Anybus® CompactCom™

Host Application Implementation Guide

HMSI-27-334 1.7 en-US ENGLISH

Important User Information

Disclaimer

Table of Contents

Page

1 Preface

1.1 About This Document

This document describes the host application example code. It provides a guide for making a simple implementation and tips for further development.

For additional related documentation and file downloads, please visit www.anybus.com/support.

1.2 Related Documents

Document	Author	Document ID
Anybus CompactCom 40 Software Design Guide	HMS	HMSI-216-125

1.3 Document history

Version	Date	Description
1.00	2015-11-20	New document
1.10	2016-02-05	Fully revised revision
1.2	2017-01-10	Converted to DOX Major updates
1.3	2018-01-23	Added disclaimer Changed document type
1.4	2018-05-31	Updated API description Added appendix on 30- and 40-series modules in the same application Minor updates
1.5	2018-10-16	Minor updates to fit latest software release
1.6	2019-02-25	Rebranding
1.7	2020-03-09	Minor updates

1.4 Document Conventions

Numbered lists indicate tasks that should be carried out in sequence:

1. First do this
2. Then do this

Bulleted lists are used for:

- Tasks that can be carried out in any order
- Itemized information

► An action

→ and a result

User interaction elements (buttons etc.) are indicated with bold text.

```
Program code and script examples
```

Cross-reference within this document: *Document Conventions, p. 4*

External link (URL): www.hms-networks.com

WARNING
Instruction that must be followed to avoid a risk of death or serious injury.

Caution
Instruction that must be followed to avoid a risk of personal injury.

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

Additional information which may facilitate installation and/or operation.

2 Introduction

When starting an implementation of the Anybus CompactCom 30 or the Anybus CompactCom 40, host application example code is available to speed up the development process. The host application example code includes a driver, which acts as glue between the Anybus CompactCom module 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 an application that can be used as a base for the final product.

This guide is developed to describe a step-by-step implementation of the Anybus CompactCom 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.

This document is based on the contents of version 3.06 of the host application example code.

The guide is divided into two steps:

***Step One**: The adaptations needed for the target hardware are done here and a simple application is developed. The goal with this step is to make sure that the hardware specific code is working and that it is possible to connect to the network and exchange a limited amount of data.*

***Step Two:** The code is adapted to the target product. The goal with this step is to customize the code and add to it, to configure the data that will be sent on the network. After this, the application can be further extended and improved.*

The driver is fully OS independent and it can even be used without an operating system, if required. Furthermore, it can be used for Anybus CompactCom 30 modules as well as Anybus CompactCom 40 modules. The driver supports multiple operating modes, where selection of one of the implemented modes can be made at runtime.

It is possible to use modules from the 30- and the 40-series in the same application, see *30- and 40-series Modules in the Same Application, p. 51*.

The host application example code is available in different versions for different platforms. When writing this guide, the platforms depicted below are available.

Each folder contains all files for a specific platform/development environment.

Platform	Reference Project/Tool Chain	Description
Generic	-	Can be ported to any platform
Xilinx, MicroZed	GNU	Used for the Microzed evaluation platform with Anybus IP
ST, STM3240-EVAL	Keil µVision	Used for the STM3240-EVAL evaluation platform
	IAR Embedded Workbench	Used for the STM3240-EVAL evaluation platform
NXP, TWRP1025	Code Warrior	Used for the NXP TWRP1025 evaluation platform
HMS, USB II Board	Visual Studio	Used for the HMS Starter Kit hardware (USB board)

2.1 Overview

Parts of the driver code need to be adapted to the host application platform. This generally includes functions which access the Anybus host interface, or functions which need to be adapted to integrate the driver into the host system. The figure below shows the different parts of the host application example code.

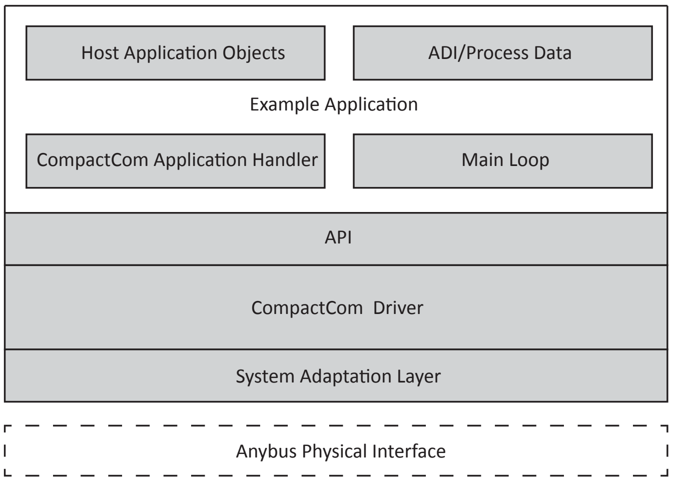


Fig. 1 Software overview

The host application example code is divided into five different folders depending on the functionality and whether the files need to be adapted or not by the user.

Folder Structure	
/abcc_abp (part of the driver - read only)	Contains all Anybus object and communication protocol definitions. Files may be updated when new Anybus CompactCom releases are available. **These files are read only and must not be changed in any way by the user**.
/abcc_drv (part of the driver - read only)	Contains source and header files for the driver. Files may be updated when new Anybus CompactCom releases are available. **These files are read only and must not be changed in any way by the user.**
/abcc_adapt	Contains configuration files. These files must be modified by the user to adapt the driver and the example code to the system environment. **Note**: If using example code adjusted to a specific platform, most of the adaptations needed in this folder are already completed.
/abcc_obj	Includes all Anybus host application object implementations. These 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 Anybus CompactCom messages. – Implementation of callbacks required by the driver. – Definition of ADIs, Application Data Instances, and default process data mapping setup. These files have to be adapted to the application by the programmer. Additionally they may be modified for optimization and/or additional features.

2.2 Preparations

Before continuing, try to answer as many of the questions below as possible. This will make the later decisions during implementation easier. It is also good to have access to the hardware schematics of the target hardware during the implementation.

Step One

Consider the following questions:

- What operating mode, or modes, shall be used in the design?
- What communication interfaces shall be used to communicate with the CompactCom in the design?
- What networks shall be used in the design?
- Are the networks available in the CompactCom 40 series or is there also a need to use CompactCom 30 series modules?
- Are the Module Identification pins connected to the host processor?
- Are the Module Detection pins connected to the host processor?

Step Two

Consider the following questions:

- Is the interrupt signal implemented in the hardware?
- What parameters/data shall be communicated on the network in the final product?
  - Name
  - Data type
  - Number of elements
  - Read/Write access
  - Acyclic access, Cyclic access
  - Max/Min/Default values
- Which events (diagnostics) shall be reported on the network?
- What network identification parameters are available? E.g. Vendor ID, Product Code, Id number etc.

3 Step One

3.1 System Adaptation and Application Development

When this step is completed you have...

- ...implemented the system specific functions needed to communicate with the Anybus CompactCom.
- ...compiled the host application example code with default settings.
- ...exchanged data between the host application and the network master/scanner.

3.2 System Set-up

These defines can be found in `abcc_adapt/abcc_td.h`.

General settings for the system environment, to be used in the driver, are configured here.

3.2.1 Big- or Little Endian

Configure if the host application is a big endian system or a little endian system. Define `ABCC_SYS_BIG_ENDIAN` if it is a big endian system. Do not define (leave as default) if the host application is a little endian system.

```
#define ABCC_SYS_BIG_ENDIAN          /* Big endian host application */

/* #define ABCC_SYS_BIG_ENDIAN  */ /* Little endian host application */
```

3.2.2 16–bit Char System

Configure if the host application is a 16-bit char system or an 8-bit char system (i.e. if the smallest addressable type is 8-bit or 16-bit). Define `ABCC_SYS_16BIT_CHAR` if it is a 16-bit char system. Do not define (leave as default) if it is an 8-bit char system. Configuring of 16-bit char for an 8-bit char system is not recommended.

```
#define ABCC_SYS_16_BIT_CHAR         /* 16 bit char system */

/* #define ABCC_SYS_16_BIT_CHAR */ /* 8 bit char system */
```

3.2.3 Extended Bus Endian Difference

If the endianness for the external parallel data bus differs from the internal data bus endianness, enable this define. If parallel 16-bit operating mode is not used, this define is ignored.

```
#define ABCC_CFG_PAR_EXT_BUS_ENDIAN_DIFF (FALSE)
```

3.2.4 Data Types

Define the Data Types for the current system. For 16-bit char systems, all 8-bit types shall be typed to 16-bit types. The following data types must be defined:

BOOL	Standard boolean data type.
BOOL8	Standard boolean data type, 8-bit.
INT8	Standard signed 8-bit data type.
INT16	Standard singed 16-bit data type.
INT32	Standard signed 32-bit data type.
UINT8	Standard unsigned 8-bit data type.
UINT16	Standard unsigned 16-bit data type.
UINT32	Standard unsigned 32-bit data type.
FLOAT32	Float (according to IEC 60559).

3.3 Anybus CompactCom Set-up

These defines and functions are found in `abcc_adapt/abcc_drv_cfg.h`. Detailed descriptions are available in `abcc_drv/inc/abcc_cfg.h`.

Settings for how to use and communicate with the Anybus CompactCom. Operating mode, interrupt handling, memory handling etc., are configured here.

3.3.1 Communication Interfaces and Operating Modes

Define the communication interfaces and the operating mode between the host application and the CompactCom (Parallel, SPI, Serial), that will be used in the implementation. There are several possibilities to set the operating mode depending on how the host application is intended to communicate with the Anybus and also depending on how the operating mode is selected by the user.

- First, define all communication interfaces that will be supported by the implementation. All interfaces that will be used must be defined here, otherwise an error will be reported later on. Only define the interfaces that will really be used, since every enabled interface will increase the compiled code size.

  Only for 40-series.

  ```
  #define ABCC_CFG_DRV_PARALLEL ( TRUE ) /* Parallel, 8/16-bit, event mode */
  #define ABCC_CFG_DRV_SPI ( FALSE ) /* SPI */
  ```

  For both 30-series and 40-series.

  ```
  #define ABCC_CFG_DRV_SERIAL ( FALSE ) /* Serial */
  #define ABCC_CFG_DRV_PARALLEL_30 ( TRUE ) /* Parallel, 8-bit, half duplex */
  ```

`ABCC_CFG_DRV_SERIAL` and `ABCC_CFG_DRV_PARALLEL_30` use the CompactCom half duplex communication protocol, with limited data sizes for process data and message data.

- Get the operating mode from external hardware - If the operating mode is set e.g. via a dip-switch connected to the host application processor or via an HMI controller, define the `ABCC_CFG_OP_MODE_GETTABLE` and implement the function `ABCC_SYS_GetOpmode()` in `abcc_adapt/abcc_sys_adapt.c`.

  ```
  #define ABCC_CFG_OP_MODE_GETTABLE ( TRUE )
  ```

  If not defined, the operating mode defines must be explicitly defined for the specific module type. (See `ABCC_CFG_ABCC_OP_MODE_30` and `ABCC_CFG_ABCC_OP_MODE_40` below).

- If the operating mode pins on the CompactCom host connector can be controlled by the host processor, define `ABCC_CFG_OP_MODE_SETTABLE` and implement the function `ABCC_SYS_SetOpmode()` in `abcc_adapt/abcc_sys_adapt.c`.

  ```
  #define ABCC_CFG_OP_MODE_SETTABLE ( TRUE )
  ```

  If not defined, it is assumed that the operating mode signals of the CompactCom host connector are fixed or controlled by external hardware, e.g. a dip-switch.

- If only one operating mode per module type (CompactCom 30 and CompactCom 40) is used, define the operating mode with `ABCC_CFG_ABCC_OP_MODE_30` and `ABCC_CFG_ABCC_OP_MODE_40`. The available operating modes (`ABP_OP_MODE_X`) can be found in `abcc_abp/abp.h`.

  ```
  #define ABCC_CFG_ABCC_OP_MODE_30   ABP_OP_MODE_8_BIT_PARALLEL
  #define ABCC_CFG_ABCC_OP_MODE_40   ABP_OP_MODE_16_BIT_PARALLEL
  ```

  If none of these defines are set, `ABCC_SYS_GetOpmode()` must be implemented to retrieve the operating mode from external hardware. See `ABCC_CFG_OP_MODE_GETTABLE` above.

3.3.2 Parallel Operating Mode Specifics

If parallel operating mode (8-bit or 16-bit) is not used, this section can be ignored.

If direct access to the CompactCom memory is available (the host controller provides dedicated signals to access external SRAM), define `ABCC_CFG_MEMORY_MAPPED_ACCESS` to `TRUE` and define the base address with `ABCC_CFG_PARALLEL_BASE_ADR` (this address must be defined to suit the host platform).

```
#define ABCC_CFG_MEMORY_MAPPED_ACCESS ( TRUE )

#define ABCC_CFG_PARALLEL_BASE_ADR ( 0x00000000 )
```

If direct access to the CompactCom memory is not available, several functions to read and write data must be implemented in `abcc_adapt/abcc_sys_adapt.c` (described in `abcc_drv/inc/abcc_sys_adapt_par.h`).

The recommendation is to have direct access to the CompactCom memory if possible for a simpler and most often faster implementation.

3.3.3 SPI Operating Mode Specifics

Only for 40-series. If SPI operating mode is not used, this section can be ignored.

The length of an SPI message fragment in bytes per SPI transaction is defined with `ABCC_CFG_SPI_MSG_FRAG_LEN`.

If the `ABCC_CFG_SPI_MSG_FRAG_LEN` value is less than the largest message to be transmitted, the sending or receiving of a message may be fragmented and take several SPI transactions to be completed. Each SPI transaction will have a message field of this length regardless if a message is present or not. If messages are important the fragment length should be set to the largest message to avoid fragmentation. If IO data are important the message fragment length should be set to a smaller value to speed up the SPI transaction.

For high message performance a fragment length up to 1524 octets is supported. The message header is 12 octets, so 16 or 32 octets will be enough to support small messages without fragmentation.

```
#define ABCC_CFG_SPI_MSG_FRAG_LEN ( 16 )
```

3.3.4 Module ID and Module Detect Settings

- If the Module Identification pins (MI) on the CompactCom host connector are not connected to the host processor, `ABCC_CFG_MODULE_ID_PINS_CONN` must be defined as `FALSE`, and `ABCC_CFG_ABCC_MODULE_ID` must be defined to the correct CompactCom module ID that corresponds to the module ID of the used device. If defined, it shall be set to the correct `ABP_MODULE_ID_X` definition from `abcc_abp/abp.h`.

  If `ABCC_CFG_MODULE_ID_PINS_CONN` is defined as `TRUE`, the function `ABCC_SYS_ReadModuleId()` in `abcc_adapt/abcc_sys_adapt.c` must be implemented.

The recommendation is to connect the Module ID pins on the application connector directly to GPIO-pins on the host processor and implement the `ABCC_SYS_ReadModuleId()` function.

```
#define ABCC_CFG_ABCC_MODULE_ID        ABP_MODULE_ID_ACTIVE_ABCC40
#define ABCC_CFG_MODULE_ID_PINS_CONN     ( TRUE )
```

- If the Module Detect pins (MD) in the host application connector are connected to the host processor, the `ABCC_CFG_MOD_DETECT_PINS_CONN` shall be set to `TRUE` and the `ABCC_SYS_ModuleDetect()` function in `abcc_adapt/abcc_sys_adapt.c` must be implemented.

```
#define ABCC_CFG_MOD_DETECT_PINS_CONN ( TRUE )
```

3.3.5 Message and Process Data Settings

Leave the following defines with the default values for now.

`#define ABCC_CFG_MAX_NUM_APPL_CMDS`	`( 2 )`
`#define ABCC_CFG_MAX_NUM_ABCC_CMDS`	`( 2 )`
`#define ABCC_CFG_MAX_MSG_SIZE`	`( 255 )`
`#define ABCC_CFG_MAX_PROCESS_DATA_SIZE`	`( 512 )`
`#define ABCC_CFG_REMAP_SUPPORT_ENABLED`	`( FALSE )`
`#define ABCC_CFG_CMD_SEQ_MAX_NUM_RETRIES`	`( 0 )`
`#define ABCC_CFG_MAX_NUM_CMD_SEQ`	`( 2 )`

The different platforms can have different default values depending on the available resources.

3.3.6 Interrupt Handling

If the IRQ pin is connected the driver can be configured to check if an event has occurred even if the interrupt is disabled. It can be used e.g. to detect the CompactCom power up event. Define `ABCC_CFG_POLL_ABCC_IRQ_PIN` to enable this functionality, and implement the function `ABCC_SYS_IsAbccInterruptActive()` in `abcc_adapt/abcc_sys_adapt.c`.

```
#define ABCC_CFG_POLL_ABCC_IRQ_PIN ( TRUE )
```

In this step, we will not use the interrupt functionality, which means that we will define `ABCC_CFG_INT_ENABLED` as `FALSE`.

If the IRQ pin is not connected, this define must be set to false.

```
#define ABCC_CFG_INT_ENABLED ( FALSE )
```

3.3.7 Communication Watchdog Settings

The timeout for the CompactCom communication watchdog is configured with `ABCC_CFG_WD_TIMEOUT_MS`. If a timeout occurs, the callback function `ABCC_CbfWdTimeout()` is called.

The watchdog functionality is only supported by the SPI-, serial- and parallel30 (half duplex) operating modes.

```
#define ABCC_CFG_WD_TIMEOUT_MS ( 1000 )
```

3.3.8 ADI Settings

Leave the following defines with the default values for now.

```
#define ABCC_CFG_STRUCT_DATA_TYPE ( FALSE )
```

```
#define ABCC_CFG_ADI_GET_SET_CALLBACK ( FALSE )
```

```
#define ABCC_CFG_64BIT_ADI_SUPPORT ( FALSE )
```

3.3.9 Debug Event Print Settings

For development purposes, a number of debug functions are available for the developer. The following defines affects debug printouts from the driver. If additional printouts are needed from the application code, use the ported function `ABCC_PORT_DebugPrint()` in `abcc_adapt/abcc_sw_port.h`.

- Enable or disable the error reporting callback function `ABCC_CbfDriverError()` with `ABCC_CFG_ERR_REPORTING_ENABLED`. The function is described in `abcc_drv/inc/abcc.h`.

  ```
  #define ABCC_CFG_ERR_REPORTING_ENABLED ( TRUE )
  ```

- Enable or disable driver support for print out of debug events within the driver with `ABCC_CFG_DEBUG_EVENT_ENABLED`. `ABCC_PORT_DebugPrint()` in `abcc_adapt/abcc_sw_port.h` will be used to print debug information.

  ```
  #define ABCC_CFG_DEBUG_EVENT_ENABLED ( TRUE )
  ```

- Enable or disable printout of debug information, such as file name and line number, when `ABCC_CbfDriverError()` is called with `ABCC_CFG_DEBUG_ERR_ENABLED`.

  ```
  #define ABCC_CFG_DEBUG_ERR_ENABLED ( FALSE )
  ```

- Enable or disable printout of received and sent messages with `ABCC_CFG_DEBUG_MESSAGING`. Related events such as buffer allocation and queuing information is also printed.

  `#define ABCC_CFG_DEBUG_MESSAGING ( FALSE )`

- Enable or disable printout of command sequencer actions with `ABCC_CFG_DEBUG_CMD_SEQ_ENABLED`.

  `#define ABCC_CFG_DEBUG_CMD_SEQ_ENABLED ( FALSE )`

3.3.10 Startup Time

If the CompactCom IRQ pin is connected, `ABCC_CFG_STARTUP_TIME_MS` will be used as a timeout while waiting for the CompactCom to become ready for communication. An error (APPL_MODULE_NOT_ANSWERING) will be reported if the start-up interrupt is not received within this time. If the interrupt pin is not available `ABCC_CFG_STARTUP_TIME_MS` will serve as time to wait before starting to communicate with the CompactCom. If not defined, the default value is 1500 ms.

`#define ABCC_CFG_STARTUP_TIME_MS ( 1500 )`

If possible, the recommendation is to use the startup interrupt (option available for the SPI and parallel communication interfaces).

3.3.11 Sync Settings

Only for 40-series.

Leave the following defines with the default values for now.

`#define ABCC_CFG_SYNC_ENABLE ( FALSE )`

`#define ABCC_CFG_SYNC_MEASUREMENT_IP ( FALSE )`

`#define ABCC_CFG_SYNC_MEASUREMENT_OP ( FALSE )`

3.4 System Adaptation Functions

A number of functions must be implemented for the driver to be able to access the Anybus CompactCom. The functions shall be implemented in `abcc_adapt/abcc_sys_adapt.c`. The functions are described per operating mode in the files specified below.

- General functions: `abcc_drv/inc/abcc_sys_adapt.h`
- SPI operating mode: `abcc_drv/inc/abcc_sys_adapt_spi.h`
- Parallel operating mode: `abcc_drv/inc/abcc_sys_adapt_par.h`
- Serial operating mode: `abcc_drv/inc/abcc_sys_adapt_ser.h`

3.4.1 General Functions

These functions can be found in `abcc_drv/inc/abcc_sys_adapt.h`.

ABCC_SYS_HwInit()

This function can be used to initiate the hardware required to communicate with the CompactCom device (e.g. configuring the direction and initial values of used host processor port pins). This function shall be called once during the power up initialization.

Note: Make sure that the CompactCom is kept in reset state when returning from this function.

ABCC_SYS_Init()

This function is called by the driver at start-up and restart of the driver. If needed, any hardware or system dependent initialization shall be done here. If not used, leave the function empty.

ABCC_SYS_Close()

Called from the driver if the driver is terminated. If resources were allocated in `ABCC_SYS_Init()` it is recommended to close or free them in this function. If not used, leave the function empty.

ABCC_SYS_HWReset()

This function must be implemented to pull the reset pin on the Anybus CompactCom interface to low.

ABCC_SYS_HWReleaseReset()

This function must be implemented to set the reset pin on the Anybus CompactCom interface to high.

ABCC_SYS_AbccInterruptEnable()

For now, interrupt will be disabled. Leave this function empty for now.

ABCC_SYS_AbccInterruptDisable()

For now, interrupt will be disabled. Leave this function empty for now.

ABCC_SYS_IsAbccInterruptActive()

If the interrupt pin (IRQ) is connected to the host processor, this function shall read the interrupt signal from the CompactCom and return `TRUE` if the interrupt pin is low (i.e. interrupt is active) and return `FALSE` if the interrupt pin is high (i.e. the interrupt is inactive). It is used to enable polling of the interrupt pin of the CompactCom interface if interrupts are not enabled.

ABCC_SYS_SyncInterruptEnable()

For now, synchronization will be disabled. Leave this function empty for now.

ABCC_SYS_SyncInterruptDisable()

For now, synchronization will be disabled. Leave this function empty for now.

3.4.2 SPI Operating Mode

Only for 40-series. If SPI operating mode is not used, the functions below are never called, and this section can be ignored.

These functions can be found in `abcc_drv/inc/abcc_sys_adapt_spi.h`.

ABCC_SYS_SpiRegDataReceived(ABCC_SYS_SpiDataReceivedCbfType pnDataReceived)

Registers the callback function that shall be called when new data is received (MISO frame received).

Example:

```
static ABCC_SYS_SpiDataReceivedCbfType pnDataReadyCbf;

void ABCC_SYS_SpiRegDataReceived( ABCC_SYS_SpiDataReceivedCbfType
pnDataReceived )
{
   pnDataReadyCbf = pnDataReceived;
}
```

ABCC_SYS_SpiSendReceive(void* pxSendDataBuffer, void* pxReceiveDataBuffer, UINT16 iLength)

Handles sending and receiving of data in SPI mode.

Two buffers are provided, one with a MOSI data frame to be sent and one buffer to store the received MISO frame.

3.4.3 Parallel Operating Mode

These functions can be found in `abcc_drv/inc/abcc_sys_adapt_par.h`.

If parallel operating mode is not used, the functions below are never called, and this section can be ignored.

If parallel operating mode is used and `ABCC_CFG_MEMORY_MAPPED_ACCESS` is defined, this section can be ignored. See Parallel Operating Mode Specifics for more information about `ABCC_CFG_MEMORY_MAPPED_ACCESS`.

ABCC_SYS_ParallelRead()

Reads an amount of octets from the CompactCom memory.

ABCC_SYS_ParallelRead8()

Only used for half duplex parallel operating mode.

Reads an octet from the CompactCom memory.

ABCC_SYS_ParallelRead16()

Reads a word from the CompactCom memory.

ABCC_SYS_ParallelWrite()

Writes an amount of octets to the CompactCom memory.

ABCC_SYS_ParallelWrite8()

Only used for half duplex parallel operating mode.

Writes an octet to the CompactCom memory.

ABCC_SYS_ParallelWrite16()

Writes a word to the CompactCom memory.

ABCC_SYS_ParallelGetRdPdBuffer()

Get the address to the received read process data.

ABCC_SYS_ParallelGetWrPdBuffer()

Get the address to store the write process data.

3.4.4 SerialOperating Mode

These functions can be found in `abcc_drv/inc/abcc_sys_adapt_ser.h`.

If serial operating mode is not used, the functions below are never called, and this section can be ignored.

ABCC_SYS_SerRegDataReceived(ABCC_SYS_SerDataReceivedCbfType pnDataReceived)

Registers a callback function that shall indicate that a new RX telegram has been received on the serial channel.

Example:

```
static ABCC_SYS_SerDataReceivedCbfType pnSerDataReadyCbf;

void ABCC_SYS_SerRegDataReceived( ABCC_SYS_SerDataReceivedCbfType
pnDataReceived )
{
   pnSerDataReadyCbf = pnDataReceived;
}
```

ABCC_SYS_SerSendReceive(void* pxTxDataBuffer, void* pxRxDataBuffer, UINT16 iTxSize, UINT16 iRxSize)

Send TX telegram and prepare for RX telegram reception.

ABCC_SYS_SerRestart(void)

Restart the serial driver. Typically used when a telegram has timed out.

This command flushes all buffers, restarts communication, and starts waiting for a RX telegram with the length of the latest provided RX telegram length.

3.5 Object Configuration

For this step, the default settings in the CompactCom will be used. No host application objects are enabled in the file `abcc_adapt/abcc_obj_cfg.h`.

In Step Two, the network identification attributes will be customized to fit the target product.

3.6 Example Application

An API layer that defines a common interface for all network applications to the Anybus CompactCom driver is available. The API is found in `abcc_drv/inc/abcc.h`. The example application is provided to give an example of how a standard application implements the CompactCom driver using the API. It can be used as it is to be able to test the CompactCom concept and can also be used as a base when implementing the driver into the final application.

3.6.1 ADI 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).

For now, the mapping described in `appl_adimap_speed_example.c` shall be used. This means that `APPL_ACTIVE_ADI_SETUP` in `/example_app/appl_adi_config.h` is defined as `APPL_ADI_SETUP_SPEED_EXAMPLE`.

- `example_app/appl_adimap_speed_example.c` - Simulation of speed and reference speed.
  - ADI 1: "Speed", UINT16, Mapped to Read process data
  - ADI 2: "Ref Speed", UINT16, Mapped to Write process data
  - Data is manipulated with the function APPL_CyclicalProcessing()
  - No structures or callbacks are used.

3.6.2 Main Loop

The main loop is where the execution of the application starts. In the generic project, it is located in the file named `main.c`. Below are some guidelines how to implement the main loop.

- `ABCC_HwInit()` - this function will initiate the hardware required to communicate with the CompactCom, and shall be called once during the power-up initialization. It must also make sure that the CompactCom is kept in reset when returning from the function. The driver can be restarted without calling this function again. `ABCC_HwInit()` will trigger the function `ABCC_SYS_HwInit()` in `abcc_adapt/abcc_sys_adapt.c`, which shall be customized to fit the current system. Make sure this function is one of the first functions called in the main function.

- `APPL_HandleAbcc()` - This function will run the CompactCom state machine and take care of reset, run, and shutdown of the driver, and it must be called periodically from the main loop. A status from the CompactCom driver is returned every time this function is called.

APPL_MODULE_NO_ERROR	The CompactCom is OK. This is the normal response if everything is running normal.
APPL_MODULE_NOT_DETECTED	No CompactCom is detected. Inform the user.
APPL_MODULE_NOT_SUPPORTED	Unsupported module detected. Inform the user.
APPL_MODULE_NOT_ANSWERING	Possible reasons: Wrong communication interface selected, defect module.
APPL_MODULE_RESET	Reset requested from the CompactCom. A reset is received from the network. The application is responsible for restarting the CompactCom.
APPL_MODULE_SHUTDOWN	Shutdown requested.
APPL_MODULE_UNEXPECTED_ERROR	Unexpected error occurred. Inform the user. If necessary, put the outputs in a fail-safe state.

- `ABCC_RunTimerSystem()` - This function shall be called periodically with a known period (ms since last call). This can be done either by having a known delay in the main loop and call the function each iteration, or by setting up a timer interrupt.

  This function is responsible for handling all timers for the CompactCom driver. It is recommended to call this function on a regular basis from a timer interrupt. Without this function no timeout and watchdog functionality will work.

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

```
int main()
{
   APPL_AbccHandlerStatusType eAbccHandlerStatus = APPL_MODULE_NO_ERROR;

   if (ABCC_HwInit() != ABCC_EC_NO_ERROR )
   {
      return ( 0 );
   }
   while ( eAbccHandlerStatus == APPL_MODULE_NO_ERROR )
   {
      eAbccHandlerStatus = APPL_HandleAbcc();
#if( !USE_TIMER_INTERRUPT )
      ABCC_RunTimerSystem( APPL_TIMER_MS );
      DelayMs( APPL_TIMER_MS );
#endif
      switch( eAbccHandlerStatus )
      {
         case APPL_MODULE_RESET:
            Reset();
            break;
         default:
            break;
      }
   }
   return ( 0 );
}
```

3.6.3 Compile and Run

To compile the project, update the make-file to include all the Anybus CompactCom 40 example code (all of the five folders described here) and compile.

- `/abcc_abp`
- `/abcc_drv`
- `/abcc_adapt`
- `/abcc_obj`
- `/example_app`

Before continuing to Step Two, make sure...

- ...the project compiles without errors.
- ...the host application can communicate with the Anybus CompactCom.
- ...data can be exchanged with the network.

4 Step Two

4.1 Adaptations and Customizations

When this step is completed you have...

- …customized the network identification, e.g. Vendor ID, Product Code, Product Name, etc.
- …created ADI:s for the target product.
- …mapped the ADI:s that shall be exchanged cyclically to process data.

4.1.1 Anybus CompactCom Setup

In Step One, some Anybus CompactCom settings were left at default values. We will revisit some of those values here.

Message and Process Data Settings

- The number of message commands that can be sent without receiving a response is configured with `ABCC_CFG_MAX_NUM_APPL_CMDS`. Increasing this value will of course increase the possible number of message commands, but it will also consume more RAM memory.

  `#define ABCC_CFG_MAX_NUM_APPL_CMDS ( 2 )`

- The number of message commands that can be received without sending a response is configured with `ABCC_CFG_MAX_NUM_ABCC_CMDS`. Increasing this value will of course increase the possible number of message commands, but it will also consume more RAM memory.

  `#define ABCC_CFG_MAX_NUM_ABCC_CMDS ( 2 )`

- The size of the largest message in bytes that will be used is configured with `ABCC_CFG_MAX_MSG_SIZE`.

Anybus CompactCom 30 supports 255 bytes messages and Anybus CompactCom 40 supports 1524 bytes messages. `ABCC_CFG_MAX_MSG_SIZE` should be set to largest size that will be sent or received. If this size is not known it recommended to set the maximum supported size.

  `#define ABCC_CFG_MAX_MSG_SIZE ( 255 )`

- The maximum size of the process data in bytes that will be used in either direction is configured with `ABCC_CFG_MAX_PROCESS_DATA_SIZE`. The maximum size is dependent on the type of network that is used. See the corresponding network guide for the networks to be used.

  `#define ABCC_CFG_MAX_PROCESS_DATA_SIZE ( 512 )`

- Enable or disable driver and Application Data object support for the remap command with `ABCC_CFG_REMAP_SUPPORT_ENABLED`. If `TRUE` the `ABCC_CbfRemapDone()` needs to be implemented by the application. The function is described in `abcc_drv/inc/abcc.h`.

  `#define ABCC_CFG_REMAP_SUPPORT_ENABLED ( FALSE )`

- The maximum number of allowed simultaneous message command sequences when using the message command sequencer.

  `#define ABCC_CFG_MAX_NUM_CMD_SEQ ( 2 )`

- Configure the number of retries the message command sequencer shall do if there is no buffer available, before an error is reported.

  `#define ABCC_CFG_CMD_SEQ_MAX_NUM_RETRIES ( 0 )`

Interrupt Handling

The Anybus CompactCom driver can be used either with the interrupt functionality enabled or disabled.

- Define if the CompactCom IRQ pin shall be used along with an interrupt routine by defining `ABCC_CFG_INT_ENABLED`. The IRQ pin can be used in both parallel mode and SPI mode. The function `ABCC_ISR()` shall be called from inside the CompactCom interrupt routine. If the interrupt is flank triggered, the interrupt shall be acknowledged before `ABCC_ISR()` is called.

  ```
  #define ABCC_CFG_INT_ENABLED ( FALSE )
  ```

- **If parallel mode is not used, this define can be ignored.** Configure which interrupts that shall be enabled when using parallel mode with the `ABCC_CFG_INT_ENABLE_MASK_PAR` define. The available options are defined in `abcc_abp/abp.h` (INT MASK Register). If an event is not notified via the CompactCom interrupt, it must be polled by the driver function `ABCC_RunDriver()` (called by `example_app/APPL_HandleAbcc()`). If not defined, the default mask is 0.

  ```
  #define ABCC_CFG_INT_ENABLE_MASK_PAR ( ABP_INTMASK_RDPDIEN | ABP_
  INTMASK_STATUSIEN | ABP_INTMASK_RDMSGIEN | ABP_INTMASK_WRMSGIEN |
  ABP_INTMASK_ANBRIEN )
  ```

- `ABCC_CFG_HANDLE_INT_IN_ISR_MASK` defines what interrupt events for the Anybus CompactCom that are handled in interrupt context. Events that are enabled in the interrupt enable mask (`ABCC_CFG_INT_ENABLE_MASK_X`) but not configured to be handled by the ISR will be translated to a bit field of `ABCC_ISR_EVENT_X` definitions (defined in `abcc_drv/inc/abcc.h`) and forwarded to the user via the `ABCC_CbfEvent()` callback. Only applicable for parallel 8/16-bit operating mode.

  If not defined, the value will be 0, i.e. no events are handled by the ISR.

  ```
  #define ABCC_CFG_HANDLE_INT_IN_ISR_MASK ( ABP_INTMASK_RDPDIEN )
  ```

ADI Settings

- Enable ADI-support for structured data types with `ABCC_CFG_STRUCT_DATA_TYPE`. This define will affect the `AD_AdiEntryType` in `abcc_drv/inc/abcc_ad_if.h`, used for defining the user ADI:s. If defined, the required memory usage will increase, i.e. it should only be defined if structured data types are needed.

  ```
  #define ABCC_CFG_STRUCT_DATA_TYPE ( FALSE )
  ```

- Enable or disable driver support for triggering of callback notifications each time an ADI is read or written with `ABCC_CFG_ADI_GET_SET_CALLBACK`. This define will affect the `AD_AdiEntryType` in `abcc_drv/inc/abcc_ad_if.h`, used for defining the user ADI:s. 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_ADI_GET_SET_CALLBACK ( FALSE )
  ```

- Enable or disable support for 64-bit data types in the Application Data object with `ABCC_CFG_64BIT_ADI_SUPPORT`.

  ```
  #define ABCC_CFG_64BIT_ADI_SUPPORT ( FALSE )
  ```

Sync Settings

Only for 40–series.

- Enable or disable driver support for sync. If TRUE, the abcc_CbfSyncIsr() must be implemented by the application.

  `#define ABCC_CFG_SYNC_ENABLE ( TRUE )`

- **ABCC_SYS_SyncInterruptEnable()**

  Enables the sync interrupt triggered by the sync-pin on the application interface (MIO/SYNC). This function will be called by the driver to enable the sync interrupt.

  Only used when synchronization functionality is enabled.

- **ABCC_SYS_SyncInterruptDisable()**

  Disables the sync interrupt triggered by the sync-pin on the application interface (MIO/SYNC). This function will be called by the driver to disable the sync interrupt.

  Only used when synchronization functionality is enabled.

If sync is not used or if the code is compiled for release, the following defines shall be disabled.

The sync measurement functions are used to measuring the input processing time and the output processing time used in a sync application.

- Enable or disable driver support for measurement of input processing time (used for sync) with ABCC_CFG_SYNC_MEASUREMENT_IP. This define is used during development by activating it and compiling special test versions of the product. When ABCC_CFG_SYNC_MEASUREMENT_IP is TRUE ABCC_SYS_GpioReset() is called when the WRPD has been sent. If running in SPI operating mode it is instead called when ABCC_SpiRunDriver() has finished sending data to the Anybus. When ABCC_CFG_SYNC_MEASUREMENT_IP is TRUE, ABCC_GpioSet() needs to be called at the Input Capture Point.

  `#define ABCC_CFG_SYNC_MEASUREMENT_IP ( FALSE )`

- Enable or disable driver support for measurement of output processing time (used for sync) with ABCC_CFG_SYNC_MEASUREMENT_OP. This define is used during development by activating it and compiling special test versions of the product. When ABCC_CFG_SYNC_MEASUREMENT_OP is TRUE, ABCC_SYS_GpioSet() is called from the RDPDI interrupt. When ABCC_CFG_SYNC_MEASUREMENT_OP is TRUE ABCC_GpioReset() needs to be called at the Output Valid Point.

  `#define ABCC_CFG_SYNC_MEASUREMENT_OP ( FALSE )`

4.1.2 System Adaptation Functions

These functions can be found in `abcc_adapt/abcc_sys_adapt.c`.

If interrupts will be used in Step Two, implement the following functions.

- **ABCC_SYS_AbccInterruptEnable()**

  Enable the CompactCom HW interrupt (IRQ_N pin on the application interface). This function will be called by the driver when the CompactCom interrupt shall be enabled.

  If `ABCC_CFG_INT_ENABLED` is not defined in `abcc_adapt/abcc_drv_cfg.h`, this function does not need to be implemented.

- **ABCC_SYS_AbccInterruptDisable()**

  Disable CompactCom HW interrupt (IRQ_N pin on the application interface).

  If `ABCC_CFG_INT_ENABLED` is not defined in `abcc_adapt/abcc_drv_cfg.h`, this function does not need to be implemented.

4.1.3 Network Identification

So far, all network settings have been left disabled and the product has identified itself as an HMS product, using default values. Now it is time to customize the network identification settings.

Host Application Objects — Networks

Define the networks to be supported by the implementation by defining their respective host application object in the file `abcc_adapt/abcc_obj_cfg.h`. Further implementations of the host application objects are done in the `abcc_obj` folder where each object has its own c- and h-files.

Example:

```
#define PRT_OBJ_ENABLE                              ( TRUE )
#define EIP_OBJ_ENABLE                              ( FALSE )
#define EPL_OBJ_ENABLE                              ( TRUE )
```

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 device is identified on the network. If the attribute is enabled (`TRUE`), the value will be used. If the attribute is disabled (`FALSE`), the attribute's default value will be used. These settings can be found in `abcc_adapt/abcc_identification.h`.

Example:

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

/*
** Attribute 2: Product Code type (UINT32 - 0x00000000-0xFFFFFFFF)
*/

#ifndef EPL_IA_PRODUCT_CODE_ENABLE
   #define EPL_IA_PRODUCT_CODE_ENABLE                 TRUE
   #define EPL_IA_PRODUCT_CODE_VALUE                  0xFFFFFFFF
#endif
```

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()`

Host Application Objects — Other

In `abcc_adapt/abcc_obj_cfg.h`, define all other host application objects that shall be supported by the implementation. If not supported, leave the value as `FALSE`.

Example:

```
          #define ETN_OBJ_ENABLE                                  ( TRUE )
          #define SYNC_OBJ_ENABLE                                 ( FALSE )
```

Host Application Objects — Advanced

The file `abcc_adapt/abcc_obj_cfg.h` contains all attributes for all supported host objects, except for those already defined in `abcc_adapt/abcc_identification.h`. All attributes in this file are disabled by default. Network specific services are labelled "not supported" 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`, `abcc_adapt/abcc_identification.h` and `abcc_adapt/abcc_drv_cfg.h`.

To override a define, just add the desired defines to the `abcc_adapt/abcc_platform_cfg.h` file or use the global defines section in the development environment.

If not used, leave the file empty.

4.1.4 Software Platform Porting

These functions can be found in `abcc_adapt/abcc_sw_port.h`.

The driver uses a number of functions, like memory copying functions, print functions, and functions for critical sections, which can be optimized for the current software platform. These functions can be found in the file `abcc_adapt/abcc_sw_port.h` (described in `abcc_drv/inc/abcc_port.h`). The default example code can be used as-is, but it should be optimized (recommended) for the desired platform later in the implementation project.

ABCC_PORT_DebugPrint()

Used by the driver for debug prints such as events or error debug information. If not defined the driver will be silent. Debug prints can e.g. be sent to a serial terminal or be saved to a logfile.

Critical Section Functions

Critical sections are used when there is a risk of resource conflicts or race conditions between CompactCom interrupt handler context and the application thread.

The following macros are used to implement the critical sections:

- `ABCC_PORT_UseCritical()`
- `ABCC_PORT_EnterCritical()`
- `ABCC_PORT_ExitCritical()`
- `ABCC_PORT_TIMER_UseCritical()`
- `ABCC_PORT_TIMER_EnterCritical()`
- `ABCC_PORT_TIMER_ExitCritical()`

Depending on the configuration of the driver there are different requirements on the critical section implementation. Please choose the most suitable implementation from the numbered list below. The first statement that is true will choose the requirement.

1. The first three macros above need to be implemented if any of the statements below are true.
   - Any message handling is done within interrupt context.

     Requirements:
   - The implementation must support that a critical section is entered from interrupt context. `ABCC_PORT_UseCritical()` should be used for any declarations needed in advance by `ABCC_PORT_EnterCritical()`.
   - When entering the critical section the required interrupts i.e. any interrupt that may lead to driver access, must be disabled. When leaving the critical section the interrupt configuration must be restored to the previous state.
2. `ABCC_PORT_EnterCritical()` and `ABCC_PORT_ExitCritical()` need to be implemented if any of the statements below are true.
   - The application is accessing the CompactCom driver message interface from different processes or threads without protecting the message interface on a higher level (semaphores or similar).

     Requirement:
   - When entering the critical section the required interrupts i.e. any interrupt that may lead to driver access, must be disabled. When leaving the critical section the interrupts must be enabled again.
3. If none of the above is true, no implementation is required.

If the application is calling ABCC_RunTimerSystem() from a timer interrupt, the last three timer-specific macros also need to be defined. If left undefined by the application in abcc_sw_port.h, these will assume the same definition as the corresponding three macros specified above:

Requirements:

- ABCC_PORT_TIMER_UseCritical() should be used for any declarations needed in advance by ABCC_PORT_TIMER_EnterCritical().
- When entering the critical section, the required interrupts i.e. any interrupt that may lead to driver access, specifically the timer system, must be disabled. When leaving the critical section the interrupt configuration must be restored to the previous state.

ABCC_PORT_UseCritical()

If any preparation is needed before calling `ABCC_PORT_EnterCritical()` or `ABCC_PORT_ExitCritical()`, this macro is used to add platform specific necessities.

ABCC_PORT_EnterCritical()

This function is called by the driver when there is a possibility of internal resource conflicts between the CompactCom interrupt handler and the application thread or main loop. The function temporarily disables interrupts to avoid conflict. Note that all interrupts that could lead to a driver access need to be disabled.

ABCC_PORT_ExitCritical()

Restore interrupts to the state they were before `ABCC_PORT_EnterCritical()` was called.

ABCC_PORT_TIMER_UseCritical()

If any preparation is needed before calling `ABCC_PORT_TIMER_EnterCritical()` or `ABCC_PORT_TIMER_ExitCritical()`, this macro is used to add platform specific necessities.

ABCC_PORT_TIMER_EnterCritical()

Disables timer based interrupts, if they are not already disabled.

ABCC_PORT_TIMER_ExitCritical()

Restore interrupts to the state they were before `ABCC_PORT_TIMER_EnterCritical()` was called.

ABCC_PORT_MemCopy()

Copy a number of octets, from the source pointer to the destination pointer.

ABCC_PORT_StrCpyToNative()

Copy a packed string to a native formatted string.

ABCC_PORT_StrCpyToPacked()

Copy a native formatted string to a packed string.

ABCC_PORT_CopyOctets()

Copy octet aligned buffer.

ABCC_PORT_Copy8()

Copy 8 bits from a source to a destination. For a 16 bit char platform octet alignment support (the octet offset is odd) need to be considered when porting this macro.

ABCC_PORT_Copy16()

Copy 16 bits from a source to a destination. Octet alignment support (the octet offset is odd) need to be considered when porting this macro.

ABCC_PORT_Copy32()

Copy 32 bits from a source to a destination. Octet alignment support (the octet offset is odd) need to be considered when porting this macro.

ABCC_PORT_Copy64()

Copy 64 bits from a source to a destination. Octet alignment support (the octet offset is odd) need to be considered when porting this macro.

4.1.5 Example Application

ADI:s and Process Data Mapping

In Step One, the example ADI mapping `appl_adimap_speed_example.h` was used. In the example application there are ADI mapping examples included, which exemplify different types of ADI:s.

Only one mapping can be used at a time. The map that is currently used in the application is configured in the file `example_app/appl_adi_config.h`, by defining `APPL_ACTIVE_ADI_SETUP` to the ADI mapping to be used. For detailed information about how to set up the ADIs, see `abcc_drv/abcc_ad_if.h`.

- `example_app/appl_adimap_speed_example.h` – Simulation of speed and reference speed.

ADI	Description
ADI 1	"Speed", UINT16 (mapped to input data)
ADI 2	"Ref Speed", UINT16 (mapped to output data)

  – Data manipulated with the function APPL_CyclicalProcessing().

  – No structures or callbacks are used.

- `example_app/appl_adimap_simple16.c` – This map loops 32 16-bit words.

ADI	Description
ADI 1	32 element array of UINT16 (mapped to input data)
ADI 2	32 element array of UINT16 (mapped to output data)

  – The ADIs are mapped to process data in each direction..

  – The 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.

ADI	Description
ADI 10	32 element array of UINT16 (mapped to output data)
ADI 11	32 element array of UINT16 (mapped to input data)
ADI 12	UINT16 (not mapped to process data)

  – ADIs 10 and 11 are mapped on process data in each direction.

  – A callback is used when the network reads ADI 11. This callback will increment the value of ADI 12 by one.

  – A callback is used when the network writes ADI 10. This callback copies the value of ADI 10 to ADI 11.

`ABCC_CFG_ADI_GET_SET_CALLBACK` has to be enabled in `abcc_adapt/abcc_drv_cfg.h` since callbacks are used. See ADI Settings for more information.

- `example_app/appl_adimap_alltypes.c` - Example of how structured data types and bit data types can be used

ADI	Description
ADI 20	UINT32 (mapped to output data)
ADI 21	UINT32 (mapped to input data)
ADI 22	SINT32 (mapped to output data)
ADI 23	SINT32 (mapped to input data)
ADI 24	UINT16 (mapped to output data)
ADI 25	UINT16 (mapped to input data)
ADI 26	SINT16 (mapped to output data)
ADI 27	SINT16 (mapped to input data)
ADI 28	BITS16 (mapped to output data)
ADI 29	BITS16 (mapped to input data)
ADI 30	UINT8 (mapped to output data)
ADI 31	UINT8 (mapped to input data)
ADI 32	SINT8 (mapped to output data)
ADI 33	SINT8 (mapped to input data)
ADI 34	PAD8 (mapped to output data, reserved space, no data)
ADI 35	PAD8 (mapped to input data, reserved space, no data)
ADI 36	BIT7 (mapped to output data)
ADI 37	BIT7 (mapped to input data)
ADI 38	Struct (mapped to output data)
ADI 39	Struct (mapped to input data)

`ABCC_CFG_STRUCT_DATA_TYPE` has to be enabled in `abcc_adapt/abcc_drv_cfg.h` since structures are used. See ADI Settings for more information.

No specific functionality is implemented to manipulate with the data in this example.

- `example_app/appl_adimap_asm.c`

  Example of an ADI setup with assembly mapping instances.

- `example_app/appl_adimap_sync.c`

  Simple example of how to handle ADI values in a sync application.

- `example_app/appl_adimap_verif.c`

  Used for internal tests at HMS Industrial Networks.

- `example_app/adimap_bacnet.c`

  BACnet specific ADI example.

The examples implement the following steps that shall be customized to fit the actual implementation:

- ADI Entry List - The ADI:s (i.e. the data instances that will be used in the implementation) must be defined as an `AD_AdiEntryType` in an ADI entry list. All parameters related to an ADI are specified here.

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, a zero length name will be 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
	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 (Only 40-series)
	ABP_BITS8	8 bit bit field (Only 40-series)
	ABP_BITS16	16 bit bit field (Only 40-series)
	ABP_BITS32	32 bit bit field (Only 40-series)
	ABP_BIT1	1 bit bit field (Only 40-series)
	ABP_BIT2	2 bit bit field (Only 40-series)
	:	:
	ABP_BIT7	7 bit bit field (Only 40-series)
	ABP_PAD0	0 pad bit field (Only 40-series)
	ABP_PAD1	1 pad bit field (Only 40-series)
	:	:
	ABP_PAD16	16 pad bit field (Only 40-series)
	DONT_CARE	Use for structured data types
bNumOfElements	For arrays: number of elements of the data type specified in bDataType. For structured data types: number of elements in the 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: ADI is mappable on write process data. ABP_APPD_DESCR_MAPPABLE_READ_PD: ADI is mappable on read process data. The 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. The 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). The type is dependent on bDataType. The use of max/min/default for acyclic messaging must be enabled in the Application Data Object (AD_IA_MIN_MAX_DEFAULT_ENABLE) in abcc_adapt/abcc_obj_cfg.h. Note: Ignored for structured data types	
psStruct	Pointer to an AD_StructDataType. Set to NULL for non structured data types. This field is enabled by defining ABCC_CFG_STRUCT_DATA_TYPE. (Optional, Only 40-series)	
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)	

The different ADI entries in the example code are defined as "const", i.e. the information will be saved in ROM. However, sometimes it is not known at compile time what the ADI list shall look like. In that case, the const declaration must be removed, and the ADI entry structure must be filled out before calling `ABCC_RunDriver()`. The information will then be saved in RAM.

For use of structured data types in an ADI, see the example in `abcc_drv/inc/abcc_ad_if.h`.

- Write and Read Process Data Mapping - ADI:s that shall be mapped as process data are mapped with `AD_MapType`. There is one combined list for both read process data and write process data.

Data Mapping Item	Description
iInstance	ADI number of the ADI to map (see ADI Entry List above)
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 the ADI is not an array or structure, enter 0

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

Note: The mapping sequence is terminated by `AD_MAP_END_ENTRY`, which MUST be present at the end of the list. During the setup sequence, the Anybus CompactCom driver will ask for this information by invoking `ABCC_CbfAdiMappingReq()`.

Example:

```
/* ADI instance no, direction, number of elements in ADI to be mapped,
index of starting element in ADI to be mapped */

AD_MapType APPL_asAdObjDefaultMap[]
{
   { 3,    PD_WRITE,  AD_MAP_ALL_ELEM , 0 },
   { 5,    PD_WRITE,  AD_MAP_ALL_ELEM , 0 },
   { 6,    PD_WRITE,  AD_MAP_ALL_ELEM , 0 },
   { 1,    PD_READ,   AD_MAP_ALL_ELEM , 0 },
   { 2,    PD_READ,   AD_MAP_ALL_ELEM , 0 },
   { 500,  PD_WRITE,  AD_MAP_ALL_ELEM , 0 },
   { 501,  PD_WRITE,  AD_MAP_ALL_ELEM , 0 },
   { 502,  PD_WRITE,  AD_MAP_ALL_ELEM , 0 },
   { 4,    PD_READ,   AD_MAP_ALL_ELEM , 0 },
   { 503,  PD_READ,   AD_MAP_ALL_ELEM , 0 },
   { AD_MAP_END_ENTRY}
};
```

See example of usage in `abcc_drv/inc/abcc_ad_if.h`.

Process Data Callbacks

There are two callback functions related to the update of the process data that must be implemented to inform the host that the read process data has been received from the network or that it is time to update the write process data. An example is available in `example_app/appl_abcc_handler.c`.

- `BOOL ABCC_CbfUpdateWriteProcessData( void* pxWritePd)` - Updates the current write process data. The data must be copied into the buffer (pxWritePd) before returning from the function.
- `void ABCC_CbfNewReadPd( void* pxReadPd)` - Called when new process data has been received from the network. The process data needs to be copied to the application ADI:s (from the buffer pxReadPd) 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 works, in general, for any process data map, but they are also slow because of all considerations needed for the general case. For better performance, please consider writing application specific update functions.

Example:

```
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 ) );
}
```

Event Handling

Only 40-series.

In event mode, all events can be configured to be forwarded to the user via the `ABCC_CbfEvent()` interface using the configuration defines below, located in the file `abcc_drv_cfg.h`.

`#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 configuration above will enable read message and read process data interrupts, but only the read process data callbacks will be executed in interrupt context directly by the driver. The read message event will be forwarded to the application by calling the function `ABCC_CbfEvent()`.

This will reduce the amount of work done in the ISR which causes jitter in the process data handling. Other configurations will of course be possible to set by the user, to increase performance for any event. At this point the user can trigger the handling of the event from any chosen context.

If the messaging is fully event driven and messages are sent in an interrupt context, please consider implementing the critical section porting in `abcc_adapt/abcc_sw_port.h`. The critical section functions are described in `abcc_drv/inc/abcc_port.h`

Example:

```
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.

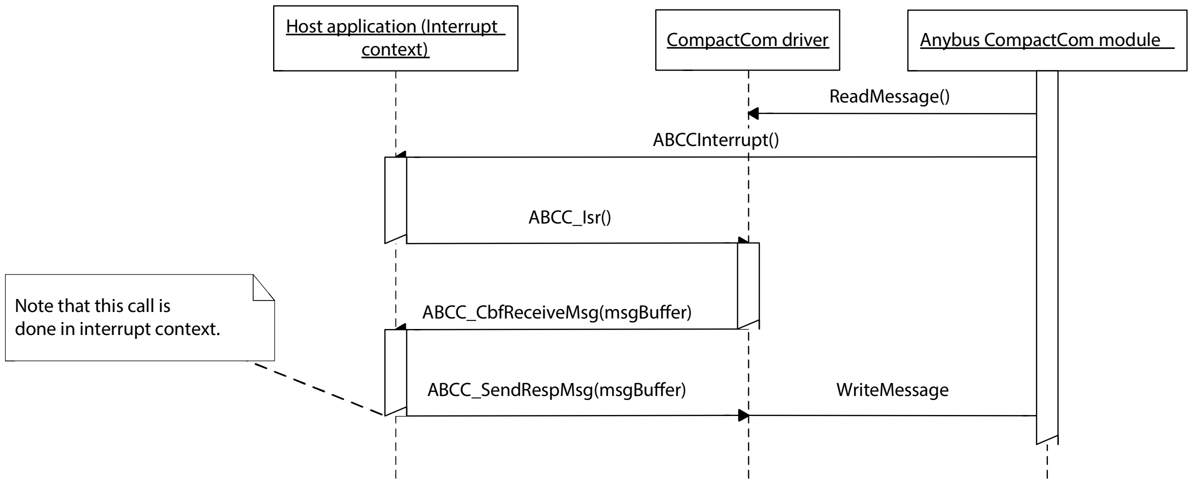
```
volatile BOOL ABCC_fRdMsgEvent = FALSE;

void Task( void )
{
   ABCC_fRdMsgEvent = FALSE;

   while ( 1 )
   {
      if( ABCC_fRdMsgEvent )
      {
         ABCC_fRdMsgEvent = FALSE;
         ABCC_TriggerReceiveMessage();
      }
   }
}
```

This code depicts a task that handles receive message events.

Handling Events in Interrupt Context

Only 40–series.

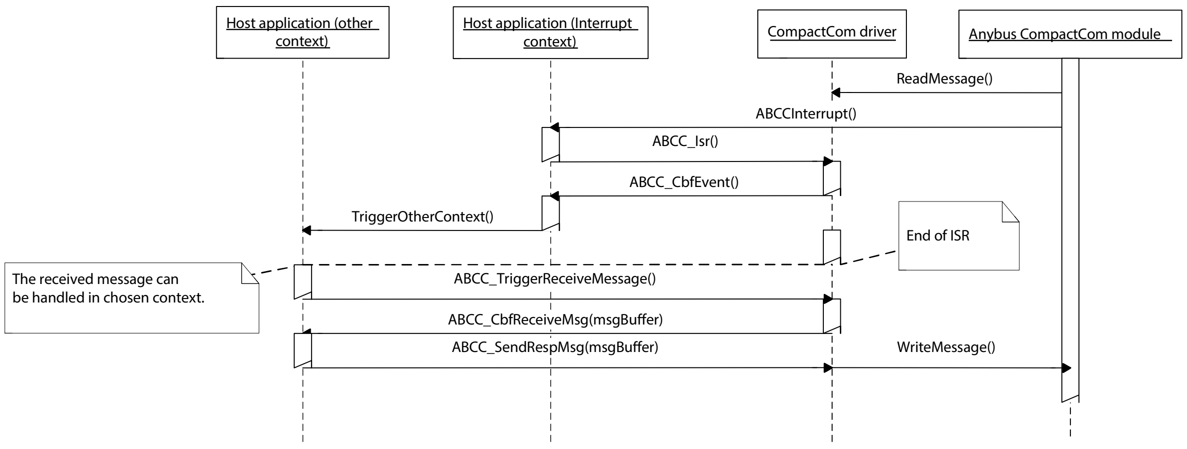
```
#define    ABCC_CFG_INT_ENABLED               ( TRUE  )
#define    ABCC_CFG_INT_ENABLE_MASK_PAR       ( ABP_INTMASK_RDMSGIEN )
#define    ABCC_CFG_HANDLE_INT_IN_ISR_MASK    ( ABP_INTMASK_RDMSGIEN )
```


Host application (Interrupt context)
CompactCom driver
Anybus CompactCom module
ReadMessage()
ABCCInterrupt()
ABCC_Isr()
Note that this call is done in interrupt context.
ABCC_CbfReceiveMsg(msgBuffer)
ABCC_SendRespMsg(msgBuffer)
WriteMessage

Handling Events Using ABCC_CbfEvent() Callback Function

Only 40–series.

```
#define    ABCC_CFG_INT_ENABLED               ( TRUE  )
#define    ABCC_CFG_INT_ENABLE_MASK_PAR       ( ABP_INTMASK_RDMSGIEN )
#define    ABCC_CFG_HANDLE_INT_IN_ISR_MASK    ( 0  )
```


Host application (other context)
Host application (Interrupt context)
CompactCom driver
Anybus CompactCom module
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()

Message Handling

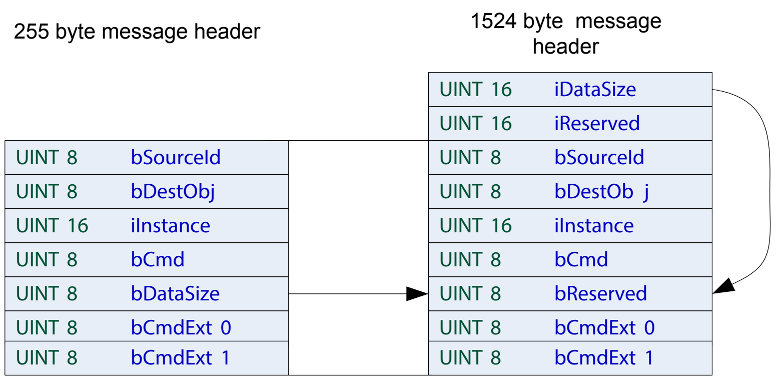
The message handling interface functions are found and described in `abcc.h`.

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.

Note: the buffer resources are configured in the file `abcc_adapt/abcc_drv_cfg.h`.

Note: The CompactCom 40-series devices handle up to 1524 bytes of messaging data, whereas the 30-series only handle 255 bytes. The message header supporting 1524 byte messages differs from the 30-series format since the size field need to be 16 bits instead of 8 bits. The driver supports communication with 30-series devices as well as 40-series devices, but only supports the new message format in the driver API. If a 30-series device is used, the driver will internally convert to the legacy message format. The figure below shows the two message formats.



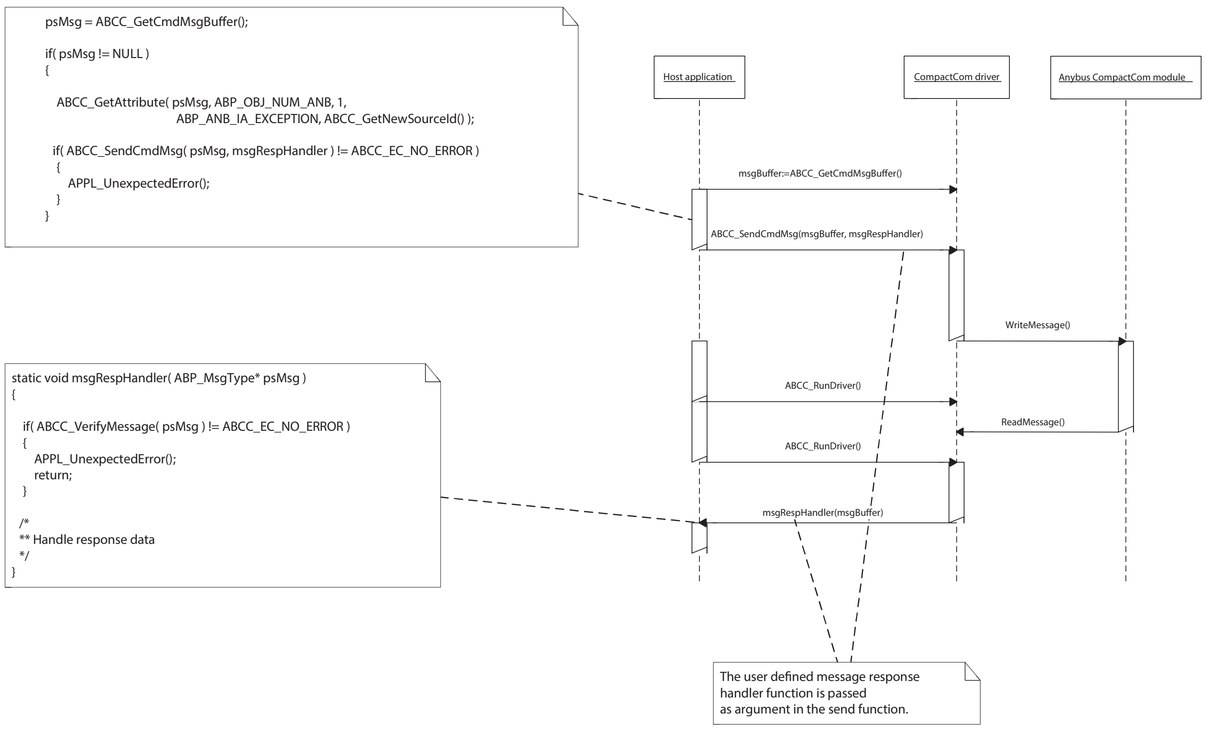
Example 1: Sending a command and receiving a response

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()`.

Sending a command to the CompactCom



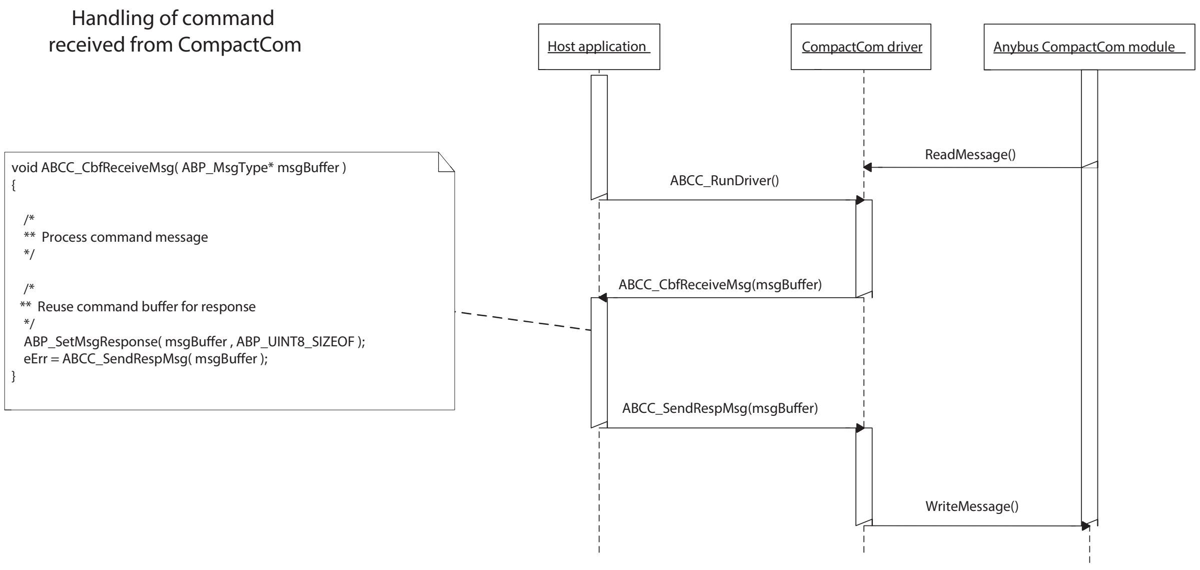
```
psMsg = ABCC_GetCmdMsgBuffer();

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
  */
}
```

The user defined message response handler function is passed as argument in the send function.

Example 2: Receiving a command and sending a response

Note: the received command buffer is reused for the response.


Handling of command received from CompactCom
Host application
CompactCom driver
Anybus CompactCom module
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 Anybus CompactCom message handling. This means that a state machine must be used to keep track of commands and responses.

Command Sequencer

An alternative way to send messages and commands to the CompactCom device is to use the command sequencer. The driver provides support for command buffer allocation, resource control and sequencing of messages. The user must provide functions to build messages and handle responses.

The command sequencer API is described in `abcc_drv\inc\abcc_cmd_seq.h`.

An array of ABCC_CmdSeqType's is provided and defines the command sequence to be executed. The last entry in the array is indicated by NULL pointers. The next command in the sequence will be executed when the previous command has successfully received a response.

If a command sequence response handler exists the response will be passed to the application.

```
static const ABCC_CmdSeqType appl_asUserInitCmdSeq[] =
{
   ABCC_CMD_SEQ( UpdateIpAddress, NULL ),     /* pnCmdHandler, pnRespHandler */
   ABCC_CMD_SEQ( UpdateNodeAddress, NULL ),
   ABCC_CMD_SEQ( UpdateBaudRate, NULL ),
   ABCC_CMD_SEQ_END()                         /* End of sequence */
};
ABCC_AddCmdSeq( appl_asUserInitCmdSeq, UserInitDone );
```

If the command sequence response handler is NULL the application will not be notified. If the error bit is set the application will be notified by the `ABCC_CbfDriverError()` callback.

If the pnCmdSeqDone function callback exists (UserInitDone in the example above) the application will be notified when the whole command sequence has finished. The number of concurrent command sequences is limited by `ABCC_CFG_MAX_NUM_CMD_SEQ` defined in `abcc_drv_cfg.h`.

In `example_app/appl_abcc_handler.c`, there are two examples of usage of the command sequencer.

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

Example 2: When the Anybus CompactCom device indicates exception state, the exception codes are read.

A Software Overview

A.1 Folders

Folders	Description
$(ROOT)/abcc_abp	This folder includes all Anybus protocol files. It may be updated when new Anybus CompactCom software releases are available, but is otherwise read only. The included files are considered read only.
$(ROOT)/abcc_drv/inc	.h files published to the application. The folder contains driver configuration files for the application as well as for the system dependent part of the driver. The included files are considered read only.
$(ROOT)/abcc_drv/src	Anybus CompactCom driver implementation. The included files are considered read only.
$(ROOT)/abcc_adapt	This folder includes all adaptation and configuration files for the driver and the objects. The files must be modified by the user to configure and adapt the driver and the example code.
$(ROOT)/abcc_obj	This folder includes all Anybus host object implementations. The files may be modified by the user.
$(ROOT)/example_app	Example application. The files may be modified by the user.

A.2 Root Files

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

A.3 CompactCom Driver Interface (Read Only)

File Name	Description
/abcc_drv/inc/abcc.h	The public interface for the Anybus CompactCom 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_port.h	Definitions for porting thee Anybus CompactCom to different platforms.
/abcc_drv/inc/abcc_sys_adapt.h	Interface for target dependent functions common to all operating modes.
/abcc_drv/inc/abcc_sys_adapt_spi.h	Interface for target dependent functions needed by abcc_spi_drv.c.
/abcc_drv/inc/abcc_sys_adapt_par.h	Interface for target dependent functions needed by abcc_par_drv.c.
/abcc_drv/inc/abcc_sys_adapt_ser.h	Interface for target dependent functions needed by abcc_ser_drv.c.
/abcc_drv/inc/abcc_cmd_seq_if.h	Interface for the command sequencer.

A.4 Internal Driver Files (Read Only)

The contents of the files in the `/abcc/drv/src` folder should not be changed.

File Name	Description
/abcc_drv/src/abcc_drv_if.h	Interface for low level driver implementing the specific operating mode.
/abcc_drv/src/abcc_debug_err.h /abcc_drv/src/abcc_debug_err.c	Help macros for debugging and error reporting.
/abcc_drv/src/abcc_link.c /abcc_drv/src/abcc_link.h	Message buffer handling and message queue handling.
/abcc_drv/src/abcc_mem.c /abcc_drv/src/abcc_mem.h	Message resource memory support used by abcc_link.c.
/abcc_drv/src/abcc_handler.h /abcc_drv/src/abcc_handler.c	Anybus CompactCom handler implementation including handler parts that are independent of operating mode.
/abcc_drv/src/abcc_setup.h /abcc_drv/src/abcc_setup.c	Anybus CompactCom handler implementation including setup state machine.
/abcc_drv/src/abcc_remap.c	Anybus CompactCom handler implementation for remapping process data at runtime.
/abcc_drv/src/abcc_timer.h /abcc_drv/src/abcc_timer.c	Support for Anybus CompactCom driver timeout functionality.
abcc_drv\src\abcc_cmd_seq.c abcc_drv\src\abcc_cmd_seq.h	Message command sequencer.

A.4.1 8/16 Bit Parallel Event Specific Files

File Name	Description
/abcc_drv/src/par/abcc_handler_par.c	Implements ABCC_RunDriver() and ABCC_ISR().
/abcc_drv/src/par/abcc_par_drv.c	Implements the driver for parallel operating mode.
/abcc_drv/src/par/abcc_drv_par_if.h	Implements the parallel driver interface.

A.4.2 SPI Specific Files

File Name	Description
/abcc_drv/src/par/abcc_handler_spi.c	Implements ABCC_RunDriver() and ABCC_ISR().
/abcc_drv/src/spi/abcc_spi_drv.c	Implements the driver for SPI operating mode.
/abcc_drv/src/spi/abcc_drv_spi_if.h	Implements the SPI driver interface.
/abcc_drv/src/spi/abcc_crc32.c /abcc_drv/src/spi/abcc_crc32.h	Crc32 implementation used by SPI.

A.4.3 8 Bit Parallel Half Duplex Specific Files

File Name	Description
/abcc_drv/src/par30/abcc_handler_par30.c	Implements ABCC_RunDriver() and ABCC_ISR().
/abcc_drv/src/par30/abcc_par30_drv.c	Implements the driver for parallel 30 half duplex operating mode.
/abcc_drv/src/par30/abcc_drv_par30_if.h	Implements the parallel 30 half duplex driver interface.

A.4.4 Serial Specific Files

File Name	Description
/abcc_drv/src/serial/abcc_handler_ser.c	Implements ABCC_RunDriver() and ABCC_ISR().
/abcc_drv/src/serial/abcc_serial_drv.c	Implements the driver for serial operating mode.
/abcc_drv/src/serial/abcc_drv_ser_if.h	Implements the serial driver interface.
/abcc_drv/src/serial/abcc_crc16.c /abcc_drv/src/serial/abcc_crc16.h	Crc16 implementation used by Serial.

A.5 System Adaptation Files

File Name	Description
/abcc_adapt/abcc_drv_cfg.h	User configuration of the CompactCom driver. The 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 an CompactCom module.
/abcc_adapt/abcc_obj_cfg.h	User configuration of the Anybus object implementation.
/abcc_adapt/abcc_sw_port.c	Platform dependent macros and functions required by the CompactCom driver and Anybus object implementation.
/abcc_adapt/abcc_sw_port.h	Platform dependent macros and functions required by the CompactCom driver and Anybus object implementation. The description of the macros are found in abcc_port.h. The file abcc_port.h is found in the public CompactCom driver interface.
/abcc_adapt/abcc_sys_adapt.c	-
/abcc_adapt/abcc_td.h	Definition of CompactCom types.
/abcc_adapt/abcc_platform_cfg.h	Platform specific defines overriding defines in abcc_adapt/abcc_obj_cfg.h, abcc_adapt/abcc_drv_cfg.h and abcc_adapt/abcc_identification.h.

B API

B.1 API Documentation

The Anybus CompactCom API layer defines a common interface for all network applications to the Anybus CompactCom driver. For more information about the interface, see /abcc_dev/inc/abcc.h.

API Functions

Function	Description
ABCC_StartDriver()	Initiates the driver, enables interrupt, and sets the operating mode. When this function has been called the timer system can be started. Note! This function will NOT release the reset of the module.
ABCC_IsReadyforCommunication()	This function must be polled after the ABCC_StartDriver() until it returns the value TRUE. This indicates that the module is ready for communication and the CompactCom setup sequence is started.
ABCC_ShutdownDriver()	Stops the driver and puts it into SHUTDOWN state.
ABCC_HWReset()	Module hardware reset. ABCC_ShutdownDriver() is called from this function. Note! This function will only set reset pin to low. It is the responsibility of the caller to make sure that the reset time (the time between the ABCC_HWReset() and ABCC_HWReleaseReset() calls) is long enough.
ABCC_HWReleaseReset()	Releases the module reset.
ABCC_RunTimerSystem()	Handles all timers for the CompactCom driver. It is recommended to call this function on a regular basis from a timer interrupt. Without this function no timeout and watchdog functionality will work.
ABCC_RunDriver()	Drives the CompactCom driver sending and receiving mechanism. This main routine should be called cyclically during polling.
ABCC_UserInitComplete()	This function should be called by the application when the last response from the user specific setup has been received. This will end the CompactCom setup sequence and ABCC_SETUP_COMPLETE will be sent.
ABCC_SendCmdMsg()	Sends a command message to the module.
ABCC_SendRespMsg()	Sends a response message to the module.
ABCC_SendRemapRespMsg()	Sends a remap response to the module.
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 module capability. This function is only supported by the parallel operating mode.
ABCC_LedStatus()	Reads the LED status. Only supported in SPI and parallel operating mode.
ABCC_AnbState()	Reads the current Anybus state.
ABCC_GetCmdQueueSize()	Sends a response message to the ABCC. The received command buffer can be reused as a response buffer. If a new buffer is used, the function ABCC_GetCmdMsgBuffer() must be used to allocate the buffer.
ABCC_GetAppStatus()	Sets the current application status. This information is only supported in SPI and parallel operating mode. When used for other operating modes the call has no effect.
ABCC_ReadModuleId()	Detects if a module is present. If the MD pins on the host connector are not connected, TRUE will be returned.
ABCC_ModuleDetect()	Reads the module capability. This function is only supported by the Anybus CompactCom 40 parallel mode.

API Functions (continued)

Function	Description
ABCC_IsSupervised()	Retrieves the network type. This function will return a valid value after ABCC_CbfAdiMappingReq has been called by the driver. If called earlier the function will return 0xFFFF which indicates that the network is unknown. The different network types ca be found in abp.h.
ABCC_HWInit	This function will initiate the driver, enable interrupt, and set the operation mode. If a firmware update is pending, a delay (iMaxStartupTime) can be defined, describing how long the driver is to wait for the startup interrupt. If the iMaxStartupTIme is set to zero (0) the driver will use the ABCC_CFG_STARTUP_TIME_MS time. When this function has been called the timer system can be started, see ABCC_ RunTimerSystem(). This function will not release the reset of the Anybus CompactCom. To release the reset, ABCC_HwReleaseReset() has to be called by the application.

API Event Related Functions

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

API Callbacks

Function	Description
ABCC_CbfAdiMappingReq()	The function is called when the driver is about to start the automatic process data mapping. It returns mapping information for read and write PD.
ABCC_CbfUserInitReq()	The function is called to trigger a user specific setup during the module setup state.
ABCC_CbfUpdateWriteProcessData()	Updates the current write process data. The data must be copied into the buffer before returning from the function.
ABCC_CbfNewReadPd()	Called when new process data has been received. The process data needs to be copied to the application ADI:s before returning from the function.
ABCC_CbfReceiveMsg()	A message has been received from the module. This is the receive function for all received commands from the module.
ABCC_CbfWdTimeout()	The function is called when communication with the module has been lost.
ABCC_CbfWdTimeoutRecovered()	Indicates a recent CompactCom watchdog timeout but now the communication is working again.
ABCC_CbfRemapDone()	This callback is invoked when REMAP response is successfully sent to the module.
ABCC_CbfAnbStateChanged()	This callback is invoked if the module changes status i.e. if Anybus state or supervision state is changed.
ABCC_CbfEvent()	Called for unhandled events. Unhandled events are events 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 will be invoked at the sync event.
ABCC_CbfDriverError()	This callback is invoked if the Anybus CompactCom changes states, see ABP_AnbStateType in abp.h for more information.

! All the API callback functions above need to be implemented by the application.

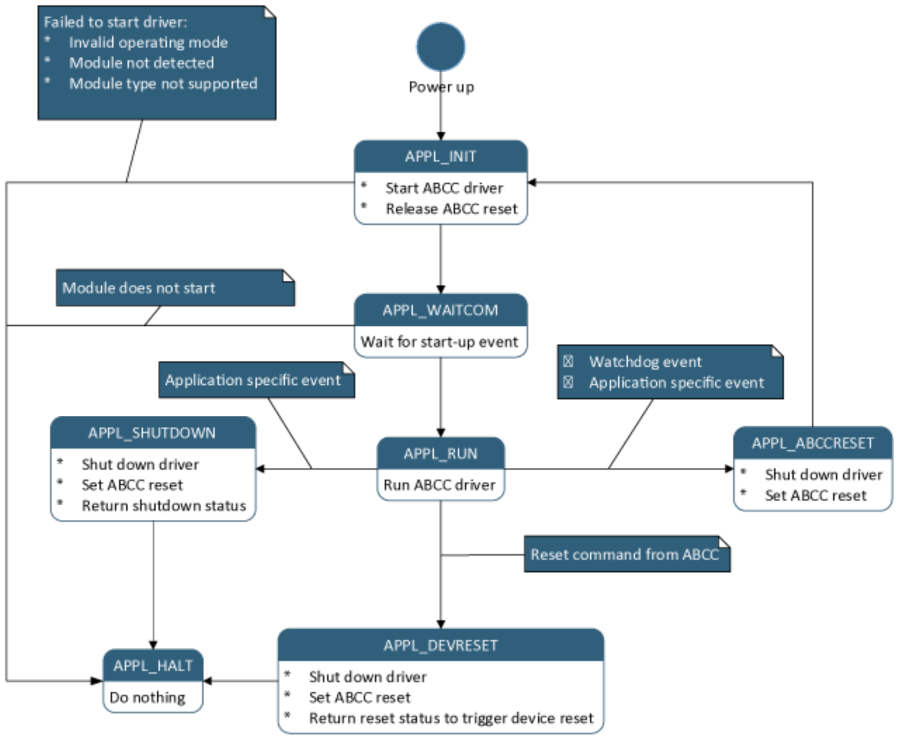
Support Functions	
Function	**Description**
ABCC_NetworkType()	Retrieves the network type.
ABCC_ModuleType()	Retrieves the module type.
ABCC_NetFormatType()	Retrieves the network endianness.
ABCC_ParameterSupport()	Retrieves the parameter support.
ABCC_GetOpmode()	Calls ABCC_SYS_GetOpmode() to read the operating mode from HW.
ABCC_GetAttribute()	Fills an Anybus CompactCom message with parameters to get an attribute.
ABCC_SetByteAttribute()	Fills an Anybus CompactCom message with parameters in order to set an attribute.
ABCC_VerifyMessage()	Verifies an Anybus CompactCom response message.
ABCC_GetDataTypeSize()	Returns the size of an ABP data type.
ABCC_SetMsgHeader()	Sets the input arguments to the ABCC message header correctly
ABCC_GetNewSourceId()	Returns a new source Id that can be used when sending a command message.
ABCC_GetDataTypeSizeInBits	Returns the size of an ABP data type in bits.

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(), found in example_app/appl_abcc_handler.h, is called cyclically from the main loop. It 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 that an Anybus CompactCom device is detected.
- The Application Data object is initiated, using the desired ADI mapping. In this example, it is one of the ADI mapping examples described in *Example Application, p. 29*.
- ABCC_StartDriver() is called to initiate the driver.
- ABCC_HwReleaseReset() is called to release the Anybus CompactCom device reset.
- Sets state to APPL_WAITCOM.

APPL_WAITCOM

- Waits for the Anybus CompactCom device to signal that it is ready to communicate.
- Sets state to APPL_RUN.

APPL_RUN

- ABCC_RunDriver() is called to run the driver. Callbacks will be invoked for specific events. All callbacks used by the driver are named ABCC_Cbf<x>(). The required callbacks are all implemented in example_app/appl_abcc_handler.c.
- During startup the following events will be triggered by the driver (in the described order):
  - ABCC_CbfAnbStateChanged() will be called when the Anybus CompactCom device has entered ABP_ANB_STATE_SETUP. If desired, set a breakpoint or use a debug function to indicate state changes.
  - ABCC_CbfAdiMappingReq() will be called when the CompactCom device is ready to send the default mapping command. The generic example code will ask the Application Data object for the configured default map.
  - ABCC_CbfUserInitReq() will be called when it is possible for the application to send commands to configure or read information to/from the CompactCom device. In the example code, the function triggers the user init state machine to start sending a command sequence to the CompactCom device. When the last message response is received, the function ABCC-UserInitComplete() is called to notify the driver that the user init sequence has ended. This will internally trigger the driver to send a SETUP_ COMPLETE command to the CompactCom device. If no user init is needed, ABCC_ UserInitComplete() can be called directly from ABCC_CbfUserInitReq().
  - When setup is complete, the CompactCom device will enter state ABP_ANB_STATE_ NW_INIT. This means that ABCC_CbfStateChanged() will be called. In this state a number of commands will be sent from the CompactCom device to the host application objects. All received commands will be handled in ABCC_CbfReceiveMsg(). The responses to the commands depend on which 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.
  - When network initiation is done, the CompactCom device will enter state ABP_ANB_ STATE_WAIT_PROCESS. Again, ABCC_CbgStateChanged() will be called by the driver. At this point, it is possible to set up an IO connection from the network.
- When the startup sequence is completed, the following events can be triggered:
  - When an IO connection is set up, the CompactCom will enter state ABP_ANB_STATE_ PROCESS_ACTIVE (or, on some networks, ABP_ANB_STATE_IDLE). When process data is received from the CompactCom device, the ABCC_CbfNewReadPd() function is called. The example code then forwards the data to the Application Data object by calling AD_ UpdatePdReadData(), to update the ADIs. The example code only loops data, so at the end of the function body, ABCC_TriggerWrPdUpdate() is called to update the write process data. The 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 write process data is available.
  - If state ABP_ANB_STATE_EXCEPTION is entered, the cause of the exception can be read from the CompactCom device by activating the exception read state machine. RunExceptionSM() will be called from state APPL_RUN when the CompactCom device is in state ABP_ANB_STATE_EXCEPTION.

  - APPL_Reset() is called to initiate a restart of the device. This will happen if the application host object receives a reset request from the CompactCom device. The CompactCom handler state machine will then enter state APPL_ABCCRESET and start over from APPL_INIT.
  - APPL_RestartAbcc() is, like APPL_Reset(), used to initiate a restart of the device. If called, the CompactCom handler state machine will then enter 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() is called to initiate a shutdown of the driver.

APPL_SHUTDOWN

- ABCC_HWReset() is called to reset the Anybus CompactCom device.
- Sets state to APPL_HALT.

APPL_ABCCRESET

- ABCC_HWReset() is called to reset the Anybus CompactCom device.
- Sets state to APPL_INIT.

APPL_DEVRESET

- ABCC_HWReset() is called to reset the Anybus CompactCom device.
- Sets state to APPL_HALT.

The return value to the main loop (via the function call from APPL_AbccHandler()) will indicate that the device should be reset.

APPL_HALT

No action.

D 30- and 40-series Modules in the Same Application

The Host Application Example Code, provided for communication with the Anybus CompactCom, supports both 30-series and 40-series in the same application. Depending on what series is mounted, the driver will adjust its settings to use the correct communication protocol.

The Host Application Example Code can be downloaded here: www.anybus.com/starterkit40.

Some adaptions of the code for the target system are needed to automatically be able to switch between the 30-series and the 40-series. There are also some things to consider when preparing the hardware design to be able to use both series in the same application.

D.1 Hardware Design Considerations

The 30-series only supports the 8-bit parallel interface and the serial interface. I.e. in order to use both the 30-series and the 40-series in the same application, at least one of those interfaces are mandatory to implement.

Use the table below, and make sure the needed interfaces are implemented when designing the host application hardware. In new designs, it is strongly recommended to use the Event protocol for the 40-series.

Communication Interface	**30-series protocol support**	**40-series protocol support**
Serial Interface	Ping-Pong protocol	Ping-Pong protocol
Parallel 8-bit	Ping-Pong protocol	Ping-Pong protocol and Event protocol
Parallel 16-bit	Not available	Event protocol
SPI	Not available	Event protocol

The OM3 signal in the host application connector shares the same pin as the Tx-signal in the 40-series, and it is not available in the 30-series. To use the serial interface and the parallel 8-bit interface with the Ping-Pong protocol in the 40-series, this pin must be pulled up to 3.3V. A weak internal pull-up is present in the Anybus CompactCom hardware.

Also see the Module Identification section below for more hardware related recommendations.

D.2 Module Identification

In order to automatically detect what module type is mounted, the Module Identification pins (MI[0..1]) need to be connected (signals are available in the host application connector), and the function `ABCC_SYS_ReadModuleId()` must be adapted in the code.

1. In abcc_adapt/abcc_drv_cfg.h:

   ```
   #define ABCC_CFG_MODULE_ID_PINS_CONN ( TRUE )
   ```

2. Adapt the function `ABCC_SYS_ReadModuleId()` and return one of the following values depending on what module type is mounted:

   ```
   ABP_MODULE_ID_ACTIVE_ABCC40
   ```

   ```
   ABP_MODULE_ID_ACTIVE_ABCC30
   ```

If the Module Identification pins are not implemented in the hardware, the Module ID must be set with the define `ABCC_CFG_ABCC_MODULE_ID`.

1. Disable the possibility to automatically read module ID in abcc_adapt/abcc_drv_cfg.h.

   ```
   #define ABCC_CFG_MODULE_ID_PINS_CONN ( FALSE )
   ```

2. Define the current module ID in abcc_adapt/abcc_drv_cfg.h.

   For a 40-series module:

```
#define ABCC_CFG_ABCC_MODULE_ID          ABP_MODULE_ID_ACTIVE_ABCC40
```

   For a 30-series module:

```
#define ABCC_CFG_ABCC_MODULE_ID          ABP_MODULE_ID_ACTIVE_ABCC30
```

If the module identity is required for some reason later in the code, the function `ABCC_ModuleType()` will return the module type for the currently mounted module.

D.3 Enable Supported Communication Interfaces

The 30-series supports the serial and the 8-bit parallel communication interfaces. The 40-series supports all communication interfaces listed below. For more information about the communication interfaces, see abcc_drv/inc/abcc_cfg.h. Only enable the communication interfaces that will actually be used, since every enabled interface will increase the memory size needed.

In abcc_adapt/abcc_drv_cfg.h, enable supported communication interfaces, e.g.:

```
    #define ABCC_CFG_DRV_PARALLEL               ( FALSE )   //Only for 40-series
    #define ABCC_CFG_DRV_SPI                    ( TRUE )    //Only for 40-series
    #define ABCC_CFG_DRV_SERIAL                 ( TRUE )    //30- and 40-series
    #define ABCC_CFG_DRV_PARALLEL_30            ( FALSE )   //30- and 40-series
```

D.4 Select Operating Mode

Since the operating mode can differ between a 40-series module and a 30-series module, it must either be configured by adapting the functions `ABCC_SYS_GetOpmode()` and `ABCC_SYS_SetOpmode()`, or by setting a fixed operating mode for each module type.

If the operating mode is configurable with the above functions, the following must be implemented, and the operating mode pins OM0-OM3 (signals available in the host application connector) must be controllable from the host processor:

1. Enable the operating mode to be gettable from an external source (e.g. configurable via a parameter).

   `#define ABCC_CFG_OP_MODE_GETTABLE ( TRUE )`

2. Implement the function `ABCC_SYS_GetOpmode()` to return the configured operating mode.

3. Enable whether the operating mode is settable via hardware pins on the host processor connected to the Anybus CompactCom.

   `#define ABCC_CFG_OP_MODE_SETTABLE ( TRUE )`

4. Implement the function `ABCC_SYS_SetOpmode()` to set the configured operating mode to the Anybus CompactCom (controlling the OM0-OM3 signals).

If the operating mode is fixed (and never changed) for each module type, the following defines can be used in abcc_adapt/abcc_drv_cfg.h.

1. Disable the function to get the operating mode from an external source.

   `#define ABCC_CFG_OP_MODE_GETTABLE ( FALSE )`

2. Set a fixed operating mode for each module type, e.g.:

```
#define ABCC_CFG_ABCC_OP_MODE_30          ABP_OP_MODE_SERIAL_115_2
#define ABCC_CFG_ABCC_OP_MODE_40          ABP_OP_MODE_SPI
```

   The following options are available for the operating mode settings:

```
ABP_OP_MODE_SPI                           //Only for 40-series
ABP_OP_MODE_16_BIT_PARALLEL               //Only for 40-series
ABP_OP_MODE_8_BIT_PARALLEL                //30- and 40-series
ABP_OP_MODE_SERIAL_19_2                   //30- and 40-series
ABP_OP_MODE_SERIAL_57_6                   //30- and 40-series
ABP_OP_MODE_SERIAL_115_2                  //30- and 40-series
ABP_OP_MODE_SERIAL_625                    //30- and 40-series
```

3. To set the operating mode physically to the Anybus CompactCom, it is also in this case possible to enable the define `ABCC_CFG_OP_MODE_SETTABLE`, and implement the function `ABCC_SYS_SetOpmode()`.

```
#define ABCC_CFG_OP_MODE_SETTABLE ( TRUE )
```

D.5 Message Data Size

The 30-series supports up to 255 bytes of message data. The 40-series supports up to 1524 bytes of message data. This value is used when compiling to set up internal buffers and shall be configured for the largest size that will be used in the application. Remember to consider the limitations for the different module types when sending a message. The define can be found in abcc_adapt/abcc_drv_cfg.h.

```
#define ABCC_CFG_MAX_MSG_SIZE ( 255 )
```

D.6 Process Data Size

The 30-series supports up to 256 bytes of process data in either direction. The 40-series supports up to 4096 bytes of process data in either direction. This value is used when compiling to set up internal buffers and shall be configured for the largest size that will be used in the application. Remember to consider the limitations for the different module types when mapping the process data. The define can be found in abcc_adapt/abcc_drv_cfg.h.

```
#define ABCC_CFG_MAX_PROCESS_DATA_SIZE ( 256 )
```

D.7 Supported Data Types

The 40-series has support for additional data types for the ADIs. When creating the ADIs to be used for the product, make sure the used data types are supported by the used modules.

The data types BOOL1, BITS8, BITS16, BITS32, OCTET, PADx, and BITx are only supported by the 40-series. Structs of any data type are only supported by the 40-series.

HMSI-27-334 1.7 en-US / 2020-03-09 / 17557