INpact Slave

Getting Started

SOFTWARE DESIGN GUIDE

4.02.0320.20000 1.5 en-US ENGLISH

Important User Information

Disclaimer

Table of Contents

Page

1 User Guide 3

1.1 Related Documents 3

1.2 Document History 3

1.3 Trademark Information 3

1.4 Conventions 4

1.5 Glossary 5

2 Introduction 6

2.1 System Requirements 6

2.2 Overview 7

3 Installation 8

3.1 Windows 8

3.2 Linux 8

4 Setup 10

4.1 System Setup 10

4.2 INpact Setup 10

4.2.1 Message and Process Data Settings 10

4.2.2 ADI Settings 10

4.2.3 Debug Event Print Settings 11

4.2.4 Startup Time 11

5 Running the Example Application 12

5.1 Selecting ADIs and Process Data Mapping 12

5.2 Implementation of Main Function 13

5.2.1 Initializing the Hardware 13

5.2.2 Main Loop 14

5.2.3 Releasing the Hardware 16

5.3 Compile and Run 16

5.3.1 Windows 16

5.3.2 Linux 17

6 Adapting and Customizing the Example Application 18
6.1 Network Identification 18
6.1.1 Host Application — Networks 18
6.1.2 Host Application — Other 19
6.1.3 Host Application Objects — Advanced 19
6.2 ADIs and Process Data Mapping 20
6.3 Process Data Callbacks 24
6.4 Event Handling 25
6.4.1 Windows 25
6.4.2 Linux 27
6.5 Message Handling 28
6.5.1 Additional Functions in Windows 28
6.5.2 Example 1: Sending a Command and Receiving a Response 29
6.5.3 Example 2: Receiving a Command and Sending a Response 30

A Software Overview 33
A.1 Files and Folders 33
A.2 Root Files 33
A.3 Driver Interface 33
A.4 Internal Driver Files (Read Only) 33
A.5 System Adaptation Files 33

B API Documentation 34
B.1 API Functions 34
B.2 API Related Functions 35
B.3 API Callbacks 35
B.4 Support Functions 35

C Host Application State Machine 36

1 User Guide

1.1 Related Documents

Document	Author
Anybus CompactCom 40 Software Design Guide (see www.anybus.com)	HMS
Anybus CompactCom M40 Network Guides	HMS
INpact PCIe User Manual	HMS

1.2 Document History

Version	Date	Description
1.0	March 2016	First release
1.1	February 2017	Adjusted application folder structure
1.2	July 2017	Added Linux information, updated Windows information
1.3	March 2018	Minor corrections
1.4	September 2018	Minor corrections
1.5	April 2019	Layout changes

1.3 Trademark Information

Ixxat® is a registered trademark of HMS Industrial Networks. All other trademarks mentioned in this document are the property of their respective holders.

1.4 Conventions

Instructions and results are structured as follows:

► instruction 1

► instruction 2

→ result 1

→ result 2

Lists are structured as follows:

- item 1
- item 2

Bold typeface indicates interactive parts such as connectors and switches on the hardware, or menus and buttons in a graphical user interface.

```
This font is used to indicate program code and other
kinds of data input/output such as configuration scripts.
```

This is a cross-reference within this document: *Conventions, p. 4*

This is an external link (URL): www.hms-networks.com

This is additional information which may facilitate installation and/or operation.

! This instruction must be followed to avoid a risk of reduced functionality and/or damage to the equipment, or to avoid a network security risk.

1.5 Glossary

ADI	Application Data Instance
API	Application Programming Interface
IDE	Integrated Development Environment
ABCC	Anybus CompactCom

2 Introduction

When starting an implementation of the INpact Slave, an host application example code is available to speed up the development process. The host application example code includes an Anybus CompactCom driver (ABCC), which acts as glue between the NP40 chip and the host application. The driver has an API (Application Programming Interface), which defines a common interface to the driver. Also included in the example code is an example application which makes use of the API to form a simple application that can be used as a base for the final product.

This guide describes a step-by-step implementation of the driver and example application. The programmer is requested to have basic knowledge in the Anybus CompactCom object model and the communication protocol before starting the implementation.

See Related Documents, p. 3 for suggested reading.

The guide describes the default settings and functions, shows how to run a simple application and how to customize an example code to the target product. The application can then be extended further if needed.

The driver is available for following operating systems:

- Windows 7/10
- Linux 32/64 bit (Intel architecture)
- INtime 6.1
- QNX x86

2.1 System Requirements

Windows

- Visual Studio 2010 or newer

IDL Linux Driver

- PC with Linux Kernel 2.6.X or higher
- HMS IDL Linux Driver
- Gcc 4.2 or newer and a working kernel build environment
- libc6 2.3.4 or newer
- libstdc++6 3.4 or newer
- libgcc1 3.0 or newer
- Eclipse CDT
- GNU Make

Driver is tested using Debian/Ubuntu and open SUSE Linux.

2.2 Overview

The driver code is specifically adapted to the INpact Slave.

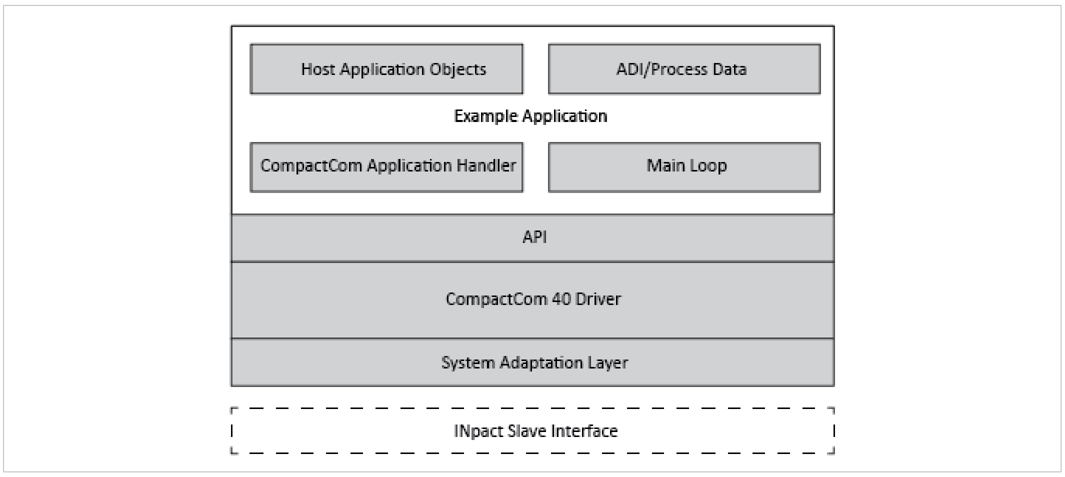


Fig. 1 Different parts of the host application example code

The libraries and headers that are used in the example code are included in the driver.

The host application example code is divided into different folders. With Windows the folders are automatically installed in the directory *C:\Users\Public\Documents\HMS\Ixxat INpact \Samples\IDL\src*. With Linux the files must be extracted from the IDL driver package (`src/IdlApp` and `src/Idl`).

/abcc_abp (part of the application)

- Contains all Anybus object and communication protocol definitions.
- Files are updated when new releases are available.
- Do not change the included files.

/abcc_adapt (part of the driver)

- Contains adaptation and configuration files.
- Files are used to adapt driver and example code to the system environment.
- Do not change the included files.

/abcc_drv (part of the driver)

- Contains source and header files for the driver.
- Files are updated when new releases are available.
- Do not change the included files.

/abcc_obj

- Contains all host application object implementations.
- Files can be modified if needed, for optimization and/or additional features.

/example_app

Example application including:

- Main state machine to handle initialization, restart, normal and error states.
- State machine patterns to show how to send messages.
- Implementation of callbacks required by the driver.
- Definition of ADIs, Application Data Instances, and default process data mapping setup.
- Files have to be adapted to the application by the programmer. Files can be modified if needed, for optimization and/or additional features.

3 Installation

3.1 Windows

► Download the VCI V4 driver from www.ixxat.com.

► To test if interface and driver are ready to use, open the file *IDLc_TestApp.exe* in directory *C:\Users\Public\Documents\HMS\Ixxat INpact\Samples\IDL\bin\x32* resp. *.\bin\x64*.

3.2 Linux

After installing the interfaces the IDL kernel module must be installed and compiled to be able to use the IDL.

► Download the IDL driver package from www.ixxat.com and unzip.

Compiling the Kernel Module

► Install the source code of the operating kernel in use in the system.

► Make sure, that the kernel configuration `/usr/src/linux/.config` file exists or that the `linux-header` package matches the kernel version.

The following steps show how to use module-assistant (m-a) to get a working kernel build environment with Debian/Ubuntu.

► To install the module assistant run `sudo apt-get install module-assistant`.

► With `sudo module-assistant prepare` the module-assistant installs the necessary packages.

► To build the kernel module, run `make` in the kernel module source directory `src/KernelModule`.

► To install the kernel module, run `sudo make install`.

→ Module is copied to the directory `/lib/modules/$(KERNEL_VERSION)/kernel/drivers/misc/`.

→ Accompanied udev rules file is copied to `/etc/udev/rules.d`.

► It is possible to edit the udev rules file to give the device nodes located in the `/dev` directory different permissions.

→ Makefile calls `depmod –a` to export the modalias file and `modprobe` to load the module.

→ Makefile copies the libraries to `/usr/lib`, sets up all necessary links and runs `idconfig`.

Depending on the distribution it may be necessary to adapt `TARGET_PATH` in the makefile. For instance on an 64-bit openSUSE system the default library path is `/usr/lib64`.

► Use command `cat/proc/idl` to check which interfaces are found and which I/O addresses and interrupts are occupied.

Embedded Systems without Kernel Headers

For example, when using a Raspberry Pi without a standard kernel release, there are no kernel or header files available. The kernel has to be recompiled and installed on the device manually to allow the IDL to function. Observe instructions on https://www.raspberrypi.org/documentation/linux/kernel/building.md.

Using IDL Application

► Extract the files from the IDL driver package to the Linux PC.

► Navigate to `IdlLinux_[ARCH]/bin/release` (`[ARCH]` can be `i386` or `amd64`) and enter `./LinuxIdlApp` in the console to start the application.

→ The application opens the first supported board and brings it in `WAIT_PROCESS` state.

→ A bus master can now perform further actions.

→ By default the application `APPL_ADI_SETUP_SPEED` is activated.

→ If the application recognizes an INpact interface it gives an output similar to the following:

Network type:	EtherCAT
Firmware version:	1.10 build 1
Network type	87
Serial number:	0:E:B1:0

4 Setup

4.1 System Setup

General setting for the system environment are configured in the driver and stored in:

- Windows: `abcc_adapt/abcc_td.h`
- Linux: `inc/abcc_td.h`

The following set-ups are supported:

- little-endian system
- 8-bit char system

4.2 INpact Setup

Defines and functions are stored in:

- Windows: `abcc_adapt/abcc_drv_cfg.h`
- Linux: `inc/abcc_drv_cfg.h`

4.2.1 Message and Process Data Settings

Values set in driver		Description
#define ABCC_CFG_MAX_NUM_APPL_ CMDS	(32)	Number of message commands that can be sent without receiving a response. Minimum two messages are required.
#define ABCC_CFG_MAX_NUM_ABCC_ CMDS	(32)	Number of message commands that can be received without sending a response. Minimum two messages are required.
#define ABCC_CFG_MAX_MSG_SIZE	(1524)	Size of largest message in bytes that will be used.
#define ABCC_CFG_MAX_PROCESS_DATA_ SIZE	(4096)	Maximum size of process data in bytes that will be used in either direction. Note: Actual usable size is dependent on protocol in use (see *Anybus CompactCom 40 Software Design Guide*).
#define ABCC_CFG_REMAP_SUPPORT_ ENABLED	(TRUE)	Enable or disable the driver and Application Data object support for the remap command.

4.2.2 ADI Settings

Values set in driver		Description
#define ABCC_CFG_STRUCT_DATA_TYPE	(TRUE)	Affects `AD_AdiEntryType` in `abcc_drv/inc/abcc_ad_if.h`, that is used for defining user ADIs. Required memory usage increases.
#define ABCC_CFG_ADI_GET_SET_ CALLBACK	(TRUE)	Affects `AD_AdiEntryType` in `abcc_drv/inc/abcc_ad_if.h`, that is used for defining user ADIs Triggers callback notifications each time an ADI is read or written. If an ADI is read by the network the callback is invoked before the action. If an ADI is written by the network the callback is invoked after the action
#define ABCC_CFG_64BIT_ADI_SUPPORT	(TRUE)	Support for 64-bit data types in application is disabled.

4.2.3 Debug Event Print Settings

For development purposes, a number of debug functions are available for the developer. The following defines affect debug printouts from the driver. The defines are described in `\abcc_adapt` (Linux: `inc/abcc_drv_cfg.h`).

- `#define ABCC_CFG_ERR_REPORTING_ENABLED ( TRUE )`

  Enabling the `ABCC_CFG_ERR_REPORTING_ENABLED` define activates the error reporting callback function `ABCC_CbfDriverError()`. The function is described in `abcc_drv/inc/abcc.h`.

- `#define ABCC_CFG_DEBUG_EVENT_ENABLED ( FALSE )`

  Enabling the `ABCC_CFG_DEBUG_EVENT_ENABLED` define activates driver support for the print out of debug events within the driver. `ABCC_PORT_DebugPrint()` is used to print debug information. The function is described in `abcc_adapt/abcc_sw_port.h`.

4.2.4 Startup Time

The startup time is given to `ABCC_StartDriver()`. If nothing is defined (0) 1500 ms are used as default.

5 Running the Example Application

An API layer that defines a common interface for all network applications to the driver is available. The example application is provided to give an example of how a standard application implements the driver using the API. For this step, the default settings are used (HMS identity).

The API is stored in:

- Windows: `abcc_drv/inc/abcc.h`
- Linux: `inc/abcc.h`

5.1 Selecting ADIs and Process Data Mapping

Process data is an integral part of the application. Process data is added to the application by creating ADIs (Application Data Instances) and mapping them to the desired process data areas (read or write).

► In Windows open *IDLc_TestApp_VS2010.vcxproj* in Visual Studio and make sure, that the mapping `APPL_ADI_SETUP_SPEED` in `appl_adi_config.h` is enabled.

► For Linux by default `APPL_ADI_SETUP_SPEED` is activated in `/src/IdlApp/example_app/appl_adi_config.h`.

► For description of the mapping see `appl_speed.c` in *ADIs and Process Data Mapping, p. 20*)

Changing ADI Mappings in Linux

ADI mappings can be found in the `IdlLinux_[ARCH]/src/IdlApp/example_app/app_adimap_*.c` files. Each file is an independent example. Within the `app_adimap_*.c` files is an array which holds all available ADIs.

► To activate an example define the `APPL_ACTIVE_ADI_SETUP` define in `IdlLinux_[ARCH]/src/IdlApp/example_app/appl_adi_config.h`.

5.2 Implementation of Main Function

The main function is where the execution of the application starts. In the generic example project, it is located in *main.c*. The Main function is divided into the three sections hardware initialization, main loop and hardware release.

5.2.1 Initializing the Hardware

Linux

With Linux it is possible to use several INpact interfaces with one application and one Linux PC. The interfaces are accessed via the functions `ABCC_OpenController()` and `ABCC_CloseController()`.

ABCC_OpenController()

This function opens the connection to the hardware. If the hardware is used by another application the call fails.

- `dwHwIndex` opens the hardware with the given index.
- `pstcHwPara` are hardware specific parameters.
- `hCtrl` is the handle to the hardware.
- `TRUE` if hardware is successfully opened.

ABCC_CloseController()

This function closes the connection to the hardware and releases the handle. If successfully closed, the handle turns to `IDL_INVALID_HANDLE`.

- `hCtrl` is the handle to be released.

ABCC_HwInit()

This function initiates the hardware that is required to communicate with the interface.

- Must be called once during the power-up initialization.
- Driver can be restarted without calling this function again.

Windows

The starter application is not prepared to access two INpact devices parallel, but the INpact driver is capable of this functionality. If required, contact HMS for further information and assistance.

ABCC_HwInit()

This function initiates the hardware that is required to communicate with the interface.

- Must be called once during the power-up initialization.
- Driver can be restarted without calling this function again.
- To use a dialog to select the hardware, set `fShowDialog` to `TRUE`.
- `pszHardwareSerial` defines the hardware serial number to select, if `fSelectDlg` is `FALSE`.
- If the initialization is successful, the function returns `ABCC_EC_NO_ERROR`.

5.2.2 Main Loop

The main loop is where the execution of the application starts. In the generic example project, it is located in `main.c` and almost identical for Windows and Linux.

APPL_HandleAbcc()

This function runs the state machine and takes care of reset, run and shutdown of the driver.

- Must be called periodically from the main loop.
- Status from the driver is returned every time this function is called.

Status	Definition
APPL_MODULE_NO_ERROR	Interface is OK. This is the response if everything is running.
APPL_MODULE_NOT_DETECTED	No interface is detected. Informing of user must be implemented.
APPL_MODULE_NOT_SUPPORTED	Unsupported interface detected. Informing of user must be implemented.
APPL_MODULE_NOT_ANSWERING	Possible reasons: Wrong API selected, defect interface.
APPL_MODULE_RESET	Reset requested from the interface. Reset is received from network. Application is responsible for restart of the system.
APPL_MODULE_SHUTDOWN	Shutdown requested.
APPL_MODULE_UNEXPECTED_ERROR	Unexpected error occurred. Informing of user must be implemented. If necessary, put outputs in fail-safe state.

For more information see *Host Application State Machine, p. 36*.

RunUI() (Linux: APPL_MustQuit())

Simple user interface to interact with the example application.

ABCC_RunTimerSystem()

- Must be called periodically with a known period (ms since last call):
  - Either by having a known delay in the main loop and calling the function each iteration
  - or by setting up a timer interrupt.
- Responsible for handling all timers for the driver.
- Recommended to call this function on a regular basis from a timer interrupt.
- Without this function the timeout and watchdog functionality do not work.

Windows

```
   while( LOOP_QUIT != bLoopState  )
    {
      bLoopState = LOOP_RUN;

      eAbccHandlerStatus = APPL_HandleAbcc();

      switch( eAbccHandlerStatus )
      {
        case APPL_MODULE_NO_ERROR:
          //
          // if a test is actually running and the INpact
          // configuration was successful quit the application !
          if (( fTest )
 && ( IsResetRequestAllowed (ABP_RESET_POWER_ON) ))
          {
            bLoopState = LOOP_QUIT;
          }
          break;
        case APPL_MODULE_RESET:
          APPL_RestartAbcc();
          break;
        default:
          bLoopState = LOOP_QUIT;
          break;
      }

      if ( bLoopState == LOOP_RUN )
      {
        bLoopState = RunUi();
      }

      switch( bLoopState )
      {
        case LOOP_RESET:
          APPL_RestartAbcc();
          bLoopState = LOOP_RUN;
          break;
        case LOOP_RUN:
        case LOOP_QUIT:
        default:
          break;
      }

      if ( bLoopState == LOOP_RUN )
      {
        Sleep( iSleepTimeMS );
        ABCC_RunTimerSystem( iSleepTimeMS );
      }
    }
```

It is recommended to use a timer interrupt with this function. However, for easier debugging when implementing, skip the timer interrupt in the beginning.

Linux

```
    while( eAbccHandlerStatus == APPL_MODULE_NO_ERROR )
    {
       eAbccHandlerStatus = APPL_HandleAbcc(hCtrl);
       if(APPL_MustQuit())
          APPL_Shutdown(hCtrl);

#if( !USE_TIMER_INTERRUPT )
       ABCC_RunTimerSystem( hCtrl, APPL_TIMER_MS );
       DelayMs( APPL_TIMER_MS );
#endif

       switch( eAbccHandlerStatus )
       {
       case APPL_MODULE_RESET:
          Reset();
          break;
       default:
          break;
       }
    }
```

5.2.3 Releasing the Hardware

Linux

APPL_Shutdown()

- `ABCC_HWReset()` is called to reset the driver.
- Sets state to `APPL_HALT`.

Windows

ABCC_Shutdown()

- Stops the driver and puts it into `SHUTDOWN` state.
- ABCC is reset.

ABCC_HwRelease();

This function releases the hardware.

5.3 Compile and Run

5.3.1 Windows

► Compile the project.

► Make sure that the project compiles without errors.

► Run the project.

► Make sure that the host application can communicate with the interface.

► Make sure that data can be exchanged with the network.

5.3.2 Linux

Compiling the IDL Application

► If using Eclipse CDT IDE, import the existing project from `IdlLinux_[ARCH]/src/IdlApp`.

  or

► If using GNU make, navigate to `IdlLinux_[ARCH]/src/IdlApp/LinuxIdlApp` and run **make all**.

► Compile and start the demo within the IDE in use.

Compiling the IDL Driver Library

► If using Eclipse CDT IDE, import the existing project from `IdlLinux_[ARCH]/src/Idl`.

  or

► If using GNU make, navigate to `IdlLinux_[ARCH]/src/IdlApp/LinuxIdlApp` and run **make all**.

► Compile the driver library within the IDE in use.

IDL API

With the following it is possible to create a user defined project (instead of adapting and customizing the demo as described in *Adapting and Customizing the Example Application, p. 18*).

IDL header files are stored in the folder `IdlLinux_[ARCH]/inc`.

► When creating a user defined project, include the `IdlLinux_[ARCH]/inc` folder and link against `libidlLinux.so` and `libidl115DriverLinux.so` (115 stands for the INpact PCIe).

► Open a device with the function `ABCC_OpenController`, which is stored in the `IdlLinux_[ARCH]/inc/IDL.h` header.

  → If succeeded the function returns a handle on the device.

  → The handle is usually the first parameter of all driver specific functions (i.e. `ABCC_StartDriver( IDL_CTRL_HDL hCtrl, UINT32 lMaxStartupTimeMs)`.

6 Adapting and Customizing the Example Application

6.1 Network Identification

If all network settings are left disabled the product is identified as an HMS product. In this step the network identification attributes are customized to fit the target product.

The identity related attributes for each enabled network object are parameters that must be set by the application. They are all related to how the interface is identified on the network. These settings are stored in `abcc_adapt/abcc_identification.h`.

6.1.1 Host Application — Networks

Network Abbreviations

Network	Abbreviation
DeviceNet	DEV
EtherCAT	ECT
Ethernet POWERLINK	EPL
EtherNet/IP	EIP
Profibus DP-V1	DPV1
PROFINET IO	PRT
Modbus TCP	EIT

- ► To enable a supported network set the respective host application object in file `abcc_ adapt/abcc_obj_cfg.h` to TRUE.
- ► In the enabled network define `vendor_id` and `product_code` according to the protocol in use (see the following example).
- ► Do further implementations of the host application in the `abcc_obj` folder where each object has its own c- and h-files.

Example Setting Vendor ID and Product Code

```
/*--------------------------------------------------------------------
** Ethernet Powerlink (0xE9)
**--------------------------------------------------------------------
*/
#if EPL_OBJ_ENABLE
/*
** Attribute 1: Vendor ID (UINT32 - 0x00000000-0xFFFFFFFF
*/
#define EPL_IA_VENDOR_ID_ENABLE                     TRUE
#define EPL_IA_VENDOR_ID_VALUE                      0xFFFFFFFF


/*
** Attribute 2: Product Code type (UINT32 - 0x00000000-0xFFFFFFFF)
*/
#define EPL_IA_PRODUCT_CODE_ENABLE                  TRUE
#define EPL_IA_PRODUCT_CODE_VALUE                   0xFFFFFFFF
```

It is also possible to define a function instead of a constant to generate the value. The serial number is a good example of where a function would be suitable. In the example below, the serial number is set during production in a specific memory area, and here the same number is fetched:

`extern char* GetSerialNumberFromProductionArea(void);`

`#define PRT_IA_IM_SERIAL_NBR_ENABLE TRUE`

`#define PRT_IA_IM_SERIAL_NBR_VALUE GetSerialNumberFromProductionArea()`

6.1.2 Host Application — Other

- In `abcc_adapt/abcc_obj_cfg.h` define all other host application objects that shall be supported by the implementation.

6.1.3 Host Application Objects — Advanced

The files `abcc_adapt/abcc_obj_cfg.h` and `abcc_adapt/abcc_identification.h.` contain all attributes for all supported host objects. All network specific services are disabled by default, and if desired they need to be implemented in the application.

The file `abcc_adapt/abcc_platform_cfg.h` can be used to override defines for objects and attributes in the files `abcc_adapt/abcc_obj_cfg.h` and `abcc_adapt/abcc_identification.h`.

To override a define add desired defines to the file `abcc_adapt/abcc_platform_cfg.h`.

6.2 ADIs and Process Data Mapping

The following ADI mapping examples, which exemplify different types of ADIs are included in the example application.

example_app/appl_speed.c

The master sets a reference value for velocity and the slave updates its corresponding speed value to the reference velocity.

- ADI1 (SINT16, mapped as input)
- ADI2 (SINT16, mapped as output)
- ADI3 (BOOL, mapped as output)
- ADI4 (SINT16, mapped as input)
- ADI5 (UINT8, mapped as output)

example_app/appl_adimap_simple16.c

Loops 32 16-bit words.

- ADI1 (32 element array of UINT16)
- ADI2 (32 element array of UINT16)
- ADIs are mapped in each direction.
- Data is looped since both ADIs refer to the same data place holder.
- No structures or callbacks are used.

example_app/appl_adimap_separate16.c

Example of how get/set callbacks can be used.

- ADI10 (32 element array of UINT16, mapped as output data)
- ADI11 (32 element array of UINT16, mapped as input data)
- ADI12 (UINT16, not mapped to process data)
- ADI10 and ADI11 are mapped on process data in each direction.
- A callback is used when the network reads ADI11. This callback will increment the value of ADI12 by one.
- A callback is used when the network writes ADI10. This callback copies the value of ADI10 to ADI11.

`ABCC_CFG_ADI_GET_SET_CALLBACK` has to be enabled in `abcc_adapt/abcc_drv_cfg.h` (Linux: `inc/abcc_drv_cfg.h`) since callbacks are used. See ADI Settings, p. 10.

example_app/appl_adimap_alltypes.c

Example of how structured data types and bit data types can be used.

ADI	Description
ADI20	UINT32 (mapped as output data)
ADI21	UINT32 (mapped as input data)
ADI22	SINT32 (mapped as output data)
ADI23	SINT32 (mapped as input data)
ADI24	UINT16 (mapped as output data)
ADI25	UINT16 (mapped as input data)
ADI26	SINT16 (mapped as output data)
ADI27	SINT16 (mapped as input data)
ADI28	BITS16 (mapped as output data)
ADI29	BITS16 (mapped as input data)
ADI30	UINT8 (mapped as output data)
ADI31	UINT8 (mapped as input data)
ADI32	SINT8 (mapped as output data)
ADI33	SINT8 (mapped as input data)
ADI34	PAD8 (mapped as output data, reserved space, no data)
ADI35	PAD8 (mapped as input data, reserved space, no data)
ADI36	BIT7 (mapped as output data)
ADI37	BIT7 (mapped as input data)
ADI38	Struct (mapped as output data)
ADI39	Struct (mapped as input data)

`ABCC_CFG_STRUCT_DATA_TYPE` has to be enabled in `abcc_adapt/abcc_drv_cfg.h` (Linux: `inc/abcc_drv_cfg.h`) since structures are used. See ADI Settings, p. 10.

Customizing the ADI Mapping

Only one mapping can be used at a time.

- Create an `AD_AdiEntryType` in an ADI entry list to define the ADIs (i.e. data instances that will be used in the implementation).
- Observe the specifications of all parameters to an ADI in *ADI Entry List, p. 21*.
- Map the ADIs to be used as process data in the list `AD_DefaultMapType`. See *Write and Read Process Data Mapping , p. 23*.

ADI Entry List

ADI entry item	Description	
iInstance	ADI instance number (1-65535). 0 is reserved for Class.	
pabName	Name of ADI (character string, ADI instance attribute #1). If NULL, zero length name is returned.	
bDataType	ABP_BOOL:	Boolean
	ABP_SINT8:	Signed 8 bit integer
	ABP_SINT16:	Signed 16 bit integer
	ABP_SINT32:	Signed 32 bit integer
	ABP_UINT8:	Unsigned 8 bit integer
	ABP_UNIT16:	Unsigned 16 bit integer
	ABP_UINT32:	Unsigned 32 bit integer
	ABP_CHAR:	Character

ADI Entry List (continued)

ADI entry item	Description	
	ABP_ENUM:	Enumeration
	ABP_SINT64:	Signed 64 bit integer
	ABP_UINT64:	Unsigned 64 bit integer
	ABP_FLOAT:	Floating point value (32 bits)
	ABP_OCTET	Undefined 8 bit data
	ABP_BITS8	8 bit bit field
	ABP_BITS16	16 bit bit field
	ABP_BITS32	32 bit bit field
	ABP_BIT1	1 bit bit field
	ABP_BIT2	2 bit bit field
	:	:
	ABP_BIT7	7 bit bit field
	ABP_PAD0	0 pad bit field
	ABP_PAD1	1 pad bit field
	:	:
	ABP_PAD16	16 pad bit field
bNumOfElements	For arrays: number of elements of data type specified in bDataType. For structured data types: number of elements in structure.	
bDesc	Entry descriptor. Bit values according to the following configurations: • ABP_APPD_DESCR_GET_ACCESS: Get service is allowed on value attribute. • ABP_APPD_DESCR_SET_ACCESS: Set service is allowed on value attribute. • ABP_APPD_DESCR_MAPPABLE_WRITE_PD: Remap service is allowed on value attribute. • ABP_APPD_DESCR_MAPPABLE_READ_PD: Remap service is allowed on value attribute. Descriptors can be logically OR:ed together. In the example, ALL_ACCESS is all of the above logically OR:ed together. Note: Ignored for structured data types	
pxValuePtr	Pointer to local value variable. Type is dependent on bDataType. Note: Ignored for structured data types	
pxValuePropPtr	Pointer to local value properties struct, if NULL, no properties are applied (max/min/default). Type is dependent on bDataType. Note: Ignored for structured data types	
psStruct	Pointer to an AD_StructDataType. Set to NULL for non structured data types. Field is enabled by defining ABCC_CFG_STRUCT_DATA_TYPE. (Optional)	
pnGetAdiValue	Pointer to an ABCC_GetAdiValueFuncType called when getting an ADI value. (Optional)	
pnSetAdivalue	Pointer to an ABCC_SetAdiValueFuncType called when setting an ADI value. (Optional)	

See example of usage in:

- Windows: `abcc_drv/inc/abcc_ad_if.h`
- Linux: `inc`

Write and Read Process Data Mapping	
Data Mapping Item	**Description**
iInstance	ADI number of ADI to map (see *ADI Entry List, p. 21*)
eDir	Direction of map. Set to PD_END_MAP to indicate end of default map list.
bNumElem	Number of elements to map. Can only be > 1 for arrays or structures. AD_DEFAULT_MAP_ALL_ELEM indicates that all elements shall be mapped. If instance == AD_MAP_PAD_ADI, bNumElem indicates number of bits to pad with.
bElemStartIndex	Element start index within an array or structure. If ADI is not an array or structure, enter 0.

The mappings are done in the same order as they show up on the network.

The mapping sequence is terminated by `AD_DEFAULT_MAP_END_ENTRY`, which MUST be present at the end of the list. During the setup sequence, the driver asks for this information by invoking `ABCC_CbfAdiMappingReq()`.

Example

```
/*---------------------------------------------------------------
** Map all adi:s in both directions
**-----------------------------------------------------------------
** 1. AD instance | 2. Direction | 3. Num elements | 4. Start index |
**-----------------------------------------------------------------
*/
const AD_DefaultMapType APPL_asAdObjDefaultMap[] =
{
   { 1, PD_WRITE, AD_DEFAULT_MAP_ALL_ELEM, 0 },
   { 2, PD_READ,  AD_DEFAULT_MAP_ALL_ELEM, 0 },
   { AD_DEFAULT_MAP_END_ENTRY }
};
```

The example can be found in `\example_app` (Linux: `src/IdlApp/example_app/appladimap.simple16.c`).

Further examples can be found in `abcc_drv/inc/abcc_ad_if.h` (Linux: `inc/abcc_ad_if.h`).

6.3 Process Data Callbacks

There are two callback functions related to the update of the process data that must be implemented. An example is available in `example_app/appl_abcc_handler.c`.

- `ABCC_CbfUpdateWriteProcessData()`: Updates the current write process data.

  Copy data into buffer before returning from function.
- `ABCC_CbfNewReadPd()`: Called when new process data is received from the network.

  Copy the process data to the application ADIs before returning from the function.

As seen below, in the example code, they both call on a service in the Application Data object to update the information. These functions work, in general, for any process data map, but they are also slow because of all considerations needed for the general case. For better performance, consider writing application specific update functions.

Windows

```
 void ABCC_CbfNewReadPd( void* pxReadPd )
{
    /*
    ** AD_UpdatePdReadData is a general function that updates all
    ** ADI:s according to current map.
    ** If the ADI mapping is fixed there is potential for doing that
    ** in a more optimized way, for example by using memcpy.
    */

     AD_UpdatePdReadData( pxReadPd );
 }
 BOOL ABCC_CbfUpdateWriteProcessData( void* pxWritePd )
 {
    /*
    ** AD_UpdatePdWriteData is a general function that updates all
    ** ADI:s according to current map.
    ** If the ADI mapping is fixed there is potential for doing that
    ** in a more optimized way, for example by using memcpy.
    */
     return( AD_UpdatePdWriteData( pxWritePd ) );
 }
```

Linux

```
BOOL ABCC_CbfUpdateWriteProcessData( IDL_CTRL_HDL hCtrl,
                                              void* pxWritePd )
{
   /*
   ** AD_UpdatePdWriteData is a general function that updates all
   ** ADI:s according to current map.
   ** If the ADI mapping is fixed there is potential for doing that
   ** in a more optimized way, for example by using memcpy.
   */
   return( AD_UpdatePdWriteData( pxWritePd ) );
}

...

void ABCC_CbfNewReadPd( IDL_CTRL_HDL hCtrl,void* pxReadPd )
{
   /*
   ** AD_UpdatePdReadData is a general function that updates all
   ** ADI:s according to current map.
   ** If the ADI mapping is fixed there is potential for doing that
   ** in a more optimized way, for example by using memcpy.
   */
   AD_UpdatePdReadData( pxReadPd );
```

6.4 Event Handling

The following defined configuration enables read message and read process data interrupts. With Linux only the read process data callbacks are executed in interrupt context directly by the driver. The read message event is forwarded to the application by calling the function `ABCC_CbfEvent()`.

`#define ABCC_CFG_INT_ENABLE_MASK_PAR (ABP_INTMASK_RDPDIEN | ABP_INTMASK_RDMSGIEN)`

`#define ABCC_CFG_HANDLE_INT_IN_ISR_MASK (ABP_INTMASK_RDPDIEN)`

The following code shows examples of how the callback event handler can trigger a task to handle an event with Windows and Linux.

6.4.1 Windows

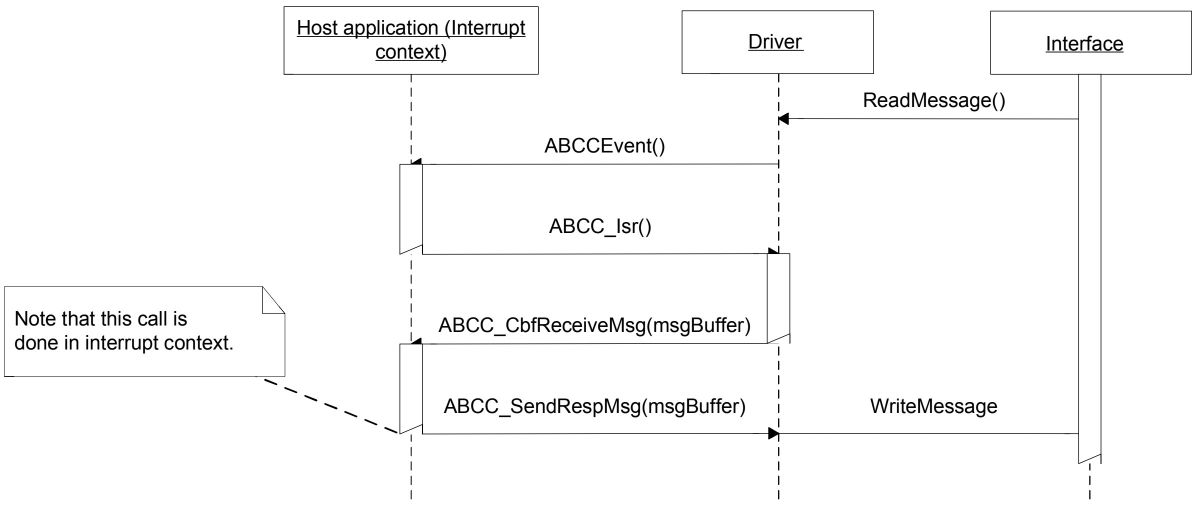
```
void ABCC_CbfEvent( UINT16 iEvents )
{
  if( iEvents & ABCC_EVENT_RDMSGI )
  {
     ABCC_fRdMsgEvent = TRUE;
  }
}
```

The code above illustrates how a task (below) can be triggered by the driver event callback.

This code depicts a task that handles receive message events.

```
volatile BOOL ABCC_fRdMsgEvent = FALSE;
void HandleEvents( void )
{
  ABCC_fRdMsgEvent = FALSE;
  while( 1 )
  {
    if( ABCC_fRdMsgEvent )
    {
      ABCC_fRdMsgEvent = FALSE;
      ABCC_TriggerReceiveMessage();
    }
  }
}
```

Handling Events

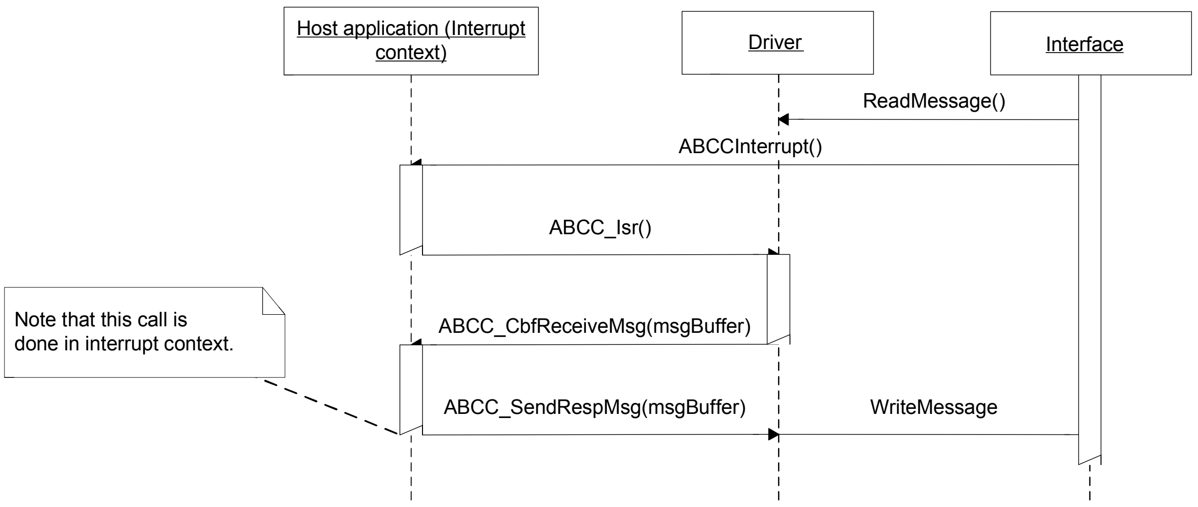

Host application (Interrupt context)
Driver
Interface
ReadMessage()
ABCCEvent()
ABCC_Isr()
Note that this call is done in interrupt context.
ABCC_CbfReceiveMsg(msgBuffer)
ABCC_SendRespMsg(msgBuffer)
WriteMessage

6.4.2 Linux

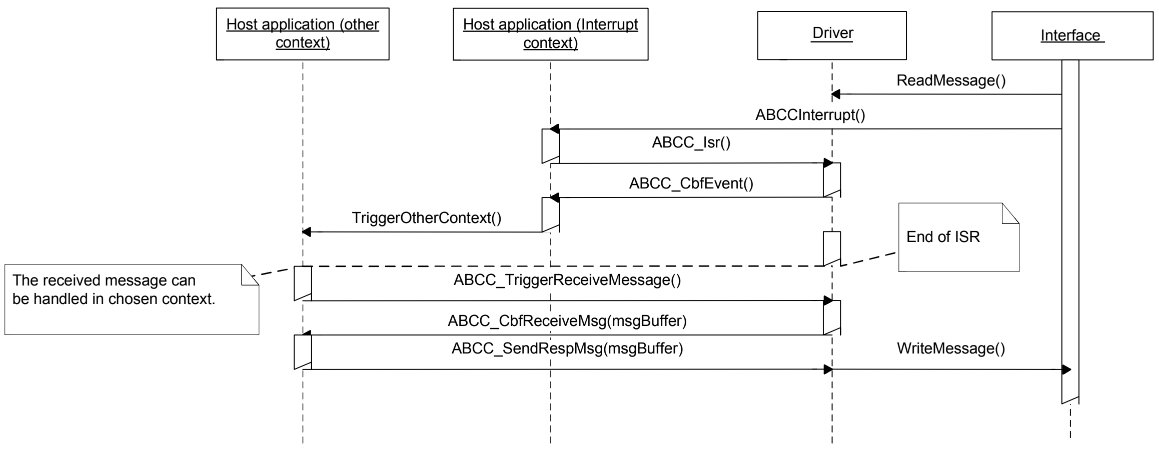
```
void ABCC_CbfEvent( IDL_CTRL_HDL hCtrl, UINT16 iEvents )
{
   if( iEvents & ABCC_ISR_EVENT_RDMSG )
   {
      ABCC_fRdMsgEvent = TRUE;
   }
}


volatile BOOL ABCC_fRdMsgEvent = FALSE;
void Task( IDL_CTRL_HDL hCtrl )
{
   ABCC_fRdMsgEvent = FALSE;
   while( 1 )
   {
      if( ABCC_fRdMsgEvent )
      {
         ABCC_fRdMsgEvent = FALSE;
         ABCC_TriggerReceiveMessage( hCtrl );
      }
   }
}
```

Handling Events in Interrupt Context


Host application (Interrupt context)
Driver
Interface
ReadMessage()
ABCCInterrupt()
ABCC_Isr()
ABCC_CbfReceiveMsg(msgBuffer)
ABCC_SendRespMsg(msgBuffer)
WriteMessage
Note that this call is done in interrupt context.

Handling Events Using ABCC_CbfEvent() Callback Function


Host application (other context)
Host application (Interrupt context)
Driver
Interface
ReadMessage()
ABCCInterrupt()
ABCC_Isr()
ABCC_CbfEvent()
TriggerOtherContext()
End of ISR
The received message can be handled in chosen context.
ABCC_TriggerReceiveMessage()
ABCC_CbfReceiveMsg(msgBuffer)
ABCC_SendRespMsg(msgBuffer)
WriteMessage()

6.5 Message Handling

The message handling interface functions are found and described in `abcc.h` (Windows and Linux).

To send a command message, the user must use the function `ABCC_GetCmdMsgBuffer()` to retrieve a message memory buffer. When receiving a response, the user must handle or copy needed data from the response buffer within the context of the response handler function. The function `ABCC_GetCmdMsgBuffer()` can return a NULL pointer, if no more memory buffers are available. It is the responsibility of the user to resend the message later or treat it as a fatal error.

6.5.1 Additional Functions in Windows

The Windows Starter application implements some additional functions to simplify the message handling. The implementation is located in *\example_app\communication\appl_com.c*.

These help functions are already used within the initialization of the Anybus module according to the description in chapter 11 in the *Anybus CompactCom 40 Software Design Guide*. See file *abcc_setup.c* to examine the use of these functions.

SetAttribute

Function to set the content of an Anybus attribute.

Parameter	Description
bObject	Object(01 == Anybus, 02 == Diagnostic, 03 == network)
wInstance	Instance within object
bAttribute	Attribute within instance
hEvent	Event to signal
wWrite	Size of data to write
pbData	Memory ptr where the data to write is stored
pwWritten	ptr on a WORD variable to store the number of written data
dwTimeout	Time to wait for a response

Possible response	Description
`TRUE`	Successful
`FALSE`	Error occurred

GetAttribute

Function to get the content of an Anybus attribute.

Parameter	Description
bObject	Object(01 == Anybus, 02 == Diagnostic, 03 == network)
wInstance	Instance within object
bAttribute	Attribute within instance
hEvent	Event to signal
wSize	Size of data to read
pbData	Memory ptr to store the read data
pwReaden	ptr on a WORD variable to store the number of read data
dwTimeout	Time to wait for a response

Possible response	Description
`TRUE`	Successful
`FALSE`	Error occurred

6.5.2 Example 1: Sending a Command and Receiving a Response

When sending the command the driver will connect the source id to the response function, in this case `appl_HandleResp()`.

The function `appl_HandleResp()` is called by the driver when a response with the matching source ID is received.

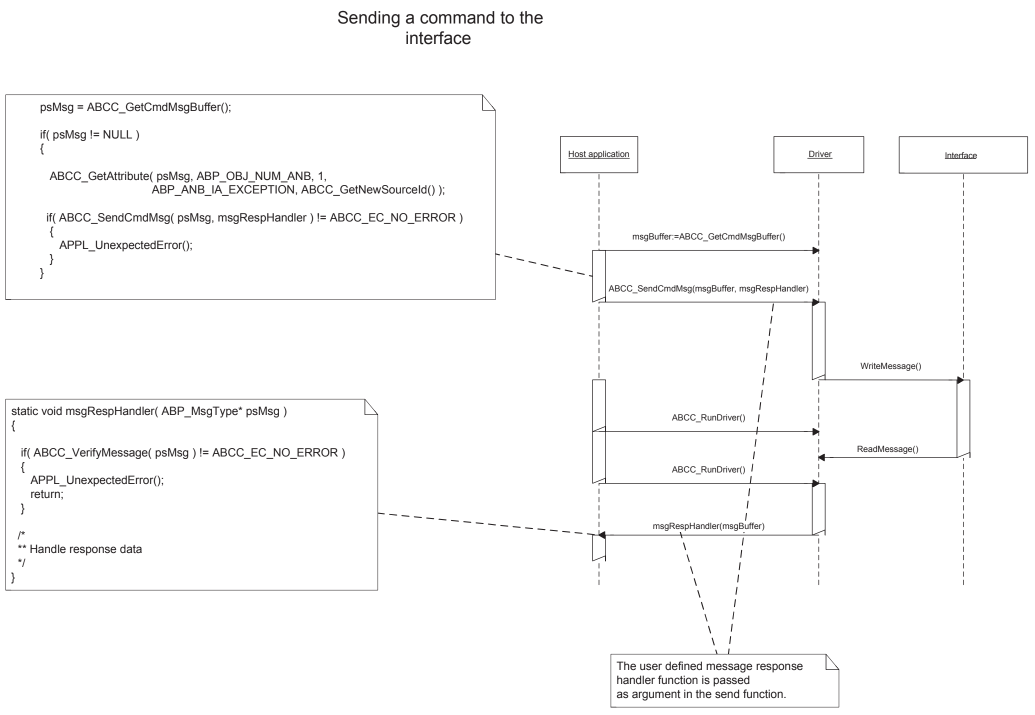
Note that the received message buffer does not need be freed, this is done internally in the driver after return from `appl_HandleResp()`.

Windows

```
void appl_HandleResp( ABP_MsgType* psMsg )
{
  HandleResponse(psMsg);
}
```

Linux

```
void appl_HandleResp( IDL_CTRL_HDL hCtrl, ABP_MsgType* psMsg )
{
   HandleResponse( hCtrl, psMsg );
}
```


Sending a command to the interface
Host application
Driver
Interface
msgBuffer:=ABCC_GetCmdMsgBuffer()
ABCC_SendCmdMsg(msgBuffer, msgRespHandler)
WriteMessage()
ABCC_RunDriver()
ReadMessage()
ABCC_RunDriver()
msgRespHandler(msgBuffer)
The user defined message response handler function is passed as argument in the send function.

```
psMsg = ABCC_GetCmdMsgBuffer();

if( psMsg != NULL )
{

  ABCC_GetAttribute( psMsg, ABP_OBJ_NUM_ANB, 1,
                     ABP_ANB_IA_EXCEPTION, ABCC_GetNewSourceId() );

  if( ABCC_SendCmdMsg( psMsg, msgRespHandler ) != ABCC_EC_NO_ERROR )
  {
    APPL_UnexpectedError();
  }
}
```

```
static void msgRespHandler( ABP_MsgType* psMsg )
{

  if( ABCC_VerifyMessage( psMsg ) != ABCC_EC_NO_ERROR )
  {
    APPL_UnexpectedError();
    return;
  }

 /*
 ** Handle response data
 */
}
```

6.5.3 Example 2: Receiving a Command and Sending a Response

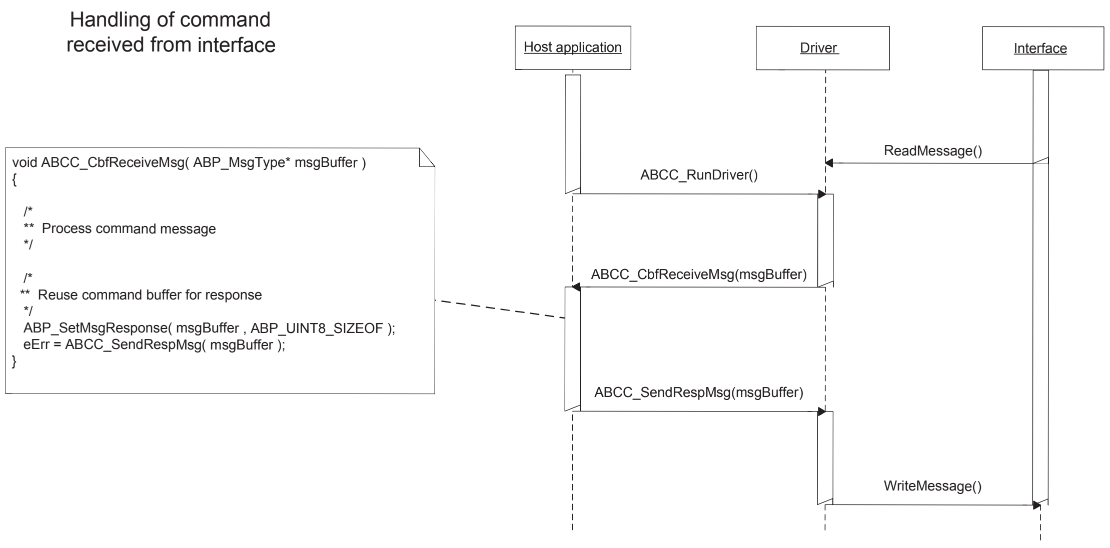
Windows

The received command buffer is reused for the response.

```
void appl_ProcessCmdMsg( ABP_MsgType* psNewMessage )
{
   /* Reuse command buffer for response */
   ABP_SetMsgResponse( psNewMessage, ABP_UINT8_SIZEOF );
    eErr = ABCC_SendRespMsg( psNewMessage );
}
```

Linux

```
void appl_ProcessCmdMsg( IDL_CTRL_HDL hCtrl, ABP_MsgType* psNewMessage )
{
   /* Reuse command buffer for response */
   ABP_SetMsgResponse( psNewMessage, ABP_UINT8_SIZEOF )
   eErr = ABCC_SendRespMsg( hCtrl, psNewMessage );
}
```


Handling of command received from interface
Host application
Driver
Interface
ReadMessage()
ABCC_RunDriver()
ABCC_CbfReceiveMsg(msgBuffer)
ABCC_SendRespMsg(msgBuffer)
WriteMessage()

```
void ABCC_CbfReceiveMsg( ABP_MsgType* msgBuffer )
{

   /*
   **  Process command message
   */

   /*
  **  Reuse command buffer for response
   */
   ABP_SetMsgResponse( msgBuffer , ABP_UINT8_SIZEOF );
   eErr = ABCC_SendRespMsg( msgBuffer );
}
```

The driver uses non-blocking message handling. This means that a state machine must be used to keep track of commands and responses. In `example_app/appl_abcc_handler.c` are two examples of state machines that can be used as templates.

Example 1: When `ABCC_CbfUserInitReq()` is called, the IP address or node address is set before `ABCC_UserInitComplete()` is called.

Example 2: When the interface indicates exception state, the exception codes are read.

A Software Overview

A.1 Files and Folders

Folders	Description
$\abcc_abp	Contains all object and communication protocol definitions. Files are updated when new releases are available.
$\abcc_adapt	Contains all adaptation and configuration. Files are used to adapt driver and example code to the system environment.
$\abcc_drv\inc	.h files published to the application. Contains driver configuration files for the application and for the system dependent part of the driver.
$\abcc_drv\src	Driver implementation
$\abcc_obj	Contains all host object implementations. Files can be modified for optimization and/or additional features.
$\example_app	Example application. Files can be modified for optimization and/or additional features.

A.2 Root Files

Folders	Description
$\main.c	Main file for example application
$\abcc_versions.h	Contains the version defines for example code, driver and abp.

A.3 Driver Interface

File name	Description
\abcc_drv\inc\abcc.h	Public interface for driver
\abcc_drv\inc\abcc_ad_if.h	Type definitions for ADI mapping
\abcc_drv\inc\abcc_cfg.h	Configuration parameters of the driver
\abcc_drv\inc\abcc_sys_adapt.h	Interface for target dependent functions common to all operating modes

A.4 Internal Driver Files (Read Only)

The contents of the files in /abcc_drv/src folder must not be changed.

File name	Description
\abcc_drv\src\abcc_handler.h \abcc_drv\src\abcc_handler.c	Handler implementation including handler parts that are independent of the operating mode.
Windows: abcc_driver.c	Implementation of the driver access

A.5 System Adaptation Files

File name	Description
\abcc_adapt\abcc_drv_cfg.h	User configuration of driver. Configuration parameters are documented in the driver's public interface abcc_cfg.h.
\abcc_adapt\abcc_identification.h	User configuration to set the identification parameters of a interface
\abcc_adapt\abcc_obj_cfg.h	User configuration of the object implementation
\abcc_adapt\abcc_sw_port.c	Macros and functions required by the driver and object implementation.
\abcc_adapt\abcc_sw_port.h	Macros and functions required by the driver and object implementation. Descriptions of macros are found in \abcc_drv\inc\abcc_port.h.
\abcc_adapt\abcc_td.h	Definition of data types

B API Documentation

The API layer defines a common interface for all network applications to the driver. The interface is found in abcc.h.

B.1 API Functions

Function	Description
Windows: ABCC_AccessDriver()	Opens the driver and establishes the connection to the INpact.
Linux: ABCC_OpenController(), ABCC_ CloseController()	Opens and closes the driver.
ABCC_StartDriver()	Initiates the driver, enables an interrupt, and sets the operating mode. When this function is called the timer system can be started. This function does NOT release the reset of the interface.
ABCC_IsReadyforCommunication()	This function must be polled after ABCC_StartDriver() until it returns the value TRUE. This indicates that the interface is ready for communication and the setup sequence is started.
ABCC_ShutdownDriver()	Stops the driver and puts it into SHUTDOWN state.
ABCC_HWReset()	Interface hardware reset ABCC_ShutdownDriver() is called from this function. This function only sets the reset pin to low. It is the responsibility of the caller to make sure that the reset time (time between ABCC_HWReset() and ABCC_HWReleaseReset() calls) is long enough.
ABCC_HWReleaseReset()	Releases the interface reset
ABCC_RunTimerSystem()	Handles all timers for the driver. It is recommended to call this function on a regular basis from a timer interrupt. Without this function no timeout nor watchdog functionality work.
ABCC_RunDriver()	Drives the driver sending and receiving mechanism. This main routine should be called cyclically during polling. TRUE: Driver is started and ready for communication. FALSE: Driver is stopped or is not started.
ABCC_UserInitComplete()	This function should be called by the application after the last response from the user specific setup is received. This ends the setup sequence and sends ABCC_SETUP_COMPLETE.
ABCC_SendCmdMsg()	Sends a command message to the interface.
ABCC_SendRespMsg()	Sends a response message to the interface.
ABCC_SendRemapRespMsg()	Sends a remap response to the interface.
ABCC_SetAppStatus()	Sets the current application status, according to ABP_AppStatusType in abp.h.
ABCC_GetCmdMsgBuffer()	Allocates the command message buffer
ABCC_ReturnMsgBuffer()	Frees the message buffer.
ABCC_TakeMsgBufferOwnership()	Takes the ownership of the message buffer.
ABCC_ModCap()	Reads the interface capability.
ABCC_LedStatus()	Reads the LED status.
ABCC_AnbState()	Reads the current Anybus state.
ABCC_HwRelease();	Releases the hardware.

B.2 API Related Functions

Function	Description
ABCC_ISR()	This function should be called from the inside interrupt routine to acknowledge and handle received events (triggered by IRQ pin on the application interface).
ABCC_TriggerRdPdUpdate()	Triggers RdPd read.
ABCC_TriggerReceiveMessage()	Triggers the message receive read.
ABCC_TriggerWrPdUpdate()	Indicates that new process data from the application is available and sends the data to the interface.
ABCC_TriggerAnbStatusUpdate()	Checks for a Anybus status change.
ABCC_TriggerTransmitMessage()	Checks the sending queue.

B.3 API Callbacks

All these functions need to be implemented by the application.

Function	Description
ABCC_CbfAdiMappingReq()	Function is called when the driver is about to start automatic process data mapping. It returns mapping information for read and write PD.
ABCC_CbfUserInitReq()	Function is called to trigger a user specific setup during the interface setup state.
ABCC_CbfUpdateWriteProcessData()	Updates the current write process data. Data must be copied into the buffer before returning from the function.
ABCC_CbfNewReadPd()	Called when new process data is received. Process data needs to be copied to application ADIs before returning from the function.
ABCC_CbfReceiveMsg()	Message is received from the interface.
ABCC_CbfWdTimeout()	Function is called when communication with the interface is lost.
ABCC_CbfRemapDone()	This callback is invoked when the REMAP response is successfully sent to the interface.
ABCC_CbfAnbStatusChanged()	This callback is invoked if the interface changes the status i.e. if Anybus state or supervision state is changed.
ABCC_CbfEvent()	Called for unhandled events. Unhandled events are events that are enabled in ABCC_USER_INT_ENABLE_MASK but not present in ABCC_ USER_HANDLE_IN_ABCC_ISR_MASK.
ABCC_CbfSync_Isr()	If sync is supported this function is invoked at a sync event.

B.4 Support Functions

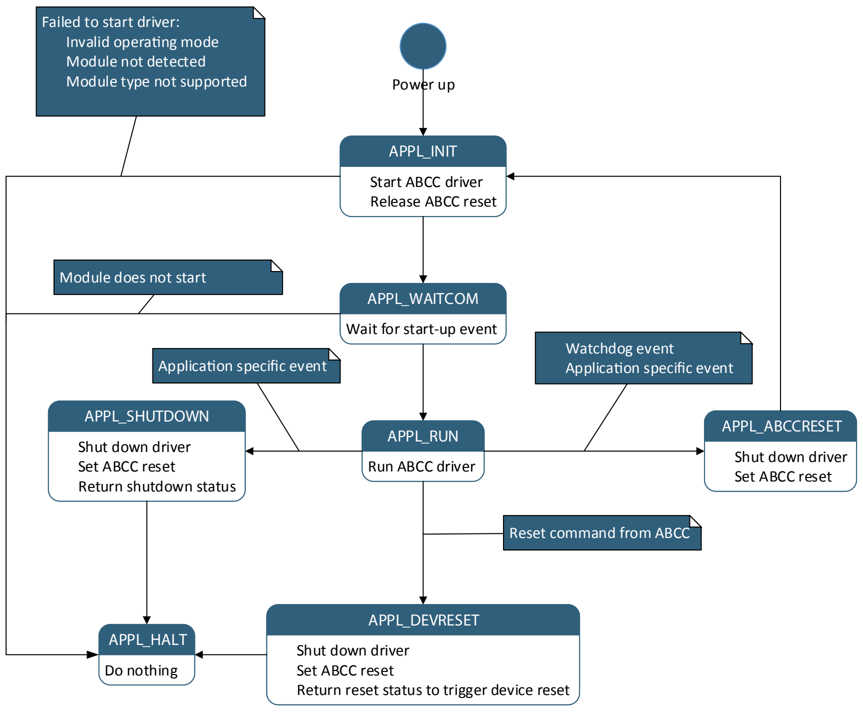
Function	Description
ABCC_NetworkType()	Retrieves the network type.
ABCC_ModuleType()	Retrieves the interface type.
ABCC_DataFormatType()	Retrieves the network endianness.
ABCC_ParameterSupport()	Retrieves the parameter support.
ABCC_GetAttribute()	Fills the message with parameters to get an attribute.
ABCC_SetByteAttribute()	Fills the message with parameters to set an attribute.
ABCC_VerifyMessage()	Verifies a response message.
ABCC_GetDataTypeSize()	Returns the size of an ABP data type.

C Host Application State Machine

The application flow in the example code is maintained using the state machine described in the flowchart below.

The function `APPL_HandleAbcc()`, that is called cyclically from the main loop, implements the state machine and is responsible for the execution of various tasks during each state.

The first time APPL_HandleAbcc() is called, state APPL_INIT is entered.



APPL_INIT

- Checks if a interface is detected.
- Application Data object is initiated, using the desired ADI mapping.
- ABCC_StartDriver() is called to initiate the driver.
- ABCC_HwReleaseReset() is called to release the interface reset.
- Sets state to APPL_WAITCOM.

APPL_WAITCOM

- Waits for the interface to signal that it is ready to communicate.
- Sets state to APPL_RUN.

APPL_RUN

- ABCC_RunDriver()
  - Called to run the driver.
  - Callbacks are invoked for specific events.
  - All callbacks used by the driver are named ABCC_Cbf<x> ().
  - Required callbacks are implemented in *appl_abcc_handler.c*.

During the startup the following events are triggered by the driver (in described order):

- ABCC_CbfStateChanged()
  - Called when the interface entered ABP_ANB_STATE_SETUP.
  - If desired, set a breakpoint or use a debug function to indicate state changes.
- ABCC_CbfAdiMappingReq()
  - Called when interface is ready to send the default mapping command.
  - The generic example code asks the Application Data object for the configured default map.
- ABCC_CbfUserInitReq()
  - Called when sending commands to configure or read information to/from interface is possible for the application.
  - In the example code, the function triggers the user init state machine to start sending a command sequence to the interface.
  - When the last message response is received, the function ABCC-UserInitComplete() is called to notify the driver that the user init sequence ended. This internally triggers the driver to send a SETUP_COMPLETE command to the interface.
  - If no user init is needed, ABCC_UserInitComplete() can be called directly from ABCC_ CbfUserInitReq().
- ABP_ANB_STATE_NW_INIT
  - When the setup is complete the interface enters this state.
  - ABCC_CbfStateChanged() is called.
  - Commands are sent from the interface to the host application objects.
  - All received commands are handled in ABCC_CbfReceiveMsg().
  - Responses to the commands depend on the host objects that are implemented, and the configuration made in *abcc_identification.h* and *abcc_obj_cfg.h.*
  - If desired, set a breakpoint in ABCC_CfgReceiveMsg() to indicate the commands that are sent and how they are handled.
- ABP_ANB_STATE_WAIT_PROCESS
  - When the network initiation is done, the interface enters this state.
  - ABCC_CbgStateChanged() is called by the driver.
  - It is possible to set up an IO connection from the network.
- ABP_ANB_STATE_PROCESS_ACTIVE
  - When an IO connection is set up, the interface enters this state (or, on some networks, ABP_ANB_STATE_IDLE).
  - When process data is received from the interface, the ABCC_CbfNewReadPd() function is called.
  - Example code forwards the data to the Application Data object by calling AD_ UpdatePdReadData(), to update the ADIs.
  - ABCC_TriggerWrPdUpdate() is called to update the write process data, because example code only loops data.

  - ABCC_TriggerWrPdUpdate() function triggers ABCC_CbfUpdateWriteProcessData(), which is called whenever the driver is ready to send new process data. ABCC_ TriggerWrPdUpdate() should always be called when updated process data is available.
- ABP_ANB_STATE_EXCEPTION
  - The cause of the exception can be read from the interface by activating the exception read state machine.
  - RunExceptionSM() is called from state APPL_RUN when the interface is in this state.
- APPL_Reset()
  - Called to initiate a restart of the interface.
  - Happens if the application host object receives a reset request from the interface. The handler state machine enters state APPL_ABCCRESET.
- APPL_RestartAbcc()
  - Used to initiate the restart of the interface is, like APPL_Reset().
  - If called, the handler state machine enters state APPL_ABCCRESET. (Currently this function is not used in the example code. It could be used instead of APPL_Reset(), since it avoids power cycling.)
- APPL_Shutdown()
  - Called to initiate the shutdown of the driver.

APPL_SHUTDOWN

- ABCC_HWReset() is called to reset the interface.
- Sets the state to APPL_HALT.

APPL_ABCCRESET

- ABCC_HWReset() is called to reset the interface.
- Sets the state to APPL_INIT.

APPL_DEVRESET

The return value to the main loop (via the function call from APPL_HandleAbcc()) indicates that the interface should be reset.

- ABCC_HWReset() is called to reset the interface.
- Sets the state to APPL_HALT.

APPL_HALT

- No action.

4.02.0320.20000 1.5 en-US / 2019-04-03 / 12685