1st CAN
 // interface to TX PDU 0 (ID = 0x124) on the 1st CAN interface
 G_Net2Run_PduRouter_RoutingPath_Config_Parameters_t  param;
```

```
  param.Flags = G_NET2RUN__PDU_ROUTER__ROUTING_PATH__CONFIG__FLAG__NONE;
  param.reserved1 = 0;
  param.reserved2 = 0;
  param.Source = G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__CAN_IF__RX;
  param.SourcePduId = 0;
  param.Target = G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__CAN_IF__TX;
  param.TargetPduId = 0;

  rc =
   G_Net2Run_PduRouter_RoutingPath_Config(
    portHandle,
    &param
   );
 }

 // close interface
 (void) G_Common_CloseInterface(portHandle);

 if (rc != G_NO_ERROR) {
  printf("\nError: %s\n", G_GetErrorDescription(rc));
 }
}
```

G_Net2Run_PduRouter_RoutingError

G_Net2Run_PduRouter_RoutingError — Configure routing error

Description

Use this command to configure a routing error.

Definition

```
G_Error_t
G_Net2Run_PduRouter_RoutingError(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_PduRouter_RoutingError_Cmd_t_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following items:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__PDU_ROUTER__ROUTING_ERROR__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__PDU_ROUTER__ROUTING_ERROR__CMD_FLAG__USE_TIME`
Use parameter `Time`

`G_NET2RUN__PDU_ROUTER__ROUTING_ERROR__CMD_FLAG__ANY_SOURCE`
Use any available source

`G_NET2RUN__PDU_ROUTER__ROUTING_ERROR__CMD_FLAG__ANY_TARGET`
Use any available target

`Mode`
Routing error mode

The following values are possible:

`G_NET2RUN__PDU_ROUTER__ROUTING_ERROR__MODE__NO_ERROR`
No error

`G_NET2RUN__PDU_ROUTER__ROUTING_ERROR__MODE__NO_SOURCE_TX_REQ`
Transmit request error

`G_NET2RUN__PDU_ROUTER__ROUTING_ERROR__MODE__NO_SOURCE_TRIG_TX`
Trigger transmit error

`reserved`
reserved parameter (must be initialized with `0`)

`Source`
Routingpath source

The following values are possible:

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__COM__TX`
TX PDU in Net2Run COM layer

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__CAN_IF__RX`
CAN RX PDU

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__CAN_TP__RX`
CAN Transport Protocol RX PDU

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__PDU_MUX__RX`
Demultiplexed RX PDU

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__PDU_MUX__TX`
Multiplexed TX PDU

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__LIN_IF__RX`
LIN RX PDU

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__FLEXRAY_IF__RX`
FrIf_RxIndication (FlexRay RX PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__FLEXRAY_NM__RX`
PduRouter_FrNm_RxIndication (FlexRay NM RX PDU)

Used to provide EIRA or ERA data to COM module.

EIRA=External Internal Requests Aggregated (aggregated state of internal and external requested partial networks)

ERA=External Requests Aggregated (aggregated state of external requested partial networks)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__PDU_MUX2__RX`
PduRouter_PduMux2_RxIndication (contained RX PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__PDU_MUX2__TX`
PduRouter_PduMux2_Transmit (container TX PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__SO_AD__RX`
PduRouter_SoAd_RxIndication (Socket Adaptor rx PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__UDP_NM__RX`
PduRouter_UdpNm_RxIndication (UDP NM rx PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__SEC_OC__RX`
PduRouter_SecOc_RxIndication (rx authentic PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__SEC_OC__TX`
PduRouter_SecOc_Transmit (tx secured PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__FVM__TX`
PduRouter_Fvm_Transmit (Fvm tx PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__BCST_TP_C1__RX`
PduRouter_BroadcastTpC1_RxIndication (BROADCAST TP C1 rx PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__SOURCE__BCST_TP_C1__TX`
PduRouter_BroadcastTpC1_Transmit (BROADCAST TP C1 tx PDU)

`Target`
Routingpath source

The following values are possible:

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__COM__RX`
RX PDU in Net2Run COM layer

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__CAN_IF__TX`
CAN TX PDU

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__CAN_TP__TX`
CAN Transport Protocol TX PDU

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__PDU_MUX__RX`
Multiplexed RX PDU

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__PDU_MUX__TX`
Demultiplexed TX PDU

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__LIN_IF__TX`
LIN TX PDU

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__FLEXRAY_IF__TX`
FlexRayIf_Transmit (FLEXRAY TX PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__FLEXRAY_NM__TX`
FlexRayNm_Transmit (FlexRay NM user data TX PDU, usually the router source is a COM TX PDU)

Used to transmit a COM TX PDU as part of the NM user data message

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__PDU_MUX2__RX`
PduMux2_RxIndication (container RX PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__PDU_MUX2__TX`
PduMux2_Transmit (contained TX PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__SO_AD__TX`
SoAd_Transmit (Socket Adaptor TX PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__SEC_OC__RX`
SecOc_RxIndication (RX secured PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__SEC_OC__TX`
SecOc_Transmit (TX authentic PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__FVM__RX`
Fvm_RxIndication (Fvm rx PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__BCST_TP_C1__RX`
PduRouter_BroadcastTpC1_RxIndication (BROADCAST TP C1 rx PDU)

`G_NET2RUN__PDU_ROUTER__ROUTING_PATH__TARGET__BCST_TP_C1__TX`
PduRouter_BroadcastTpC1_Transmit (BROADCAST TP C1 rx PDU)

`SourcePduId`
Source PDU ID

`TargetPduId`
Target PDU ID

`NumberOfErrors`
0 : error is not disabled by number

else : error is disabled after `NumberOfErrors` errors

`Time`
Error time in microseconds

Used if flag `G_NET2RUN__PDU_ROUTER__ROUTING_ERROR__CMD_FLAG__USE_TIME` is set

0 : error is not disabled by time

else : error is disabled after `Time` microseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

18 Secure Onboard Communication

Configuration functions for the **Sec** ure **O** nboard **C** ommunication module of **Net2Run**

The **SecOC** module aims for resource-efficient and practicable authentication mechanisms for critical data on the level of PDUs.

For many parameters in this section, references are made to the corresponding AUTOSAR specification **Specification of Module Secure Onboard Communication 4.3.0** that can be downloaded at http://www.autosar.org.

G_Net2Run_SecOc_RxPduProcessing

G_Net2Run_SecOc_RxPduProcessing — Configuration of the RX PDUs to be verified by the SecOC module

Description

Use this command for configuration of the Rx PDUs to be verified by the SecOC module.

Definition

```
G_Error_t
G_Net2Run_SecOc_RxPduProcessing(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SecOc_RxPduProcessing_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SEC_OC__RX_PDU_PROCESSING__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__SEC_OC__RX_PDU_PROCESSING__CMD_FLAG__USE_AUTH_DATA_FRESHNESS`
Use authentic data freshness values

ECUC_SecOC_00083

If it is set to **TRUE**, the values `SecOCAuthDataFreshnessStartPosition` and `SecOCAuthDataFreshnessLen` must be set to specify the bit position and length within the **Authentic_PDU**.

`G_NET2RUN__SEC_OC__RX_PDU_PROCESSING__CMD_FLAG__IGNORE_VERIFICATION_RESULT`
Ignore verification result

ECUC_SecOc_00052

The result of the authentication process (e.g. MAC Verify) is ignored after the first try and the **SecOC** proceeds like the result was a success. The calculation of the authenticator is still done, only its result will be ignored.

`RxPduProcessingId`
RX Processing PDU ID

`RxAuthenticPduId`
ECUC_SecOC_00041

RX Authentic PDU ID

`RxSecuredPduId`
RX Secured PDU ID

ECUC_SecOc_00044

`DataId`
Data ID

ECUC_SecOC_00030

This parameter defines a unique numerical identifier for Secured I-PDU.

`FreshnessValueLength`
Freshness value length, in bits (**0** .. **64**)

ECUC_SecOc_00031

This parameter defines the complete length in bits of the Freshness Value. As long as the key doesn't change, the counter shall not overflow. The length of the counter shall be determined based on the expected life time of the corresponding key and frequency of usage of the counter.

`FreshnessValueTxLength`
Freshness value TX length, in bits (**0** .. **64**)

ECUC_SecOC_00032

This parameter defines the length in bits of the Freshness Value to be included in the payload of the Secured I-PDU. This length is specific to the least significant bits of the complete Freshness Counter. If the parameter is 0 no Freshness Value is included in the Secured I-PDU.

`FreshnessValueId`
Freshness value ID

ECUC_SecOC_00038

This parameter defines the ID of the Freshness Value. The Freshness Value might be a normal counter or a time value.

`AuthDataFreshnessLength`
Authentic data freshness Length, in bits

ECUC_SecOc_00082

This parameter defines the complete length in bits of the Freshness Value. As long as the key doesn't change the counter shall not overflow. The length of the counter shall be determined based on the expected life time of the corresponding key and frequency of usage of the counter.

`AuthDataFreshnessStartPosition`
Authentic data freshness start position, in bits

ECUC_SecOc_00081

This parameter defines the length in bits of the Freshness Value to be included in the payload of the Secured I-PDU. This length is specific to the least significant bits of the complete Freshness Counter. If the parameter is **0** no Freshness Value is included in the Secured I-PDU.

`AuthVerifyAttempts`
Authentication verify attempts

ECUC_SecOC_00080

This parameter specifies the number of authentication verify attempts that are to be carried out when the verification of the authentication information failed for a given Secured I-PDU. If zero is set, then only one authentication verification attempt is done.

`AuthInfoTxLength`
Authentication info TX length, in bits

ECUC_SecOC_00034

This parameter defines the length in bits of the authentication code to be included in the payload of the Secured I-PDU.

`AuthServiceConfig_CsmJobId`
ECUC_SecOC_00048

AUTOSAR name: "SecOCRxAuthServiceConfigRef"

This reference is used to define which crypto service function is called for authentication.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SecOc_RxSecuredPdu

G_Net2Run_SecOc_RxSecuredPdu — Specifies the PDU that is received by the SecOC module from the PduRouter

Description

Use this command to specify the PDU that is received by the SecOC module from the PduRouter.

Definition

```
G_Error_t
G_Net2Run_SecOc_RxSecuredPdu(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SecOc_RxSecuredPdu_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__SEC_OC__RX_SECURED_PDU__CMD_FLAG__NONE`
No flag is set

`RxSecuredPduId`
ECUC_SecOC_00042/00043

RX Secured PDU ID

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SecOc_RxSecuredPduCollection

G_Net2Run_SecOc_RxSecuredPduCollection — Specifies two PDUs that are received by SecOC module from the PduRouter and a message linking between them

Description

Use this command to specify two PDUs that are received by SecOC module from the PduRouter and a message linking between them

For this PDU the Mac verification is provided.

Definition

```
G_Error_t
G_Net2Run_SecOc_RxSecuredPduCollection(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SecOc_RxSecuredPduCollection_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SEC_OC__RX_SECURED_PDU_COLLECTION__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__SEC_OC__RX_SECURED_PDU_COLLECTION__CMD_FLAG__USE_MSG_LINK`
ECUC_SecOC_00074

SecOC links an Authentic I-PDU and Cryptographic I-PDU together by repeating a specific part (Message Linker) of the Authentic I-PDU in the Cryptographic I-PDU.

`RxSecuredPduId_AuthenticPart`
RX Secured PDU ID authentic part

ECUC_SecOC_00061

`RxSecuredPduId_CryptographicPart`
RX Secured PDU ID cryptographic part

ECUC_SecOC_00064

`PduLengthMax_AuthenticPart`
Maximum PDU size in bytes for authentic PDU

`PduLengthMax_CryptographicPart`
maximum PDU size in bytes for cryptographic part

`MsgLinkPosition`
ECUC_SecOC_00059

The position of the Message Linker inside the Authentic I-PDU in bits.

`MsgLinkLength`
ECUC_SecOC_00060

Length of the Message Linker inside the Authentic I-PDU in bits.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SecOc_RxAuthenticPdu

G_Net2Run_SecOc_RxAuthenticPdu — Specify the Authentic PDU that is received by the SecOC module from the PduRouter

Description

Use this command to specify the Authentic PDU that is received by the SecOC module from the PduRouter.

Definition

```
G_Error_t
G_Net2Run_SecOc_RxAuthenticPdu(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SecOc_RxAuthenticPdu_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__SEC_OC__RX_AUTHENTIC_PDU__CMD_FLAG__NONE`
No flag is set

`RxAuthenticPduId`
RX Authentic PDU ID

ECUC_SecOC_00062/00063

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SecOc_TxPduProcessing

G_Net2Run_SecOc_TxPduProcessing — Configuration of the TX PDUs to be secured by the SecOC module

Description

Use this command for configuration of the TX PDUs to be secured by the SecOC module.

Definition

```
G_Error_t
G_Net2Run_SecOc_TxPduProcessing(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SecOc_TxPduProcessing_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SEC_OC__TX_PDU_PROCESSING__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__SEC_OC__TX_PDU_PROCESSING__CMD_FLAG__USE_TX_CONFIRMATION`
ECUC_SecOC_00085

A Boolean value that indicates if the function SecOC_SPduTxConfirmation shall be called for this PDU.

`G_NET2RUN__SEC_OC__TX_PDU_PROCESSING__CMD_FLAG__PROVIDE_TX_TRUNC_FV`
ECUC_SecOC_00084

This parameter specifies if the TX query freshness function provides the truncated freshness info instead of generating this by SecOC. In this case, SecOC shall add this data to the Authentic PDU instead of truncating the freshness value.

`TxPduProcessingId`
TX PDU Processing ID

`TxAuthenticPduId`
ECUC_SecOC_00023

TX Authentic PDU ID

`TxSecuredPduId`
ECUC_SecOc_00024

TX Secured PDU ID

`DataId`
ECUC_SecOC_00014

This parameter defines a unique numerical identifier for Secured I-PDU.

`FreshnessValueLength`
Freshness value length, in bits (**0** .. **64**)

ECUC_SecOc_00015

This parameter defines the complete length in bits of the Freshness Value. As long as the key doesn't change the counter shall not overflow. The length of the counter shall be determined based on the expected life time of the corresponding key and frequency of usage of the counter.

`FreshnessValueTxLength`
Freshness value TX length, in bits (**0** .. **64**)

ECUC_SecOC_00016

This parameter defines the length in bits of the Freshness Value to be included in the payload of the Secured I-PDU. This length is specific to the least significant bits of the complete Freshness Counter. If the parameter is **0** , no Freshness Value is included in the Secured I-PDU.

`FreshnessValueId`
Freshness value ID

ECUC_SecOC_00021

This parameter defines the ID of the Freshness Value. The Freshness Value might be a normal counter or a time value.

`AuthInfoTxLength`
Authentication info TX length, in bits

ECUC_SecOC_00018

This parameter defines the length in bits of the authentication code to be included in the payload of the Secured I-PDU.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`AuthServiceConfig_CsmJobId`
ECUC_SecOC_00013

AUTOSAR name: "SecOCRxAuthServiceConfigRef"

This reference is used to define which crypto service function is called for authentication.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SecOc_TxPduProcessing_Error

G_Net2Run_SecOc_TxPduProcessing_Error — manipulate message authentication code

Description

Use this command to manipulate message authentication code of the TX PDUs to be secured by the SecOC module.

Definition

```
G_Error_t
G_Net2Run_SecOc_TxPduProcessing_Error(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SecOc_TxPduProcessing_Error_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SEC_OC__TX_PDU_PROCESSING__ERROR__FLAG__NONE`
No flag is set

`TxPduProcessingId`
TX PDU Processing ID

`ErrorMode`
Error Mode

The following values are possible :

`G_NET2RUN__SEC_OC__ERROR_MODE__NO_ERROR`
The message authentication code is not be manipulated

`G_NET2RUN__SEC_OC__ERROR_MODE__ADD_ONE_TO_FIRST_BYTE`
After the message authentication code (MAC) is calculated one is added to the first byte of the MAC

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`NumberOfCycles`
0=set error mode "ErrorMode" permanently, else

error mode is set to "ErrorMode" for "NumberOfCycles" cycles and then it is set to "G_NET2RUN__SEC_OC__ERROR_MODE__NO_ERROR"

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SecOc_TxAuthenticPdu

G_Net2Run_SecOc_TxAuthenticPdu — Specify the PDU that is received by the SecOC module from the PduRouter

Description

Use this command to specify the PDU that is received by the SecOC module from the PduRouter.

Definition

```
G_Error_t
G_Net2Run_SecOc_TxAuthenticPdu(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SecOc_TxAuthenticPdu_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__SEC_OC__TX_AUTHENTIC_PDU__CMD_FLAG__NONE`
No flag is set

`TxAuthenticPduId`
TX Authentic PDU ID

ECUC_SecOC_00055/00056

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SecOc_TxSecuredPdu

G_Net2Run_SecOc_TxSecuredPdu — Specify one PDU that is transmitted by the SecOC module to the PduRouter after the Mac was generated.

Description

Use this command to specify one PDU that is transmitted by the SecOC module to the PduRouter after the Mac was generated. This PDU contains the cryptographic information.

Definition

```
G_Error_t
G_Net2Run_SecOc_TxSecuredPdu(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SecOc_TxSecuredPdu_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible:

`G_NET2RUN__SEC_OC__TX_SECURED_PDU__CMD_FLAG__NONE`
No flag is set

`TxSecuredPduId`
TX Secured PDU ID

ECUC_SecOC_00027/00028

`PduLengthMax`
Maximum PDU length, in bytes

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SecOc_TxSecuredPduCollection

G_Net2Run_SecOc_TxSecuredPduCollection — Specify the PDU that is transmitted by the SecOC module to the PduRouter after the Mac was generated.

Description

Use this command to specify the PDU that is transmitted by the SecOC module to the PduRouter after the Mac was generated. Two separate PDUs are transmitted to the PduRouter: Authentic I-PDU and Cryptographic I-PDU.

Definition

```
G_Error_t
G_Net2Run_SecOc_TxSecuredPduCollection(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SecOc_TxSecuredPduCollection_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SEC_OC__TX_SECURED_PDU_COLLECTION__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__SEC_OC__TX_SECURED_PDU_COLLECTION__CMD_FLAG__USE_MSG_LINK`
Use message link

ECUC_SecOC_00074

SecOC links an Authentic I-PDU and Cryptographic I-PDU together by repeating a specific part (Message Linker) of the Authentic I-PDU in the Cryptographic I-PDU.

`TxSecuredPduId_AuthenticPart`
TXSecured PDU ID authentic part

ECUC_SecOC_00072

`TxSecuredPduId_CryptographicPart`
TX Secured PDU ID cryptographic part

ECUC_SecOC_00073

`PduLengthMax_AuthenticPart`
Maximum PDU length for authentic part, in bytes

`PduLengthMax_CryptographicPart`
Maximum PDU length for cryptographic part, in bytes

`MsgLinkPosition`
The position of the Message Linker inside the Authentic I-PDU in bits.

Used if flag `G_NET2RUN__SEC_OC__TX_SECURED_PDU_COLLECTION__CMD_FLAG__USE_MSG_LINK` is set.

`MsgLinkLength`
Length of the Message Linker inside the Authentic I-PDU in bits.

Used if flag `G_NET2RUN__SEC_OC__TX_SECURED_PDU_COLLECTION__CMD_FLAG__USE_MSG_LINK` is set.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SecOc_GetVerificationStat

G_Net2Run_SecOc_GetVerificationStat — Get verification statistics

Description

Use this command to get verification statistics.

Definition

```
G_Error_t
G_Net2Run_SecOc_GetVerificationStat(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SecOc_GetVerificationStat_Cmd_t * const cmd,
 G_Net2Run_SecOc_GetVerificationStat_Rsp_t * const rsp
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SEC_OC__GET_VERIFICATION_STAT__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__SEC_OC__GET_VERIFICATION_STAT__CMD_FLAG__CLEAR_STAT`
Clear statistics after query

`RxPduProcessingId`
RX PDU processing ID

`rsp`
Returns response parameters

The structure contains the following elements:

`Flags`
Response flags

The following values are possible and can be combined:

`G_NET2RUN__SEC_OC__GET_VERIFICATION_STAT__RSP_FLAG__NONE`
No flag is set

`G_NET2RUN__SEC_OC__GET_VERIFICATION_STAT__RSP_FLAG__STAT_CLEARED`
Statistics have been cleared

`LastVerificationErrorCode`
Last verification error code

`VerificationPassCounter`
Number of successful verifications

`VerificationFailCounter`
Number of failed verifications

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

19 Signal Functions

Functions for configuring, reading and writing **Rest Bus Simulation** signals.

G_Net2Run_Signal_Config

G_Net2Run_Signal_Config — Configure **Rest Bus Simulation** signals

Description

Use this function to configure **Rest Bus Simulation** signals.

The **Rest Bus Simulation** signal parameters used for the configuration are taken from the **.rbs** file that is created with the Net2Run GUI.

This command is mandatory when signals of the **Rest Bus Simulation** have to be read or written, therefore it should be called before accessing any signal of the **Rest Bus Simulation** .

Definition

```
G_Error_t
G_Net2Run_Signal_Config(
 const G_PortHandle_t  portHandle,
 const char * const filename
);
```

Parameters

`portHandle`
: Handle to the communication port

`filename`
: Name of the **Rest Bus Simulation** file

  This file is created by the Net2Run GUI and has usually a **.rbs** file ending.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Signal_Config2

G_Net2Run_Signal_Config2 — Configure **Rest Bus Simulation** signals

Description

Use this function to configure **Rest Bus Simulation** signals without the need of an rbs file.

This command is mandatory when signals of the **Rest Bus Simulation** have to be read or written, therefore it should be called before accessing any signal of the **Rest Bus Simulation** .

Definition

```
G_Error_t
G_Net2Run_Signal_Config2(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Signal_Config2_CmdFlags_t  cmdFlags,
 const u32_t  numberOfSignals,
 const G_Net2Run_Signal_Config2_Signal_t * const net2RunSignals
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following parameters are possible and can be combined:

`G_API__NET2RUN__SIGNAL__CONFIG_2__CMD_FLAG__NONE`
No flag is set

`numberOfSignals`
Number of signals to be configured

`net2RunSignals`
Signal data array

The structure contains the following elements:

`Name`
Signal name

The signal name must not exceed 256 characters!

`ComId`
COM layer ID of the signal

`TransmissionMode`
Transmission mode

The following values are possible:

`G_NET2RUN__SIGNAL__TRANSMISSION_MODE__UNKNOWN`
The signal transmission mode is unknown

G_NET2RUN__SIGNAL__TRANSMISSION_MODE__TX
The signal is transmitted

G_NET2RUN__SIGNAL__TRANSMISSION_MODE__RX
The signal is received

reserved
reserved parameter (must be initialized with **0**)

Internal
Structure with parameters of the **internal** (raw) signal data

The structure contains the following elements:

Type
Signal type

The following values are possible:

G_NET2RUN__SIGNAL__TYPE__BOOLEAN
Boolean signal (value size = 1 byte)

G_NET2RUN__SIGNAL__TYPE__FLOAT32
32 bit floating point signal (value size = 4 bytes)

G_NET2RUN__SIGNAL__TYPE__FLOAT64
64 bit floating point signal (value size = 8 bytes)

G_NET2RUN__SIGNAL__TYPE__S8
Signed decimal 8 bit signal (value size = 1 byte)

G_NET2RUN__SIGNAL__TYPE__S16
Signed decimal 16 bit signal (value size = 2 bytes)

G_NET2RUN__SIGNAL__TYPE__S32
Signed decimal 32 bit signal (value size = 4 bytes)

G_NET2RUN__SIGNAL__TYPE__U8
Unsigned decimal 8 bit signal (value size = 1 byte)

G_NET2RUN__SIGNAL__TYPE__U16
Unsigned decimal 16 bit signal (value size = 2 bytes)

G_NET2RUN__SIGNAL__TYPE__U32
Unsigned decimal 32 bit signal (value size = 4 bytes)

G_NET2RUN__SIGNAL__TYPE__U8_N
Array of unsigned decimal 8 bit values (value size varies)

G_NET2RUN__SIGNAL__TYPE__U8_DYN
Array of unsigned decimal 8 bit value with dynamic length (value size = 0 bytes)

reserved1
reserved parameter (must be initialized with **0**)

reserved2
reserved parameter (must be initialized with **0**)

reserved3
reserved parameter (must be initialized with **0**)

MinValue
Minimum value of the **internal** signal range

`Maximum`
Maximum value of the **internal** signal range

`DefaultValue`
Internal default signal value

`Physical`
Structure with parameters of the **physical** (application) signal data

The structure contains the following elements:

`Type`
Signal type

The following values are possible:

`G_NET2RUN__SIGNAL__TYPE__BOOLEAN`
Boolean signal (value size = 1 byte)

`G_NET2RUN__SIGNAL__TYPE__FLOAT32`
32 bit floating point signal (value size = 4 bytes)

`G_NET2RUN__SIGNAL__TYPE__FLOAT64`
64 bit floating point signal (value size = 8 bytes)

`G_NET2RUN__SIGNAL__TYPE__S8`
Signed decimal 8 bit signal (value size = 1 byte)

`G_NET2RUN__SIGNAL__TYPE__S16`
Signed decimal 16 bit signal (value size = 2 bytes)

`G_NET2RUN__SIGNAL__TYPE__S32`
Signed decimal 32 bit signal (value size = 4 bytes)

`G_NET2RUN__SIGNAL__TYPE__U8`
Unsigned decimal 8 bit signal (value size = 1 byte)

`G_NET2RUN__SIGNAL__TYPE__U16`
Unsigned decimal 16 bit signal (value size = 2 bytes)

`G_NET2RUN__SIGNAL__TYPE__U32`
Unsigned decimal 32 bit signal (value size = 4 bytes)

`G_NET2RUN__SIGNAL__TYPE__U8_N`
Array of unsigned decimal 8 bit values (value size varies)

`G_NET2RUN__SIGNAL__TYPE__U8_DYN`
Array of unsigned decimal 8 bit value with dynamic length (value size = 0 bytes)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`reserved4`
reserved parameter (must be initialized with **0**)

`reserved5`
reserved parameter (must be initialized with **0**)

`reserved6`
reserved parameter (must be initialized with **0**)

`reserved7`
reserved parameter (must be initialized with **0**)

`MinValue`
Minimum value of the **physical** signal range

`Maximum`
Maximum value of the **physical** signal range

`Offset`
Calculation Offset

The calculation offset is used for calculating the **internal** (raw) value from a given **physical** value

(**Internal Value = (Physical Value - Offset) / Factor**)

or vice versa

(**Physical Value = (Internal Value * Factor) + Offset**).

`Factor`
see `Offset`

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Signal_Write

G_Net2Run_Signal_Write — Write signal value

Description

Use this function to write the value of a **Rest Bus Simulation** signal.

Parameter `mode` specifies whether the value entered is interpreted as **internal** or **physical** value.

While **physical data** is defined by **value** and **unit**, **internal data** represents the **raw data** on the bus.

When entering a **physical** value, it is converted to its **internal** equivalent automatically.

The function takes a **void pointer** for the value to be written. Therefore it is possible to pass different value types to this function. In return, it is necessary to pass the size of the value type in bytes. While the **physical** signal type mostly is **float64_t**, the **internal** signal type varies from signal to signal.

An example of how to write signal values can be found at the end of this section.

Definition

```
G_Error_t
G_Net2Run_Signal_Write(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Signal_Write_Mode_t  mode,
 const G_Net2Run_Signal_Write_CmdFlags_t  cmdFlags,
 const G_Net2Run_SignalHandle_t  signalHandle,
 const u16_t  valueSize,
 const void * const value
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
Specifies whether the value entered is interpreted as **internal** or **physical** value

The following values are possible:

`G_NET2RUN__SIGNAL__WRITE__MODE__INTERNAL`
The value will be interpreted as **internal** (raw data) and the data will be written without any conversion.

`G_NET2RUN__SIGNAL__WRITE__MODE__PHYSICAL`
The value will be interpreted as **physical** and the data will be converted to its **internal** (raw) representation.

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SIGNAL__WRITE__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__SIGNAL__WRITE__CMD_FLAG__DISREGARD_LIMITS`
Value limits will be disregarded

Each signal has its specified value range. When you enter a value that is out of this range, the signal won't be written and an error is returned. This behavior is overridden when this flag is set.

`signalHandle`
Signal handle

The signal handle is used to identify the signal. It is retrieved with function G_Net2Run_Signal_GetSignalHandle .

`valueSize`
Size of the value type in bytes (e.g. the size of type **float64_t** is **8 bytes**)

`value`
Signal value to be written

Depending on `mode`, the value needs to be in the format of the **internal** or the **physical** data type and `valueSize` needs to be set accordingly.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

Read / Write Example

```
#include <stdio.h>
#include <windows.h>
#include "g_api_common.h"
#include "g_api_net2run.h"


//------------------------------------------------------------------------
// function:    ErrorHandler
// description: evaluates error code and closes application if an error
//              occurred
// parameters:  portHandle - port handle
//              errorCode - error code
//------------------------------------------------------------------------
void
ErrorHandler(
 const G_PortHandle_t  portHandle,
 const G_Error_t  errorCode
)
{
 if (errorCode != G_NO_ERROR) {
  printf("Error: %s\n", G_GetLastErrorDescription());
  (void) G_Common_CloseInterface(portHandle);

  ExitProcess(1);
 }
}
```

```
//--------------------------------------------------------------------------
// function:    main
// description: main function
// parameters:  none
//--------------------------------------------------------------------------
void
main(
 void
)
{
 G_PortHandle_t  portHandle;

 // The name of the RBS File, built with the Net2Run GUI.
 const char * const configFilename = "TestProject.rbs";

 G_Net2Run_Pdu_PduGroupControl_Group_t  pduGroup;
 G_Net2Run_SignalHandle_t  userTestSignal01;
 float64_t  value64;
 u8_t  value8;
 G_Net2Run_Signal_Read_RspFlags_t  rspFlags;
 u16_t  valueSize;
 G_Error_t  rc;

 // Open a port to the Net2Run interface with name "NET2RUN1".
 rc =
  G_Common_OpenInterface(
   "NET2RUN1",
   &portHandle
  );

 ErrorHandler(
  portHandle,
  rc
 );

 // Init all interface parameters and restore default interface state.
 rc =
  G_Common_InitInterface(
   portHandle
  );

 ErrorHandler(
  portHandle,
  rc
 );

 // Configure Rest Bus Simulation from rbs file "TestProject.rbs",
 // built with the Net2Run GUI.
 rc =
  G_Net2Run_Config(
   portHandle,
   G_NET2RUN__CONFIG__CMD_FLAG__NONE,
   configFilename
  );
```

```
ErrorHandler(
 portHandle,
 rc
);

// Configure Rest Bus Simulation signals from rbs file "TestProject.rbs",
// built with the Net2Run GUI.
// This step is necessary for reading and writing signals.
rc =
 G_Net2Run_Signal_Config(
  portHandle,
  configFilename
 );

ErrorHandler(
 portHandle,
 rc
);

// Request communication for all ECUs
rc =
 G_Net2Run_ComManager_ComMode_Request(
  portHandle,
  G_NET2RUN__COM_MANAGER__COM_MODE__REQUEST__CMD_FLAG__ALL_USERS,
  0,
  G_NET2RUN__COM_MANAGER__COM_MODE__FULL_COMMUNICATION
 );

ErrorHandler(
 portHandle,
 rc
);

// Start PDU Groups.
// The group ID "0xFFFFFFFF" represents all available PDU Groups.
pduGroup.Flags = G_NET2RUN__PDU__PDU_GROUP_CONTROL__GROUP__FLAG__START;
pduGroup.PduGroupId = 0xFFFFFFFF;

rc =
 G_Net2Run_Pdu_PduGroupControl(
  portHandle,
  1,
  &pduGroup
 );

ErrorHandler(
 portHandle,
 rc
);

// Get the signal handle for the signal named "UserTestSignal01".
// This handle is used for read and write operations.
rc =
 G_Net2Run_Signal_GetSignalHandle(
  portHandle,
  "UserTestSignal01",
  &userTestSignal01
 );
```

```
ErrorHandler(
 portHandle,
 rc
);

// Write a physical signal value. The physical type of this signal is
// float64.
value64 = 102;
valueSize = sizeof(value64);

rc =
 G_Net2Run_Signal_Write(
  portHandle,
  G_NET2RUN__SIGNAL__WRITE__MODE__PHYSICAL,
  G_NET2RUN__SIGNAL__WRITE__CMD_FLAG__NONE,
  userTestSignal01,
  valueSize,
  &value64
 );

ErrorHandler(
 portHandle,
 rc
);

// Write an internal signal value. The physical type of this signal is u8.
value8 = 255;
valueSize = sizeof(value8);

rc =
 G_Net2Run_Signal_Write(
  portHandle,
  G_NET2RUN__SIGNAL__WRITE__MODE__INTERNAL,
  G_NET2RUN__SIGNAL__WRITE__CMD_FLAG__NONE,
  userTestSignal01,
  valueSize,
  &value8
 );

ErrorHandler(
 portHandle,
 rc
);

// Read the physical value.
valueSize = sizeof(value64);

rc =
 G_Net2Run_Signal_Read(
  portHandle,
  G_NET2RUN__SIGNAL__READ__MODE__PHYSICAL,
  G_NET2RUN__SIGNAL__READ__CMD_FLAG__NONE,
  userTestSignal01,
  &rspFlags,
  &valueSize,
  &value64
 );
```

```
 ErrorHandler(
  portHandle,
  rc
 );

 // Read the internal value.
 valueSize = sizeof(value8);

 rc =
  G_Net2Run_Signal_Read(
   portHandle,
   G_NET2RUN__SIGNAL__READ__MODE__INTERNAL,
   G_NET2RUN__SIGNAL__READ__CMD_FLAG__NONE,
   userTestSignal01,
   &rspFlags,
   &valueSize,
   &value8
  );

 ErrorHandler(
  portHandle,
  rc
 );

 // Close the port to the interface.
 rc = G_Common_CloseInterface(portHandle);

 ErrorHandler(
  portHandle,
  rc
 );
}
```

After configuring the **Rest Bus Simulation** with `G_Net2Run_Config` and configuring the signals with `G_Net2Run_Signal_Config`, the PDU groups are started with `G_Net2Run_Pdu_PduGroupControl`. At this point, the **Rest Bus Simulation** is running and the frames are on the bus.

In the next step, the signal handle for signal `UserTestSignal01` is queried. With the use of this handle, a **physical** and an **internal** value is written and subsequently read back.

G_Net2Run_Signal_Read

G_Net2Run_Signal_Read — Read signal value

Description

Use this function to read the value of a **Rest Bus Simulation** signal.

Parameter `mode` specifies whether to read the **internal** or the **physical** value.

While **physical data** is defined by **value** and **unit**, **internal data** represents the **raw data** on the bus.

The function takes a **void pointer** for the value to be read. Therefore it is possible to return different value types. In return, it is necessary to pass the size of the value buffer in bytes. While the **physical** signal type mostly is **float64_t**, the **internal** signal type varies from signal to signal.

For an example of how to read signal values, see Read / Write Example.

Definition

```
G_Error_t
G_Net2Run_Signal_Read(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Signal_Read_Mode_t  mode,
 const G_Net2Run_Signal_Read_CmdFlags_t  cmdFlags,
 const G_Net2Run_SignalHandle_t  signalHandle,
 G_Net2Run_Signal_Read_RspFlags_t * const rspFlags,
 u16_t * const valueSize,
 void * const value
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
Specifies whether to read the **internal** or the **physical** value

The following values are possible:

`G_NET2RUN__SIGNAL__READ__MODE__INTERNAL`
The **internal** (raw) value will be read

`G_NET2RUN__SIGNAL__READ__MODE__PHYSICAL`
The **physical** value will be read

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SIGNAL__READ__CMD_FLAG__NONE`
No flag is set

`signalHandle`
Signal handle

The signal handle is used to identify the signal. It is retrieved with function G_Net2Run_Signal_GetSignalHandle .

`rspFlags`
Returns response Flags

The following values are possible and can be combined:

`G_NET2RUN__SIGNAL__READ__RSP_FLAG__NONE`
No flag is set

`G_NET2RUN__SIGNAL__READ__RSP_FLAG__SIGNAL_INVALID`
The returned signal value is out of its specified range

`valueSize`
In : Size of buffer `value` in bytes

Out : Size of the returned signal value in bytes

`value`
Returns signal value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Signal_UpdateCallback_Set

G_Net2Run_Signal_UpdateCallback_Set — Activate callback function

Description

Use this function to activate a callback function that will be called when the specified signal is received (for RX signals) or the signal value has changed (after calling G_Net2Run_Signal_Write for TX signals).

When the callback function is not needed anymore, it can be reset with command G_Net2Run_Signal_UpdateCallback_Reset .

Definition

```
G_Error_t
G_Net2Run_Signal_UpdateCallback_Set(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Signal_Read_Mode_t  mode,
 const G_Net2Run_Signal_Read_CmdFlags_t  cmdFlags,
 const G_Net2Run_SignalHandle_t  signalHandle,
 const G_Net2Run_Signal_UpdateCallback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`mode`
Specifies whether to read the **internal** or the **physical** value

The following values are possible:

`G_NET2RUN__SIGNAL__READ__MODE__INTERNAL`
The **internal** (raw) value will be read

`G_NET2RUN__SIGNAL__READ__MODE__PHYSICAL`
The **physical** value will be read

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SIGNAL__READ__CMD_FLAG__NONE`
No flag is set

`signalHandle`
Signal handle

The signal handle is used to identify the signal. It is retrieved with function G_Net2Run_Signal_GetSignalHandle .

`callback`
Function pointer of the callback function that will be called if the specified signal is received (for RX signals) or the signal value has changed (after calling G_Net2Run_Signal_Write for TX signals)

The callback function needs to have the following parameters:

`mode`
Returns whether the value read is **internal** or **physical**

The following values are possible:

`G_NET2RUN__SIGNAL__READ__MODE__INTERNAL`
The **internal** (raw) value is read

`G_NET2RUN__SIGNAL__READ__MODE__PHYSICAL`
The **physical** value is read

`signalHandle`
Signal handle

The signal handle is used to identify the signal.

`rspFlags`
Returns response Flags

The following values are possible and can be combined:

`G_NET2RUN__SIGNAL__READ__RSP_FLAG__NONE`
No flag is set

`G_NET2RUN__SIGNAL__READ__RSP_FLAG__SIGNAL_INVALID`
The returned signal value is out of its specified range

`valueSize`
Size of the returned signal value in bytes

`value`
Returns signal value

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
#include <stdio.h>
#include <windows.h>
#include "g_api_common.h"
#include "g_api_net2run.h"


//----------------------------------------------------------------------------
// function:    ErrorHandler
// description: evaluates error code and closes application if an error
//              occurred
// parameters:  portHandle - port handle
//              errorCode - error code
//----------------------------------------------------------------------------
static void
ErrorHandler(
 const G_PortHandle_t  portHandle,
 const G_Error_t  errorCode
)
{
 if (errorCode != G_NO_ERROR) {
  printf("Error: %s\n", G_GetLastErrorDescription());
  (void) G_Common_CloseInterface(portHandle);
```

```
  ExitProcess(1);
 }
}


//---------------------------------------------------------------------------
// function:    UpdateCallbackFunction
// description: signal update callback function
// parameters:  portHandle - port handle
//              mode - signal read mode (internal or physical)
//              rspFlags - response flags
//              valueSize - size of signal value type in bytes
//              value - signal value
//---------------------------------------------------------------------------
static void
G_API
UpdateCallbackFunction(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_Signal_Read_Mode_t  mode,
 const G_Net2Run_SignalHandle_t  signalHandle,
 const G_Net2Run_Signal_Read_RspFlags_t  rspFlags,
 const u16_t  valueSize,
 const void * const value
)
{
 // This is the update callback function, that is called when a signal is
 // received or a signal value has been written.
 G_Net2Run_SignalInfo_t  info;
 G_Net2Run_Signal_Type_t  type;
 G_Error_t  rc;

 rc =
  G_Net2Run_Signal_GetSignalInfo(
   portHandle,
   signalHandle,
   &info
  );

 ErrorHandler(
  portHandle,
  rc
 );

 switch (mode) {
  case G_NET2RUN__SIGNAL__READ__MODE__INTERNAL:
   type = info.Internal.Type;
   break;

  case G_NET2RUN__SIGNAL__READ__MODE__PHYSICAL:
   type = info.Physical.Type;
   break;

  default:
   return;
 }

 printf("Signal value: ");
```

```
 switch (type) {
  case G_NET2RUN__SIGNAL__TYPE__BOOLEAN:
  case G_NET2RUN__SIGNAL__TYPE__U8:
   printf("%u", *(u8_t *) value);
   break;

  case G_NET2RUN__SIGNAL__TYPE__FLOAT32:
   printf("%f", *(float32_t *) value);
   break;

  case G_NET2RUN__SIGNAL__TYPE__FLOAT64:
   printf("%f", *(float64_t *) value);
   break;

  case G_NET2RUN__SIGNAL__TYPE__S8:
   printf("%d", *(s8_t *) value);
   break;

  case G_NET2RUN__SIGNAL__TYPE__S16:
   printf("%d", *(s16_t *) value);
   break;

  case G_NET2RUN__SIGNAL__TYPE__S32:
   printf("%d", *(s32_t *) value);
   break;

  case G_NET2RUN__SIGNAL__TYPE__U16:
   printf("%u", *(u16_t *) value);
   break;

  case G_NET2RUN__SIGNAL__TYPE__U32:
   printf("%u", *(u32_t *) value);
   break;

  default:
   printf("unknown");
   break;
 }

 printf("\n");
}


//-----------------------------------------------------------------------------
// function:    main
// description: main function
// parameters:  none
//-----------------------------------------------------------------------------
void
main(
 void
)
{
 G_PortHandle_t  portHandle;

 // The name of the RBS File, built with the Net2Run GUI.
 const char * const configFilename = "TestProject.rbs";
```

```
G_Net2Run_Pdu_PduGroupControl_Group_t  pduGroup;
G_Net2Run_SignalHandle_t  userTestSignal01;
float64_t  value64;
u8_t  value8;
u16_t  valueSize;
G_Error_t  rc;

// Open a port to the Net2Run interface with name "NET2RUN1".
rc =
 G_Common_OpenInterface(
  "NET2RUN1",
  &portHandle
 );

ErrorHandler(
 portHandle,
 rc
);

// Init all interface parameters and restore default interface state.
rc =
 G_Common_InitInterface(
  portHandle
 );

ErrorHandler(
 portHandle,
 rc
);

// Configure Rest Bus Simulation from rbs file "TestProject.rbs",
// built with the Net2Run GUI.
rc =
 G_Net2Run_Config(
  portHandle,
  G_NET2RUN__CONFIG__CMD_FLAG__NONE,
  configFilename
 );

ErrorHandler(
 portHandle,
 rc
);

// Configure Rest Bus Simulation signals from rbs file "TestProject.rbs",
// built with the Net2Run GUI.
// This step is necessary for reading and writing signals.
rc =
 G_Net2Run_Signal_Config(
  portHandle,
  configFilename
 );

ErrorHandler(
 portHandle,
 rc
);
```

```
// Request communication for all ECUs
rc =
 G_Net2Run_ComManager_ComMode_Request(
  portHandle,
  G_NET2RUN__COM_MANAGER__COM_MODE__REQUEST__CMD_FLAG__ALL_USERS,
  0,
  G_NET2RUN__COM_MANAGER__COM_MODE__FULL_COMMUNICATION
 );

ErrorHandler(
 portHandle,
 rc
);

// Start PDU Groups.
// The group ID "0xFFFFFFFF" represents all available PDU Groups.
pduGroup.Flags = G_NET2RUN__PDU__PDU_GROUP_CONTROL__GROUP__FLAG__START;
pduGroup.PduGroupId = 0xFFFFFFFF;

rc =
 G_Net2Run_Pdu_PduGroupControl(
  portHandle,
  1,
  &pduGroup
 );

ErrorHandler(
 portHandle,
 rc
);

// Get the signal handle for the signal named "UserTestSignal01".
// This handle is used for read and write operations.
rc =
 G_Net2Run_Signal_GetSignalHandle(
  portHandle,
  "UserTestSignal01",
  &userTestSignal01
 );

ErrorHandler(
 portHandle,
 rc
);

// Set up update callback.
// This function will be called when the specified signal is received or
// written.
rc =
 G_Net2Run_Signal_UpdateCallback_Set(
  portHandle,
  G_NET2RUN__SIGNAL__WRITE__MODE__PHYSICAL,
  G_NET2RUN__SIGNAL__READ__CMD_FLAG__NONE,
  userTestSignal01,
  &UpdateCallbackFunction
 );
```

```
ErrorHandler(
 portHandle,
 rc
);

// Write a physical signal value. The physical type of this signal is
// float64.
value64 = 102;
valueSize = sizeof(value64);

rc =
 G_Net2Run_Signal_Write(
  portHandle,
  G_NET2RUN__SIGNAL__WRITE__MODE__PHYSICAL,
  G_NET2RUN__SIGNAL__WRITE__CMD_FLAG__NONE,
  userTestSignal01,
  valueSize,
  &value64
 );

ErrorHandler(
 portHandle,
 rc
);

// Write an internal signal value. The physical type of this signal is u8.
value8 = 200;
valueSize = sizeof(value8);
rc =
 G_Net2Run_Signal_Write(
  portHandle,
  G_NET2RUN__SIGNAL__WRITE__MODE__INTERNAL,
  G_NET2RUN__SIGNAL__WRITE__CMD_FLAG__NONE,
  userTestSignal01,
  valueSize,
  &value8
 );

ErrorHandler(
 portHandle,
 rc
);

// The update callback function should be called twice
// (once for every write action).
Sleep(1000);

// Reset update callback function
rc =
 G_Net2Run_Signal_UpdateCallback_Reset(
  portHandle,
  userTestSignal01
 );

ErrorHandler(
 portHandle,
 rc
);
```

```
 // Close the port to the interface.
 rc = G_Common_CloseInterface(portHandle);

 ErrorHandler(
  portHandle,
  rc
 );
}
```

In this example, the update callback function `UpdateCallbackFunction` is used to print the value of the signal when it is received or written.

G_Net2Run_Signal_UpdateCallback_Reset

G_Net2Run_Signal_UpdateCallback_Reset — Reset update callback function

Description

Use this function to reset (deactivate) the update callback function for a signal.

Definition

```
G_Error_t
G_Net2Run_Signal_UpdateCallback_Reset(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SignalHandle_t  signalHandle
);
```

Parameters

`portHandle`
Handle to the communication port

`signalHandle`
Signal handle

The signal handle is used to identify the signal. It is retrieved with function G_Net2Run_Signal_GetSignalHandle .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Signal_GetSignalList

G_Net2Run_Signal_GetSignalList — Query signal information

Description

Use this function to query signal information of all available signals.

Definition

```
G_Error_t
G_Net2Run_Signal_GetSignalList(
 const G_PortHandle_t  portHandle,
 u32_t * const numberOfSignals,
 G_Net2Run_SignalInfo_t * const net2RunSignals
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfSignals`
In : Number of signal information the buffer `signals` can be filled with

The number of available signals can be queried with G_Net2Run_Signal_GetNumberOfSignals .

Out : Number of signal information the buffer `signals` contains

`net2RunSignals`
Returns list with signal information

A list element has the following structure:

`Name`
Signal name

`Offset`
Calculation Offset

The calculation offset is used for calculating the **internal** (raw) value from a given **physical** value

(**Internal Value = (Physical Value - Offset) / Factor**)

or vice versa

(**Physical Value = (Internal Value * Factor) + Offset**).

`Factor`
see `Offset`

`Internal`
Structure with parameters of the **internal** (raw) signal data

The structure contains the following elements:

`MinValue`
Minimum value of the **internal** signal range

`Maximum`
Maximum value of the **internal** signal range

`DefaultValue`
Internal default signal value

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Type`
Internal signal type

The following values are possible:

`G_NET2RUN__SIGNAL__TYPE__BOOLEAN`
Boolean signal (value size = 1 byte)

`G_NET2RUN__SIGNAL__TYPE__FLOAT32`
32 bit floating point signal (value size = 4 bytes)

`G_NET2RUN__SIGNAL__TYPE__FLOAT64`
64 bit floating point signal (value size = 8 bytes)

`G_NET2RUN__SIGNAL__TYPE__S8`
Signed decimal 8 bit signal (value size = 1 byte)

`G_NET2RUN__SIGNAL__TYPE__S16`
Signed decimal 16 bit signal (value size = 2 bytes)

`G_NET2RUN__SIGNAL__TYPE__S32`
Signed decimal 32 bit signal (value size = 4 bytes)

`G_NET2RUN__SIGNAL__TYPE__U8`
Unsigned decimal 8 bit signal (value size = 1 byte)

`G_NET2RUN__SIGNAL__TYPE__U16`
Unsigned decimal 16 bit signal (value size = 2 bytes)

`G_NET2RUN__SIGNAL__TYPE__U32`
Unsigned decimal 32 bit signal (value size = 4 bytes)

`G_NET2RUN__SIGNAL__TYPE__U8_N`
Array of unsigned decimal 8 bit values (value size varies)

`G_NET2RUN__SIGNAL__TYPE__U8_DYN`
Array of unsigned decimal 8 bit value with dynamic length (value size = 0 bytes)

`reserved3`
reserved parameter (must be initialized with **0**)

`Size`
Size of internal type, in bytes

`Physical`
Structure with parameters of the **physical** signal data

The structure contains the following elements:

`MinValue`
Minimum value of the **physical** signal range

`Maximum`
Maximum value of the **physical** signal range

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`Type`
Physical signal type

The following values are possible:

`G_NET2RUN__SIGNAL__TYPE__BOOLEAN`
Boolean signal (value size = 1 byte)

`G_NET2RUN__SIGNAL__TYPE__FLOAT32`
32 bit floating point signal (value size = 4 bytes)

`G_NET2RUN__SIGNAL__TYPE__FLOAT64`
64 bit floating point signal (value size = 8 bytes)

`G_NET2RUN__SIGNAL__TYPE__S8`
Signed decimal 8 bit signal (value size = 1 byte)

`G_NET2RUN__SIGNAL__TYPE__S16`
Signed decimal 16 bit signal (value size = 2 bytes)

`G_NET2RUN__SIGNAL__TYPE__S32`
Signed decimal 32 bit signal (value size = 4 bytes)

`G_NET2RUN__SIGNAL__TYPE__U8`
Unsigned decimal 8 bit signal (value size = 1 byte)

`G_NET2RUN__SIGNAL__TYPE__U16`
Unsigned decimal 16 bit signal (value size = 2 bytes)

`G_NET2RUN__SIGNAL__TYPE__U32`
Unsigned decimal 32 bit signal (value size = 4 bytes)

`G_NET2RUN__SIGNAL__TYPE__U8_N`
Array of unsigned decimal 8 bit values (value size varies)

`G_NET2RUN__SIGNAL__TYPE__U8_DYN`
Array of unsigned decimal 8 bit value with dynamic length (value size = 0 bytes)

`reserved3`
reserved parameter (must be initialized with **0**)

`Size`
Size of physical type, in bytes

`reserved4`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Signal_GetSignalInfo

G_Net2Run_Signal_GetSignalInfo — Query signal information for a specific signal

Description

Use this function to query signal information for a signal, specified by its signal handle.

Definition

```
G_Error_t
G_Net2Run_Signal_GetSignalInfo(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SignalHandle_t  signalHandle,
 G_Net2Run_SignalInfo_t * const info
);
```

Parameters

`portHandle`
Handle to the communication port

`signalHandle`
Signal handle

The signal handle is used to identify the signal. It is retrieved with function G_Net2Run_Signal_GetSignalHandle .

`info`
Returns signal information (see Net2Run Signal Info Structure)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Signal_GetSignalTransmissionMode

G_Net2Run_Signal_GetSignalTransmissionMode — Query signal transmission mode

Description

Use this function to query the transmission mode of a signal.

Definition

```
G_Error_t
G_Net2Run_Signal_GetSignalTransmissionMode(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SignalHandle_t  signalHandle,
 G_Net2Run_Signal_TransmissionMode_t * const transmissionMode
);
```

Parameters

`portHandle`
Handle to the communication port

`signalHandle`
Signal handle

The signal handle is used to identify the signal. It is retrieved with function G_Net2Run_Signal_GetSignalHandle .

`transmissionMode`
Returns signal transmission mode

The following values are possible:

`G_NET2RUN__SIGNAL__TRANSMISSION_MODE__UNKNOWN`
The signal transmission mode is unknown

`G_NET2RUN__SIGNAL__TRANSMISSION_MODE__TX`
The signal is transmitted

`G_NET2RUN__SIGNAL__TRANSMISSION_MODE__RX`
The signal is received

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Signal_GetSignalHandle

G_Net2Run_Signal_GetSignalHandle — Query a signal handle

Description

Use this function to query the handle to a signal by its name.

For an example, see Read / Write Example.

Definition

```
G_Error_t
G_Net2Run_Signal_GetSignalHandle(
 const G_PortHandle_t  portHandle,
 const char * const signalName,
 G_Net2Run_SignalHandle_t * const signalHandle
);
```

Parameters

`portHandle`
Handle to the communication port

`signalName`
Name of the signal

`signalHandle`
Returns signal handle

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_Signal_GetNumberOfSignals

G_Net2Run_Signal_GetNumberOfSignals — Query the number of available signals

Description

Use this function to query the number of available signals.

This can be useful to determine the size of the signal buffer before calling G_Net2Run_Signal_GetSignalList .

Definition

```
G_Error_t
G_Net2Run_Signal_GetNumberOfSignals(
 const G_PortHandle_t  portHandle,
 u32_t * const numberOfSignals
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfSignals`
Returns number of available signals

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

20 Signal Block

Functions for controlling Signal Blocks of a Net2Run **Rest Bus Simulation**

G_Net2Run_SignalBlock_Config

G_Net2Run_SignalBlock_Config — Configure Net2Run Signal Block

Description

Use this command to configure a Net2Run Signal Block.

Definition

```
G_Error_t
G_Net2Run_SignalBlock_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SignalBlock_Config_CmdFlags_t  cmdFlags,
 G_Net2Run_SignalBlock_t ** signalBlock,
 const u32_t  numberOfSignals,
 const G_Net2Run_SignalBlock_Signals_t * const signals
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SIGNAL_BLOCK__CONFIG__CMD_FLAG__NONE`
No flag is set

`signalBlock`
Returns signal block handle

`numberOfSignals`
Number of signals the signal block contains

`signals`
Signal information of the signal block signals

The structure contains the following elements:

`SignalHandle`
Signal handle of the signal block signal

Can be obtained with G_Net2Run_Signal_GetSignalHandle

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`SignalFlags`
Signal flags

The following values are possible and can be combined:

`G_NET2RUN__SIGNAL_BLOCK__SIGNAL_FLAG__NONE`
No flag is set

`G_NET2RUN__SIGNAL_BLOCK__SIGNAL_FLAG__USE_INTERNAL_VALUE`
not set : The physical signal value is used for reading and writing (mostly `float64_t`)

set : The internal signal value is used for reading and writing (different types possible)

`G_NET2RUN__SIGNAL_BLOCK__SIGNAL_FLAG__USE_TIME_STAMP`
not set : The signal's timestamp is not included in rsp stream when reading

set : The signal's timestamp (64 bit) is put before signal length or signal data when reading

`G_NET2RUN__SIGNAL_BLOCK__SIGNAL_FLAG__USE_SIGNAL_LENGTH`
not set : The signal length is not included in cmd/rsp stream (exception: the signal length for signals of type UINT8_DYN is always included, so for these signals this flag is ignored)

set : The signal length is included in cmd/rsp stream before signal data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SignalBlock_Delete

G_Net2Run_SignalBlock_Delete — Delete Net2Run Signal Block

Description

Use this command to delete a Net2Run Signal Block and free its resources.

Definition

```
G_Error_t
G_Net2Run_SignalBlock_Delete(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SignalBlock_Delete_CmdFlags_t  cmdFlags,
 G_Net2Run_SignalBlock_t ** signalBlock
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SIGNAL_BLOCK__DELETE__CMD_FLAG__NONE`
No flag is set

signalBlock
Signal block to be deleted

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SignalBlock_Write

G_Net2Run_SignalBlock_Write — Write Net2Run Signal Block values

Description

Use this command to write Net2Run Signal Block values

Definition

```
G_Error_t
G_Net2Run_SignalBlock_Write(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SignalBlock_t * const signalBlock,
 const u8_t * const stream
);
```

Parameters

`portHandle`
Handle to the communication port

`signalBlock`
Signal block handle

`stream`
Stream with signal data

The stream contains the following data structure:

```
struct {
 u8_t  SignalLength;  // if signal flag "USE_SIGNAL_LENGTH" is set
 u8_t  SignalData[SignalLength];
} Signals[numberOfSignals];
```

`SignalLength`
Length of the signal in bytes

Only needed when signal flag `G_NET2RUN__SIGNAL_BLOCK__SIGNAL_FLAG__USE_SIGNAL_LENGTH` is set or when signal type is **`UINT8_DYN`**.

`SignalData`
Signal data

If `SignalLength` is declared, the size of the signal data corresponds to `SignalLength`.

If signal flag `G_NET2RUN__SIGNAL_BLOCK__SIGNAL_FLAG__USE_INTERNAL_VALUE` is set, the size of the signal data corresponds to the size of the internal signal type.

If signal flag `G_NET2RUN__SIGNAL_BLOCK__SIGNAL_FLAG__USE_INTERNAL_VALUE` is **not** set, the size of the signal data corresponds to the size of the physical signal type.

G_Net2Run_SignalBlock_Read

G_Net2Run_SignalBlock_Read — Read Net2Run Signal Block signals

Description

Use this command to read Net2Run signal block signals

Definition

```
G_Error_t
G_Net2Run_SignalBlock_Read(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SignalBlock_t * const signalBlock,
 u8_t * const stream
);
```

Parameters

`portHandle`
Handle to the communication port

`signalBlock`
Signal block handle

`stream`
Returns stream with signal data

The stream contains the following data structure:

```
struct {
 u64_t  TimeStamp;  // if signal flag "USE_TIME_STAMP" is set
 u8_t  SignalLength;  // if signal flag "USE_SIGNAL_LENGTH" is set
 u8_t  SignalData[SignalLength];
} Signals[NumberOfSignals];
```

`TimeStamp`
Only present when signal flag `G_NET2RUN__SIGNAL_BLOCK__SIGNAL_FLAG__USE_TIME_STAMP` is set.

`SignalLength`
Length of the signal in bytes

Only present when signal flag `G_NET2RUN__SIGNAL_BLOCK__SIGNAL_FLAG__USE_SIGNAL_LENGTH` is set or signal type is **`UINT8_DYN`**.

`SignalData`
Signal data

If signal flag `G_NET2RUN__SIGNAL_BLOCK__SIGNAL_FLAG__USE_SIGNAL_LENGTH` is set or signal type is **`UINT8_DYN`** the size of the signal data corresponds to `SignalLength`.

If signal flag `G_NET2RUN__SIGNAL_BLOCK__SIGNAL_FLAG__USE_INTERNAL_VALUE` is set, the size of the signal data corresponds to the size of the internal signal type.

If signal flag `G_NET2RUN__SIGNAL_BLOCK__SIGNAL_FLAG__USE_INTERNAL_VALUE` is **not** set, the size of the signal data corresponds to the size of the physical signal type.

Depending on its content, the data structure of a signal block entry may not be aligned. Therefore attention should be paid when accessing the different elements of the structure. If you can not guarantee that your signal bock configuration always delivers aligned data, it is adviseable to access the signal block data byte by byte.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SignalBlock_Buffer_Config

G_Net2Run_SignalBlock_Buffer_Config — Signal Block Buffer configuration

Description

Use this command for configuration of the Signal Block Buffer.

The Signal Block Buffer can be used to automatically poll and store signal block data in an internal buffer. This data can then be read and processed later.

Definition

```
G_Error_t
G_Net2Run_SignalBlock_Buffer_Config(
 const G_PortHandle_t  portHandle,
 G_Net2Run_SignalBlock_t * const signalBlock,
 const G_Net2Run_SignalBlock_Buffer_Config_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`signalBlock`
Signal block handle

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SIGNAL_BLOCK__BUFFER__CONFIG__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__SIGNAL_BLOCK__BUFFER__CONFIG__CMD_FLAG__START`
0 : Only configure the signal block buffer

1 : Configure and start the signal block buffer

`BufferSize`
Size of the internal Signal Block Buffer in bytes

0 = default size (10 Mb)

If the signal block data exceeds the buffer size, the buffer overrun flag will be set and the data will be lost.

`PollingCycle`
Polling cycle for signal block data in milliseconds

0 = default (5 ms)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SignalBlock_Buffer_Read

G_Net2Run_SignalBlock_Buffer_Read — Read Signal Block Data from buffer

Description

Use this command to read signal block data from the internal signal block data buffer.

Each command call will return one signal block data instance.

The signal block data buffer can be configured with G_Net2Run_SignalBlock_Buffer_Config .

Definition

```
G_Error_t
G_Net2Run_SignalBlock_Buffer_Read(
 const G_PortHandle_t  portHandle,
 G_Net2Run_SignalBlock_t * const signalBlock,
 const G_Net2Run_SignalBlock_Buffer_Read_CmdFlags_t  cmdFlags,
 G_Net2Run_SignalBlock_Buffer_Read_RspFlags_t * const rspFlags,
 u8_t * const stream
);
```

Parameters

`portHandle`
Handle to the communication port

`signalBlock`
Signal block handle

`cmdFlags`
Command flags

The following values are possible:

`G_NET2RUN__SIGNAL_BLOCK__BUFFER__READ__CMD_FLAG__NONE`
No flag is set

`rspFlags`
Response flags

The following values are possible and can be combined:

`G_NET2RUN__SIGNAL_BLOCK__BUFFER__READ__RSP_FLAG__NONE`
No flag is set

`G_NET2RUN__SIGNAL_BLOCK__BUFFER__READ__RSP_FLAG__BUFFER_OVERRUN`
A Buffer overrun occurred, data is lost.

`G_NET2RUN__SIGNAL_BLOCK__BUFFER__READ__RSP_FLAG__DATA_AVAILABLE`
More data is available in the internal signal block buffer.

`stream`
Returns Signal Block data

The stream contains the following data structure:

```
struct {
 u64_t  TimeStamp;  // if signal flag "USE_TIME_STAMP" is set
 u8_t  SignalLength;  // if signal flag "USE_SIGNAL_LENGTH" is set
 u8_t  SignalData[SignalLength];
} Signals[NumberOfSignals];
```

`TimeStamp`
Only present when signal flag `G_NET2RUN__SIGNAL_BLOCK__SIGNAL_FLAG__USE_TIME_STAMP` is set.

`SignalLength`
Length of the signal in bytes

Only present when signal flag `G_NET2RUN__SIGNAL_BLOCK__SIGNAL_FLAG__USE_SIGNAL_LENGTH` is set or signal type is **UINT8_DYN** .

`SignalData`
Signal data

If signal flag `G_NET2RUN__SIGNAL_BLOCK__SIGNAL_FLAG__USE_SIGNAL_LENGTH` is set or signal type is **UINT8_DYN** the size of the signal data corresponds to `SignalLength`.

If signal flag `G_NET2RUN__SIGNAL_BLOCK__SIGNAL_FLAG__USE_INTERNAL_VALUE` is set, the size of the signal data corresponds to the size of the internal signal type.

If signal flag `G_NET2RUN__SIGNAL_BLOCK__SIGNAL_FLAG__USE_INTERNAL_VALUE` is **not** set, the size of the signal data corresponds to the size of the physical signal type.

Depending on its content, the data structure of a signal block entry may not be aligned. Therefore attention should be paid when accessing the different elements of the structure. If you can not guarantee that your signal bock configuration always delivers aligned data, it is adviseable to access the signal block data byte by byte.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

21 Signal Group

Functions for controlling **Signal Groups** of a Net2Run **Rest Bus Simulation**

Signal Groups are used to bundle multiple signals that can be read or be written with a single command.

G_Net2Run_SignalGroup_TxPdu_Config

G_Net2Run_SignalGroup_TxPdu_Config — Configuration of a signal group within a TX PDU

Description

Use this command to build a signal group within a TX PDU.

Definition

```
G_Error_t
G_Net2Run_SignalGroup_TxPdu_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SignalGroup_TxPdu_Config_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SIGNAL_GROUP__TX_PDU__CONFIG__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__SIGNAL_GROUP__TX_PDU__CONFIG__CMD_FLAG__USE_UPDATE_BIT`
0 : the signal group has no update bit

1 : the signal group has an update bit at `UpdateBitPosition`

`TxPduId`
ID of the TX PDU in COM layer

`TxPduSignalGroupId`
ID of the signal group (0..0xFFFFFFFF within TX PDU)

Here you can choose the ID that identifies the signal group.

`UpdateBitPosition`
Bit position of update-bit inside I-PDU (only relevant if flag `G_NET2RUN__SIGNAL_GROUP__TX_PDU__CONFIG__CMD_FLAG__USE_UPDATE_BIT` is set)

`TransferProperty`
Transfer property of the signal group

The following values are possible:

`G_NET2RUN__COM__TRANSFER_PROPERTY__TRIGGERED`
When a TX signal with this transfer property is changed, the COM layer initiates an immediate transmission of the TX PDU, except if the defined transmission mode (see "G_Net2Run_Com_TxMode_t") for the TX PDU is configured to PERIODIC or NONE

`G_NET2RUN__COM__TRANSFER_PROPERTY__PENDING`
When a TX signal with this transfer property is changed, the COM layer doesn't initiate an immediate transmission of the TX PDU. The transmission keeps pending until an other TX signal within the same TX PDU with transfer property TRIGGERED is changed or the timer for periodic transmission expires in case of the TX PDU's transmission mode PERIODIC or MIXED (see "G_Net2Run_Com_TxMode_t").

`G_NET2RUN__COM__TRANSFER_PROPERTY__TRIGGERED_ON_CHANGE`
Depending on the transmission mode, a write access to a signal with this transfer property can trigger the transmission of the corresponding tx PDU, but only in case the written value is different to the locally stored (last sent or initial value) in length or value.

AUTOSAR name: ComTransferProperty::TRIGGERED_ON_CHANGE

`G_NET2RUN__COM__TRANSFER_PROPERTY__TRIGGERED_ON_CHANGE_WITHOUT_REP`
Depending on the transmission mode, a write access to a signal with this transfer property can trigger the transmission of the corresponding I-PDU just once without a repetition, but only in case the written value is different to the locally stored (last sent or initial value) in length or value.

AUTOSAR name:
ComTransferProperty::TRIGGERED_ON_CHANGE_WITHOUT_REPETITION

`G_NET2RUN__COM__TRANSFER_PROPERTY__TRIGGERED_WITHOUT_REP`
Depending on the transmission mode, a write access to a signal with this transfer property can trigger the transmission of the corresponding I-PDU just once without a repetition.

AUTOSAR name: ComTransferProperty::TRIGGERED_WITHOUT_REPETITION

`reserved`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SignalGroup_TxPdu_Mapping

G_Net2Run_SignalGroup_TxPdu_Mapping — Map a signal to the signal group

Description

Use this command to map a signal to the signal group of a TX PDU.

After the signal is mapped to the signal group, the whole signal group can be read or be written by reading or writing the signal.

The signal needs to be configured with "G_Net2Run_Com_Signal_Config" first and the signal type has to be `G_NET2RUN__SIGNAL__TYPE__U8_N`.

Definition

```
G_Error_t
G_Net2Run_SignalGroup_TxPdu_Mapping(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SignalGroup_TxPdu_Mapping_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SIGNAL_GROUP__TX_PDU__MAPPING__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__SIGNAL_GROUP__TX_PDU__MAPPING__CMD_FLAG__UPDATE_NOTIFICA-TION`
0 : disable signal update notification

1 : enable signal update notification: the signal is notified as updated when the signal group was transmitted

`SignalId`
ID of the signal that is mapped to the signal group

`SignalMappingId`
ID of the signal mapping

Has to be within the range of maximum number of mapping for this signal (specified by "G_Net2Run_Com_Signal_Config").

`TxPduId`
ID of the TX PDU that contains the signal group

`TxPduSignalGroupId`
ID of the signal group within the TX PDU (as specified by G_Net2Run_SignalGroup_TxPdu_Config)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SignalGroup_RxPdu_Config

G_Net2Run_SignalGroup_RxPdu_Config — Configuration of a signal group within an RX PDU

Description

Use this command to build a signal group within an RX PDU.

Definition

```
G_Error_t
G_Net2Run_SignalGroup_RxPdu_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SignalGroup_RxPdu_Config_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SIGNAL_GROUP__RX_PDU__CONFIG__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__SIGNAL_GROUP__RX_PDU__CONFIG__CMD_FLAG__USE_UPDATE_BIT`
0 : the signal group has no update bit

1 : the signal group has an update bit at `UpdateBitPosition`

`G_NET2RUN__SIGNAL_GROUP__RX_PDU__CONFIG__CMD_FLAG__UPDATE_WITHOUT_UPDATE_BIT`
This flag is only relevant if flag `G_NET2RUN__SIGNAL_GROUP__RX_PDU__CONFIG__CMD_FLAG__USE_UPDATE_BIT` is set

0 : if the byte which contains the update bit is not received, but the signal group bytes are received (all group signals of the signal group are received), the signal group is not updated

1 : if the byte which contains the update bit is not received, but the signal group bytes are received (all group signals of the signal group are received), the signal group is updated

`RxPduId`
ID of the RX PDU in COM layer

`RxPduSignalGroupId`
ID of the signal group (0..0xFFFFFFFF within RX PDU)

Here you can choose the ID that identifies the signal group.

`UpdateBitPosition`
Bit position of update-bit inside I-PDU (only relevant if flag `G_NET2RUN__SIGNAL_GROUP__RX_PDU__CONFIG__CMD_FLAG__USE_UPDATE_BIT` is set)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SignalGroup_RxPdu_Mapping

G_Net2Run_SignalGroup_RxPdu_Mapping — Map a signal to the signal group

Description

Use this command to map a signal to the signal group of an RX PDU.

After the signal is mapped to the signal group, the whole signal group can be read or be written by reading or writing the signal.

The signal needs to be configured with "G_Net2Run_Com_Signal_Config" first and the signal type has to be `G_NET2RUN__SIGNAL__TYPE__U8_N`.

Definition

```
G_Error_t
G_Net2Run_SignalGroup_RxPdu_Mapping(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SignalGroup_RxPdu_Mapping_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SIGNAL_GROUP__RX_PDU__MAPPING__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__SIGNAL_GROUP__RX_PDU__MAPPING__CMD_FLAG__UPDATE_NOTIFICA-TION`
0 : disable signal update notification

1 : enable signal update notification: the signal is notified as updated when the signal group was received

`G_NET2RUN__SIGNAL_GROUP__RX_PDU__MAPPING__CMD_FLAG__COPY_TO_SHADOW_ON_WRITE`
1 : all contained group signal values are copied to the corresponding shadow buffers on a write to this signal mapping

The same effect can be achieved by calling G_Net2Run_SignalGroup_RxPdu_Read for the specified signal group.

`G_NET2RUN__SIGNAL_GROUP__RX_PDU__MAPPING__CMD_FLAG__COPY_TO_SHADOW_ON_READ`
1 : all contained group signal values are copied to the corresponding shadow buffers when a read operation is performed

The same effect can be achieved by calling
G_Net2Run_SignalGroup_RxPdu_Read for the specified signal group.

`SignalId`
ID of the signal that is mapped to the signal group

`SignalMappingId`
ID of the signal mapping

Has to be within the range of maximum number of mapping for this signal (specified by "G_Net2Run_Com_Signal_Config").

`RxPduId`
ID of the RX PDU that contains the signal group

`RxPduSignalGroupId`
ID of the signal group within the RX PDU (as specified by G_Net2Run_SignalGroup_RxPdu_Config)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SignalGroup_TxPdu_Write

G_Net2Run_SignalGroup_TxPdu_Write — Write a signal group of a TX PDU

Description

Use this command to write a signal group of a TX PDU.

Definition

```
G_Error_t
G_Net2Run_SignalGroup_TxPdu_Write(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SignalGroup_TxPdu_Write_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`TxPduId`
ID of the TX PDU containing the signal group

`TxPduSignalGroupId`
ID of the signal group within TX PDU

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SignalGroup_RxPdu_Read

G_Net2Run_SignalGroup_RxPdu_Read — Read a signal group of an RX PDU

Description

Use this command to read a signal group of an RX PDU.

The values of the signals in the signal group are written to the shadow buffers.

Definition

```
G_Error_t
G_Net2Run_SignalGroup_RxPdu_Read(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SignalGroup_RxPdu_Read_Cmd_t * const cmd
);
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`RxPduId`
ID of the RX PDU containing the signal group

`RxPduSignalGroupId`
ID of the signal group within RX PDU

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

22 Socket Adaptor for Ethernet

Functions for controlling **Socket Adaptor** of a Net2Run **Rest Bus Simulation**

The Socket Adaptor is used to connect the AUTOSAR communication mechanisms and patterns to the ones of TCP/IP. While AUTOSAR is based on a static approach, TCP/IP uses dynamic configurations and routing. The Socket Adaptor module links the AUTOSAR-World to the Socket-World used in Ethernet communication.

G_Net2Run_SoAd_Init

G_Net2Run_SoAd_Init — Initialise the socket adaptor module.

Description

Use this command to initialise the socket adaptor module. This is the last and final step of configuring the socket adaptor. After this command was called the socket adaptor cannot be configured anymore.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_Init(
 const G_PortHandle_t  portHandle
)
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_DeInit

G_Net2Run_SoAd_DeInit — DeInitialise the socket adaptor module.

Description

Use this command to deinitialise the Socket Adaptor Module. This operation is the inversion of SoAd_Init and neccessary for continuing the configuration process if SoAd_Init was called before.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_DeInit(
 const G_PortHandle_t  portHandle
)
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_PduRoute_Config

G_Net2Run_SoAd_PduRoute_Config — Configuration of a PDU route within the socket adaptor.

Description

Use this command to configure a PDU route within the Socket Adaptor.

AUTOSAR 4.4.0 ECUC_SoAd_00007 Describes the path of a PDU from an upper layer of the SoAd to the socket in the TCP/IP stack for transmission. [...]

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_PduRoute_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_PduRoute_Config_Cmd_t * const cmd
)
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SO_AD__PDU_ROUTE__FLAG__NONE`
No flag is set

`G_NET2RUN__SO_AD__PDU_ROUTE__FLAG__ENABLE_TX_CONFIRMATION`
Enable TX Confirmation to other application layers

Since the TX confirmation is necessary for almost all applications, it is strongly recommended to configure all TX PDUs with this flag set.

`TxPduId`
Tx Pdu Id AUTOSAR 4.4.0 ECUC_SoAd_00031

`CollectionSemantics`
AUTOSAR 4.4.0: ECUC_SoAd_00160 AUTOSAR name: SoAdTxPduCollectionSemantics

following values are possible:

`G_NET2RUN__SO_AD__TX_COLLECTION_SEMANTICS__LAST_IS_BEST`
The PDU data will be fetched via Up_[SoAd][If]TriggerTransmit just before the transmission executes.

`G_NET2RUN__SO_AD__TX_COLLECTION_SEMANTICS__QUEUED`
The PDU data will instantly be stored in the context of the SoAd_IfTransmit API.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`NumberOfPduRouteDests`
Number of PDU route destinations

`PduRouteDestId`
Array of PDU route destination Ids

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_PduRoute_Delete

G_Net2Run_SoAd_PduRoute_Delete — Delete a PDU Route

Description

Use this command to delete a PDU Route.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_PduRoute_Delete(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_PduRoute_Delete_CmdFlags_t cmdFlags,
 const u32_t  TxPduId
)
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SO_AD__PDU_ROUTE__DELETE__CMD_FLAG__NONE`
No flag is set

`TxPduId`
Tx PDU ID

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_PduRoute_GetFreeTxPduId

G_Net2Run_SoAd_PduRoute_GetFreeTxPduId — Return a free Tx PDU ID.

Description

Use this command to query a free Tx PDU ID.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_PduRoute_GetFreeTxPduId(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_PduRoute_GetFreeTxPduId_CmdFlags_t cmdFlags,
 u32_t  * const TxPduId
)
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SO_AD__SO_ROUTE__GET_FREE_RX_PDU_ID__CMD_FLAG__NONE`
No flag is set

`TxPduId`
Returns a free Tx PDU ID

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_PduRouteDest_Config

G_Net2Run_SoAd_PduRouteDest_Config — Configuration of a PDU route destination within the socket adaptor.

Description

Use this command to configure a PDU route destination.

AUTOSAR 4.4.0 ECUC_SoAd_00119 Specifies the PDU route destination.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_PduRouteDest_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_PduRouteDest_Config_Cmd_t * const cmd
)
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SO_AD__PDU_ROUTE_DEST__FLAG__NONE`
No flag is set

`PduRouteDestId`
Unique identifier of this PDU route destination

`TxUdpTiggerMode`
AUTOSAR 4.4.0 ECUC_SoAd_00136 AUTOSAR name: SoAdTxUdpTriggerMode Specifies whether a PDU triggers the transmission of the nPduUdpTxBuffer. If this parameter is set to TRIGGER_NEVER, SoAd shall use an nPduUdpTxBuffer for the related socket connection. nPduUdpTxBuffer can only be used for upper layers with IF API, i.e. this parameter shall only be set to TRIGGER_NEVER if all upper layers belonging to the related socket connection have SoAdTxUpperLayerType set to "IF". This parameter is only relevant for UDP connections.

following values are possible:

`G_NET2RUN__SO_AD__TX_UDP_TRIGGER_MODE__ALWAYS`
PDU triggers the transmission

`G_NET2RUN__SO_AD__TX_UDP_TRIGGER_MODE__NEVER`
PDU does not trigger the transmission

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`TxPduHeaderId`
Optional in case PduHeaderIds are used AUTOSAR: ECUC_SoAd_00120

`TxUdpTriggerTimeout`
Specifies the timeout in [us] the nPduUdpTxBuffer shall be transmitted at the latest after this PDU is put into the buffer. This optional parameter is only relevant if SoAdTxUdpTriggerMode is TRIGGER_NEVER. AUTOSAR: ECUC_SoAd_00150

`NumberOfSoConRefIds`
number of (max. 128) ids in the stream that refer either to SocketConnections (if PDU_ROUTE_DEST__SO_CON_FLAG__USE_GROUP_ID is not set) or to SocketConnection-Groups (if PDU_ROUTE_DEST__FLAG__SO_CON_GROUP is set)

`NumberOfRoutingGroupIds`
number of (max. 128) routing group ids in the stream

`So_ConRefIdS`
array of Socket Connection Ref IDs (type SoAd_PduRouteDest_SoConRef_t)

`RoutingGroupIds`
array of routing group IDs

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_Routing_Disable

G_Net2Run_SoAd_Routing_Disable — Disable routing to or from a socket connection.

Description

Use this command to disable routing for the given routing group.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_Routing_Disable(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_Routing_Disable_Cmd_t * const cmd
)
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SO_AD__ROUTING__DISABLE__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__SO_AD__ROUTING__DISABLE__CMD__FLAG__SPECIFIC`
Use this flag to disable the routing of only one specific socket connection controlled by the given routing group instead of all socket connections of this routing group.

`RoutingGroupId`
Routing Group Id

`SpecificSoConId`
Specific Socket Connnection Id

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_Routing_Enable

G_Net2Run_SoAd_Routing_Enable — Enable routing to or from a socket connection.

Description

Use this command to enable routing for the given routing group.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_Routing_Enable(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_Routing_Enable_Cmd_t * const cmd
)
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SO_AD__ROUTING__ENABLE__CMD_FLAG__NONE`
No flag is set

`G_NET2RUN__SO_AD__ROUTING__ENABLE__CMD__FLAG__SPECIFIC`
Use this flag to enable the routing of only one specific socket connection controlled by the given routing group instead of all socket connections of this routing group.

`RoutingGroupId`
Routing Group Id

`SpecificSoConId`
Specific Socket Connnection Id

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_RoutingGroup_Config

G_Net2Run_SoAd_RoutingGroup_Config — Configuration of a routing group.

Description

Use this command to configure a routing group.

AUTOSAR 4.4.0 ECUC_SoAd_00109 Each container describes a specific routing group which can be enabled or disabled. A routing group consists of PDUs. Routing of PDUs can either be forwarding of PDUs from the upper layer to a TCP or UDP socket of the TCP/IP stack specified by a SoAdPduRoute or the other way around specified by a SoAdSocketRoute.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_RoutingGroup_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_RoutingGroup_Config_CmdFlags_t cmdFlags,
 const u32_t  RoutingGroupId  // AUTOSAR 4.4.0 ECUC_SoAd_00121
)
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SO_AD__SO_ROUTING_GROUP__FLAG__NONE`
No flag is set

`G_NET2RUN__SO_AD__ROUTING_GROUP__FLAG__ENABLE_AT_INIT`
Enables the routing for this routing group at initialisation.

If this flag is not set, the routing will be disable at initialisation. Hence connections that are part of this routing group might not be able to receive or transmit data.

`RoutingGroupId`
Routing Group ID AUTOSAR 4.4.0 ECUC_SoAd_00121

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_RoutingGroup_Delete

G_Net2Run_SoAd_RoutingGroup_Delete — Delete a routing group.

Description

Use this command to delete a routing group.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_RoutingGroup_Delete(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_RoutingGroup_Delete_CmdFlags_t cmdFlags,
 const u32_t  RoutingGroupId
)
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SO_AD__ROUTING_GROUP__DELETE__CMD_FLAG__NONE`
No flag is set

`RoutingGroupId`
Routing Group ID

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_RoutingGroup_GetFreeRoutingGroupId

G_Net2Run_SoAd_RoutingGroup_GetFreeRoutingGroupId — Return a free routing group ID.

Description

Use this command to query a free routing group ID.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_RoutingGroup_GetFreeRoutingGroupId(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_RoutingGrp_GetFreeRoutingGrpId_CmdFlags_t cmdFlags,
 u32_t  * const RoutingGroupId
)
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SO_AD__ROUTING_GRP__GET_FREE_ROUTING_GRP_ID__CMD_FLAG__NONE`
No flag is set

`RoutingGroupId`
Returns a free Routing Group ID

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_SoCon_Close

G_Net2Run_SoAd_SoCon_Close — Close a socket connection.

Description

Use this command to close a socket connection.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_SoCon_Close(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_SoCon_Close_CmdFlags_t cmdFlags,
 const u32_t Id
)
```

Parameters

`portHandle`
: Handle to the communication port

`cmdFlags`
: Command flags

  The following values are possible and can be combined:

  `G_NET2RUN__SO_AD__SO_CON__CLOSE__CMD__FLAG__NONE`
  : No flag is set

  `G_NET2RUN__SO_AD__SO_CON__CLOSE__CMD__FLAG__SO_CON_GROUP`
  : Use this flag to close all socket connections that are part of the socket connection group with the given id.

`Id`
: Id either of the socket connection or of the socket connection group, if the flag "SO_CON_GROUP" is used.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_SoCon_Config

G_Net2Run_SoAd_SoCon_Config — Configuration of a socket connection.

Description

Use this command to configure a socket connection.

AUTOSAR 4.4.0 ECUC_SoAd_00009 Specifies the socket connection (Id and remote address information). Note: Parameters which are common to all socket connections of a socket connection group are specified directly at the group.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_SoCon_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_SoCon_Config_Cmd_t * const cmd
)
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SO_AD__SO_CON__CONFIG_FLAG__NONE`
No flag is set

`G_NET2RUN__SO_AD__SO_CON__CONFIG_FLAG__SET_REMOTE_ADDR`
Set the given remote address.

`G_NET2RUN__SO_AD__SO_CON_GROUP__FLAG__PDU_HEADER_OPTION`
AUTOSAR 4.4.0 ECUC_SoAd_00131 Enables the transmission of the PDU header (ID, length) on this socket connection. TRUE: add SoAd PDU header before PDU data FALSE: No SoAd PDU header is used

`G_NET2RUN__SO_AD__SO_CON_GROUP__FLAG__AUTOMATIC_SO_CON_SETUP`
AUTOSAR 4.4.0 ECUC_SoAd_00110 Specifies if the setup of the socket connection shall be done automatically (TRUE) or manually (FALSE) via SoAd_OpenSoCon() and SoAd_CloseSoCon().

`G_NET2RUN__SO_AD__SO_CON_GROUP__FLAG__MSG_ACCEPTANCE_FILTER`
AUTOSAR 4.4.0 ECUC_SoAd_00137 Specifies if the message acceptance filter is enabled (TRUE) or not (FALSE).

`SoConId`
Socket Connection Id

`SoConGroupId`
Socket Connection Group Id

`RemotePort`
AUTOSAR 4.4.0 ECUC_SoAd_00020 Remote UDP or TCP port used for this connection. To accept any remote port, set SoAdSocketRemotePort to 0. See message acceptance policy for more details.

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`RemoteAddress`
Either a IPv4 or IPv6 address in network byte order. AUTOSAR 4.4.0 ECUC_SoAd_00019 IP address of remote node. The configured address must be of the same TcpIpDomainType (i.e. IPv4 or IPv6) as the TcpIpLocalAddr referred by SoAdSocketLocalAddressRef. To accept any remote IP address, set SoAdSocketRemoteIpAddress to "ANY". See message acceptance policy for more details. ["ANY" = '0.0.0.0' or '::']

`RxBufferSize`
Protocol Size of Rx Buffer

`TxBufferSize`
Size of Tx Buffer

`NumberOfTxBuffers`
Number of Tx Buffers

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_SoCon_Delete

G_Net2Run_SoAd_SoCon_Delete — Delete a socket connection group.

Description

Use this command to delete a socket connection.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_SoCon_Delete(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_SoCon_Delete_CmdFlags_t cmdFlags,
 const u32_t  SoConId
)
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SO_AD__SO_CON__DELETE__CMD_FLAG__NONE`
No flag is set

`SoConId`
Socket Connection ID

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_SoCon_GetFreeSoConId

G_Net2Run_SoAd_SoCon_GetFreeSoConId — Return a free socket connection ID.

Description

Use this command to query a free socket connection ID.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_SoCon_GetFreeSoConId(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_SoCon_GetFreeSoConId_CmdFlags_t cmdFlags,
 u32_t  * const SoConId
)
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SO_AD__SO_CON__GET_FREE_SO_CON_ID__CMD__FLAGS__NONE`
No flag is set

`SoConId`
Returns a free Socket Connection ID

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_SoCon_GetState

G_Net2Run_SoAd_SoCon_GetState — Return the state of the socket connection.

Description

Use this command to query the state of the socket connection.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_SoCon_GetState(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_SoCon_GetState_CmdFlags_t cmdFlags,
 const u32_t  SoConId,
 G_Net2Run_SoAd_SoCon_State_t  * const state
)
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SO_AD__SO_CON__GET_STATE__CMD_FLAG__NONE`
No flag is set

`SoConId`
Socket Connection Id

`state`
Returns the Socket Connection state

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_SoCon_Open

G_Net2Run_SoAd_SoCon_Open — Open a socket connection.

Description

Use this command to open a socket connection.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_SoCon_Open(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_SoCon_Open_CmdFlags_t cmdFlags,
 const u32_t  Id
)
```

Parameters

`portHandle`
: Handle to the communication port

`cmdFlags`
: Command flags

  The following values are possible and can be combined:

  `G_NET2RUN__SO_AD__SO_CON__OPEN__CMD_FLAG__NONE`
  : No flag is set

  `G_NET2RUN__SO_AD__SO_CON__OPEN__CMD__FLAG__SO_CON_GROUP`
  : Use this flag to open all socket connections that are part of the socket connection group with the given id.

`Id`
: ID either of the socket connection or of the socket connection group, if the flag "SO_CON_GROUP" is used.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_SoCon_RemoteAddr_Set

G_Net2Run_SoAd_SoCon_RemoteAddr_Set — Set the remote address of a socket connection

Description

Use this command to set the remote address of a socket connection

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_SoCon_RemoteAddr_Set(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_SoCon_RemoteAddr_Set_Cmd_t * const cmd
)
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SO_AD__SO_CON__REMOTE_ADDR__SET__CMD_FLAG__NONE`
No flag is set

`SoConId`
Socket Connnection Id

`RemoteAddress`
contains the remote IP address ("`u8_t RemoteAddress[16]`") Either a IPv4 or IPv6 address in network byte order. Must be of the same type as the configured local address of the corresponding SoConGroup

`RemotePort`
Remote Port

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_SoConGroup_Config

G_Net2Run_SoAd_SoConGroup_Config — Configuration of a socket connection group.

Description

Use this command to configure a socket connection group.

AUTOSAR 4.4.0 ECUC_SoAd_00130 Specifies the configuration of a socket connection group, i.e. specifies the socket connections belonging to the group and the parameters which are common for all socket connections of the group. A socket connection specifies how data can be received and transmitted via a TCP or UDP socket.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_SoConGroup_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_SoConGroup_Config_Cmd_t * const cmd
)
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SO_AD__SO_CON_GROUP__FLAG__NONE`
No flag is set

`G_NET2RUN__SO_AD__SO_CON_GROUP__FLAG__USE_VLAN_ID`
Use VLAN ID

A **V**irtual **L**ocal **A**rea **N**etwork can be configured with G_Ethernet_Node_Vlan_Config .

`G_NET2RUN__SO_AD__SO_CON_GROUP__FLAG__PDU_HEADER_OPTION`
AUTOSAR 4.4.0 ECUC_SoAd_00131 Enables the transmission of the PDU header (ID, length) on this socket connection. TRUE: add SoAd PDU header before PDU data FALSE: No SoAd PDU header is used

`G_NET2RUN__SO_AD__SO_CON_GROUP__FLAG__AUTOMATIC_SO_CON_SETUP`
AUTOSAR 4.4.0 ECUC_SoAd_00110 Specifies if the setup of the socket connection shall be done automatically (TRUE) or manually (FALSE) via SoAd_OpenSoCon() and SoAd_CloseSoCon().

`G_NET2RUN__SO_AD__SO_CON_GROUP__FLAG__MSG_ACCEPTANCE_FILTER`
AUTOSAR 4.4.0 ECUC_SoAd_00137 Specifies if the message acceptance filter is enabled (TRUE) or not (FALSE).

`UdpFlags`
Udp flags

The following values are possible and can be combined:

`G_NET2RUN__SO_AD__SO_CON_GROUP__UDP_FLAGS__ZERO`
No flag is set

`G_NET2RUN__SO_AD__SO_CON_GROUP__UDP_FLAG__LISTEN_ONLY`
AUTOSAR 4.4.0 ECUC_SoAd_00024 Specifies if the socket connection group is only used for reception (TRUE) or used for both reception and transmission (FALSE).

`G_NET2RUN__SO_AD__SO_CON_GROUP__UDP_FLAG__STRICT_HDR_LEN_CHECK`
AUTOSAR 4.4.0 ECUC_SoAd_00154 Specifies if UDP messages shall be dropped (TRUE) if the length of all contained PDUs does not match the length of the whole message or not (FALSE). Shall only be set to TRUE if SoAdPduHeaderEnable is also set to TRUE.

`G_NET2RUN__SO_AD__SO_CON_GROUP__UDP_FLAG__ALIVE_SUPERVISION`
Use this flag if alive supervision should be used. AUTOSAR 4.4.0 ECUC_SoAd_00149

`TcpFlags`
Tcp flags

The following values are possible and can be combined:

`G_NET2RUN__SO_AD__SO_CON_GROUP__TCP_FLAGS__ZERO`
No flag is set

`G_NET2RUN__SO_AD__SO_CON_GROUP__TCP_FLAG__INITIATE_CONNECTION`
AUTOSAR 4.4.0 ECUC_SoAd_00022 Specifies the initiator for this TCP connection. [...] TRUE: This TCP connection is initiated by this module. [client] FALSE: This TCP connection is to be initiated in the listen mode. [server]

`G_NET2RUN__SO_AD__SO_CON_GROUP__TCP_FLAG__USE_CONGESTION_CONTROL`
compare with AUTOSAR 4.4.0 ECUC_SoAd_00023 Specifies not to use the congestion control mechanism for this connection. TRUE: This TCP connection will use congestion control. (Nagle algorithm) FALSE: This TCP connection will NOT use congestion control. (TcpNoDelay)

`G_NET2RUN__SO_AD__SO_CON_GROUP__TCP_FLAG__USE_TCP_KEEP_ALIVE`
compare with AUTOSAR 4.4.0 ECUC_SoAd_00148 Specifies not to use the congestion control mechanism for this connection. TRUE: This TCP connection will use the keep-alive mechanism. FALSE: This TCP connection will not use the keep-alive mechanism.

`G_NET2RUN__SO_AD__SO_CON_GROUP__TCP_FLAG__USE_TLS`
compare with AUTOSAR 4.4.0 ECUC_SoAd_00148 If set the TCP socket is assigned to a TLS connection. The SoAd needs to call Tcpip_ChangeParameter with the reference to the TLS connection as the parameter. TRUE: This TCP connection will use TLS. FALSE: This TCP connection will not use TLS.

`SoConGroupId`
Socket Connection Group Id

`ControllerId`
Controller Id 0=ETHERNET1, 1=ETHERNET2, ...

`EcuId`
Ecu Id

`reserved1`
reserved parameter (must be initialized with **0**)

`Protocol`
AUTOSAR 4.4.0 ECUC_SoAd_00139 Specifies the transport protocol and transport protocol specific parameters used for the socket connections of the socket connection group. The following values are possible:

`G_NET2RUN__SO_AD__SOCKET_PROTOCOL__UDP4`
User Datagram Protocol Ipv4

`G_NET2RUN__SO_AD__SOCKET_PROTOCOL__UDP6`
User Datagram Protocol Ipv6

`G_NET2RUN__SO_AD__SOCKET_PROTOCOL__TCP4`
Transmission Control Protocol Ipv4

`G_NET2RUN__SO_AD__SOCKET_PROTOCOL__TCP6`
Transmission Control Protocol Ipv6

`G_NET2RUN__SO_AD__SOCKET_PROTOCOL__UNKNOWN`
This is just a placeholder

`VlanId`
Vlan Id

`FlowLabel`
AUTOSAR 4.4.0 ECUC_SoAd_00157 The 20-bit Flow Label field in the IPv6 header may be used by a source to label sequences of packets for which it requests special handling by the IPv6 routers, such as non-default quality of service. If not set a Flow Label of zero is used to indicate packets that have not been labeled. only IPv6

`DifferentiatedServicesField`
AUTOSAR 4.4.0 ECUC_SoAd_00158 The 6-bit Differentiated Service Field in the IP headers may be used for classifying network traffic. If not set a value of zero is used to indicate packets that have not been classified. only IPv6

`FramePriority`
AUTOSAR 4.4.0 ECUC_SoAd_00138 Specifies the priority of the Ethernet frame. If IEEE 802.1Q VLAN Tags are used, the specified priority will be used in the VLAN Tag PCP filed. If this optional parameter is not available the default priority specified in the TcpIp module is used. only for Vlans

`LocalPort`
AUTOSAR 4.4.0 ECUC_SoAd_00018 Local UDP or TCP port used for this connection. If this parameter set to 0 SoAd requests TcpIp to select an ephemeral port.

`LocalAddress`
AUTOSAR 4.4.0 ECUC_SoAd_00017 Local IP address and interface used for this connection. Either a IPv4 or IPv6 address in network byte order.

`TcpKeepAliveProbesMax`
AUTOSAR 4.4.0 ECUC_SoAd_00151 Just considered if tcp-flag "USE_TCP_KEEP_ALIVE" is used. Maximum number of times that TCP retransmits an individual data segment before aborting the connection.

`reserved2`
reserved parameter (must be initialized with **0**)

`reserved3`
reserved parameter (must be initialized with **0**)

`TcpKeepAliveInterval`
AUTOSAR 4.4.0 ECUC_SoAd_00152 Just considered if tcp-flag "USE_TCP_KEEP_ALIVE" is used. Specifies the interval in [us] between subsequent keep-alive probes

`TcpKeepAliveTime`
AUTOSAR 4.4.0 ECUC_SoAd_00153 Just considered if tcp-flag "USE_TCP_KEEP_ALIVE" is used. Specifies the time in [us] between the last data packet sent and the first keep-alive probe.

`UdpTriggerTimeout`
AUTOSAR 4.4.0 ECUC_SoAd_00135 Just considered if "TriggerTimeout" is used. Specifies the timeout in [us] a nPduUdpTxBuffer is waiting for a PDU with TriggerMode = TRIGGER_ALWAYS, i.e. when the timeout expires the nPduUdpTxBuffer is transmitted. Timer is reset after each UDP transmission. This optional parameter is only relevant if a nPduUdpTxBuffer is used.

`UdpAliveSupervisionTimeout`
AUTOSAR 4.4.0 ECUC_SoAd_00137 Just considered if udp-flag "ALIVE_SUPERVISION" is used. Specifies the time in [us] a UDP socket connection remains in the mode SOAD_SOCON_ONLINE after the latest reception of a frame from the remote peer specified by the remote address. If this optional parameter is not enabled UDP Alive Supervision is deactivated for the related socket connection group.

`ConnectionRetryWaitTime`
Only considered if Flag "AutomaticSoConSetup" is set. Specifies the time in [us] the SoCon waits before it tries to reopen the socket connection

`ConnectionRetries`
Only considered if Flag "AutomaticSoConSetup" is set. Specifies the number of retries when establishing a connection failed.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_SoConGroup_Delete

G_Net2Run_SoAd_SoConGroup_Delete — Delete a socket connection group.

Description

Use this command to delete a socket connection group.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_SoConGroup_Delete(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_SoConGroup_Delete_CmdFlags_t cmdFlags,
 const u32_t  SoConGroupId
)
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SO_AD__SO_CON_GROUP__DELETE__CMD_FLAG__NONE`
No flag is set

`SoConGroupId`
Socket Connection Group ID

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_SoConGroup_GetFreeSoConGroupId

G_Net2Run_SoAd_SoConGroup_GetFreeSoConGroupId — Return a free socket connection group ID.

Description

Use this command to query a free socket connection group ID.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_SoConGroup_GetFreeSoConGroupId(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_SoCon_GetFreeSoConId_CmdFlags_t cmdFlags,
 u32_t  * const SoConGroupId
)
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SO_AD__SO_CON_GRP__GET_FREE_SO_CON_GRP_ID__CMD_FLAG__NONE`
No flag is set

`SoConGroupId`
Returns a free Socket Connection Group ID

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_SoRoute_Config

G_Net2Run_SoAd_SoRoute_Config — Configuration of a socket route

Description

Use this command to configure a socket route.

AUTOSAR 4.4.0 ECUC_SoAd_00008 Describes the path of a PDU from a socket in the TCP/IP stack to an upper layer of the SoAd after reception in the TCP/IP Stack.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_SoRoute_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_SoRoute_Config_Cmd_t * const cmd
)
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SO_AD__SO_ROUTE_CONF__FLAG__NONE`
No flag is set

`RxPduId`
Rx Pdu ID

`PduHeaderId`
Optional in case PduHeaderIds are used AUTOSAR 4.4.0 ECUC_SoAd_00036 ID contained in the packet received on the TCP/IP connection if the PDU header option is enabled.

`NumberOfSoConRefIds`
number of (max. 128) ids in the stream that refer to SocketConnections (if SO_ROUTE__SO_CON_FLAG__USE_GROUP_ID is not set) or to SocketConnectionGroups (if SO_ROUTE__SO_CON_FLAG__USE_GROUP_ID is set)

`NumberOfRoutingGroupIds`
number of (max. 128) routing group ids in the stream

`So_ConRefIdS`
array of Socket Connection Ref IDs (type SoRoute_SoConRef_t)

`RoutingGroupIds`
array of Routing Group IDs

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_SoRoute_Delete

G_Net2Run_SoAd_SoRoute_Delete — Delete a socket route.

Description

Use this command to delete a socket route.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_SoRoute_Delete(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_SoRoute_Delete_CmdFlags_t cmdFlags,
 const u32_t  RxPduId
)
```

Parameters

`portHandle`
: Handle to the communication port

`cmdFlags`
: Command flags

  The following values are possible and can be combined:

  `G_NET2RUN__SO_AD__SO_ROUTE__DELETE__CMD_FLAG__NONE`
  : No flag is set

`RxPduId`
: Rx Pdu ID

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_SoAd_SoRoute_GetFreeRxPduId

G_Net2Run_SoAd_SoRoute_GetFreeRxPduId — Return a free rx PDU ID.

Description

Use this command to query a free Rx PDU ID.

Definition

```
G_Error_t
G_API
G_Net2Run_SoAd_SoRoute_GetFreeRxPduId(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_SoAd_SoRoute_GetFreeRxPduId_CmdFlags_t cmdFlags,
 u32_t  * const RxPduId
)
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__SO_AD__SO_ROUTE__GET_FREE_RX_PDU_ID__CMD_FLAG__NONE`
No flag is set

`RxPduId`
Returns a free Rx Pdu ID

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

23 UDP Network management

Functions for controlling the UDP network management of a Net2Run **Rest Bus Simulation**

G_Net2Run_UdpNm_Autosar_Ecu_Config

G_Net2Run_UdpNm_Autosar_Ecu_Config — Configuration of Autosar Ecu

Description

Use this command for configuration of an Autosar Ecu.

Definition

```
G_Error_t
G_API
G_Net2Run_UdpNm_Autosar_Ecu_Config(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_UdpNm_Autosar_Ecu_Config_Cmd_t * const cmd
)
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`Flags`
Command flags

The following values are possible and can be combined:

`G_NET2RUN__UDP_NM__AUTOSAR__ECU__FLAG__NONE`
No flag is set

`G_NET2RUN__UDP_NM__AUTOSAR__ECU__FLAG__PASSIVE_MODE_ENABLED`
In Passive Mode the node is only receiving NM messages but not transmitting any NM messages.

`G_NET2RUN__UDP_NM__AUTOSAR__ECU__FLAG__NODE_DETECTION_ENABLED`
Enables UDP NM Node detection.

`G_NET2RUN__UDP_NM__AUTOSAR__ECU__FLAG__USER_DATA_ENABLED`
Enable user data support.

`G_NET2RUN__UDP_NM__AUTOSAR__ECU__FLAG__COM_USER_DATA_SUPPORT`
Enable the COM user data support.

`G_NET2RUN__UDP_NM__AUTOSAR__ECU__FLAG__PDU_RX_INDICATION_ENABLED`
Enable RX Indication to other application layers

Since the RX indication is necessary for almost all applications, it is strongly recommended to configure all RX PDUs with this flag set.

`G_NET2RUN__UDP_NM__AUTOSAR__ECU__FLAG__REMOTE_SLEEP_IND_ENABLED`
Enable remote sleep indication support. This feature is required for gateway nodes only. It must not be defined if UDPNM_PASSIVE_MODE_ENABLED is defined. This parameter shall be derived from NM_REMOTE_SLEEP_IND_ENABLED.

G_NET2RUN__UDP_NM__AUTOSAR__ECU__FLAG__STATE_CHANGE_IND_ENABLED
Enable UDP NM state change notification.

G_NET2RUN__UDP_NM__AUTOSAR__ECU__FLAG__ACTIVE_WAKE_UP_BIT_ENABLED
Enable the handling of the Active Wakeup bit.

G_NET2RUN__UDP_NM__AUTOSAR__ECU__FLAG__USE_NID_IN_PDU
Use node identifier bit in the network management PDU.

G_NET2RUN__UDP_NM__AUTOSAR__ECU__FLAG__USE_CBV_IN_PDU
Use node control bit vector in the network management PDU.

G_NET2RUN__UDP_NM__AUTOSAR__ECU__FLAG__USE_NMS_IN_PDU
Use network management state in the network management PDU.

`ControllerId`
ETH controller ID

0 = ETH1, **1** = ETH2, **2** = ETH3, ...

`EcuId`
Ecu Id beginning with 0

`NmNodeId`
Identifier of the network management node.

`reserved1`
reserved parameter (must be initialized with **0**)

`NmTimeoutTime`
NM-Timeout Time in microseconds

Network Timeout for NM-Messages. It denotes how long the NM shall stay in the Network Mode before transition into Prepare Bus-Sleep Mode shall take place. It shall be equal for all nodes in the cluster.

`WaitBusSleepTime`
Timeout for bus calm down phase in microseconds.

It denotes how long the NM shall stay in the Prepare Bus-Sleep Mode before transition into Bus-Sleep Mode shall take place. It shall be equal for all nodes in the cluster. It shall be long enough to make all Tx-buffer empty.

`RepeatMessageTime`
Timeout for Repeat Message State in microseconds.

It defines how long the NM shall stay in the Repeat Message State.

`MsgCycleTime`
Period of a NM-message in microseconds.

It determines the periodic rate in the "periodic transmission mode with bus load reduction" and is the basis for transmit scheduling in the "periodic transmission mode without bus load reduction".

`MsgCycleOffset`
Time offset in the periodic transmission mode in microseconds.

It determines the start delay of the transmission.

`RemoteSleepIndTime`
Timeout for Remote Sleep Indication in microseconds.

It defines the time how long it shall take to recognize that all other nodes are ready to sleep. The value "0" denotes that no Remote Sleep Indication functionality is configured.

AUTOSAR name: UdpNmRemoteSleepIndTime {UDPNM_REMOTE_SLEEP_IND_TIME}

`PduNidPosition`
Node id position within PDU.

Specifies the position of the **Node Identifier** within the PDUs when flag `G_NET2RUN__UDP_NM__AUTOSAR__ECU__FLAG__USE_NID_IN_PDU` is set

The following values are possible:

`G_BET2RUN__UDP_NM__AUTOSAR__PDU_NID_POSITION__BYTE_0`
Node id position is in PDU byte #0 (AUTOSAR default).

`G_BET2RUN__UDP_NM__AUTOSAR__PDU_NID_POSITION__BYTE_1`
Node id position is in PDU byte #1.

`PduCbvPosition`
Control bit vector position within PDU.

Specifies the position of the **Control bit vector** within the PDUs when flag `G_NET2RUN__UDP_NM__AUTOSAR__ECU__FLAG__USE_CBV_IN_PDU` is set

The following values are possible:

`G_BET2RUN__UDP_NM__AUTOSAR__PDU_CBV_POSITION__BYTE_0`
Control bit vector is in PDU byte #0.

`G_BET2RUN__UDP_NM__AUTOSAR__PDU_CBV_POSITION__BYTE_1`
Control bit vector is in PDU byte #1 (AUTOSAR default).

`PduNmsPosition`
NM state position within PDU.

Specifies the position of the **NM state** within the PDUs when flag `G_NET2RUN__UDP_NM__AUTOSAR__ECU__FLAG__USE_NMS_IN_PDU` is set

The following values are possible:

`G_BET2RUN__UDP_NM__AUTOSAR__PDU_NMS_POSITION__BYTE_0`
NM state position is in PDU byte #0.

`G_BET2RUN__UDP_NM__AUTOSAR__PDU_NMS_POSITION__BYTE_1`
NM state position is in PDU byte #1.

`G_BET2RUN__UDP_NM__AUTOSAR__PDU_NMS_POSITION__BYTE_2`
NM state position is in PDU byte #2.

`UserDataOffset`
offset within NM-Data PDU where user data begins.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_UdpNm_Autosar_RxPdu

G_Net2Run_UdpNm_Autosar_RxPdu — Configuration of Autosar Rx-Pdu

Description

Use this command for Autosar Rx-Pdu configuration.

Definition

```
G_Error_t
G_API
G_Net2Run_UdpNm_Autosar_RxPdu(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_UdpNm_Autosar_RxPdu_Cmd_t * const cmd
)
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`ControllerId`
ETH controller ID

0 = ETH1, **1** = ETH2, **2** = ETH3, ...

`EcuId`
Ecu ID

ECU ID, starting with 0

`NmNodeId`
NM node identifier that is used as source node identifier.

`reserved1`
reserved parameter (must be initialized with **0**)

`SoAd_RxPduId`
RxPduId of the corresponsing SoRoute of the Socket Adaptor.

The corresponding SoRoute must be configured first.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Net2Run_UdpNm_Autosar_TxPdu

G_Net2Run_UdpNm_Autosar_TxPdu — Configuration of Autosar Tx-Pdu

Description

Use this command for Autosar Tx-Pdu configuration.

Definition

```
G_Error_t
G_API
G_Net2Run_UdpNm_Autosar_TxPdu(
 const G_PortHandle_t  portHandle,
 const G_Net2Run_UdpNm_Autosar_TxPdu_Cmd_t * const cmd
)
```

Parameters

`portHandle`
Handle to the communication port

`cmd`
Command parameters

The structure contains the following elements:

`ControllerId`
ETH controller ID

0 = ETH1, **1** = ETH2, **2** = ETH3, ...

`EcuId`
Ecu ID

ECU ID, starting with 0

`NmNodeId`
NM node identifier that is used as source node identifier.

`reserved1`
reserved parameter (must be initialized with **0**)

`SoAd_TxPduId`
TxPduId of the corresponsing PduRoute of the Socket Adaptor.

The corresponding PduRoute must be configured first.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Resistor Functions

1 Common Resistor Functions

These functions provide control over Resistor function of **GOEPEL electronic** devices.

G_Resistors_GetNumberOfResistors

G_Resistors_GetNumberOfResistors — Query number of resistors

Description

This command is used to query the number of resistors of the device.

Definition

```
u32_t
G_Resistors_GetNumberOfResistors(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

u32_t number of resistors

G_Resistors_Values_Set

G_Resistors_Values_Set — Set resistor values

Description

Use this command to set values for several resistors of a device.

Definition

```
G_Error_t
G_Resistors_Values_Set(
 const G_PortHandle_t  portHandle,
 const u32_t  numberOfResistors,
 const u32_t * const indexes,
 const u32_t * const values
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfResistors`
Number of resistors to be set

`indexes`
Array with indexes of resistors to be set

Since resistor indexes are zero-based, the **index** for the **1st** resistor of a device is **0**.

`values`
Array with values for resistors to be set in **milliohm**

Each values-element belongs to the `indexes`-element with the same array-index.

E.g. the value in `values[3]` belongs to the resistor specified by the index in `indexes[3]`.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

In this example, two resistors are set to 1 Ohm (resistor 1) and 4200 Ohm (resistor 2).

Note that the resistor index is **zero-based** and the resistor value is specified in **milliohm** .

```
// 1st resistor
indexes[0] = 0;
values[0] = 1000;  // 1 ohm

// 2nd resistor
indexes[1] = 1;
values[1] = 4200000;  // 4200 ohm

return
 G_Resistors_Values_Set(
  PortHandle1,
  2,  // number of resistors
  indexes,
  values
 );
```

G_Resistors_Values_Get

G_Resistors_Values_Get — Query resistor values

Description

Use this command to query values for several resistors of a device.

Definition

```
G_Error_t
G_Resistors_Values_Get(
 const G_PortHandle_t  portHandle,
 const u32_t  numberOfResistors,
 const u32_t * const indexes,
 u32_t * const values
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfResistors`
Number of resistors to be queried

`indexes`
Array with indexes of resistors to be queried

Since resistor indexes are zero-based, the **index** for the **1st** resistor of a device is **0**.

`values`
Returns array with resistor values in **milliohm**

Each values-element belongs to the `indexes`-element with the same array-index.

E.g. the value in `values[3]` belongs to the resistor specified by the index in `indexes[3]`.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

In this example, the first two resistors of a device are queried.

Note that the resistor index is **zero-based** and the resistor value is returned in **milliohm** .

```
u32_t  indexes[2];
u32_t  values[2];

indexes[0] = 0;  // 1st resistor
indexes[1] = 1;  // 2nd resistor

return
 G_Resistors_Values_Get(
  PortHandle1,
  2,  // number of resistors
  indexes,
  values
 );
```

G_Resistors_Reset

G_Resistors_Reset — Reset resistors

Description

Use this command to reset several resistors of a device (R = ∞).

Definition

```
G_Error_t
G_Resistors_Reset(
 const G_PortHandle_t  portHandle,
 const u32_t  numberOfResistors,
 const u32_t * const indexes
);
```

Parameters

`portHandle`
Handle to the communication port

`numberOfResistors`
Number of resistors to be reset

`indexes`
Array with indexes of resistors to be reset

Since resistor indexes are zero-based, the **index** for the **1st** resistor of a device is **0**.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

In this example, the first two resistors of a device are reset.

Note that the resistor index is **zero-based** and the resistor value is returned in **milliohm**.

```
u32_t  indexes[2];

indexes[0] = 0;  // 1st resistor
indexes[1] = 1;  // 2nd resistor

return
 G_Resistors_Values_Reset(
  PortHandle1,
  2,  // number of resistors
  indexes
 );
```

Sequence Functions

These **G-API** functions provide control over the Sequence-Functionality of your hardware, including replay files. If the return value of type **G_Error_t** does not correspond to G_NO_ERROR, an error has occurred. A description of this error can be retrieved by calling G_GetLastErrorDescription or G_GetLastErrorByPortHandle .

1 Common Functions

Common functions for the Sequence interface

G_Sequence_InitInterface

G_Sequence_InitInterface — Init Sequence interface

Description

This command resets the selected Sequence interface without software reset to the initial state.

Definition

```
G_Error_t
G_Sequence_InitInterface(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

2 Replay Functions

These functions provide control over the replay sequences.

Replay sequences are sequences of commands, that can be recorded, stored and started directly on the hardware. This provides an increased performance because there is no communication with the host.

G_Sequence_Replay_Allocate

G_Sequence_Replay_Allocate — Allocate a Replay Sequence

Description

This command allocates a replay sequence and returns its handle.

Definition

```
G_Error_t
G_Sequence_Replay_Allocate(
 const G_PortHandle_t  portHandle,
 G_Sequence_Replay_Handle_t * const sequenceHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

`sequenceHandle`
: Returns the handle to the allocated sequence

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Sequence_Replay_Deallocate

G_Sequence_Replay_Deallocate — Deallocate a Replay Sequence

Description

This command deallocates a replay sequence.

Definition

```
G_Error_t
G_Sequence_Replay_Deallocate(
 const G_PortHandle_t  portHandle,
 const G_Sequence_Replay_Handle_t  sequenceHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

`sequenceHandle`
: Handle to the sequence

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Sequence_Replay_Recording_Start

G_Sequence_Replay_Recording_Start — Start recording of a replay sequence

Description

This command starts the recording of a replay sequence.

Definition

```
G_Error_t
G_Sequence_Replay_Recording_Start(
 const G_PortHandle_t  portHandle,
 const G_Sequence_Replay_Handle_t  sequenceHandle,
 const G_Sequence_Replay_Recording_CmdFlags_t  cmdFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`sequenceHandle`
Handle to the sequence

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_SEQUENCE__REPLAY__RECORDING__CMD_FLAG__ZERO`
No flag is set

`G_SEQUENCE__REPLAY__RECORDING__CMD_FLAG__DISABLE_CMD_EXECUTION`
Disable command execution

No commands are executed on the hardware. As an acknowledge for a successfully recorded command, the return code `rc` will be **`0xE1`** (command not executed).

`G_SEQUENCE__REPLAY__RECORDING__CMD_FLAG__DELETE_EXISTING_ENTRIES`
Delete existing entries

Already recorded commands for this sequence are deleted before the recording is started.

`G_SEQUENCE__REPLAY__RECORDING__CMD_FLAG__AUTOMATIC_INSERT_DELAYS`
Insert a delay after each recorded command

A delay is inserted after each command. The delay time equals the time between the command calls. Therefore the command execution during the replay sequence will have the same timing as the command calls during recording.

If not set, the recorded commands will be executed without delay.

It is possible to manually insert delays via command G_Sequence_Replay_AddDelay.

Note that each delay takes one entry in the replay sequence!

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Sequence_Replay_Recording_Stop

G_Sequence_Replay_Recording_Stop — Stop recording of a replay sequence

Description

This command stops the recording of a replay sequence.

Definition

```
G_Error_t
G_Sequence_Replay_Recording_Stop(
 const G_PortHandle_t  portHandle,
 const G_Sequence_Replay_Handle_t  sequenceHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

`sequenceHandle`
: Handle to the sequence

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Sequence_Replay_Playback_Start

G_Sequence_Replay_Playback_Start — Start playback of a replay sequence

Description

This command starts the playback of a replay sequence.

Definition

```
G_Error_t
G_Sequence_Replay_Playback_Start(
 const G_PortHandle_t  portHandle,
 const G_Sequence_Replay_Handle_t  sequenceHandle,
 const G_Sequence_Replay_Playback_CmdFlags_t  cmdFlags
);
```

Parameters

`portHandle`
Handle to the communication port

`sequenceHandle`
Handle to the sequence

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_SEQUENCE__REPLAY__PLAYBACK__CMD_FLAG__ZERO`
No flag is set

`G_SEQUENCE__REPLAY__PLAYBACK__CMD_FLAG__ALLOW_STARTING_WHEN_PLAYING`
Allow to restart the sequence if it is already running.

This property can also be set by G_Sequence_Replay_Property_SetById .

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Sequence_Replay_Playback_Stop

G_Sequence_Replay_Playback_Stop — Stop playback of a replay sequence

Description

This command stops the playback of a replay sequence.

Definition

```
G_Error_t
G_Sequence_Replay_Playback_Stop(
 const G_PortHandle_t  portHandle,
 const G_Sequence_Replay_Handle_t  sequenceHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

`sequenceHandle`
: Handle to the sequence

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Sequence_Replay_GetState

G_Sequence_Replay_GetState — Get state of a replay sequence

Description

This command queries the state of a replay sequence.

Definition

```
G_Error_t
G_Sequence_Replay_GetState(
 const G_PortHandle_t  portHandle,
 const G_Sequence_Replay_Handle_t  sequenceHandle,
 G_Sequence_Replay_GetState_RspFlags_t * const rspFlags,
 u32_t * const actualEntry,
 u32_t * const numberOfEntries
);
```

Parameters

`portHandle`
Handle to the communication port

`sequenceHandle`
Handle to the sequence

`rspFlags`
Response flags

The following values are possible and can be combined:

`G_SEQUENCE__REPLAY__GET_STATE__RSP_FLAG__ZERO`
No flag is set

`G_SEQUENCE__REPLAY__GET_STATE__RSP_FLAG__IS_RECORDING`
A recording is currently in progress

`G_SEQUENCE__REPLAY__GET_STATE__RSP_FLAG__IS_PLAYING`
A playback is currently in progress

`actualEntry`
Returns the number of the actual sequence entry

`numberOfEntries`
Returns the number of all sequence entries

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Sequence_Replay_AddDelay

G_Sequence_Replay_AddDelay — Add delay to a replay sequence

Description

This command adds a delay to a replay sequence.

Definition

```
G_Error_t
G_Sequence_Replay_AddDelay(
 const G_PortHandle_t  portHandle,
 const G_Sequence_Replay_Handle_t  sequenceHandle,
 const u32_t  delay
);
```

Parameters

`portHandle`
: Handle to the communication port

`sequenceHandle`
: Handle to the sequence

`delay`
: Delay time in microseconds

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Sequence_Replay_SaveToFile

G_Sequence_Replay_SaveToFile — Save a replay sequence

Description

This command saves a replay sequence into the internal memory of the hardware.

Definition

```
G_Error_t
G_Sequence_Replay_SaveToFile(
 const G_PortHandle_t  portHandle,
 const G_Sequence_Replay_Handle_t  sequenceHandle,
 const G_Sequence_Replay_SaveToFile_CmdFlags_t  cmdFlags,
 const char * const filename
);
```

Parameters

`portHandle`
Handle to the communication port

`sequenceHandle`
Handle to the sequence

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_SEQUENCE__REPLAY__SAVE_TO_FILE__CMD_FLAG__ZERO`
No flag is set

`G_SEQUENCE__REPLAY__SAVE_TO_FILE__CMD_FLAG__APPEND`
The replay sequence is appended to an already existing file

`filename`
Filename for the replay sequence file

The file is saved within the permanent memory on the hardware. Therefore it is still available after a POR (Power On Reset).

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Sequence_Replay_LoadFromFile

G_Sequence_Replay_LoadFromFile — Load a replay sequence

Description

This command loads a replay sequence from the internal memory of the hardware.

Definition

```
G_Error_t
G_Sequence_Replay_LoadFromFile(
 const G_PortHandle_t  portHandle,
 const G_Sequence_Replay_Handle_t  sequenceHandle,
 const G_Sequence_Replay_LoadFromFile_CmdFlags_t  cmdFlags,
 const char * const filename
);
```

Parameters

`portHandle`
Handle to the communication port

`sequenceHandle`
Handle to the sequence

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_SEQUENCE__REPLAY__LOAD_FROM_FILE__CMD_FLAG__ZERO`
No flag is set

`G_SEQUENCE__REPLAY__LOAD_FROM_FILE__CMD_FLAG__APPEND`
The replay sequence is appended to the existing sequence

`filename`
Filename of the replay sequence file

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Sequence_Replay_Property_SetById

G_Sequence_Replay_Property_SetById — Set sequence property

Description

Set a property of the sequence

Definition

```
G_Error_t
G_Sequence_Replay_Property_SetById(
 const G_PortHandle_t  portHandle,
 const G_Sequence_Replay_Handle_t  sequenceHandle,
 const G_Sequences_Replay_PropertyId_t  id,
 const u32_t  value
);
```

Parameters

`portHandle`
Handle to the communication port

`sequenceHandle`
Handle to the sequence

`id`
Property ID

The following values are possible:

`G_SEQUENCE__REPLAY__PROPERTY_ID__ALLOW_STARTING_WHEN_PLAYING`
Allow to restart the sequence if it is already running.

This property can also be set by starting the sequence with G_Sequence_Replay_Playback_Start and command flag `G_SEQUENCE__REPLAY__PLAYBACK__CMD_FLAG__ALLOW_STARTING_WHEN_PLAYING` set.

`G_SEQUENCE__REPLAY__PROPERTY_ID__IGNORE_PLAYBACK_START__NUMBER`
Number of times the command G_Sequence_Replay_Playback_Start will be ingored

0xFFFFFFFF = always ignore

When changing the value of this property, the property value of `G_SEQUENCE__REPLAY__PROPERTY_ID__IGNORE_PLAYBACK_START__COUNTER` will automatically be set to the same value.

`G_SEQUENCE__REPLAY__PROPERTY_ID__IGNORE_PLAYBACK_START__COUNTER`
Counter of disregards of the G_Sequence_Replay_Playback_Start command (decreasing)

0xFFFFFFFF = always ignore

If the counter is zero and command G_Sequence_Replay_Playback_Start is called, the call will not be ignored and the counter will be set to the value of `G_SEQUENCE__REPLAY__PROPERTY_ID__IGNORE_PLAYBACK_START__NUMBER`.

If the counter is not zero and command G_Sequence_Replay_Playback_Start is called, the call will be ignored and the counter will be decremented by one.

`G_SEQUENCE__REPLAY__PROPERTY_ID__REPETITIONS__NUMBER`
Number of repetitions of a sequence

0xFFFFFFFF = always repeat sequence

When changing the value of this property, the property value of `G_SEQUENCE__REPLAY__PROPERTY_ID__REPETITIONS__COUNTER` will automatically be set to the same value.

`G_SEQUENCE__REPLAY__PROPERTY_ID__REPETITIONS__COUNTER`
Counter of the repetitions of a sequence (decreasing)

0xFFFFFFFF = always repeat sequence

When the counter is zero and the end of the sequence is reached, the playback will be stopped and the counter will be set to the value of `G_SEQUENCE__REPLAY__PROPERTY_ID__REPETITIONS__NUMBER`.

If the counter is not zero while the end of the sequence is reached, the playback will be restarted and the counter will be decreased by one.

`value`
Property value (e.g. **0** for **false**, **1** for **true**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Sequence_Replay_Property_GetById

G_Sequence_Replay_Property_GetById — Query sequence property

Description

Query a property of the sequence

Definition

```
G_Error_t
G_Sequence_Replay_Property_GetById(
 const G_PortHandle_t  portHandle,
 const G_Sequence_Replay_Handle_t  sequenceHandle,
 const G_Sequences_Replay_PropertyId_t  id,
 u32_t * const value
);
```

Parameters

`portHandle`
Handle to the communication port

`sequenceHandle`
Handle to the sequence

`id`
Property ID

The following values are possible:

`G_SEQUENCE__REPLAY__PROPERTY_ID__ALLOW_STARTING_WHEN_PLAYING`
Allow to restart the sequence if it is already running.

`G_SEQUENCE__REPLAY__PROPERTY_ID__AUTOPLAY_SEQUENCE_HANDLE`
If the **AutoPlay** feature of the device is enabled and a sequence was started at Power On of the device, the handle of this sequence is returned.

`G_SEQUENCE__REPLAY__PROPERTY_ID__IGNORE_PLAYBACK_START__NUMBER`
Number of times the command G_Sequence_Replay_Playback_Start will be ingored

0xFFFFFFFF = always ignore

When changing the value of this property, the property value of `G_SEQUENCE__REPLAY__PROPERTY_ID__IGNORE_PLAYBACK_START__COUNTER` will automatically be set to the same value.

`G_SEQUENCE__REPLAY__PROPERTY_ID__IGNORE_PLAYBACK_START__COUNTER`
Counter of disregards of the G_Sequence_Replay_Playback_Start command (decreasing)

0xFFFFFFFF = always ignore

If the counter is zero and command G_Sequence_Replay_Playback_Start is called, the call will not be ignored and the counter will be set to the value of `G_SEQUENCE__REPLAY__PROPERTY_ID__IGNORE_PLAYBACK_START__NUMBER`.

If the counter is not zero and command G_Sequence_Replay_Playback_Start is called, the call will be ignored and the counter will be decremented by one.

`G_SEQUENCE__REPLAY__PROPERTY_ID__REPETITIONS__NUMBER`
Number of repetitions of a sequence

0xFFFFFFFF = always repeat sequence

When changing the value of this property, the property value of `G_SEQUENCE__REPLAY__PROPERTY_ID__REPETITIONS__COUNTER` will automatically be set to the same value.

`G_SEQUENCE__REPLAY__PROPERTY_ID__REPETITIONS__COUNTER`
Counter of the repetitions of a sequence (decreasing)

0xFFFFFFFF = always repeat sequence

When the counter is zero and the end of the sequence is reached, the playback will be stopped and the counter will be set to the value of `G_SEQUENCE__REPLAY__PROPERTY_ID__REPETITIONS__NUMBER`.

If the counter is not zero while the end of the sequence is reached, the playback will be restarted and the counter will be decreased by one.

`value`
Returns property value (e.g. **0** for **false**, **1** for **true**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Sequence_Replay_Autoplay_Enable

G_Sequence_Replay_Autoplay_Enable — Enable AutoPlay feature

Description

Enable the **AutoPlay** feature of the device

When the **AutoPlay** feature is enabled, the device can automatically play back a sequence after a power on.

Definition

```
G_Error_t
G_Sequence_Replay_AutoPlay_Enable(
 const G_PortHandle_t  portHandle,
 const G_Sequence_Replay_AutoPlay_Enable_CmdFlags_t  cmdFlags,
 const char * const filename
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_SEQUENCE__REPLAY__AUTOPLAY__ENABLE__CMD_FLAG__ZERO`
No command flag is set

`G_SEQUENCE__REPLAY__AUTOPLAY__ENABLE__CMD_FLAG__DIS_FILE_EXISTING_CHECK`
Disable the "No file existing" - check

When set, you can specify a sequence file for **AutoPlay** that does not exist, yet. The file can be added later.

If the device powers up and the sequence file for **AutoPlay** is not available, no sequence will be played.

`filename`
Filename of the replay sequence file

A recorded sequence can be saved to a file with command G_Sequence_Replay_SaveToFile.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Example

```
G_Sequence_Replay_Handle_t  sequenceHandle;
G_PortHandle_t  cmdPort;
G_PortHandle_t  seqPort;
char  rsp[256];
u32_t  rspLength = 256;
G_Error_t  rc;
```

```
// open command interface
rc =
 G_Common_OpenInterface(
  &cmdPort,
  "CAN1"
 );

if (rc != G_NO_ERROR) {
 return;
}

// open sequence interface
rc =
 G_Common_OpenInterface(
  &seqPort,
  "SEQ1"
 );

if (rc != G_NO_ERROR) {
 return;
}

// allocate sequence handle

rc =
 G_Sequence_Replay_Allocate(
  seqPort,
  &sequenceHandle
 );

if (rc != G_NO_ERROR) {
 return;
}

// start recording
rc =
 G_Sequence_Replay_Recording_Start(
  seqPort,
  sequenceHandle,
  G_SEQUENCE__REPLAY__RECORDING__CMD_FLAG__ZERO
 );

if (rc != G_NO_ERROR) {
 return;
}

// the sequence is made of one call to 'G_Common_GetFirmwareVersion'

rc =
 G_Common_GetFirmwareVersion(
  cmdPort,
  rsp,
  &rspLength
 );

if (rc != G_NO_ERROR) {
 return;
```

```
}

// stop recording

rc =
 G_Sequence_Replay_Recording_Stop(
  seqPort,
  sequenceHandle
 );

if (rc != G_NO_ERROR) {
 return;
}

// save sequence to file

rc =
 G_Sequence_Replay_SaveToFile(
  seqPort,
  sequenceHandle,
  G_SEQUENCE__REPLAY__SAVE_TO_FILE__CMD_FLAG__ZERO,
  "sequence1"
 );

if (rc != G_NO_ERROR) {
 return;
}

// deallocate sequence handle to free resources

rc =
 G_Sequence_Replay_Deallocate(
  seqPort,
  sequenceHandle
 );

if (rc != G_NO_ERROR) {
 return;
}

// enable AutoPlay feature

rc =
 G_Sequence_Replay_AutoPlay_Enable(
  seqPort,
  G_SEQUENCE__REPLAY__AUTOPLAY__ENABLE__CMD_FLAG__ZERO,
  "sequence1"
 );

if (rc != G_NO_ERROR) {
 return;
}
```

G_Sequence_Replay_AutoPlay_Disable

G_Sequence_Replay_AutoPlay_Disable — Disable AutoPlay feature

Description

Disable the **AutoPlay** feature of the device

Definition

```
G_Error_t
G_Sequence_Replay_AutoPlay_Disable(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Transport Protocol Functions

The functions provided in file "g_api_tp.dll" can be used to perform common transport protocol tasks including multichannel management and accessing transport protocol functions directly without a diagnostics protocol.

Before the common transport protocol functions can be used, the transport protocol has to be initialized. This is done by the bus specific transport protocol functions of each communication DLL (e.g. G_FlexRay_Tp_Init_Iso10681).

Currently, only **FlexRay** and **Ethernet** interfaces are supported by the common transport protocol functions in this DLL. When other buses protocols are used, refer to the specific functions in the communication DLLs.

1 Common Tp Functions

These commands provide access to common transport protocol functions including multisession channel management and reset.

Multisession Note

A multisession channel is required if multiple applications are working with the transport protocol on the same interface. For each channel, the transport protocol can be configured with different parameters.

Usually, it is not neccessary to use multisession channels. It is recommended to use channel **0** and to disregard the multisession channel management functions in these cases.

G_Tp_Channel_Get

G_Tp_Channel_Get — Allocate a multisession channel

Description

Use this command to allocate a multisession channel. (See Multisession Note for multisession details)

Definition

```
G_Error_t
G_Tp_Channel_Get(
 const G_PortHandle_t portHandle,
 u8_t * const channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Returns multisession channel

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Tp_Channel_Free

G_Tp_Channel_Free — Free a multisession channel

Description

Use this command to free a multisession channel. (See Multisession Note for multisession details)

Definition

```
G_Error_t
G_Tp_Channel_Free(
 const G_PortHandle_t portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
    Handle to the communication port

`channel`
    Multisession channel to be freed

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Tp_Reset

G_Tp_Reset — Reset transport protocol parameters

Description

Use this command to reset all transport protocol parameters.

Definition

```
G_Error_t
G_Tp_Reset(
 const G_PortHandle_t portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

2 Tp Direct

These commands provide direct access to transport protocol functions.

Usually the transport protocol is used by a diagnostics protocol and therefore the user cannot influence its behavior directly. With the help of the Tp Direct functions, it is possible to handle data using a transport protocol directly without the need of a diagnostics protocol.

Before the tp direct functions can be used, the transport protocol has to be initialized. This is done by the bus specific transport protocol functions of each communication DLL (e.g. G_FlexRay_Tp_Init_Iso10681). In addition to this, Tp Direct has to be initialized with G_Tp_Direct_Init .

G_Tp_Direct_Reset

G_Tp_Direct_Reset — Reset all Tp Direct parameters

Description

Use this function to reset all Tp Direct parameters.

Definition

```
G_Error_t
G_Tp_Direct_Reset(
 const G_PortHandle_t portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (see Multisession Note for details)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Tp_Direct_Init

G_Tp_Direct_Init — Initialize Tp Direct

Description

Use this command to initialize the direct access to the transport protocol.

Definition

```
G_Error_t
G_Tp_Direct_Init(
 const G_PortHandle_t portHandle,
 const u8_t  channel,
 const G_Tp_Direct_Init_Parameters_t * const parameters
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (see Multisession Note for details)

`parameters`
Structure with Tp Direct parameters

The structure contains the following elements:

`Flags`
Tp Direct flags

The following values are possible and can be combined:

`G_TP__DIRECT__INIT__FLAG__NONE`
No flag is set

`RxBufferSize`
Size of receive buffer in bytes (**0** = use maximum Tp message size)

`TxBufferSize`
Size of transmit buffer in bytes (**0** = use maximum Tp message size)

`NumberOfRxBuffers`
Number of receive buffers of the specified size (shared by physical and functional messages), e.g. **3**

`NumberOfTxBuffers`
Number of transmit buffers of the specified size (shared by physical and functional messages), e.g. **2**

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Tp_Direct_GetState

G_Tp_Direct_GetState — Query current Tp Direct state

Description

Use this command to query the current state of the Tp Direct.

Definition

```
G_Error_t
G_Tp_Direct_GetState(
 const G_PortHandle_t portHandle,
 const u8_t  channel,
 const G_Tp_Direct_GetState_CmdFlags_t  cmdFlags,
 G_Tp_Direct_GetState_Response_t * const response
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (see Multisession Note for details)

`cmdFlags`
Command Flags

The following values are possible and can be combined:

`G_TP__DIRECT__GET_STATE__CMD_FLAG__NONE`
No flag is set

`G_TP__DIRECT__GET_STATE__CMD_FLAG__RESET_LAST_ERROR`
The Tp Direct error buffer will be reset after the last error code is read

`response`
Returns structure with response data

The structure contains the following elements:

`Flags`
Response Flags

The following values are possible and can be combined:

`G_TP__DIRECT__GET_STATE__RSP_FLAG__NONE`
No flag is set

`G_TP__DIRECT__GET_STATE__RSP_FLAG__RX_BUFFER_NOT_EMPTY`
The receive buffer contains data

`G_TP__DIRECT__GET_STATE__RSP_FLAG__TX_BUSY`
A transmission is currently in progress

`G_TP__DIRECT__GET_STATE__RSP_FLAG__TX_BUSY__PHYSICAL`
A physical (unicast) transmission is currently in progress

G_TP__DIRECT__GET_STATE__RSP_FLAG__TX_BUSY__FUNCTIONAL
A functional (multicast) transmission is currently in progress

G_TP__DIRECT__GET_STATE__RSP_FLAG__RX_BUSY
A data reception is currently in progress

G_TP__DIRECT__GET_STATE__RSP_FLAG__RX_BUSY__PHYSICAL
A physical (unicast) data reception is currently in progress

G_TP__DIRECT__GET_STATE__RSP_FLAG__RX_BUSY__FUNCTIONAL
A functional (multicast) data reception is currently in progress

LastErrorCode
The Tp Direct error code that occurred last

A description for the error code can be obtained by G_Common_GetFirmwareErrorDescription.

State
Tp Direct state

The following values are possible:

G_TP__DIRECT__STATE__UNKNOWN
Unknown state

G_TP__DIRECT__STATE__NOT_INITIALIZED
Tp Direct not initialized

G_TP__DIRECT__STATE__DISCONNECTED
Tp Direct is not connected

G_TP__DIRECT__STATE__CONNECT_IN_PROGRESS
Tp Direct is currently connecting

G_TP__DIRECT__STATE__CONNECTED
Tp Direct is connected

G_TP__DIRECT__STATE__DISCONNECT_IN_PROGRESS
Tp Direct is currently disconnecting

Type
Tp Direct type

The following values are possible:

G_TP__DIRECT__TYPE__UNKNOWN
Unknown type

G_TP__DIRECT__TYPE__OFF
The Tp Direct is turned off (not initialized)

G_TP__DIRECT__TYPE__NORMAL
The Tp Direct is in normal operation state

reserved1
reserved parameter (must be initialized with **0**)

reserved2
reserved parameter (must be initialized with **0**)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Tp_Direct_Start

G_Tp_Direct_Start — Start Tp Direct

Description

Use this function to start Tp Direct after it has been initialized with G_Tp_Direct_Init .

`mode`, `length` and `data` can be used if a request should be sent when Tp Direct is started. If no request should be sent, set `length` to **`0`** .

Definition

```
G_Error_t
G_Tp_Direct_Start(
 const G_PortHandle_t portHandle,
 const u8_t  channel,
 const G_Tp_Direct_Mode_t  mode,
 const u32_t  length,
 const u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (see Multisession Note for details)

`mode`
Tp Direct request mode

The following values are possible:

`G_TP__DIRECT__MODE__UNKNOWN`
Unknown mode

`G_TP__DIRECT__MODE__PHYSICAL`
Physical request mode (unicast)

`G_TP__DIRECT__MODE__FUNCTIONAL`
Functional request mode (multicast)

`length`
Request length

If `length` > 0, a request is sent when starting Tp Direct.

`data`
Request data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Tp_Direct_Stop

G_Tp_Direct_Stop — Stop Tp Direct

Description

Use this function to stop Tp Direct.

`mode`, `length` and `data` can be used if a request should be sent when Tp Direct is stopped. If no request should be sent, set `length` to **`0`**.

Definition

```
G_Error_t
G_Tp_Direct_Stop(
 const G_PortHandle_t portHandle,
 const u8_t  channel,
 const G_Tp_Direct_Mode_t  mode,
 const u32_t  length,
 const u8_t * const data
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (see Multisession Note for details)

`mode`
: Tp Direct request mode (see G_Tp_Direct_Mode_t)

`length`
: Request length

  If `length` > 0, a request is sent when stopping Tp Direct.

`data`
: Request data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Tp_Direct_GetResponse

G_Tp_Direct_GetResponse — Get Tp Direct response

Description

Use this function to query a Tp Direct Response from the response buffer.

Definition

```
G_Error_t
G_Tp_Direct_GetResponse(
 const G_PortHandle_t portHandle,
 const u8_t  channel,
 G_Tp_Direct_GetResponse_RspParameters_t * const rspParameters,
 u32_t * const length,
 u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (see Multisession Note for details)

`rspParameters`
Returns structure with additional response information

The structure contains the following elements:

`Flags`
Response Flags

The following values are possible and can be combined:

`G_TP__DIRECT__GET_RESPONSE__RSP_FLAG__NONE`
No flag is set

`G_TP__DIRECT__GET_RESPONSE__RSP_FLAG__TX_BUSY`
A transmission is currently in progress

`G_TP__DIRECT__GET_RESPONSE__RSP_FLAG__RX_BUSY`
A reception is currently in progress

`G_TP__DIRECT__GET_RESPONSE__RSP_FLAG__VALID`
The response data is valid

If this flag is not set, an error occurred during reception.

`G_TP__DIRECT__GET_RESPONSE__RSP_FLAG__BUFFER_NOT_EMPTY`
More responses are available

`G_TP__DIRECT__GET_RESPONSE__RSP_FLAG__BUFFER_OVERRUN`
Indicates that the internal RX buffer was too small and data got lost

`G_TP__DIRECT__GET_RESPONSE__RSP_FLAG__TX_BUSY__PHYSICAL`
A physical (unicast) transmission is currently in progress

`G_TP__DIRECT__GET_RESPONSE__RSP_FLAG__TX_BUSY__FUNCTIONAL`
A functional (multicast) transmission is currently in progress

`G_TP__DIRECT__GET_RESPONSE__RSP_FLAG__RX_BUSY__PHYSICAL`
A physical (unicast) reception is currently in progress

`G_TP__DIRECT__GET_RESPONSE__RSP_FLAG__RX_BUSY__FUNCTIONAL`
A functional (multicast) reception is currently in progress

`ErrorCode`
Error Code value in case of an error, otherwise **0x00** (NO_ERROR)

A description for the error code can be obtained by G_Common_GetFirmwareErrorDescription.

`State`
Tp Direct state (see G_Tp_Direct_State_t for details)

`Mode`
Tp Direct Mode (see G_Tp_Direct_Mode_t for details)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`length`
in : size of data buffer `data` in bytes

out : size of data that is returned (in bytes)

`data`
Data buffer

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Tp_Direct_GetResponse_Async_AddCallback

G_Tp_Direct_GetResponse_Async_AddCallback — Set the callback function for asynchronous TP direct responses

Description

Use this command to set the callback function for asynchronous TP direct responses.

The following sequence of commands is necessary for receiving asynchronous TP Direct responses:

- Initialize TP (e.g. G_Can_Tp_Config_Iso15765)
- Initialize TP direct with G_Tp_Direct_Init
- Add callback function for responses with G_Tp_Direct_GetResponse_Async_AddCallback
- Enable asynchronous communication with G_AsyncCommunication_Enable
- Enable asynchronous TP Direct resonses with G_Tp_Direct_GetResponse_Async_Enable
- Start TP Direct with G_Tp_Direct_Start

To reset TP Direct, call these commands in reverse order.

Definition

```
G_Error_t
G_Tp_Direct_GetResponse_Async_AddCallback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Tp_Direct_GetResponse_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (see Multisession Note for details)

`callback`
Function pointer of the callback function (see G_Tp_Direct_GetResponse_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Tp_Direct_GetResponse_Async_Callback

G_Tp_Direct_GetResponse_Async_Callback — Callback function for asynchronous TP direct responses

Description

Callback function for asynchronous TP direct responses

The following sequence of commands is necessary for receiving asynchronous TP Direct responses:

- Initialize TP (e.g. G_Can_Tp_Config_Iso15765)
- Initialize TP direct with G_Tp_Direct_Init
- Add callback function for responses with G_Tp_Direct_GetResponse_Async_AddCallback
- Enable asynchronous communication with G_AsyncCommunication_Enable
- Enable asynchronous TP Direct resonses with G_Tp_Direct_GetResponse_Async_Enable
- Start TP Direct with G_Tp_Direct_Start

To reset TP Direct, call these commands in reverse order.

Definition

```
void
G_Tp_Direct_GetResponse_Async_Callback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Tp_Direct_GetResponse_RspParameters_t * const rspParameters,
 const u32_t  length,
 const u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (see Multisession Note for details)

`rspParameters`
Returns structure with additional response information

The structure contains the following elements:

`Flags`
Response Flags

The following values are possible and can be combined:

`G_TP__DIRECT__GET_RESPONSE__RSP_FLAG__NONE`
No flag is set

`G_TP__DIRECT__GET_RESPONSE__RSP_FLAG__TX_BUSY`
A transmission is currently in progress

`G_TP__DIRECT__GET_RESPONSE__RSP_FLAG__RX_BUSY`
A reception is currently in progress

`G_TP__DIRECT__GET_RESPONSE__RSP_FLAG__VALID`
The response data is valid

If this flag is not set, an error occurred during reception.

`G_TP__DIRECT__GET_RESPONSE__RSP_FLAG__BUFFER_NOT_EMPTY`
More responses are available

`G_TP__DIRECT__GET_RESPONSE__RSP_FLAG__BUFFER_OVERRUN`
Indicates that the internal RX buffer was too small and data got lost

`G_TP__DIRECT__GET_RESPONSE__RSP_FLAG__TX_BUSY__PHYSICAL`
A physical (unicast) transmission is currently in progress

`G_TP__DIRECT__GET_RESPONSE__RSP_FLAG__TX_BUSY__FUNCTIONAL`
A functional (multicast) transmission is currently in progress

`G_TP__DIRECT__GET_RESPONSE__RSP_FLAG__RX_BUSY__PHYSICAL`
A physical (unicast) reception is currently in progress

`G_TP__DIRECT__GET_RESPONSE__RSP_FLAG__RX_BUSY__FUNCTIONAL`
A functional (multicast) reception is currently in progress

`ErrorCode`
Error Code value in case of an error, otherwise **0x00** (NO_ERROR)

A description for the error code can be obtained by G_Common_GetFirmwareErrorDescription.

`State`
Tp Direct state (see G_Tp_Direct_State_t for details)

`Mode`
Tp Direct Mode (see G_Tp_Direct_Mode_t for details)

`reserved1`
reserved parameter (must be initialized with **0**)

`reserved2`
reserved parameter (must be initialized with **0**)

`length`
in : size of data buffer `data` in bytes

out : size of data that is returned (in bytes)

`data`
Data buffer

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Tp_Direct_GetResponse_Async_Enable

G_Tp_Direct_GetResponse_Async_Enable — Enable asynchronous responses for TP direct

Description

Use this command to enable asynchronous responses for TP direct.

The following sequence of commands is necessary for receiving asynchronous TP Direct responses:

- Initialize TP (e.g. G_Can_Tp_Config_Iso15765)
- Initialize TP direct with G_Tp_Direct_Init
- Add callback function for responses with G_Tp_Direct_GetResponse_Async_AddCallback
- Enable asynchronous communication with G_AsyncCommunication_Enable
- Enable asynchronous TP Direct resonses with G_Tp_Direct_GetResponse_Async_Enable
- Start TP Direct with G_Tp_Direct_Start

To reset TP Direct, call these commands in reverse order.

Definition

```
G_Error_t
G_Tp_Direct_GetResponse_Async_Enable(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (see Multisession Note for details)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Tp_Direct_GetResponse_Async_Disable

G_Tp_Direct_GetResponse_Async_Disable — Disable asynchronous responses for TP direct

Description

Use this command to disable asynchronous responses for TP direct.

Definition

```
G_Error_t
G_Tp_Direct_GetResponse_Async_Disable(
 const G_PortHandle_t  portHandle,
 const u8_t  channel
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (see Multisession Note for details)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Tp_Direct_GetResponse_Async_RemoveCallback

G_Tp_Direct_GetResponse_Async_RemoveCallback — Remove the callback function for asynchronous TP direct responses

Description

Use this command to remove the callback function for asynchronous TP direct responses.

Definition

```
G_Error_t
G_Tp_Direct_GetResponse_Async_AddCallback(
 const G_PortHandle_t  portHandle,
 const u8_t  channel,
 const G_Tp_Direct_GetResponse_Async_Callback_t  callback
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (see Multisession Note for details)

`callback`
Function pointer of the callback function (see G_Tp_Direct_GetResponse_Async_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Tp_Direct_QueueRequest

G_Tp_Direct_QueueRequest — Queue Tp Direct Request

Description

Use this command to queue a Tp Direct Request.

In contrast to G_Tp_Direct_Request , the transmission is only triggered and the command returns before the request has been sent completely.

Definition

```
G_Error_t
G_Tp_Direct_QueueRequest(
 const G_PortHandle_t portHandle,
 const u8_t  channel,
 const G_Tp_Direct_Mode_t  mode,
 const u32_t  length,
 const u8_t * const data
);
```

Parameters

`portHandle`
: Handle to the communication port

`channel`
: Multisession channel (see Multisession Note for details)

`mode`
: Tp Direct request mode (see G_Tp_Direct_Mode_t for details)

`length`
: Request data length

`data`
: Request data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_Tp_Direct_Request

G_Tp_Direct_Request — Send Tp Direct Request

Description

Use this command to send a Tp Direct Request.

In contrast to G_Tp_Direct_QueueRequest , the command returns not before the request has been sent completely or a timeout occurred.

Definition

```
G_Error_t
G_Tp_Direct_Request(
 const G_PortHandle_t portHandle,
 const u8_t  channel,
 const G_Tp_Direct_Mode_t  mode,
 const u32_t  txTimeout,
 const u32_t  length,
 const u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`channel`
Multisession channel (see Multisession Note for details)

`mode`
Tp Direct request mode (see G_Tp_Direct_Mode_t for details)

`txTimeout`
Maximum time for sending the request in milliseconds

If the request can not be sent within `txTimeout`, the command returns with an error.

`length`
Request data length

`data`
Request data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

UserCode Functions

The **User Code** interface provides access to functions for storing, executing, debugging and deleting code, written by the user with the optionally available **Net2Run IDE** toolkit.

The **Net2Run IDE** is a comprehensive C/C++ code development tool chain for **GOEPEL electronic** intelligent communication controllers of **Series 61** . Giving developers an industry standard IDE, a cross compiler tool chain and Ethernet based remote debugging facilities. Enabling users to develop test programs for CAN, LIN, K-Line and FlexRay vehicle networks, that can be executed directly on the Series 61's PowerPC CPU. Bringing an excellent real-time performance, low latency responses as well as standalone operation capabilities to your test stands.

1 File Operations

These functions provide control over User Code Files, created with the **Net2Run IDE** .

G_UserCode_File_AutoPlay_Enable

G_UserCode_File_AutoPlay_Enable — Enable automatic execution of User Code

Description

Use this command to enable automatic execution of a User Code file when the device is started.

Definition

```
G_Error_t
G_UserCode_File_AutoPlay_Enable(
 const G_PortHandle_t  portHandle,
 const G_UserCode_File_Autoplay_Enable_CmdFlags_t  cmdFlags,
 const char * const filePath
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command Flags

The following values are possible and can be combined:

`G_USER_CODE__FILE__AUTOPLAY__ENABLE__CMD_FLAG__NONE`
No flag is set

`G_USER_CODE__FILE__AUTOPLAY__ENABLE__CMD_FLAG__NO_FILE_EXISTING_CHECK`
Disable check for existing file

No error is returned if the file defined by `filePath` does not exist

`filePath`
Path and name of the file on the device

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_UserCode_File_AutoPlay_Disable

G_UserCode_File_AutoPlay_Disable — Disable automatic execution of User Code

Description

Use this command to disable automatic execution of a User Code file when the device is started.

Definition

```
G_Error_t
G_UserCode_File_AutoPlay_Disable(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
    Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_UserCode_File_Delete

G_UserCode_File_Delete — Delete a User Code File from the device

Description

This command deletes a User Code File from the device.

Definition

```
G_Error_t
G_UserCode_File_Delete(
 const G_PortHandle_t  portHandle,
 const char * const filePath
);
```

Parameters

`portHandle`
: Handle to the communication port

`filePath`
: Path and name of the file on the device

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_UserCode_File_GetNumberOfFiles

G_UserCode_File_GetNumberOfFiles — Return number of files for directory

Description

Use this command to query the number of files that are stored in a specific directory of the device.

The value provided by `directory` specifies a sub-directory within the User Code Directory of the device. Therefore a value of `""` specifies the root User Code Directory.

Definition

```
G_Error_t
G_UserCode_File_GetNumberOfFiles(
 const G_PortHandle_t  portHandle,
 const char * const directory,
 u32_t * const numberOfFiles
);
```

Parameters

`portHandle`
: Handle to the communication port

`directory`
: Sub-directory within User Code Directory on the device

  Value `""` specifies the root User Code Directory.

`numberOfFiles`
: Returns the number of files that are stored in the specified directory (excluding files stored in sub-directories)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_UserCode_File_GetFileInfoByNumber

G_UserCode_File_GetFileInfoByNumber — Return file information

Description

Use this command to query the file name and the file size for a file in the User Code Directory on the device or any of its sub-directories specified by `directory`.

The total number of files in a directory can be obtained by G_UserCode_File_GetNumberOfFiles .

Definition

```
G_Error_t
G_UserCode_File_GetFileInfoByNumber(
 const G_PortHandle_t  portHandle,
 const char * const directory,
 const u32_t  fileNumber,
 u32_t * const fileSize,
 u32_t * const fileNameSize,
 char * const fileName
);
```

Parameters

`portHandle`
Handle to the communication port

`directory`
Sub-directory within User Code Directory on the device

Value "" specifies the root User Code Directory.

`fileNumber`
Number of the file whose information is queried

The total number of files in a directory can be obtained by G_UserCode_File_GetNumberOfFiles .

`fileSize`
Returns the size of the file in bytes

`fileNameSize`
in : Size of the buffer provided by `fileName` (in bytes)

out : Length of the file name returned in `fileName` (in characters, without terminating zero)

`fileName`
Returns file name

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_UserCode_File_GetNumberOfDirectories

G_UserCode_File_GetNumberOfDirectories — Return number of subdirectories for a parent directory

Description

Use this command to query the number of subdirectories that are located within a specific directory of the device.

The value provided by `parentDirectory` specifies a sub-directory within the User Code Directory of the device. Therefore a value of `""` specifies the root User Code Directory.

Definition

```
G_Error_t
G_UserCode_File_GetNumberOfDirectories(
 const G_PortHandle_t  portHandle,
 const char * const parentDirectory,
 u32_t * const numberOfDirectories
);
```

Parameters

`portHandle`
: Handle to the communication port

`parentDirectory`
: Directory within User Code Directory on the device

  Value `""` specifies the root User Code Directory.

`numberOfDirectories`
: Returns the number of sub directories that are located in the specified parent directory

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_UserCode_File_GetDirectoryNameByNumber

G_UserCode_File_GetDirectoryNameByNumber — Return directory name

Description

Use this command to query the name of a subdirectory that is located within a specific directory of the device.

The value provided by `parentDirectory` specifies a sub-directory within the User Code Directory of the device. Therefore a value of `""` specifies the root User Code Directory.

The total number of subdirectories within a directory can be obtained by G_UserCode_File_GetNumberOfDirectories .

Definition

```
G_UserCode_File_GetDirectoryNameByNumber(
 const G_PortHandle_t  portHandle,
 const char * const parentDirectory,
 const u32_t  directoryNumber,
 u32_t * const directoryNameSize,
 char * const directoryName
);
```

Parameters

`portHandle`
Handle to the communication port

`parentDirectory`
Parent directory within User Code Directory on the device

Value `""` specifies the root User Code Directory.

`directoryNumber`
Number of the directory whose name is queried

The total number of subdirectories within a directory can be obtained by G_UserCode_File_GetNumberOfDirectories .

`directoryNameSize`
in : Size of the buffer provided by `directoryName` (in bytes)

out : Length of the directory name returned in `directoryName` (in characters, without terminating zero)

`directoryName`
Returns directory name

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_UserCode_File_Read

G_UserCode_File_Read — Read User Code File from device

Description

Use this command to read a User Code File from the device and store it on the local system.

Definition

```
G_Error_t
G_UserCode_File_Read(
 const G_PortHandle_t  portHandle,
 const char * const sourceFilePath,
 const char * const destinationFilePath
);
```

Parameters

`portHandle`
: Handle to the communication port

`sourceFilePath`
: Path of the file on the device

  The path consists of any sub-directory within the User Code File Directory (optional) and the name of the file that should be read.

  For the file "*testFile.x*", stored in the sub-directory "*test*", `sourceFilePath` would be "*test/testFile.x*".

`destinationFilePath`
: Path and name of the file that is saved on the local system

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_UserCode_File_Save

G_UserCode_File_Save — Save User Code file

Description

This command saves a User Code File, created with Net2Run IDE, on the device.

Definition

```
G_Error_t
G_UserCode_File_Save(
 const G_PortHandle_t  portHandle,
 const char * const sourceFilePath,
 const char * const destinationFilePath
);
```

Parameters

`portHandle`
: Handle to the communication port

`sourceFilePath`
: Path of the file to be saved

`destinationFilePath`
: Path and name for the saved file

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_UserCode_File_Start

G_UserCode_File_Start — Start User Code on the device

Description

Use this command to start User Code stored on the device.

Definition

```
G_Error_t
G_UserCode_File_Start(
 const G_PortHandle_t  portHandle,
 const char * const filePath
);
```

Parameters

`portHandle`
: Handle to the communication port

`filePath`
: Path and name of the file on the device

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_UserCode_File_Stop

G_UserCode_File_Stop — Stop User Code on the device

Description

Use this command to stop User Code stored on the device.

Definition

```
G_Error_t
G_UserCode_File_Stop(
 const G_PortHandle_t  portHandle,
 const char * const filePath
);
```

Parameters

`portHandle`
: Handle to the communication port

`filePath`
: Path and name of the file on the device

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

2 Debugger

These functions provide control over the debugger module for User Code Files, created with the **Net2Run IDE .**

G_UserCode_Debugger_Start

G_UserCode_Debugger_Start — Start User Code Debugger service

Description

Use this command to start the debugger service for User Code on the device.

The debugger service enables the Net2Run IDE Debugger to communicate with the device over the selected port.

This is neccessary for every User Code File that needs to be debugged. Note that the `port` remains the same for every session.

Definition

```
G_Error_t
G_UserCode_Debugger_Start(
 const G_PortHandle_t  portHandle,
 const u16_t  port
);
```

Parameters

`portHandle`
: Handle to the communication port

`port`
: Port number of the debugger service

  The debug sessions in the Net2Run IDE have to be configured to use this port number.

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_UserCode_Debugger_Stop

G_UserCode_Debugger_Stop — Stop User Code Debugger service

Description

Use this command to stop the debugger service for User Code on the device.

Definition

```
G_Error_t
G_UserCode_Debugger_Stop(
 const G_PortHandle_t  portHandle
);
```

Parameters

`portHandle`
: Handle to the communication port

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

3 Message Fifo

These functions provide control over the message FIFO functionality of the user code interface.

Message fifos provide a way of communication between different programs (e.g. communication between onboard programs and host programs). After the FIFO has been created by one of both sides, each side can write to and read from the FIFO. A write operation to the FIFO can optionally be signaled by a callback function for the reader side.

G_UserCode_MsgFifo_Fifo_CreateById

G_UserCode_MsgFifo_Fifo_CreateById — Create a message FIFO with a specific id

Description

Use this command to create a message FIFO with a specific FIFO ID.

Definition

```
G_Error_t
G_UserCode_MsgFifo_Fifo_CreateById(
 const G_PortHandle_t  portHandle,
 const G_UserCode_MsgFifo_Fifo_CreateById_CmdFlags_t  cmdFlags,
 const u32_t  fifoId
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_USER_CODE__MSG_FIFO__FIFO__CREATE_BY_ID__CMD_FLAG__NONE`
No flag is set

`G_USER_CODE__MSG_FIFO__FIFO__CREATE_BY_ID__CMD_FLAG__DELETE_MSGS`
If a FIFO with the specified ID already exists, all messages are deleted from the FIFO

`fifoId`
ID of the FIFO (0x00000001..0x7FFFFFFF)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_UserCode_MsgFifo_Fifo_CreateByName

G_UserCode_MsgFifo_Fifo_CreateByName — Create a FIFO with a specific name

Description

Use this command to create a message FIFO with a specific name

Definition

```
G_Error_t
G_UserCode_MsgFifo_Fifo_CreateByName(
 const G_PortHandle_t  portHandle,
 const G_UserCode_MsgFifo_Fifo_CreateByName_CmdFlags_t  cmdFlags,
 const char * const name,
 const u32_t  instanceId,
 u32_t * const fifoId
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_USER_CODE__MSG_FIFO__FIFO__CREATE_BY_NAME__CMD_FLAG__NONE`
No flag is set

`G_USER_CODE__MSG_FIFO__FIFO__CREATE_BY_NAME__CMD_FLAG__DELETE_MSGS`
If a FIFO with the specified name already exists, all messages are deleted from the FIFO

`name`
Name of the FIFO

Zero terminated string, recommendation: use the name of the on board program and a prefix that marks the direction of the FIFO, e.g.: 'OnBoardProgram1_In' for a FIFO from PC to onboard program and 'OnBoardProgram1_Out' for a FIFO from onboard program to PC.

`instanceId`
Instance ID of the FIFO

Allows multiple FIFOs with the same name (but different instance IDs).

`fifoId`
Returns FIFO ID

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_UserCode_MsgFifo_Fifo_Delete

G_UserCode_MsgFifo_Fifo_Delete — Delete a FIFO

Description

Use this command to delete a message FIFO.

Definition

```
G_Error_t
G_UserCode_MsgFifo_Fifo_Delete(
 const G_PortHandle_t  portHandle,
 const G_UserCode_MsgFifo_Fifo_Delete_CmdFlags_t  cmdFlags,
 const u32_t  fifoId
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_USER_CODE__MSG_FIFO__FIFO__DELETE__CMD_FLAG__NONE`
No flag is set

`fifoId`
ID of the FIFO

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_UserCode_MsgFifo_Msg_Write

G_UserCode_MsgFifo_Msg_Write — Write into FIFO

Description

Use this command to write into a message FIFO.

Definition

```
G_Error_t
G_UserCode_MsgFifo_Msg_Write(
 const G_PortHandle_t  portHandle,
 const G_UserCode_MsgFifo_Msg_Write_CmdFlags_t  cmdFlags,
 const u32_t  fifoId,
 const u32_t  length,
 const u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_USER_CODE__MSG_FIFO__MSG__WRITE__CMD_FLAG__NONE`
No flag is set

`fifoId`
ID of the FIFO

`length`
Length of the data to be written (in bytes)

`data`
Data to be written

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_UserCode_MsgFifo_Msg_Read

G_UserCode_MsgFifo_Msg_Read — Read from FIFO

Description

Use this command to read data from a message FIFO.

Definition

```
G_Error_t
G_UserCode_MsgFifo_Msg_Read(
 const G_PortHandle_t  portHandle,
 const G_UserCode_MsgFifo_Msg_Read_CmdFlags_t  cmdFlags,
 const u32_t  fifoId,
 G_UserCode_MsgFifo_Msg_Read_RspFlags_t * const rspFlags,
 u32_t * const length,
 u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`cmdFlags`
Command flags

The following values are possible and can be combined:

`G_USER_CODE__MSG_FIFO__MSG__READ__CMD_FLAG__NONE`
No flag is set

`fifoId`
ID of the FIFO

`rspFlags`
Response flags

The following values are possible and can be combined:

`G_USER_CODE__MSG_FIFO__MSG__READ__RSP_FLAG__NONE`
No flag is set

`G_USER_CODE__MSG_FIFO__MSG__READ__RSP_FLAG__MSG_IS_VALID`
The returned data is a valid message

If this flag is not set, an error occurred (e.g. FIFO was too small, buffer was too small, …).

`G_USER_CODE__MSG_FIFO__MSG__READ__RSP_FLAG__FIFO_NOT_EMPTY`
There are more messages in the FIFO

`length`
Returns the length of the message (in bytes)

`data`
Returns message data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_UserCode_MsgFifo_Msg_Read_Callback_Enable

G_UserCode_MsgFifo_Msg_Read_Callback_Enable — Enable signaling of available data with a callback function

Description

Use this function to enable the signaling of available data with a callback function.

Definition

```
G_Error_t
G_UserCode_MsgFifo_Msg_Read_Callback_Enable(
 const G_PortHandle_t  portHandle,
 const u32_t  fifoId,
 const G_UserCode_MsgFifo_Msg_Read_Callback_t  callbackFunction
);
```

Parameters

`portHandle`
Handle to the communication port

`fifoId`
ID of the FIFO

`callbackFunction`
Function that is called if new data is available (see G_UserCode_MsgFifo_Msg_Read_Callback)

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_UserCode_MsgFifo_Msg_Read_Callback

G_UserCode_MsgFifo_Msg_Read_Callback — Signal available data

Description

This is a callback function that is called when new data is available.

The callback function has to be enabled by G_UserCode_MsgFifo_Msg_Read_Callback_Enable .

Definition

```
void
G_UserCode_MsgFifo_Msg_Read_Callback(
 const G_PortHandle_t  portHandle,
 const u32_t  fifoId,
 const G_UserCode_MsgFifo_Msg_Read_RspFlags_t  rspFlags,
 const u32_t  length,
 const u8_t * const data
);
```

Parameters

`portHandle`
Handle to the communication port

`fifoId`
Returns the ID of the FIFO

`rspFlags`
Returns response flags (see G_UserCode_MsgFifo_Msg_Read_RspFlags_t)

`length`
Returns the length of the message (in bytes)

`data`
Returns message data

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_UserCode_MsgFifo_Msg_Read_Callback_Disable

G_UserCode_MsgFifo_Msg_Read_Callback_Disable — Disable signaling of available data with a callback function

Description

Use this function to disable the signaling of available data with a callback function.

Definition

```
G_Error_t
G_UserCode_MsgFifo_Msg_Read_Callback_Disable(
 const G_PortHandle_t  portHandle,
 const u32_t  fifoId
);
```

Parameters

`portHandle`
: Handle to the communication port

`fifoId`
: ID of the FIFO

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

G_UserCode_MsgFifo_Property_GetById

G_UserCode_MsgFifo_Property_GetById — Query a FIFO property

Description

Use this command to query a message FIFO property.

Definition

```
G_Error_t
G_UserCode_MsgFifo_Property_GetById(
 const G_PortHandle_t  portHandle,
 const u32_t  fifoId,
 const G_UserCode_MsgFifo_PropertyId_t  propertyId,
 u32_t * const value
);
```

Parameters

`portHandle`
Handle to the communication port

`fifoId`
ID of the FIFO

`propertyId`
Message FIFO property to be queried

The following values are possible:

`G_USER_CODE__MSG_FIFO__PROPERTY_ID__CALLBACK_ENABLED`
Returns if a callback function is enabled (use G_UserCode_MsgFifo_Msg_Read_Callback_Enable to enable a callback function for a message FIFO)

`G_USER_CODE__MSG_FIFO__PROPERTY_ID__NUMBER_OF_MSGS_IN_FIFO`
Returns number of available messages in the message FIFO

`G_USER_CODE__MSG_FIFO__PROPERTY_ID__NUMBER_OF_FIFOS`
The number of fifos is returned

`value`
Returns the value for the selected property

Return Value

G_Error_t error code, see file *g_error.h* for a list of all error codes.

Asynchronous Functionality

For dedicated **G-API** functions, there also is an asynchronous version that allows a non-blocking operation because there is no need to wait for a response after the function call. As soon as the firmware response is received, the related callback function will be called.

In case of failure in the firmware while executing the command, the error callback function specified by G_AsyncCommunication_Enable is called.

For using the asynchronous functionality of a function, there are several steps to take which are exemplified with the command G_Common_GetFirmwareVersion_Async .

Asynchonous functionality - using G_Common_GetFirmwareVersion_Async as an example.

1. Establishing connection to the hardware

```
rc = G_Common_OpenInterface(name, &portHandle);
```

2. Activating the asynchronous communication

```
rc =
  G_AsyncCommunication_Enable(
    portHandle,
    ErrorCallbackFunction
  );
```

3. Installing the callback function

```
rc =
  G_Common_GetFirmwareVersion_Async_AddCallback(
    portHandle,
    GetFirmwareVersionCallbackFunction
  );
```

4. Asynchronous processing of the firmware query

```
rc = G_Common_GetFirmwareVersion_Async(portHandle);
```

5. Execution of other commands (the programming flowchart is not blocked by waiting for the firmware version)

```
printf(
  "This text is printed after calling "
  "'G_Common_GetFirmwareVersion_Async'\n"
);
```

6. Deleting the callback function

```
rc =
  G_Common_GetFirmwareVersion_Async_RemoveCallback(
    portHandle
  );
```

7. Deactivating the asynchronous communication

```
            rc = G_AsyncCommunication_Disable(portHandle);
```

8. Disconnecting the connection to the hardware

```
            rc = G_Common_CloseInterface(portHandle);
```

The complete source code:

```
#include <windows.h>
#include <stdio.h>
#include "g_api.h"
#include "g_api_common.h"

// this is the error callback function
void
ErrorCallbackFuntion(
 const G_PortHandle_t  portHandle,
 const G_Error_t  error,
 const u8_t * const errorInfo,
 const u32_t  errorInfoLength
)
{
 UNREFERENCED_PARAMETER(portHandle);
 UNREFERENCED_PARAMETER(errorInfo);
 UNREFERENCED_PARAMETER(errorInfoLength);

 // print error
 printf("ErrorCallBack\n");
 printf("ErrorCode: %d\n", error);
}


// this is the callback function for
// 'G_Common_GetFirmwareVersion_Async'
void GetFirmwareVersionCallbackFunction(
 const G_PortHandle_t  portHandle,
 const char * const version,
 const u32_t  versionLength
)
{
 UNREFERENCED_PARAMETER(portHandle);
 UNREFERENCED_PARAMETER(versionLength);

 // print firmware version
 printf("GetFirmwareVersionCallback\n");
 printf("%s", version);
}

void main(void) {
 G_PortHandle_t  portHandle;
```

```
G_Error_t  rc;
const char * name = "MOST1";

// open interface
rc = G_Common_OpenInterface(name, &portHandle);

if (rc != G_NO_ERROR) {
 printf("%s\n", G_GetLastErrorDescription());
 return;
}

// enable async communication
rc =
 G_AsyncCommunication_Enable(
  portHandle,
  ErrorCallbackFuntion
 );

if (rc != G_NO_ERROR) {
 printf("%s\n", G_GetLastErrorDescription());

 rc = G_Common_CloseInterface(portHandle);

 if (rc != G_NO_ERROR) {
  printf("%\ns", G_GetLastErrorDescription());
 }

 return;
}

// add callback function
rc =
 G_Common_GetFirmwareVersion_Async_AddCallback(
  portHandle,
  GetFirmwareVersionCallbackFunction
 );

if (rc != G_NO_ERROR) {
 printf("%s\n", G_GetLastErrorDescription());

 rc = G_Common_CloseInterface(portHandle);

 if (rc != G_NO_ERROR) {
  printf("%\ns", G_GetLastErrorDescription());
 }

 return;
}

// get firmware version
rc = G_Common_GetFirmwareVersion_Async(portHandle);

if (rc != G_NO_ERROR) {
 printf("%s\n", G_GetLastErrorDescription());

 rc = G_Common_CloseInterface(portHandle);

 if (rc != G_NO_ERROR) {
```

```
   printf("%\ns", G_GetLastErrorDescription());
  }

  return;
 }
```

```
 // here you can call other commands
 // in the meantime the callback function should be called

 printf(
  "This text is printed on screen after calling "
  "'G_Common_GetFirmwareVersion_Async'\n"
 );

 // remove callback function
 rc =
  G_Common_GetFirmwareVersion_Async_RemoveCallback(
   portHandle
  );

 if (rc != G_NO_ERROR) {
  printf("%s\n", G_GetLastErrorDescription());

  rc = G_Common_CloseInterface(portHandle);

  if (rc != G_NO_ERROR) {
   printf("%\ns", G_GetLastErrorDescription());
  }

  return;
 }

 // disable async communication
 rc = G_AsyncCommunication_Disable(portHandle);

 if (rc != G_NO_ERROR) {
  printf("%s\n", G_GetLastErrorDescription());
 }

 // close interface
 rc = G_Common_CloseInterface(portHandle);

 if (rc != G_NO_ERROR) {
  printf("%\ns", G_GetLastErrorDescription());
 }
}
```

Appendix A Connector assignment of the supported hardware

A.1 PCI/PXI 3051

A.1.1 Connector assignment

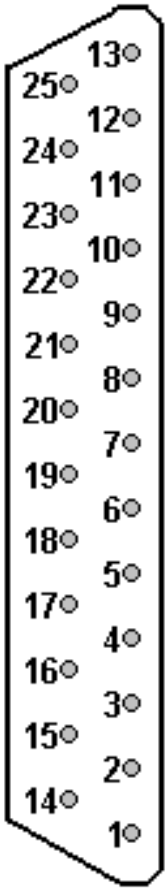

Figure A.1 Connector assignment PXI/PCI 3051

Pin	Signal name	Comment
14	CAN1_High	CAN bus line High
2	CAN1_Low	CAN bus line Low
15	V_Bat1	Reference potential Plus Transceiver
1	GND	Ground Transceiver
17	CAN2_High	CAN bus line High
5	CAN2_Low	CAN bus line Low
18	V_Bat2	Reference potential Plus Transceiver
4	Ground Transceiver	GND
20	CAN3_High	CAN bus line High

Pin	Signal name	Comment
8	CAN3_Low	CAN bus line Low
21	V_Bat3	Reference potential Plus Transceiver
7	GND	Ground Transceiver
23	CAN4_High	CAN bus line High
11	CAN4_Low	CAN bus line Low
24	V_Bat4	Reference potential Plus Transceiver
10	GND	Ground Transceiver
3	INPUT1	Input CAN controller1
16	OUTPUT1	Output CAN controller1
6	INPUT2	Input CAN controller2
19	OUTPUT2	Output CAN controller2
9	INPUT3	Input CAN controller3
22	OUTPUT3	Output CAN controller3
12	INPUT4	Input CAN controller4
25	OUTPUT4	Output CAN controller4
13	GND	Ground

Table A.1 Pin content PXI/PCI 3051

A.2 PCI/PXI 3052

A.2.1 Connector assignment

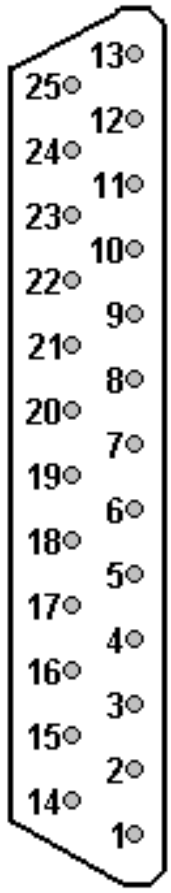

Figure A.2 Connector assignment PXI/PCI 3052

Pin	Signal name	Comment
14	CAN1_High	CAN bus line High
2	CAN1_Low	CAN bus line Low
15	V_Bat1	Reference potential Plus Transceiver
1	GND	Ground Transceiver
17	CAN2_High	CAN bus line High
5	CAN2_Low	CAN bus line Low
18	V_Bat2	Reference potential Plus Transceiver
4	GND	Ground Transceiver
20	CAN3_High	CAN bus line High
8	CAN3_Low	CAN bus line Low
21	V_Bat3	Reference potential Plus Transceiver
7	GND	Ground Transceiver
23	CAN4_High	CAN bus line High

Pin	Signal name	Comment
11	CAN4_Low	CAN bus line Low
24	V_Bat4	Reference potential Plus Transceiver
10	GND	Ground Transceiver
3	INPUT1	Input CAN controller1
16	OUTPUT1	Output CAN controller1
6	INPUT2	Input CAN controller2
19	OUTPUT2	Output CAN controller2
9	INPUT3	Input CAN controller3
22	OUTPUT3	Output CAN controller3
12	INPUT4	Input CAN controller4
25	OUTPUT4	Output CAN controller4
13	GND	Ground

Table A.2 Connector assignment PXI/PCI 3052

A.2.2 LED display

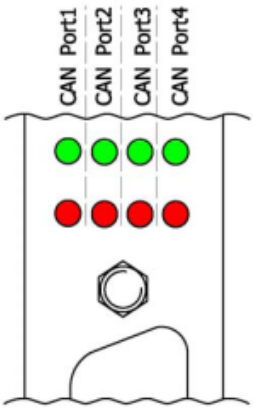

Figure A.3 LED display PXI/PCI 3052

LED red	LED green	Description
twinkle permanently		Controller does not run (Xilinx download?)
twinkle alternately		Bootloader software is running (software reset?)
do not twinkle		Firmware is running
twinkles (for a moment)	does not twinkle	Firmware command will be processed

Table A.3 LED display PXI/PCI 3052

A.3 PXI/PCI 3060

A.3.1 Connector assignment

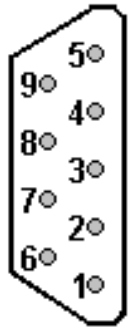

Figure A.4 Connector assignment PXI/PCI 3060

Pin	Signal	Comment	Conductor colouring
1	TRGO_1	Trigger output 1	black
6	TRGO_2	Trigger output 2	green
2	TRGO_3	Trigger output 3	yellow
7	TRGO_4	Trigger output 4	blue
4	TRGI_1	Trigger input 1	orange
5	TRGI_2	Trigger input 2	red
3	GND	Ground	brown
8	UBatt	Reference potential Diagnosing ring interruptions	white
9	RingDiag	Data line Diagnosing ring interruptions	violet
		No function	grey

Table A.4 Connector assignment PXI/PCI 3060

A.3.2 LED display

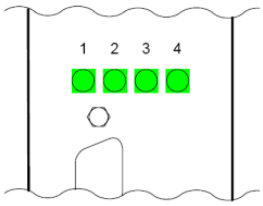

Figure A.5 LED display PXI/PCI 3060

LED 1	LED 2	LED 3	LED 4	Description
twinkle permanently				Controller does not run
twinkle alternately				Bootloader software is running
do not twinkle				Firmware is running
twinkles (for a moment)	does not twinkle			Firmware commands will be processed
		twinkles		MOST bus: Light on / Light off

LED 1	LED 2	LED 3	LED 4	Description
			twinkles	MOST bus: locked / unlocked

Table A.5 LED display PXI/PCI 3060

A.4 PXI/PCI 3072

A.4.1 Connector assignment

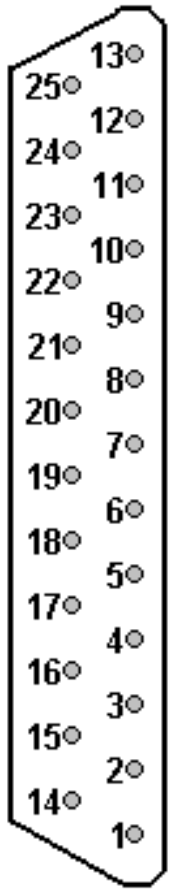

Figure A.6 Connector assignment PXI/PCI 3072

Pin	Signals name	comment
1	LIN1 / K-Line1	
2	LIN2 / K-Line2	
3	LIN3 / K-Line3	
4	Trigger output LIN1	TTL signal
5	Trigger output LIN2	TTL signal
6	Triggeroutput LIN3	TTL signal
7	VBAT 1 Out	
8	VBAT 2 Out	
9	VBAT 3 Out	
10	n.c.	not assigned
11	n.c.	not assigned
12	n.c.	not assigned
13	GND	Ground potential

Pin	Signals name	comment
14	Wake1	
15	Wake2	
16	Wake3	
17	VBAT 1 In	
18	VBAT 2 In	
19	VBAT 3 In	
20	Trigger input LIN3	Signal level: 5V...VBAT
21	Triggerinput LIN3	Signal level: 5V ... VBAT
22	Triggerinput LIN3	Signal level: 5V ... VBAT
23	n.c.	not assigned
24	n.c.	not assigned
25	n.c.	not assigned

Table A.6 Connector assignment PXI/PCI3072

A.5 PXI/PCI 3080

A.5.1 Connector assignment

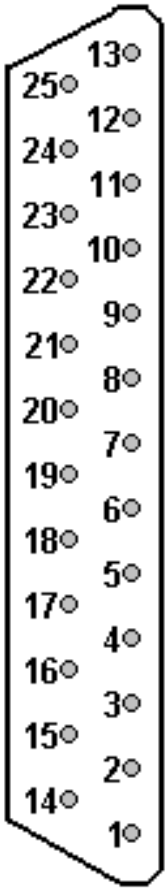

Figure A.7 Connector assignment PXI/PCI 3080

Pin	Signals name	Comment
1	CAN1_High	
14	CAN1_Low	
2	CAN2_High	
15	CAN2_Low	
3	VBAT	Reference potential plus transceiver
16	GND	Ground potential transceiver
4	K-Line1 / LIN1	depends on transceiver
17	L-Line1 / WAKE1	depends on transceiver
5	K-Line2 / LIN2/ J1850 PWM+	depends on transceiver
18	L-Line2 / WAKE2/ J1850 PWM-	depends on transceiver
6	J1850 VPW	
19	n.c.	Do not assign
7	VBAT.	Reference potential plus transceiver
20	GND	Ground potential transceiver
8	Analog Input 1	

Pin	Signals name	Comment
21	Analog Input 2	
9	Digital Input 1	
22	Digital Output 1	
10	Digital Input 2	
23	Digital Output 3	
11	Digital Input 3	
24	Digital Output 3	
12	Digital Input 4	
25	Digital Output 4	
13	Wake line	

Table A.7 Connector assignment PXI/PCI 3080

A.5.2 LED display

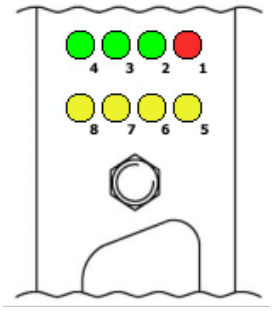

Figure A.8 LED display PXI/PCI 3080

Nr.	Name	Color	Meaning
1	D100	red	/HDRST "Hardware Reset Indication"-output of the microcontroller
2	D700	green	Voltage 5V status
3	D701	green	Voltage 3,3V status
4	D702	green	Voltage 2,5V status
5	D804	yellow	Microcontroller K-Line2 / LIN2 status
6	D803	yellow	Microcontroller K-Line1 / LIN1 status
7	D802	yellow	Microcontroller CAN2 status
8	D801	yellow	Microcontroller CAN1 status

Table A.8 LED display PXI/PCI 3080

A.6 PXI/PCI 3090

A.6.1 Connector assignment

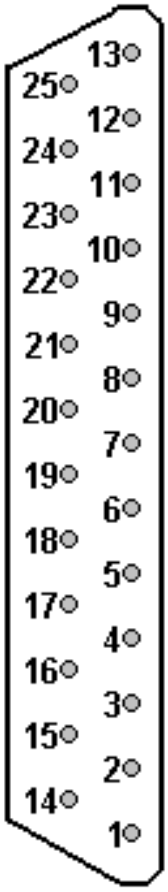

Figure A.9 Connector assignment PXI/PCI 3090

Pin	Signals name	Comment
1	FlexRay1A_BP	FlexRay node 1 (plus bus line **Channel A**)
14	FlexRay1A_BM	FlexRay node 1 (minus bus line **Channel A**)
2	GND	Ground potential transceiver
15	FlexRay1B_BP	FlexRay node 1 (plus bus line **Channel B**)
3	FlexRay1B_BM	FlexRay node 1 (minus bus line **Channel B**)
16	GND	Ground potential transceiver
4	CAN1_H	CAN node 1 (High)
17	CAN1_L	CAN node 1 (Low)
5	GND	Ground potential transceiver
18	VBAT1	Reference potential transceiver (node 1)
6	INPUT1	Digital input FlexRay node 1
19	OUTPUT1	Digital output FlexRay node 1
7	n.c.	Not assigned
20	FlexRay2A_BP	FlexRay node 2 (plus bus line **Channel A**)
8	FlexRay2A_BM	FlexRay node 2 (minus bus line **Channel A**)

Pin	Signals name	Comment
21	GND	Ground potential transceiver
9	FlexRay2B_BP	FlexRay node 2 (plus bus line **Channel B**)
22	FlexRay2B_BM	FlexRay node 2 (minus bus line **Channel B**)
10	GND	Ground potential transceiver
23	CAN2_H	CAN node 2 (High)
11	CAN2_L	CAN node 2 (Low)
24	GND	Ground potential transceiver
12	VBAT2	Reference potential transceiver (node 2)
25	INPUT2	Digital input FlexRay node 2
13	OUTPUT2	Digital output FlexRay node 2

Table A.9 Connector assignment PXI/PCI 3090

A.6.2 LED display

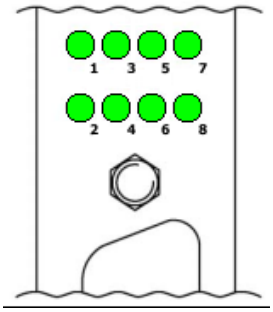

Figure A.10 LED display PXI/PCI 3090

LED 1	LED 2	LED 3	LED 4	LED 5..8	Description
permanently ON				not used	µC1: Controller not running
alternately blinking					µC1: Bootloader software is running, probable error cause: software reset not executed
permanently OFF					µC1: Firmware is running
ON (shortly)	OFF				µC1: Firmware runs executing commands
		permanently ON			µC2: Controller not running
		alternately blinking			µC2: Bootloader software is running, probable error cause: software reset not executed
		permanently OFF			µC1: Firmware is running
		ON (shortly)	OFF		µC2: Firmware runs executing commands

Table A.10 LED display PXI/PCI 3090

A.7 Series61

A.7.1 Connector assignment

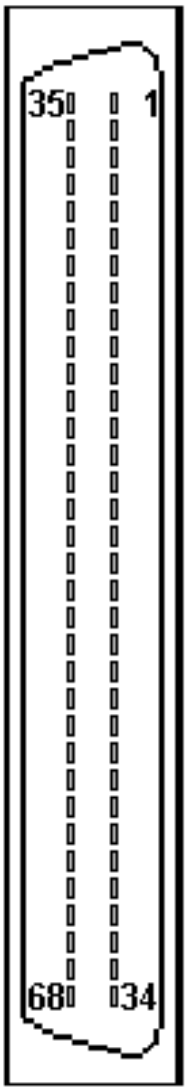

Figure A.11 Connector assignment Series61

The assignment of pins **23 .. 34** and **57 .. 68** varies depending on the extension board. Please see your related hardware manual.

Pin	Signals name	Comment
1	CAN1_H / LIN1 / KLINE1	CAN node 1 (High) / LIN Node 1 / K-Line Node 1
2	CAN1_L	CAN node 1 (Low)
3	GND	
4	CAN2_H / LIN2 / KLINE2	CAN node 2 (High) / LIN Node 2 / K-Line Node 2
5	CAN2_L	CAN node 2 (Low)
6	GND	
7	CAN3_H / LIN3 / KLINE3	CAN node 3 (High) / LIN Node 3 / K-Line Node 3

Pin	Signals name	Comment
8	CAN3_L	CAN node 3 (Low)
9	GND	
10	CAN4_H / LIN4 / KLINE4	CAN node 4 (High) / LIN Node 4 / K-Line Node 4
11	CAN4_L	CAN node 4 (Low)
12	GND	
13	FlexRay1A_BP	FlexRay node 1 (plus bus line **Channel A**)
14	FlexRay1A_BM	FlexRay node 1 (minus bus line **Channel A**)
15	GND	
16	FlexRay2A_BP	FlexRay node 2 (plus bus line **Channel A**)
17	FlexRay2A_BM	FlexRay node 2 (minus bus line **Channel A**)
18	GND	
19	OUT1_digital	
20	OUT2_digital	
21	OUT3_digital	
22	OUT4_digital	
23	OUT5_digital	
24	OUT6_digital	
25	OUT7_digital	
26	OUT8_digital	
27	GND	
28	OUT1_analog	
29	OUT2_analog	
30	OUT3_analog	
31	OUT4_analog	
32	OUT5_analog	
33	OUT6_analog	
34	GND	
35	CAN1_RH	
36	CAN1_RL	
37	VBAT1	
38	CAN2_RH	
39	CAN2_RL	
40	VBAT2	
41	CAN3_RH	
42	CAN3_RL	
43	VBAT3	
44	CAN4_RH	
45	CAN4_RL	
46	VBAT4	
47	FlexRay1B_BP	FlexRay node 1 (plus bus line **Channel B**)

Pin	Signals name	Comment
48	FlexRay1B_BM	FlexRay node 1 (minus bus line **Channel B**)
49	GND	
50	FlexRay2B_BP	FlexRay node 2 (plus bus line **Channel B**)
51	FlexRay2B_BM	FlexRay node 2 (minus bus line **Channel B**)
52	GND	
53	IN1_digital	
54	IN2_digital	
55	IN3_digital	
56	IN4_digital	
57	IN5_digital	
58	IN6_digital	
59	IN7_digital	
60	IN8_digital	
61	VBAT_IO	
62	IN1_analog	
63	IN2_analog	
64	IN3_analog	
65	IN4_analog	
66	IN5_analog	
67	IN6_analog	
68	GND	

Table A.11 Connector assignment Series61xx

A.8 smartCAR

A.8.1 Connector assignment RJ45

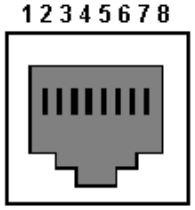

Figure A.12 Connector assignment smartCAR RJ45

Pin	CAN	LIN / K-Line
1	UBatt	UBatt
2	n.c.	n.c.
3	CAN-High	n.c.
4	CAN-Low	n.c.
5	n.c.	LIN / K-Line
6	n.c.	n.c.
7	n.c.	n.c.
8	GND	GND

Table A.12 Connector assignment smartCAR RJ45

A.8.2 Connector assignment DSub-9

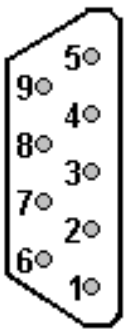

Figure A.13 Connector assignment smartCAR DSub-9

Pin	CAN	LIN / K-Line
1	n.c.	n.c.
2	CAN-Low	n.c.
3	GND	GND
4	n.c.	n.c.
5	n.c.	n.c.
6	n.c.	n.c.
7	CAN-High	LIN / K-Line
8	n.c.	n.c.

Pin	CAN	LIN / K-Line
9	UBatt	UBatt

Table A.13 Connector assignment smartCAR DSub-9

A.9 USB 3052 / basicCAN

A.9.1 Connector assignment

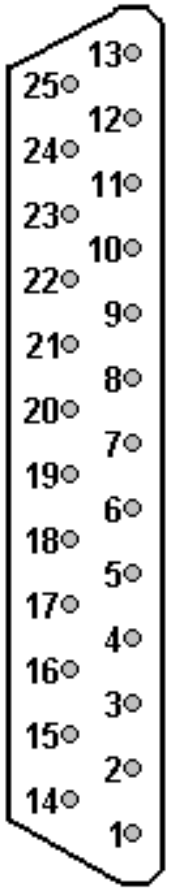

Figure A.14 Connector assignment USB 3052 / basicCAN

Pin	Signals name	Comment
14	CAN1_High	CAN bus line High
2	CAN1_Low	CAN bus line Low
15	V_Bat1	Reference potential Plus Transceiver
1	GND	Ground Transceiver
17	CAN2_High	CAN bus line High
5	CAN2_Low	CAN bus line Low
18	V_Bat2	Reference potential Plus Transceiver
4	GND	Ground Transceiver
20	CAN3_High	CAN bus line High
8	CAN3_Low	CAN bus line Low
21	V_Bat3	Reference potential Plus Transceiver
7	GND	Ground Transceiver
23	CAN4_High	CAN bus line High
11	CAN4_Low	CAN bus line Low
24	V_Bat4	Reference potential Plus Transceiver

Pin	Signals name	Comment
10	GND	Ground Transceiver
3	INPUT1	Input CAN controller1
16	OUTPUT1	Output CAN controller1
6	INPUT2	Input CAN controller2
19	OUTPUT2	Output CAN controller2
9	INPUT3	Input CAN controller3
22	OUTPUT3	Output CAN controller3
12	INPUT4	Input CAN controller4
25	OUTPUT4	Output CAN controller4
13	GND	Ground

Table A.14 Connector assignment USB 3052 / basicCAN

A.9.2 LED display

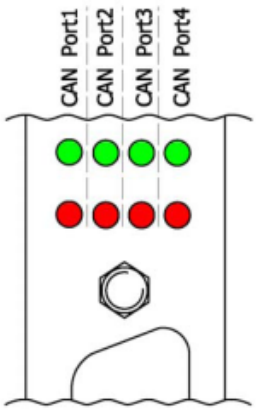

Figure A.15 LED display USB 3052 / basicCAN

LED red	LED green	Description
twinkle permanently		Controller does not run (Xilinx download?)
twinkle alternately		Bootloader software is running (software reset?)
do not twinkle		Firmware is running
twinkles (for a moment)	does not twinkle	Firmware command will be processed

Table A.15 LED display USB 3052 / basicCAN

A.10 USB 3060 / basicMOST

A.10.1 Connector assignment

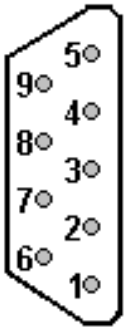

Figure A.16 Connector assignment USB 3060 / basicMOST

Pin	Signal	Comment	Conductor colouring
1	TRGO_1	Trigger output 1	black
6	TRGO_2	Trigger output 2	green
2	TRGO_3	Trigger output 3	yellow
7	TRGO_4	Trigger output 4	blue
4	TRGI_1	Trigger input 1	orange
5	TRGI_2	Trigger input 2	red
3	GND	Ground	brown
8	UBatt	Reference potential Diagnosing ring interruptions	white
9	RingDiag	Data line Diagnosing ring interruptions	violet
		No function	grey

Table A.16 Pin content USB 3060 / basicMOST

A.10.2 LED display

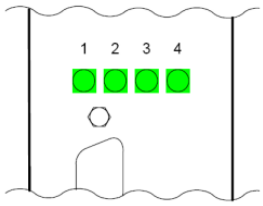

Figure A.17 LED display USB 3060 / basicMOST

LED 1	LED 2	LED 3	LED 4	Description
twinkle permanently				Controller does not run
twinkle alternately				Bootloader software is running
do not twinkle				Firmware is running
twinkles (for a moment)	does not twinkle			Firmware commands will be processed
		twinkles		MOST bus: Light on / Light off

LED 1	LED 2	LED 3	LED 4	Description
			twinkles	MOST bus: locked / unlocked

Table A.17 LED display USB 3060 / basicMOST

A.11 USB3072 / basicLIN

A.11.1 Connector assignment

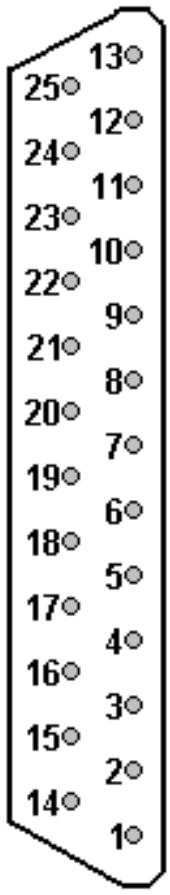

Figure A.18 Connector assignment USB3072 / basicLIN

Pin	Signals name	Comment
1	LIN1 / K-Line1	LIN-(K-Line) Bus line 1
2	LIN2 / K-Line2	LIN-(K-Line) Bus line 2
3	LIN3 / K-Line3	LIN-(K-Line) Bus line 3
4	Trig out1	Trigger output 1
5	Trig out 2	Trigger output 2
6	Trig out 3	Trigger output 3
7	VBAT out 1	Measuring output for supply voltage of transceiver 1 OR supply voltage input for transceier 1 without reverse polarity diode
8	VBAT out 2	Measuring output for supply voltage of transceiver 2 OR supply voltage input for transceier 2 without reverse polarity diode
9	VBAT out 3	Measuring output for supply voltage of transceiver 3 OR supply voltage input for transceier 3 without reverse polarity diode
10	n.c.	not assigned

Pin	Signals name	Comment
11	n.c.	not assigned
12	n.c.	not assigned
13	GND	Ground potential
14	Wake1	Transceiver Wake input 1
15	Wake2	Transceiver Wake input 2
16	Wake3	Transceiver Wake input 3
17	VBAT in1	Supply voltage for transceiver 1 via reverse polarity diode
18	VBAT in2	Supply voltage for transceiver 2 via reverse polarity diode
19	VBAT in3	Supply voltage for transceiver 3 via reverse polarity diode
20	Trig in1	Trigger input 1
21	Trig in2	Trigger input 2
22	Trig in3	Trigger input 3
23	n.c.	not assigned
24	n.c.	not assigned
25	n.c.	not assigned

Table A.18 Connector assignment USB3072 / basicLIN

A.11.2 LED display

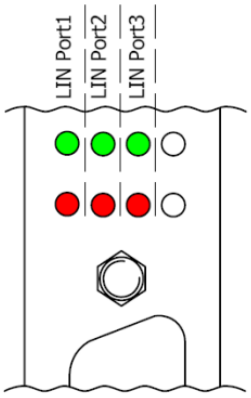

Figure A.19 LED display

LED green	LED red	Bemerkung
Permanently ON		Controller is not running (Xilinx download?)
Alternately blinking		Bootloader software is running
OFF		Firmware is running
ON (shortly)	OFF	Firmware runs executing firmware commands

Table A.19 LED display USB3072 / basicLIN

A.12 USB3080 / basicCAR

A.12.1 Connector assignment

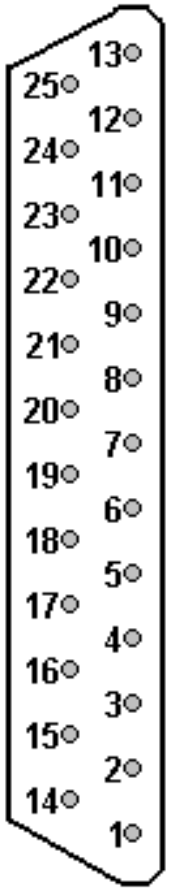

Figure A.20 Connector assignment USB3080 / basicCAR

Pin	Signals name	Comment
1	CAN1_High	CAN bus high
14	CAN1_Low	CAN bus low
2	CAN2_High	CAN bus high
15	CAN2_Low	CAN bus low
3	VBAT	Power supply input for transceiver
16	GND	Ground potential communicaction interfaces
4	K-Line1 / LIN1	depending on transceiver
17	L-Line1 / WAKE1	depending on transceiver
5	K-Line2 / LIN2/ J1850 PWM+	depending on transceiver
18	L-Line2 / WAKE2/ J1850 PWM-	depending on transceiver
6	J1850 VPW	
19	n.c.	Do not assign
7	VBAT.	Power supply input for transceiver
20	GND	Ground potential communication interfaces
8	Analog Input 1	

Pin	Signals name	Comment
21	Analog Input 2	
9	Digital Input 1	
22	Digital Output 1	
10	Digital Input 2	
23	Digital Output 3	
11	Digital Input 3	
24	Digital Output 3	
12	Digital Input 4	
25	Digital Output 4	
13	Wake line	

Table A.20 Connector assignment USB3080 / basicCAR

A.12.2 LED display

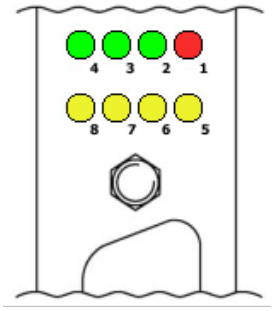

Figure A.21 LED display USB3080 / basicCAR

Nr.	Name	Color	Meaning
1	D100	red	/HDRST "Hardware Reset Indication" output of the microcontroller
2	D700	green	Voltage 5V status (internal)
3	D701	green	Voltage 3,3V status (internal)
4	D702	green	Voltage 5V status (internal)
5	D804	yellow	K-Line2 / LIN2 status
6	D803	yellow	K-Line1 / LIN1 status
7	D802	yellow	CAN2 status
8	D801	yellow	CAN1 status

Table A.21 LED display USB3080 / basicCAR

A.13 USB 4120 / basicCON 4120

A.13.1 Connector assignment - LVDS interface

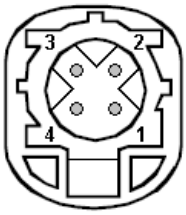

Figure A.22 Connector assignment LVDS USB 4120 / basicCON 4120

Pin	Signal
1	GND
2	LVDS+
3	NC
4	LVDS-
Shield	GND

Table A.22 Pin content LVDS USB 4120 / basicCON 4120

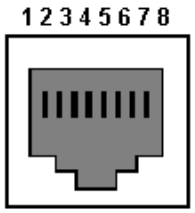

Figure A.23 Connector assignment LVDS USB 4120 / basicCON 4120

Pin	Signal
1	LVDS In0-
2	LVDS In0+
3	LVDS In1-
4	LVDS In2+
5	LVDS In2-
6	LVDS In1+
7	Clk In-
8	Clk In+
Shield	GND

Table A.23 Pin content LVDS USB 4120 / basicCON 4120

A.13.2 Connector assignment - Digital I/O interface

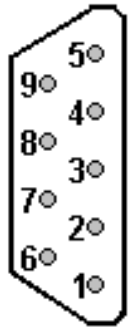

Figure A.24 Connector assignment DIO USB 4120 / basicCON 4120

Pin	Signal	Comment
1	FG_Ready_Out	Framegrabber output ready for DIO control
2	Comp_Fail_Out	Picture comparator has detected at least one failure
3	Ext_Trigger_In	The external trigger input starts a picture grip followed by the comparison with the reference picture
4	Ext_Res1_In	Reserve 1 input
5	DIO_VCC	DIO power supply (+5..+27V), **Pos DIO** setting of the **DIO power supply switch**
6	FG_Pass_Out	The **Framegrabber Pass** output indicates that triggering has been finished and the picture has been compared error-free
7	FG_Lock_Out	The **Framegrabber Lock** output indicates that the LVDS signal is locked
8	Ext_Res0_In	Reserve 0 input
9	DIO_GND	DIO ground pin

Table A.24 Pin content DIO USB 4120 / basicCON 4120

A.13.3 LED display

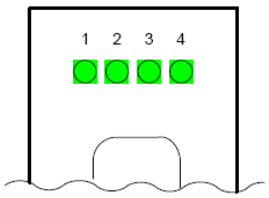

Figure A.25 LED display USB 4120 / basicCON 4120

LED	Description
LED 1 ON	Pixel error during the onboard picture comparison
LED2 ON	Error at picture scan (sync or resolution error)
LED 3 ON	External trigger mode ON
LED 4 ON	LVDS lock active

Table A.25 LED display USB 4120 / basicCON 4120

A.14 USB3085 / basicCAR

A.14.1 Connector assignment

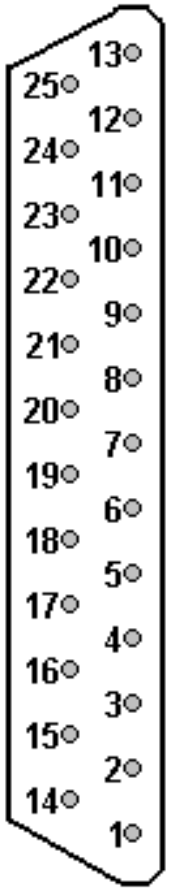

Figure A.26 Connector assignment USB3085 / basicCAR

Pin	Signals name	Comment
1	CAN1_High	CAN bus high
14	CAN1_Low	CAN bus low
2	CAN2_High	CAN bus high
15	CAN2_Low	CAN bus low
3	VBAT	Power supply input for transceiver
16	GND	Ground potential communicaction interfaces
4	K-Line1 / LIN1	depending on transceiver
17	L-Line1 / WAKE1	depending on transceiver
5	K-Line2 / LIN2/ J1850 PWM+	depending on transceiver
18	L-Line2 / WAKE2/ J1850 PWM-	depending on transceiver
6	J1850 VPW	
19	n.c.	Do not assign
7	VBAT.	Power supply input for transceiver
20	GND	Ground potential communication interfaces
8	Analog Input 1	

Pin	Signals name	Comment
21	Analog Input 2	
9	Digital Input 1	
22	Digital Output 1	
10	Digital Input 2	
23	Digital Output 3	
11	Digital Input 3	
24	Digital Output 3	
12	Digital Input 4	
25	Digital Output 4	
13	Wake line	

Table A.26 Connector assignment USB3085 / basicCAR

A.14.2 LED display

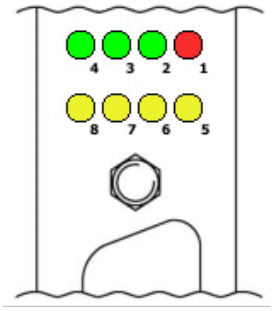

Figure A.27 LED display USB3085 / basicCAR

Nr.	Name	Color	Meaning
1	D100	red	/HDRST "Hardware Reset Indication" output of the microcontroller
2	D700	green	Voltage 5V status (internal)
3	D701	green	Voltage 3,3V status (internal)
4	D702	green	Voltage 5V status (internal)
5	D804	yellow	K-Line2 / LIN2 status
6	D803	yellow	K-Line1 / LIN1 status
7	D802	yellow	CAN2 status
8	D801	yellow	CAN1 status

Table A.27 LED display USB3085 / basicCAR