XDK110: Getting Started Guide with Extension Bus

Cross-Domain Development Kit XDK110

Platform for Application Development

XDK110: Extension Bus Advanced Guide	
Document revision	1.0
Document release date	**11. May 2018**
Workbench version	3.3.0 and above
Document number	BCDS-XDK110-GUIDE Extension Bus Advanced
Technical reference code(s)	
Notes	Data in this document is subject to change without notice. Product photos and pictures are for illustration purposes only and may differ from the real product's appearance. This document is confidential and under NDA inherent with the purchase of an XDK110. **Advance information – Subject to change without notice**

XDK110
Platform for Application Development

Table of contents

This guide postulates a basic understanding of the XDK and the according workspace. For new users we recommend going through the following guides at xdk.io/guides first:

- *Workbench Installation*
- *Workbench First Steps*
- *XDK Guide FreeRTOS*

1. General Description

1.1 General Introduction

The XDK comes with the ability to attach a 26 pin XDK gateway called extension bus to its own 26 pin connector. The extension bus allows over the 26 additional pins access to pin functionality of the XDKs MCU pins by attaching external devices such as sensors or actuators to it. All MCU pins are GPIOs and can be configured freely within the limitations of the MCU. This way, developers can use the hardware and software of XDK to test new components with minimal application effort.

The availability and the names of the different MCU pins will be explained in section 1.3. For further information on the MCU pin functionality, please refer to the MCU data sheet(LINK).

Other than the basic Extension Bus guide, the Extension Bus Advanced guide will focus on the three most common serial bus protocols UART, SPI and I2C, all of which are mature, industry standard technologies.

1.2 Connecting the Extension Bus

Connect the 26-pin cable (included in delivery) to the extension bus board and to the 26-pin connector of the XDK. The extension bus offers a simple way to implement additional functions with external components. It is optimized for the use with breadboards.

1.3 MCU pins on the Extension Bus

The extension bus I/O pins are labeled with the respective MCU pin designator. The following table is an excerpt of the available pins on the extension bus. It only contains the pins used for the serial bus protocols used in this guide. For the complete table please refer to the basic Extension Bus guide.

Table 1. Pin labelling and configuration of the extension bus

CONNECTOR PIN	MCU PIN	SUGGESTED USE	MULTIPLEX LOCATION	MACRO
B1	PB9	UART1 TX	UART1 #2	EXTENSION_UART1_TX
B2	PB10	UART1 RX	UART1 #2	EXTENSION_UART1_RX
B3	PB2	UART1 RTS	n/a	EXTENSION_UART1_RTS
B4	PF6	UART1 CTS	n/a	EXTENSION_UART1_CTS
B5	PB4	SPI MISO	USART2 #1	EXTENSION_US2_MISO
B6	PB3	SPI MOSI	USART2 #1	EXTENSION_US2_MOSI
B7	PB5	SPI Clock	USART2 #1	EXTENSION_US2_SCK
B8	PD8	SPI Chip select	n/a	EXTENSION_US2_CS
B9	PB11	I2C1 data line (pull-up)	I2C1 #1	EXTENSION_I2C1_SDA
B10	PB12	I2C1 clock line (pull-up)	I2C1 #1	EXTENSION_I2C1_SCL
B13	GND	Power	n/a	

Please note that the pins B9 and B10 are internally modified to suit the standards set for I2C pins. These pins are equipped with an internal pull-up resistor of 3.32 kOhm and are therefore usable for I2C communication only.

Additionally, please note that the pins of the extension bus are protected against electrostatic discharge (ESD) up to 4 kV. This includes a serial resistance of 40 Ohm into the signal path. Please note that this is no problem for communication, but it need to be considered when powering external components via GPIO, e.g. an LED.

Keep those limitations of the Extension Bus gateway in mind when interacting with the UART, SPI and I2C pins. For further information about the general characteristics and limitations, please refer to the basic Extension Bus guide.

Before you start to connect external components to the XDK, please note the following limitations.

The pin voltage of every general purpose extension bus pin is 2.5 V. If you connect external components that work on a different voltage level, a level shifter has to be used to ensure communication with the MCU.
Failing to do so can lead to permanent damage of the XDK!

The micro controller supports four general purpose pin current drive strength settings: 0.5, 2,6 and 20 mA. The default drive strength is set to 6 mA and an alternative level can be defined for each port. For how to configure these current drive strengths, please refer to chapter 1.X in the Extension Bus Basis guide. While each pin can receive or provide maximum 20 mA current, there is a limit on the maximum combined current sourced or sank by all GPIOs.
As the XDK board itself has many useful peripherals connected to the MCU, which may require specific alternative drive strength settings if enabled, it is recommended to use the default 6 mA drive strength for all I/O pins of the extension board connector and to limit the total amount of current sourced or sank through the extension board bus to 50 mA. **Exceeding these limits might permanently damage the micro controller!**

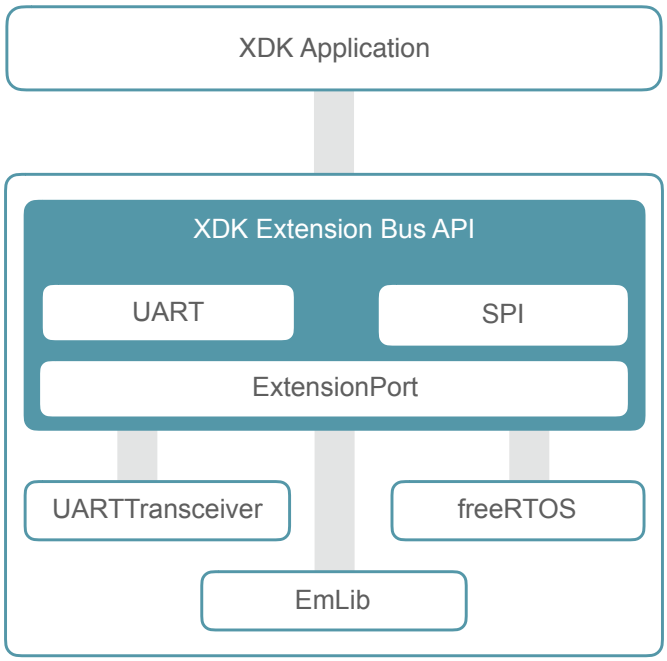
2. Extension Bus Advanced API Overview

It is generally recommended to develop an application based on the highest API level the XDK framework supports, although it is possible to access deeper API levels if the highest level does not provide the functionality required for a specific purpose.

As with most XDK functionalities, there is an API allowing simple access to necessary functions for developing applications that use the serial bus protocols on the Extension Bus. For that the BSP and Utils API provided by the Platform layer will be used.

Picture 1 illustrates the interfaces for UART, SPI and I2C from the corresponding API.

Picture 1. API Overview



2.1 Preparation

The application in this guide is based on an empty XdkApplicationTemplate project, which can be opened from the Welcome-screen of the XDK-Workbench. The code snippets should be placed into `appInitSystem()` one after another. It is recommended to create a new project for every chapter (UART, SPI, I2C), based on an empty XdkApplicationTemplate.

3. UART (Universal Asynchronous Receiver-Transmitter)

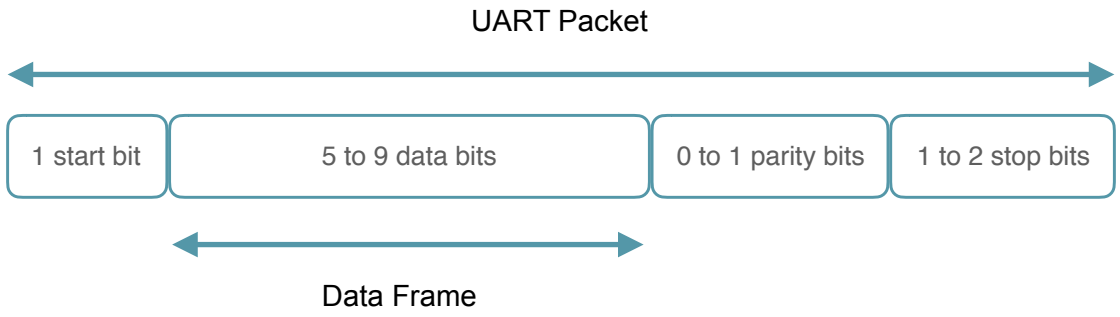
3.1 General Introduction

UART stands short for Universal Asynchronous Receiver Transmitter and it is a physical circuit in a micro controller or a stand-alone IC. A UART's main purpose is to transmit and receive serial data over two wires. In UART communication, two UARTs communicate directly with each other.

UARTs transmit data asynchronously, which means there is no clock signal to synchronize the output of bits from the transmitting UART to the sampling of bits by the receiving UART. Instead of a clock signal, the transmitting UART adds start and stop bits to the data packet being transferred. These bits define the beginning and end of the data packet so the receiving UART knows when to start reading the bits.

Data transmitted via UART is organized into packets. Each packet contains 1 start bit, 5 to 9 data bits (depending on the UART hardware interface), an optional parity bit, and 1 or 2 stop bits:

Picture 2. UART Packet Frame



The data rate for UART range from 230.4kbps to 16Mbps.

Configurable parameters for UART are:

- **Baud rate:** The Baud rate is a measure of the speed of data transfer at which data can be transferred and incoming data is read. It is expressed in bits per second (bps).
- **Start Bit:** The first bit of an ongoing UART data packet transmission. It indicates that the data line is leaving its idle state, typically from logic high to logic low. It simplifies communication between transmitter and receiver, but does not deliver meaningful data.
- **Stop Bit:** The last bit of an ongoing UART data packet transmission represents the end of the data transmission. It raises the logic level back into the idle state, typically to logic high.
- **Parity bit:** Parity describes the evenness or oddness of a number. The parity bit is a way for the receiving UART to tell if any data has changed during transmission. After the receiving UART reads the data frame, it counts the number of bits with a value of 1 and checks if the total is an even or odd number. If the parity bit is a 0 (even parity), the bits in the data frame should total to an even number. If the parity bit is a 1 (odd parity), the 1 bits in the data frame should total to an odd number. When the parity bit matches the data, the UART knows that the transmission was free of error. But if the parity bit is a 0, and the total is odd or the parity bit is a 1, and the total is even, the UART knows that bits in the data frame have changed.

Please note that for a working UART communication it is necessary that both UART endpoints operate at the same baud rate. Furthermore, the same configuration about the start, stop and parity bits need to be applied to both endpoints.

3.2 Initializing and configuring UART

This section describes how to initialize UART either to receive or transmit data. It also covers the necessary configuration for the baud rate, the start bit, stop bit and parity bit. For that, the following interfaces will be used.

Code 1. Including the required UART interfaces

```
#include "BSP_ExtensionPort.h"
#include "BCDS_MCU_UART.h"
#include "BCDS_UARTTransceiver.h "
```

The `BSP_ExtensionPort.h` is used to connect and configure the UART interface on the Extension Bus. With it, the configuration of the Baud rate, the start, stop and parity bits will be implemented.

The following Table shows an excerpt of the functions used to implement a working UART connection.

Table 2. Function overview `BSP_ExtensionPort.h`

FUNCTION	DESCRIPTION
`BSP_ExtensionPort_Connect()`	This function is called to enable the power control for the extension bus and disables all GPIO pins to conserve power
`BSP_ExtensionPort_ConnectUart()`	This function is called to configure the corresponding UART receive (RX) and transmit (TX) pins on the extension bus
`BSP_ExtensionPort_SetUartConfig()`	This function is called to configure UART settings, such as Baud rate, start bit, stop bit and parity bit
`BSP_ExtensionPort_GetUartHandle()`	This function returns the handle for the UART communication holding all relevant configuration data
`BSP_ExtensionPort_EnableUart()`	This function is called to enable the UART module for transmitting and reading data

The interface `BCDS_MCU_UART.h` is used to initialize UART on the driver level to handle the events when data is received or transmitted via UART over the function `MCU_UART_Initialize()`.

Furthermore, the interface `BCDS_UARTTransceiver.h` is used to extend UART to add functionality such as a ring buffer to store the incoming data and to send out data streams greater than one UART data packet.

Table 3 shows an excerpt of the used functionsTable 3. Function overview `BCDS_UARTTransceiver.h`

FUNCTION	DESCRIPTION
`UARTTransceiver_Initialize()`	This function is used to initialize the UART transceiver module
`UARTTransceiver_StartInAsyncMode()`	This function is used to start the UART transceiver for asynchronous data transmission
`UARTTransceiver_ReadData()`	This function is used to read stored UART data from the UART transceiver
`UARTTransceiver_WriteData()`	This function is used to send multiple UART data packets via the UART transceiver

Code 2 shows an outline of the first configurations to the UART module.

Code 2. UART Extension Bus configuration

```
// Place this code snippet inside AppInitSystem() after BCDS_UNUSED(param2);
BSP_ExtensionPort_Connect();
BSP_ExtensionPort_ConnectUart();

BSP_ExtensionPort_SetUartConfig(BSP_EXTENSIONPORT_UART_PARITY,
BSP_EXTENSIONPORT_UART_NO_PARITY, NULL);
BSP_ExtensionPort_SetUartConfig(BSP_EXTENSIONPORT_UART_BAUDRATE, UINT32_C(9600), NULL);
BSP_ExtensionPort_SetUartConfig(BSP_EXTENSIONPORT_UART_STOPBITS,
BSP_EXTENSIONPORT_UART_STOPBITS_ONE, NULL);
```

First, the function `BSP_ExtensionPort_Connect()` is called to enable the power control on the Extension Bus and to disable all pins on the Extension Bus. Afterwards, the function `BSP_ExtensionPort_ConnectUart()` configures the UART pins PB9 and PB10 on the Extension Bus to be used as TX and RX pins.

Afterwards, `BSP_ExtensionPort_SetUartConfig()` is called to configure the baud rate, the number of stop bits and the parity bits. As for this outline, a baud rate of 9600, one stop bit and no parity bit is configured.

Now the initialization of the UART module on the driver level can begin. For that, a global variable of the type `UARTTranceiver_T` as shown in Code 3 needs to be declared.

Code 3. Declaration of a global variable of type `UARTTranceiver_T`

```
static UARTTransceiver_T UartTransceiverInstance;
```

This variable will hold all relevant information in regards to the UART data transmission as well as the UART module itself. The specific information will be for example the amount of received data stored in an internal ring buffer.

Now, a callback function with the type and parameter signature as shown in Code 4 needs to be declared.

Code 4. Callback function for MCU_UART_Initialize()

```
void UartDriverCallBack(UART_T uart, struct MCU_UART_Event_S event){
    if (UartTransceiverInstance.handle == uart){
        UARTTransceiver_LoopCallback(&UartTransceiverInstance, event);
    }
}
```

This function invokes the UART receive and transmit event by passing it to the `UARTTransceiver_LoopCallback()` function. `UARTTransceiver_LoopCallback()` takes the triggered events and reads for example data from the UART module or sends the next UART data packet over it.

Afterwards, the UART driver API can be initialized. Code 5 shows how this is done.

Code 5. Initializing the UART driver API

```
HWHandle_T UartHandle = BSP_ExtensionPort_GetUartHandle();
MCU_UART_Initialize(UartHandle, UartDriverCallBack);
```

Before the initialization function `MCU_UART_Initialize()` can be called, the hardware handle with the configurations made in Code 2 needs to be obtained by calling the function `BSP_ExtensionPort_GetUartHandle()` and passed into a variable of the type `HWHandle_T`. Then the function `MCU_UART_Initialize()` can be called by passing the callback function declared in Code 4 UART handle.

Now that the configuration on the UART hardware level is completed and the driver API is initialized, the utility API for the UARTTransceiver can be initialized.

For that, preparation as shown in Code 6 needs to be done.

Code 6. Preparation for UARTTransceiver

```
#define MAX_UART_RING_BUFFERSIZE   UINT32_C(45)
static uint8_t UartRingBuffer[MAX_UART_RING_BUFFERSIZE];
```

For using the UARTTransceiver a static buffer needs to be declared. This buffer called `UartRingBuffer` will be used as a ring buffer for the UART module and temporarily store the received UART data packets. After it is created, it is recommended to leave it as it is and to do no further operation with it in other code snippets as the proposed ones.

Afterwards, the UARTTransceiver can be initialized. Code 7 showcases how.

Code 7. Initializing the UARTTransceiver

```
enum UARTTransceiver_UartType_E type = UART_TRANSCEIVER_UART_TYPE_UART;
uint8_t rxBuffSize = (MAX_UART_RING_BUFFERSIZE-1);

UARTTransceiver_Initialize(&UartTransceiverInstance, UartHandle, UartRingBuffer,
rxBuffSize, type);
```

First, two variables, `type` and `rxBuffSize,` are declared for better readability. The variable type represents the UART module the UARTTranceiver will take data from or transmit over. Since only UART is currently supported by the UARTTransceiver module, it is simply set to use UART by setting it to the enumeration `UART_TRANSCEIVER_UART_TYPE_UART`.
Please note: Currently LEUART is not supported by the UARTTransceiver module.
Afterwards, the variable `rxBuffSize` is set to the maximum length of elements the ring buffer declared in Code 6 can hold. Both of this variables, as well as the `UartTransceiverInstance`, declared in Code 3 and the obtained hardware handle for the UART module are passed to the function `UARTTransceiver_Initialize()`, which initializes the UARTTransceiver.

Now all configuration and initialization for UART are made. Only a few steps are necessary to start the module to receive and transmit data.

For that, a callback for checking every incoming UART data packet is declared as shown in Code 8.

Code 8. UART data packet frame check callback

```
static bool UartFrameEndCheck(uint8_t lastByte){
    BCDS_UNUSED(lastByte);
    return true;
}
```

The callback simply returns true for every incoming byte. Depending on the application use case, this function can be adapted to trigger custom events for certain received bytes.

Afterwards, a callback function is declared for the receive and transmit events as shown in Code 9.

Code 9. UART event callback

```
static void UartTxRxCallbacks(struct MCU_UART_Event_S event){
    if (event.RxComplete){
        portYIELD_FROM_ISR(pdTRUE);
    }
}
```

Since the UART module is interrupt driven, only a return from the interrupt service routine is required after a bit is received to inform freeRTOS to continue with the interrupted initial task.

Now that all necessary callback functions are declared, the UART module can be started as shown in Code 10.

Code 10. Starting the UART module

```
BSP_ExtensionPort_EnableUart();

UARTTransceiver_StartInAsyncMode(&UartTransceiverInstance, UartFrameEndCheck,
UartTxRxCallbacks);
```

For that, the function `BSP_ExtensionPort_EnableUart()` is called, which enables the UART module to use the configured Extension Bus pins. Afterwards, the `UARTTransceiver_StartInAsyncMode()` is called to start the UARTTransceiver for incoming and outgoing data transmission, and is passed `UartTransceiverInstance` declared in Code 3 and the callback functions declared in Code 8 and 9.

3.3 Transmitting data over UART

Now that all configuration and initialization is done, data can be transmitted via UART as shown in Code 11. For that, a function which takes a buffer holding the data and the buffer length can be build, as shown in Code 11.

Code 11. UART data transmit function

```
void UartDataWrite(uint8_t* buffer, uint32_t bufferLength){

    uint32_t writeTimeout = UINT32_MAX;
    UARTTransceiver_WriteData(&UartTransceiverInstance,buffer, bufferLength, writeTimeout);
}
```

The function simply calls `UARTTransceiver_WriteData()`, which takes the passed buffer and buffer length and the `UartTransceiverInstance` variable holding all UART related information as well as a predefined timeout for the write operation.

It is recommended to wrap the writing function, since only the data buffer itself and the buffer length can vary.

An example on how to transmit data with it is shown in Code 12, please note that for a working data transmission a secondary device using UART need to be attached to the Extension Bus of the XDK.

Code 12. UART writing example

```
void appInitSystem(void * CmdProcessorHandle, uint32_t param2){

    // all other initialization and preparation code here

    uint8_t buffer[] = "Hello";
    uint32_t bufferLength = sizeof(buffer)/sizeof(uint8_t);

    UartDataWrite(buffer,bufferLength);
}
```

Simply a buffer is declared within the string `Hello`. The size of this buffer is dynamically calculated and passed to the function `UartDataWrite()` to transmit the passed data.

3.4 Receiving data over UART

On the opposite, data can also be read from UART. For that, it is recommended to declare a reading function in the same manner as the writing function. A possible example is shown in Code 13.

Code 13. UART da reading function

```
void UartDataRead(uint8_t * buffer, uint32_t bytesRead){

    uint32_t timeout = UINT32_MAX;
    uint32_t actualLength = 0;

    UARTTransceiver_ReadData(&UartTransceiverInstance, buffer, bytesRead, &actualLength,
timeout);
}
```

Similar as in Code 11 the function simply takes a buffer were the data read from UART is stored, and the number of bytes read from the internal ring buffer of the UARTTransceiver. As such, these variables are passed to the function `UARTTransceiver_ReadData(),` along the global variable `UartTransceiverInstance` and predefined timeout for the read operation.

An example of how to use the function is shown in Code 14.

Code 14. UART read example

```
void appInitSystem(void * CmdProcessorHandle, uint32_t param2){

    // all other initialization and preparation code here

    uint8_t buffer[20];
    uint32_t dataLength = 5;

    UartDataRead(buffer,dataLength);
}
```

For its usage, simply a declaration of a buffer with a fixed data size is made and the number of read bytes from the internal ring buffer of the UARTTransceiver is set. Both variables are passed to the `UartDataRead()` function.

Please note that this is only the easiest approach to read data from the internal ring buffer of the UARTTransceiver. It does not contain any detection if there is any stored data in the ring buffer or if a data transmission is ongoing. For that kind of logic, the callback function declared in Code 8 can be adapted to trigger, for example, an event when a data transmission occurs. An alternative approach could also be to build a logic with the properties of the variable `UartTransceiverInstance`, which holds relative information about the internal ring buffer.

4. SPI (serial peripheral interface bus)

4.1 General Introduction

SPI is short for Serial Peripheral Interface and is a synchronous serial bus specification to transfer data between integrated circuits. It uses four lines for the data transmission and integrates the roles of master and slave. With the selection through a chip select line, multiple slave devices can be connected to one master. The data transmission is a full-duplex synchronous communication between the master and multiple slaves.

The serial lines of SPI are named and have a certain functionality as follows:

- **Clock:** Serial Clock controlled by the master device and provided to the slave device for synchronized data exchange.
- **CS (Chip Select):** Chip select is controlled by the master device and as active low line an indication that the master is sending data to or requesting data from the corresponding slave device.
- **MOSI (Master Out Slave In):** Data transmission from the master device to the slave device. The MOSI line on the master device needs to be connected to the MOSI line on the slave device.
- **MISO (Master In Slave Out):** Data transmission from the slave device to the master device. The MISO line on the master device needs to be connected to the MISO line on the slave device.

Possible SPI data rates are between 600 and 230400 bps.

Additional configurable parameters for SPI are:

- **Baud rate:** The Baud rate is a measure of the speed of data transfer at which data can be transferred and incoming data is read. It is expressed in bits per second (bps).
- **SPI mode:** The SPI mode allows manipulation over the clock polarity and the rising or falling edge to modify when incoming or outgoing data is sampled. For a detailed information about which modes are available on the XDK, please refer to the interface `BSP_ExtensionPort.h`.
- **SPI bit order:** The SPI bit order parameter allows the configuration about the sampled byte order of outgoing or incoming data stream is represented with the most significant bit or the least significant bit first.

4.2 Initializing and configuring SPI

This section describes the steps necessary to configure and initialize the SPI module to transmit and receive data. It also covers the necessary configuration for the baud rate, the SPI mode and the bit order. For that, the following interfaces will be used.

Code 15. Including the required SPI interfaces

```
#include "BCDS_MCU_SPI.h"
#include "BSP_ExtensionPort.h"
#include "semphr.h"
```

The `BSP_ExtensionPort.h` is mainly used to connect and configure the SPI interface on the Extension Bus. With it, the configuration of the Baud rate, the SPI mode and the SPI bit order will be implemented.

The following Table shows an excerpt of the functions used to implement a working SPI connection.

Table 4. Function overview `BSP_ExtensionPort.h`

FUNCTION	DESCRIPTION
`BSP_ExtensionPort_Connect()`	This function is called to enable the power control for the extension bus and disables all GPIO pins to preserve power
`BSP_ExtensionPort_ConnectSpi()`	This function is called to configure the corresponding SPI MOSI, MISO and Clock pins on the extension bus
`BSP_ExtensionPort_SetSpiConfig()`	This function is called to configure SPI settings, such as Baud rate, SPI mode and SPI bit order
`BSP_ExtensionPort_GetSpiHandle()`	This function returns the handle for the SPI communication holding all relevant configuration data
`BSP_ExtensionPort_EnableSpi()`	This function is called to enable the SPI module for transmitting and reading data
`BSP_ExtensionPort_ConnectGpio()`	This function is called to connect to a certain GPIO pin on the extension bus
`BSP_ExtensionPort_SetGpioConfig()`	This function is called to set configuration for a specific GPIO pin on the extension bus
`BSP_ExtensionPort_EnableGpio()`	This function is called to enable a specific GPIO pin on the extension bus
`BSP_ExtensionPort_ClearGpio()`	This function is called to clear a state on a certain GPIO pin
`BSP_ExtensionPort_SetGpio()`	This function is called to set a state on a certain GPIO pin

The interface `BCDS_MCU_SPI.h` will be used to initialize SPI on driver level and to transmit and to receive data.

An excerpt of the functions used is shown in the following table,

Table 5 Function overview `BSP_MCU_SPI.h`

FUNCTION	DESCRIPTION
`MCU_SPI_Initialize()`	This function is used to initialize the SPI interface on driver level
`MCU_SPI_Send()`	This function is used to transmit data over the SPI interface
`MCU_SPI_Receive()`	This function is used to receive data from the SPI interface

Furthermore, the interface `semphr.h` is used to add control logic to the implementation of SPI, shown in the next chapter ,via the functions `xSemaphoreTake()`, `xSemaphoreGiveFromISR()` and `xSemaphoreCreateBinary()`.

Code 16 shown an example of the first configuration of SPI with the `BSP_ExtensionPort.h`.

Code 16. SPI Extension Bus configuration

```
BSP_ExtensionPort_Connect();
BSP_ExtensionPort_ConnectSpi();

BSP_ExtensionPort_SetSpiConfig(BSP_EXTENSIONPORT_SPI_BAUDRATE, UINT32_C(2000000), NULL);
BSP_ExtensionPort_SetSpiConfig(BSP_EXTENSIONPORT_SPI_MODE, BSP_EXTENSIONPORT_SPI_MODE0,
NULL);
BSP_ExtensionPort_SetSpiConfig(BSP_EXTENSIONPORT_SPI_BIT_ORDER,
BSP_EXTENSIONPORT_SPI_MSB_FIRST, NULL);
```

First, the function `BSP_ExtensionPort_Connect()` is called to enable the power control on the Extension Bus and to disable all pins on the Extension Bus. Afterwards, the function `BSP_ExtensionPort_ConnectSpi()` configures the SPI pins PB4, PB3 and PB5 on the Extension Bus to be used as MISO, MOSI and Clock pins.

Afterwards, `BSP_ExtensionPort_SetSpiConfig()` is called to configure the baud rate, the SPI mode and the SPI bit order. As for this outline, a baud rate of 9600, the SPI mode 0 and the bit order with the most significant bit first is chosen.

Now the initialization of the SPI module at the driver level can begin. For that, a global variable of the type `HWHandle_T` and `SemaphoreHandle_t` as shown in Code 17 need to be declared.

Code 17. Global variables for the SPI module initializaton

```
HWHandle_T SpiHandle = NULL;
static SemaphoreHandle_t spiCompleteSync = NULL;
```

The global variable of the type `HWHandle_t` will be used to store all SPI related configuration and initialization data. The global variable of the type `SemaphoreHandle_t` will be used for the control logic for sending and receiving data.

Afterwards, the implementation of the function callback for the SPI initialization can begin. For that, a callback function as shown in Code 18 is used.

Code 18. SPI event callback function

```
void SpiAppCallback(SPI_T spi, struct MCU_SPI_Event_S event){

    BCDS_UNUSED(spi);

    if ((event.TxComplete) || (event.RxComplete)){

        portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

        if (xSemaphoreGiveFromISR(spiCompleteSync, &xHigherPriorityTaskWoken) == pdTRUE){
            portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
       }
    }
}
```

The code adds control logic to the triggered events, when a SPI data frame is transmitted or received via the SPI module. A semaphore is inserted for controlled receiving and transmitting of data. After the semaphore `spiCompleteSync` is released, new data can be transmitted or received. This will be further explained and cleared up in the context of code 23.

Afterwards, the SPI module on driver level can be initialized. Code 19 shows how to proceed with that.

Code 19. Initializing the SPI module

```
SpiHandle = BSP_ExtensionPort_GetSpiHandle();

MCU_SPI_Initialize(SpiHandle, SpiAppCallback);
```

First, the function `BSP_ExtensionPort_GetSpiHandle()` is used to store all configurations made with the BSP_ExtensionPort.h interface in the global variable `SpiHandle`. Then this variable and the callback function, declared in Code 18 is passed into the function `MCU_SPI_Initialize()` to initialize the SPI module.

Now, the SPI module itself is configured, including the MISO, MOSI and Clock line. Only the CS line is left to be configured. For that, configuration applied to the Extension Bus pin PB8 needs to be done. Code 20 shows a outline on how to proceed with that.

Code 20. Configuring the chip select line

```
BSP_ExtensionPort_ConnectGpio(BSP_EXTENSIONPORT_GPIO_PD8);
BSP_ExtensionPort_SetGpioConfig(BSP_EXTENSIONPORT_GPIO_PD8, BSP_EXTENSIONPORT_GPIO_PINMODE,
(uint32_t) BSP_EXTENSIONPORT_PUSHPULL, NULL);
BSP_ExtensionPort_EnableGpio(BSP_EXTENSIONPORT_GPIO_PD8);
```

First, we connect to the GPIO pin PD8 on the Extension Bus with the function `BSP_ExtensionPort_ConnectGpio()`. After that, configuration to that pin is applied via the function `BSP_ExtensionPort_SetGpioConfig()` to act as pushpull output.

Afterwards, the pin is enabled by using the function `BSP_ExtensionPort_EnableGpio()`.

Now the pin can be used as chip select line to enable the data transmission between the master and slave.

For that, two functions are implemented to set the chip select level to high and low. The Codes 21 and 22 show an outline of the corresponding implementations.

Code 21. Example CS low function

```
void SpiSetCSLow(void){
    BSP_ExtensionPort_ClearGpio(BSP_EXTENSIONPORT_GPIO_PD8);
}
```

The function from Code 21 is simply structured and only wraps the function call `BSP_ExtensionPort_ClearGpio()` into the functions body. The function `BSP_ExtensionPort_ClearGpio()` on the other hand sets the voltage level on the GPIO pin PD8 to active low.

Code 22. Example CS high function

```
void SpiSetCSHigh(void){
    BSP_ExtensionPort_SetGpio(BSP_EXTENSIONPORT_GPIO_PD8);
}
```

However, the function from Code 22 simply wraps the function call of `BSP_ExtensionPort_SetGpio()`, which ensures that the voltage level on the GPIO pin PD8 is set to high.

Both functions from Code 21 and 22 will be used in the upcoming two sections to transmit and receive data via the SPI module.

4.3 Transmitting data over SPI

Now that all configuration and initialization is done, data can be transmitted via SPI as outlined in Code 11. For that, a function is implement, which can work with sensors or other slave devices attached to the Extension Bus.

Code 23. Write data to a sensor slave over SPI

```
void SpiWriteRegister(uint8_t regAddr, uint8_t *writeVal, uint32_t writeLength){

    Retcode_T sendReturn;
    SpiSetCSLow();

    if ((writeLength > 0) && (spiCompleteSync != NULL)){

       sendReturn = MCU_SPI_Send(SpiHandle, &regAddr, writeLength);

       if(sendReturn == RETCODE_OK){
              xSemaphoreTake(spiCompleteSync, UINT32_C(0));
       }

       sendReturn = MCU_SPI_Send(SpiHandle, writeVal, writeLength);

       if(sendReturn == RETCODE_OK){
           xSemaphoreTake(spiCompleteSync, UINT32_C(0));
       }

    SpiSetCSHigh();
}
```

The function outlined in Code 23 takes three parameters, a register address called `regAddr`, a pointer to the buffer containing the data to be transmitted called `writeVal`, and the length of the data buffer called `writeLength`.

The body of the function defines first a variable of the type `Retcode_T`, this variable will be used to store the return value of the function `MCU_SPI_Send()`. Afterwards, the function `SpiSetCSLow()`, which was implemented in Code 21, is called to set the voltage level of the chip select line to low. This informs the connected slave device that the master is about to send or read data to or from it. Afterwards, an if condition is used to check if the length of the transmit data is is greater than zero and if the `spiCompleteSync` semaphore is not null.

Then the register address itself and its length is send via the function `SPI_MCU_Send()`. The function takes a handle from the type `SPI_T` containing all information regarding the SPI interface, the send byte or buffer, and its length as parameters. **Please note**: since only one byte is send in both write attempts, the write length is identically.

Afterwards, the return value of the function `SPI_MCU_Send()` is evaluated. If the evaluation is okay, then the function `xSemaphoreTake()` is called using the `spiCompleteSync` semaphore with a second parameter, which is used as timeout. In this case here, the parameter given to the function is zero to indicate that no timeout is used.

By taking the semaphore, it is ensured that no previous transmission over SPI is still processed because the semaphore would not be available if that would be the case. The semaphore is then release by the implemented event for `event.TxComplete` in the function `SpiAppCallback()`. Then the procedure is done once more to send the write value.

Afterwards, the chip select line is raised to a high voltage level by using the function `SpiSetCSHigh()`, implemented in Code 22.

A call of the function `SpiWriteRegister()` could then be used, as shown in Code 24.

Code 24. SPI write appInitSystem() function call

```
void appInitSystem(void * CmdProcessorHandle, uint32_t param2)
{
    if (CmdProcessorHandle == NULL)
    {
        printf("Command processor handle is null \n\r");
        assert(false);
    }
    BCDS_UNUSED(param2);

    vTaskDelay(5000);

    // Other Initialization code from the code snipppets 16-20 here

    // Ensuring that CS is at its high state initially
    SpiSetCSHigh();

    uint8_t regAddr = 0x01;
    uint8_t regWriteVal = 0x09;

    uint8_t dataSize = UINT8_C(1);

    SpiWriteRegister(regAddr, &regWriteVal, dataSize);

}
```

Here the function `SpiWriteRegister()` is simply called, by passing a register address, a value written to the register and the length of the value into it.

An alternative approach to send data to the slave device would be achieved by lowering the chip select line to a low voltage level and then call `MCU_SPI_Send()` to transmit then desired data to the slave device. Afterwards, only raising the voltage level on the chip select line to high would be required.

4.4 Receiving data over SPI

Now that transmitting data over SPI was explained, receive data over SPI is explained in this section. For that, a function called `SpiReadRegister()` is implemented in the following outline.

Code 25. Read data from a sensor slave

```
void SpiReadRegister(uint8_t regAddr, uint8_t *readVal, uint32_t readLength){
    /* CS pin Set to Low */
    SpiSetCSLow();

    Retcode_T recvReturn;
    uint8_t readData = 0xFF;

    if ((readLength > 0) && (spiCompleteSync != NULL)){

        recvReturn = MCU_SPI_Send(SpiHandle, &regAddr, readLength);

        if (recvReturn == RETCODE_OK){
            xSemaphoreTake(spiCompleteSync, SPI_DATA_TRANSFER_TIMEOUT_MILLISEC);
        }

        recvReturn = MCU_SPI_Receive(SpiHandle, &readData, readLength);

        if (recvReturn == RETCODE_OK) {
            xSemaphoreTake(spiCompleteSync, SPI_DATA_TRANSFER_TIMEOUT_MILLISEC);
            *readVal = readData;
        }
    }
    /* CS pin Set to High */
    SpiSetCSHigh();
}
```

The function takes three parameters: a register address from where the data is read called `regAddr`, a pointer which will store the read data called `readVal`, and the length of read data called `readLength`. The function body itself calls the function `SpiSetCSLow()` to drive the chip select line to low to inform the slave device that a data transmission is initiated.

First, the local variable `readData` is declared and initialized. This variable will temporarily hold the incoming data received over SPI. Then, an if condition is used to check if the read length is greater than zero and if the `spiCompleteSync` semaphore is not null. After that, the function `MCU_SPI_Send()` is called to initiate data transmission. As inputs, the SPI handle, the register address and the read length are used, to inform the slave device from which register a value should be read.

Afterwards, the return value of this function call is evaluated. If the return value corresponds to `RETCODE_OK`, then the function `xSemaphoreTake()` is called on `spiCompleteSync` to block the function until sending is completed. If sending is not completed within 10ms - as specified by the macro `SPI_DATA_TRANSFER_TIMEOUT_MILLISEC` - a timeout is issued.

After sending is completed, either successfully or due to the timeout, the function `MCU_SPI_Receive()` is called after the semaphore `spiCompleteSync` had been released by a TX complete event. This function receives the SPI handle, the local variable `readData` for storing the incoming data, and the read length.

Again, the return value is evaluated and if `RETCODE_OK` is returned, `xSemaphoreTake()` is called on the semaphore `spiCompleteSync`.

Afterwards, the value of the variable `readData`, which now contains the data received by the SPI module, is stored in `readVal`, as a return value of the function. Finally, the chip select pin is raised back to a high voltage level by calling `SpiSetCSHigh()`, which indicates that the SPI transmission is over.

A call of the function `SpiReadRegister()` could then be used as shown in Code 26.

Code 26. SPI write appInitSystem() function call

```
void appInitSystem(void * CmdProcessorHandle, uint32_t param2)
{
    if (CmdProcessorHandle == NULL)
    {
        printf("Command processor handle is null \n\r");
        assert(false);
    }
    BCDS_UNUSED(param2);

    vTaskDelay(5000);

    // Other Initialization code from the code snipppets 16-20 here

    // Ensuring that CS is at its initial state high
    SpiSetCSHigh();

    uint8_t regAddr = 0x02;
    uint8_t regReadVal = 0x00;

    uint8_t dataSize = UINT8_C(1);

    SpiReadRegister(regAddr, &regReadVal, dataSize);

}
```

Here the function `SpiReadRegister()` is called, passing a register address, a pointer to a variable that stores the incoming data from the register, the length of the data to be read as inputs.

An alternative approach to read incoming data from the slave device would be easily achieved, by only lowering the chip select line to a low voltage level and then call `MCU_SPI_Read()` to read the incoming data from the slave device. Afterwards only raising the voltage level on the chip select line to high would be required.

Note: While this example uses read / write buffers with a size of one byte, the functions `MCU_SPI_SEND()` and `MCU_SPI_READ()` also allow that multiple bytes are read and written within one call. The code must be adapted accordingly.

REV. NO.	CHAPTER	DESCRIPTION OF MODIFICATION/CHANGES	EDITOR	DATE
1.0		Version 1.0 initial release	AFS	2018-05-11