How to use iMOTION™ Configurable UART

About this document

Scope and purpose

This application note provides examples of how to use the iMOTION™ Configurable UART for a given application and describes the methods available to the Configurable UART. Currently iMOTION™ has two major firmware versions: FW 1.03.03 and FW 5.X.X. These have different development tools, namely the MCEWizard/MCEDesigner and the iMOTION™ Solution Designer (iSD) respectively. Therefore, this application note provides script examples for each of the MCEWizard/MCEDesigner and the iSD.

Intended audience

This application note is intended for customers who want to understand how to use the iMOTION™ Configurable UART for their application.

Table of contents

1 Configurable UART Overview

1.1 Introduction

The latest software release of the iMOTION™ Motion Control Engine (MCE) supports two kinds of UART communication options for customers. One option is to follow the predefined User Mode UART communication protocol, and the other option is to implement a customized UART communication protocol by using the Configurable UART function.

The User Mode UART communication protocol is designed to provide a simple, reliable, and scalable communication method for motor control applications. This protocol can easily be implemented in a wide spectrum of microcontrollers, which work as a master to control the MCE and the motor. It supports one-master-multiple-slave networking topology (up to 15 slave nodes on the same network), which is required in some industrial fan/pump applications. Each UART command is processed every 1ms. If you want to know detailed information about the User Mode UART communication protocol, you can refer to section 2.3 of the MCE Software Reference Manual [1] or the MCE Functional Reference Manual [2].

If users want to implement a customized UART communication protocol, it can be realized by using the Configurable UART API described in this document. The Configurable UART function is supported by the Script Engine. The Script Engine is a lightweight virtual machine running in the MCE and enables users to implement system-level functionalities beyond motor control and PFC.

1.2 Overview

The Configurable UART function is a customizable communication protocol that can realize user defined or industry standard communication protocols. The Configurable UART function has two different modes of operation, with each being more useful depending on the communication protocol used. These modes are the Buffer Mode UART and the FIFO Mode UART. The Buffer Mode UART is a simple mode that handles the network layer processing at the firmware level. In contrast, the FIFO mode UART does not do any network layer handling but lets the user handle the network layer using script code. Figure 1 shows the process of using the Configurable UART function. It shows the MCE firmware and part of the relevant hardware peripheral handling the physical layer and the data link layer, while allowing the user to implement the network and the application layer using scripting.

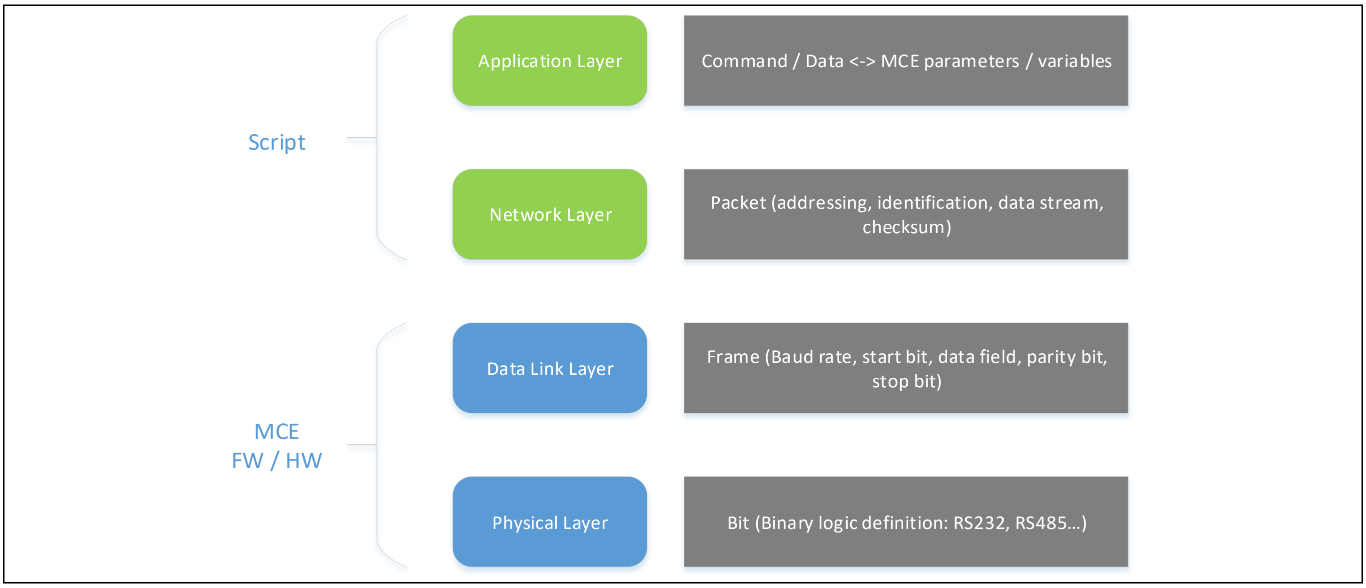

Figure 1 Communication Protocol Layers

Configurable UART Overview

To implement a desired protocol, one must use the Configurable UART APIs along with the Script Engine. Table 1 is a complete list of the Configurable UART APIs available to a user. For more information regarding the Script Engine or Configurable UART APIs please refer to [3] and [2] (or [1]) respectively.

Table 1 Configurable UART API

API name	Brief description
UART_DriverInit()	Initializes the UART hardware driver.
UART_DriverDeinit()	De-initializes the UART hardware driver.
UART_FifoInit()	Initialize UART hardware FIFO.
UART_BufferInit()	Initialize UART software buffer.
UART_GetStatus()	Get the status word for the UART communication status.
UART_GetRxDelay()	Returns the delay time between receive frames.
UART_Control()	Writes to the Control Word that defines UART control commands.
UART_RxFifo()	Returns one byte from the receive FIFO.
UART_TxFifo()	Puts one byte to the transmit FIFO.
UART_RxBuffer()	Returns one byte from the receive buffer from a specified location.
UART_TxBuffer()	Puts one byte in the transmit buffer at a specified location.

1.3 UART Hardware Driver

Figure 2 shows the structure of the UART driver. Using `UART_DriverInit()` users are able to select important parameters related to the UART hardware such as: UART channel, baudrate, data bits, stop bits, parity, or inversion of the tx and rx signals. Before either Buffer Mode or FIFO Mode can be used the user must first initialize the hardware driver. For details about UART driver initialization please refer to [1] and [2].

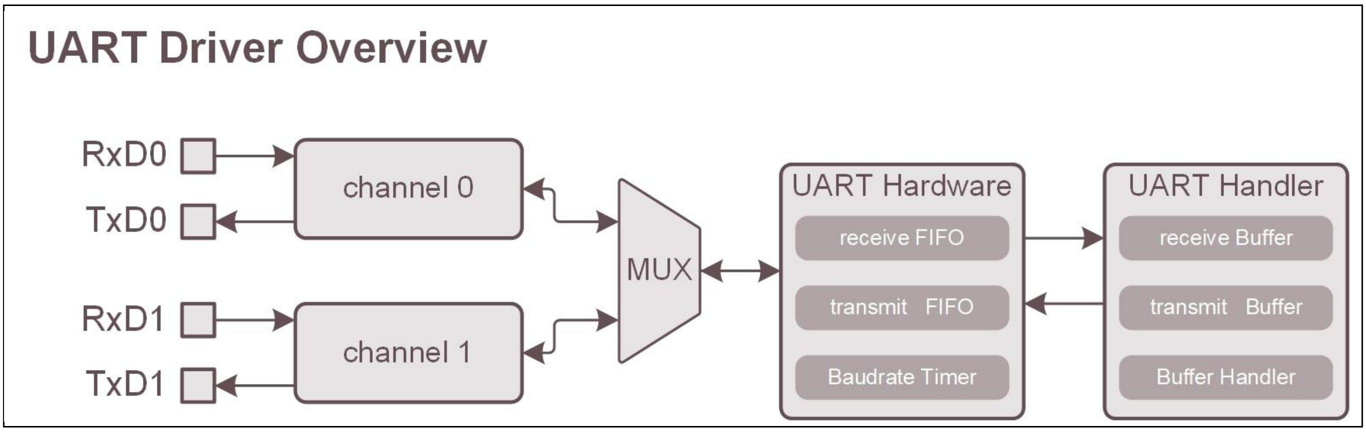

Figure 2 UART Hardware Driver Overview

2 Buffer Mode

2.1 Buffer Mode Description

The Buffer Mode is a UART mode that utilizes the MCE firmware to handle the physical layer, data link layer, and timing related parts of the network layer. As a result, the user can access the buffered data and handle the upper layers without needing to fuss around with the network layer. One limitation with Buffer Mode is that the number of data bytes in a frame needs to be fixed. The Buffer Mode is configurable by initialization and provides access to the data buffers and status information during runtime. Figure 3 is an overview of the Buffer Mode state machine.

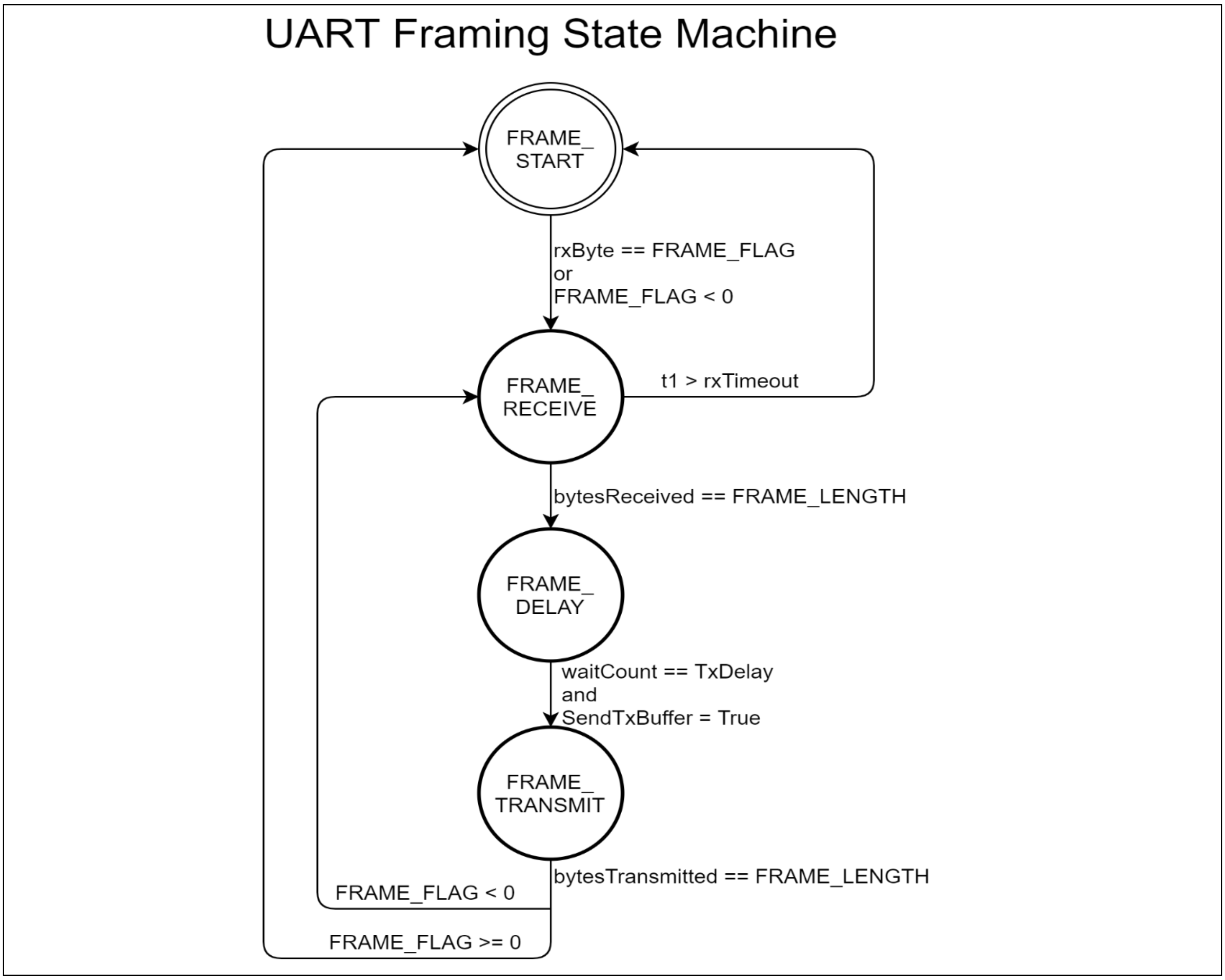

Figure 3 Buffer Mode State Machine

Table 2 State Description and Transition

State	State Functionality	Transition Event	Next Sequence State
FRAME_START	Bytes are received and compared with a FRAME_FLAG. Any bytes not matching FRAME_FLAG are ignored, and a matching byte signifies the transition event.	Received byte matches a known FRAME_FLAG.	FRAME_RECEIVE
FRAME_RECEIVE	Bytes are received up to the frame length, and the receive delay timer is stopped. Once all bytes have been received, the receive delay timer starts again[1].	If bytes received is equal to the frame length.	FRAME_DELAY
		If time from first received byte to last received byte is greater than rxTimeout.	FRAME_START
FRAME_DELAY	The state machine remains in this state for the configured transmit delay.	When txDelay is met, and sendTxBuffer is true.	FRAME_TRANSMIT
FRAME_TRANSMIT	The transmit buffer is sent. A delay between each byte can be configured[2].	When frame flag is invalid.	FRAME_RECEIVE
		When all bytes of transmit frame has been sent, and frame flag is valid.	FRAME_START

Note: For more information regarding timing related parameters please refer to [1] and [2].

2.2 Buffer Mode Custom Protocol Example

Let's implement a custom protocol to give an idea on how one would implement their own. The following are requirements of our custom protocol:

Table 3

Requirements	Details
baudrate	115,200 bps
physical layer	RS-232
UART frame bits	1 stop bit, 8 data bits, no parity
bytes per frame	7
maximum transmit delay	20 ms, this is the maximum acceptable delay upon the MCE slave receiving a data frame.
Commands	Must support at least 3 commands: Start motor/Set Speed, stop motor, Get status
frame checking	Must perform checksum of each frame

2.2.1 Initializing Buffer Mode

We first need to initialize our Script Engine settings and UART driver, then configure the Buffer mode to meet our requirements. In the case of the script for the MCEWizard/MCEDesigner, the script version, the script start command, the execution steps and period for Task1 are initialized in Code Listing 1 lines 003 – 012. In the case of the script for the iSD, these are set in the Property Window as shown in Figure 4. To meet our maximum transmit delay, we set the execution period of Task1 to 20 ms. This ensures we do not miss a data frame within a 20 ms interval. For more information regarding Script settings please refer to [3].

Using `UART_DriverInit()` we set the baudrate to 115,200 bps, set 1 stop bit, set 8 data bits, set no parity, set the UART channel to UART 1, and disable logic inversion of the UART signals. After this, we call `UART_BufferInit()` to set a few important settings with respect to our protocol:

- We have a max transmit delay of 20 ms but no minimum transmit delay. To ensure we meet this requirement we set all delays (txDelay, txByteDelay) to zero.
- RxTimeout is the time between receiving the first and last byte of a receive frame. If our baudrate is 115,200 bps we expect to receive our entire frame of 7 bytes within 1 ms. To give some room for error we set our RxTimeout to 3 ms.
- We set txDataLength and rxDataLength to 6 to meet our 7 byte per frame requirement. Buffer Mode automatically inserts an additional byte at the beginning of a frame for signifying the start of a receive, and transmit data frame. This beginning byte is specified by the rxFlag and txFlag respectively.

The following Code Listing 1 shows the initialization code for the MCEWizard/MCEDesigner which initialized the UART driver and Buffer handler based on our protocol requirements.

Code Listing 1 Driver and Buffer Initialization for MCEWizard/MCEDesigner

```
001    /*****Script Settings****************************************/
002    /*Script version value should be 255.255*/
003    #SET SCRIPT_USER_VERSION (1.02)
004    /*Script execution time for Task1 in 10mS, maximum value
005    65535*/
006    #SET SCRIPT_TASK1_EXECUTION_PERIOD (2)
007    /* Start command, Task0: Bit0, Task1: Bit1; if bit is set,
008    script executes after init */
009    #SET SCRIPT_START_COMMAND (0x3)
010    /* Script Task1 step, this defines the number of lines to be
011    executed every 10mS*/
012    #SET SCRIPT_TASK1_EXECUTION_STEP (200)
013    /***********************************************************/
014
015    const int RX_FLAG_BYTE = 0xA5;
016    const int TX_FLAG_BYTE = 0x5A;
017    const int ER_CODE_BYTE = 0xEE;
018    const int LOW_BYTE_MASK = 0xFF;
019    const int MCE_CMD_MOTOR_STOP = 0;
020    const int MCE_CMD_MOTOR_START = 1;
021    const int MCE_CMD_PFC_STOP = 0;
022    const int MCE_CMD_PFC_START = 1;
023
024    Script_Task1_init()
025    {
026      /* Driver initialization */
027      UART_DriverInit(
028                      1,      /* channel */
```

Code Listing 1 Driver and Buffer Initialization for MCEWizard/MCEDesigner

```
                          0,       /* rxInvert */
                          0,       /* txInvert */
                          115200,  /* baudrate */
                          8,       /* dataBits */
                          0,       /* parity */
                          1        /* stopBits */
                          );

          /* Buffer initialization */
          UART_BufferInit(
                          0,             /* halfDuplex */
                          3,             /* rxTimeout */
                          0,             /* txDelay */
                          0,             /* txByteDelay */
                          RX_FLAG_BYTE,  /* rxFlag */
                          TX_FLAG_BYTE,  /* txFlag */
                          6,             /* rxDataLength */
                          6              /* txDataLength */
                          );
        }
```

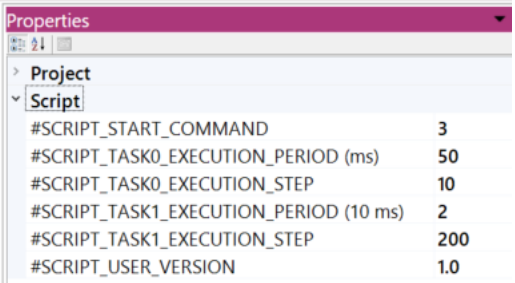

Figure 4 Execution period and step of Task1 for iSD

The following Code Listing 2 shows the initialization code for the iSD which initialized the UART driver and Buffer handler based on our protocol requirements.

Code Listing 2 Driver and Buffer Initialization for iSD (Script_Task1.mcs)

```
/* Task1 init function */
Script_Task1_init()
{
  UART_DriverDeinit();

  /* Driver initialization */
  UART_DriverInit(
                  1,       /* channel */
                  0,       /* rxInvert */
                  0,       /* txInvert */
                  115200,  /* baudrate */
                  8,       /* dataBits */
                  0,       /* parity */
                  1        /* stopBits */
                  );

  /* Buffer initialization */
```

Code Listing 2 Driver and Buffer Initialization for iSD (Script_Task1.mcs)

```
        UART_BufferInit(
                        0,              /* halfDuplex */
                        3,              /* rxTimeout(ms) */
                        0,              /* txDelay(ms) */
                        0,              /* txByteDelay(ms) */
                        RX_FLAG_BYTE,   /* rxFlag */
                        TX_FLAG_BYTE,   /* txFlag */
                        6,              /* rxDataLength */
                        6               /* txDataLength */
                        );
      }
```

2.2.2 Receive Frame Structure

Next is the need to construct a receive frame that meets our requirements. Figure 5 is an example of a receive data frame that meets our basic requirements along with some null data to pad the rest of the frame.

Master ⟶ Slave	0xA5	Process Command (1 byte)	TargetSpeed (2 byte)	0x00	0x00	Checksum (1 byte)

Figure 5 Receive frame example

Table 4 specifies the details of the receive data frame structure. The master is responsible for sending a data frame in this format to the MCE slave.

Table 4 Receive frame structure details

Byte number	Name	Description
1	rxFlag	The first byte signifying the beginning of a receive data frame, specified in `UART_BufferInit()`.
2	Process Command	This byte specifies which command is to be executed by the MCE slave. 1: Start motor, set speed 2: Stop motor 3: Get status information
3,4	TargetSpeed	Two bytes, in little endian ordering, that specify the TargetSpeed of the motor.
5,6	Null data	These bytes are filled with zeros to pad the rest of the frame.
7	Checksum	This byte is the checksum value for bytes 1-6. Checksum = -1*(byte1+byte2+…byte6)

2.2.3 Transmit Frame Structure

After deciding what data is going to be received from a master we need to construct a frame to transmit back to the master. Figure 6 is an example of a transmit data frame that contains all of the information needed to meet our protocol.

Slave ⟶ Master	0x5A	TargetSpeed (2 byte)	Command (1 byte)	PFC Command (1 byte)	0x00	Checksum (1 byte)

Figure 6 Transmit frame example

Table 5 specifies the details of the transmit data frame structure. The MCE slave will send this receive frame in response to a correct command from the Master.

Table 5 Transmit frame structure details

Byte number	Name	Description
1	txFlag	The first byte signifying the beginning of a transmit data frame, specified in `UART_BufferInit()`.
2, 3	TargetSpeed	Two bytes in little endian ordering, that specify the TargetSpeed of the motor.
4	Command	This byte specifies whether the motor is in a stop or start state. 1: Start 0: Stop
5	PFC_Command	This byte specifies whether the PFC is in a stop or start state. 1: Start 0: Stop
6	Null data	This byte is filled with zeros to pad the rest of the frame.
7	Checksum	This byte is the checksum value for bytes 1-6. Checksum = -1*(byte1+byte2+…byte6)

2.2.4 Error Frame

We need to construct an error frame when an invalid checksum is received by the MCE slave. Figure 7 is an example of an error frame that is sent when an invalid checksum is received.

Slave ⟶ Master	0x5A	Checksum (1 byte)	0xEE	0xEE	Checksum (1 byte)	0x00	0x00

Figure 7 Error frame example

Table 6 specifies the details of the error frame structure. The MCE slave will send this frame in response if the Master sends a data frame with an incorrect checksum.

Table 6 Error frame structure details

Byte number	Name	Description
1	txFlag	The first byte signifying the beginning of a transmit data frame, specified in `UART_BufferInit()`.
2	Checksum	The correctly calculated checksum from the last received data frame.
3,4	Constants	Two-byte constants placed in the frame to signify an error.
5	Checksum	The correctly calculated checksum from the last received data frame.
6,7	Null data	Null data to pad the frame.

2.2.5 Protocol Implementation using Buffer Mode

In `Script_Task1()`, using `UART_GetStatus()`, we poll for the `isRxBufferFull` bit. Polling for this bit lets us know that we have received one frame that has filled the size of the Buffer.

Next, we calculate the checksum and compare it against the checksum from the received data frame. If it's correct, we execute one of the commands based on the Command byte. If the checksum is not correct, we send an error frame with the correct checksum.

Finally, we insert bytes into our transmit data frame using `UART_TxBuffer()`, while specifying an index for each byte. Once our entire transmit data frame has been constructed we can initiate a transmission by calling `UART_Control()` and setting the `SendTxBuffer` bit.

The following Code Listing 3 shows the Buffer Mode Code Implementation for the MCEWizard/MCEDesigner.

Code Listing 3 Buffer Mode Code Implementation for MCEWizard/MCEDesigner

```
/*****************************************************************/
/* Task1 function */
Script_Task1()
{
  const int PROC_CMD_MOTOR_START = 1;
  const int PROC_CMD_MOTOR_STOP = 2;
  const int PROC_CMD_GET_STATUS = 3;
  const int UART_SendTxBuffer = 0x0400;
  const int UART_ClrRxBufferFlag = 0x0100;
  const int UART_IsRxBufferFull = 0x0100;

  int checksum_rx;
  int checksum_tx;
  int uart_status;

  /* Get Config UART Status */
  uart_status = UART_GetStatus();

  /* UART_IsRxBufferFull */
  if(uart_status & UART_IsRxBufferFull)
    {
      /* Receive Buffer frame */
      /* UART_RxBuffer(0): Process Command */
      /* UART_RxBuffer(1): Target Speed Lower Byte */
```

Code Listing 3 Buffer Mode Code Implementation for MCEWizard/MCEDesigner

```
        /* UART_RxBuffer(2): Target Speed Upper Byte */
        /* UART_RxBuffer(3): 0x00 */
        /* UART_RxBuffer(4): 0x00 */
        /* UART_RxBuffer(5): Checksum */

          checksum_rx = (
                        -(RX_FLAG_BYTE + UART_RxBuffer(0) + UART_RxBuffer(1) + UART_RxBuffer(2))
                        & LOW_BYTE_MASK;

        if(checksum_rx == UART_RxBuffer(5))
          {
            /* Set Speed, Start motor, Start PFC */
            if(UART_RxBuffer(0) == PROC_CMD_MOTOR_START)
              {
                TargetSpeed = (UART_RxBuffer(1) | (UART_RxBuffer(2) << 8));
                PFC_Command = MCE_CMD_PFC_START;
                Command = MCE_CMD_MOTOR_START;
                checksum_tx = - (TX_FLAG_BYTE + (TargetSpeed >> 8) + (TargetSpeed & LOW_BYTE_MASK) + (Command & LOW_BYTE_MASK) + (PFC_Command & LOW_BYTE_MASK)) & LOW_BYTE_MASK;
                UART_TxBuffer(0, TargetSpeed >> 8);
                UART_TxBuffer(1, TargetSpeed & LOW_BYTE_MASK);
                UART_TxBuffer(2, Command & LOW_BYTE_MASK);
                UART_TxBuffer(3, PFC_Command & LOW_BYTE_MASK);
                UART_TxBuffer(4, 0x00);
                UART_TxBuffer(5, checksum_tx);
              }
            /* Set speed to min speed, Stop motor, stop PFC */
            if(UART_RxBuffer(0) == PROC_CMD_MOTOR_STOP)
              {
                Command = 0;
                PFC_Command = 0;
                TargetSpeed = MinSpd;
                checksum_tx = -(TX_FLAG_BYTE + (TargetSpeed >> 8) + (TargetSpeed & LOW_BYTE_MASK) + (Command & LOW_BYTE_MASK) + (PFC_Command & LOW_BYTE_MASK)) & LOW_BYTE_MASK;

                UART_TxBuffer(0, TargetSpeed >> 8);
                UART_TxBuffer(1, TargetSpeed & LOW_BYTE_MASK);
                UART_TxBuffer(2, Command & LOW_BYTE_MASK);
                UART_TxBuffer(3, PFC_Command & LOW_BYTE_MASK);
                UART_TxBuffer(4, 0x00);
                UART_TxBuffer(5, checksum_tx);
              }
            /* Get status information */
            if(UART_RxBuffer(0) == PROC_CMD_GET_STATUS)
              {
                checksum_tx = -(TX_FLAG_BYTE + (TargetSpeed >> 8) + (TargetSpeed & LOW_BYTE_MASK) + (Command & LOW_BYTE_MASK) + (PFC_Command & LOW_BYTE_MASK)) & LOW_BYTE_MASK;
                UART_TxBuffer(0, TargetSpeed >> 8);
                UART_TxBuffer(1, TargetSpeed & LOW_BYTE_MASK);
                UART_TxBuffer(2, Command & LOW_BYTE_MASK);
                UART_TxBuffer(3, PFC_Command & LOW_BYTE_MASK);
                UART_TxBuffer(4, 0x00);
                UART_TxBuffer(5, checksum_tx);
              }
```

Code Listing 3 Buffer Mode Code Implementation for MCEWizard/MCEDesigner

```
        }
      /* incorrect checksum received, send correct checksum */
      else
        {
          UART_TxBuffer(0, checksum_rx);
          UART_TxBuffer(1, ER_CODE_BYTE);
          UART_TxBuffer(2, ER_CODE_BYTE);
          UART_TxBuffer(3, checksum_rx);
          UART_TxBuffer(4, 0x00);
          UART_TxBuffer(5, 0x00);
        }
      /* UART_SendTxBuffer | UART_ClrRxBufferFlag */
      UART_Control(UART_SendTxBuffer | UART_ClrRxBufferFlag);
    }
}
```

The following Code Listing 4 and Code Listing 5 shows the Buffer Mode Code Implementation for the iSD.

Code Listing 4 Buffer Mode Code Implementation for iSD (Global.mcs)

```
/************************************************************/
/*Global variables*/
/************************************************************/
const int RX_FLAG_BYTE = 0xA5;
const int TX_FLAG_BYTE = 0x5A;
const int ER_CODE_BYTE = 0xEE;
const int LOW_BYTE_MASK = 0xFF;
const int MCE_CMD_MOTOR_STOP = 0;
const int MCE_CMD_MOTOR_START = 1;
const int MCE_CMD_PFC_STOP = 0;
const int MCE_CMD_PFC_START = 1;
```

Code Listing 5 Buffer Mode Code Implementation for iSD (Script_Task1.mcs)

```
/************************************************************/
/* Task1 function */
Script_Task1()
{
  const int PROC_CMD_MOTOR_START = 1;
  const int PROC_CMD_MOTOR_STOP = 2;
  const int PROC_CMD_GET_STATUS = 3;
  const int UART_SendTxBuffer = 0x0400;
  const int UART_ClrRxBufferFlag = 0x0100;
  const int UART_IsRxBufferFull = 0x0100;

  int checksum_rx;
  int checksum_tx;
  int uart_status;

  /* Get Config UART Status */
  uart_status = UART_GetStatus();

  /* UART_IsRxBufferFull */
```

Code Listing 5 Buffer Mode Code Implementation for iSD (Script_Task1.mcs)

```
    if(uart_status & UART_IsRxBufferFull)
      {
        /* Receive Buffer frame */
        /* UART_RxBuffer(0): Process Command */
        /* UART_RxBuffer(1): Target Speed Lower Byte */
        /* UART_RxBuffer(2): Target Speed Upper Byte */
        /* UART_RxBuffer(3): 0x00 */
        /* UART_RxBuffer(4): 0x00 */
        /* UART_RxBuffer(5): Checksum */

        checksum_rx = (-(
                         RX_FLAG_BYTE
                         + UART_RxBuffer(0)
                         + UART_RxBuffer(1)
                         + UART_RxBuffer(2)
                         + UART_RxBuffer(3)
                         + UART_RxBuffer(4)
                         )
                       & LOW_BYTE_MASK);

        if(checksum_rx == UART_RxBuffer(5))
          {
            /* Set Speed, Start motor, Start PFC */
            if(UART_RxBuffer(0) == PROC_CMD_MOTOR_START)
              {
                APP_MOTOR0.TargetSpeed = (UART_RxBuffer(1) |
                                          (UART_RxBuffer(2) <<
8));
                APP_MOTOR0.Command = MCE_CMD_MOTOR_START;
                APP_PFC.Command = MCE_CMD_PFC_START;
                checksum_tx = (-(
                                 TX_FLAG_BYTE
                                 + (APP_MOTOR0.TargetSpeed >> 8)
                                 + (APP_MOTOR0.TargetSpeed &
LOW_BYTE_MASK)
                                 + (APP_MOTOR0.Command &
LOW_BYTE_MASK)
                                 + (APP_PFC.Command &
LOW_BYTE_MASK)
                                 )
                               & LOW_BYTE_MASK);
                UART_TxBuffer(0, APP_MOTOR0.TargetSpeed >> 8);
                UART_TxBuffer(1, APP_MOTOR0.TargetSpeed &
LOW_BYTE_MASK);
                UART_TxBuffer(2, APP_MOTOR0.Command &
LOW_BYTE_MASK);
                UART_TxBuffer(3, APP_PFC.Command &
LOW_BYTE_MASK);
                UART_TxBuffer(4, 0x00);
                UART_TxBuffer(5, checksum_tx);
              }
            /* Set speed to min speed, Stop motor, stop PFC */
            if(UART_RxBuffer(0) == PROC_CMD_MOTOR_STOP)
              {
```

Code Listing 5 Buffer Mode Code Implementation for iSD (Script_Task1.mcs)

```
                        APP_MOTOR0.Command = MCE_CMD_MOTOR_STOP;
                        APP_PFC.Command = MCE_CMD_PFC_STOP;
                        APP_MOTOR0.TargetSpeed = APP_MOTOR0.MinSpd;
                        checksum_tx = (-(
                                         TX_FLAG_BYTE
                                         + (APP_MOTOR0.TargetSpeed >> 8)
                                         + (APP_MOTOR0.TargetSpeed & LOW_BYTE_MASK)
                                         + (APP_MOTOR0.Command & LOW_BYTE_MASK)
                                         + (APP_PFC.Command & LOW_BYTE_MASK)
                                         )
                                       & LOW_BYTE_MASK);
                        UART_TxBuffer(0, APP_MOTOR0.TargetSpeed >> 8);
                        UART_TxBuffer(1, APP_MOTOR0.TargetSpeed & LOW_BYTE_MASK);
                        UART_TxBuffer(2, APP_MOTOR0.Command & LOW_BYTE_MASK);
                        UART_TxBuffer(3, APP_PFC.Command & LOW_BYTE_MASK);
                        UART_TxBuffer(4, 0x00);
                        UART_TxBuffer(5, checksum_tx);
                      }
                    /* Get status information */
                    if(UART_RxBuffer(0) == PROC_CMD_GET_STATUS)
                      {
                        checksum_tx = (-(
                                         TX_FLAG_BYTE
                                         + (APP_MOTOR0.TargetSpeed >> 8)
                                         + (APP_MOTOR0.TargetSpeed & LOW_BYTE_MASK)
                                         + (APP_MOTOR0.Command & LOW_BYTE_MASK)
                                         + (APP_PFC.Command & LOW_BYTE_MASK)
                                         )
                                       & LOW_BYTE_MASK);
                        UART_TxBuffer(0, APP_MOTOR0.TargetSpeed >> 8);
                        UART_TxBuffer(1, APP_MOTOR0.TargetSpeed & LOW_BYTE_MASK);
                        UART_TxBuffer(2, APP_MOTOR0.Command & LOW_BYTE_MASK);
                        UART_TxBuffer(3, APP_PFC.Command & LOW_BYTE_MASK);
                        UART_TxBuffer(4, 0x00);
                        UART_TxBuffer(5, checksum_tx);
                      }
                  }
                /* incorrect checksum received, send correct checksum */
                else
                  {
                    UART_TxBuffer(0, checksum_rx);
                    UART_TxBuffer(1, ER_CODE_BYTE);
```

Code Listing 5 Buffer Mode Code Implementation for iSD (Script_Task1.mcs)

```
                    UART_TxBuffer(2, ER_CODE_BYTE);
                    UART_TxBuffer(3, checksum_rx);
                    UART_TxBuffer(4, 0x00);
                    UART_TxBuffer(5, 0x00);
                  }
                /* UART_SendTxBuffer | UART_ClrRxBufferFlag */
                UART_Control(UART_SendTxBuffer | UART_ClrRxBufferFlag);
              }
          }
```

2.2.6 Performance Evaluation

Given a 10 kHz inverter frequency and 33 kHz PFC frequency, while both motor and PFC are running, the following metrics were taken:

- When a data frame is received, the script takes less than 1 ms to run and begin transmission of a data frame.
- Sending data frames at an interval of 150 ms for an extended period of time, the MCE slave was able to respond correctly, within 20 ms, and with no issues.
- The CPU load average was 69 % and CPU load peak was 72 %.
  - Script_Task1 does not affect CPU load as it gets what is left of the CPU bandwidth.
- It required 3 RAM variables, and 14 constants (although one could do without the constants).

3 FIFO Mode

3.1 FIFO Mode Description

FIFO Mode is another configuration of the Configurable UART. It is a simple protocol based on the first-in-first-out principle that ensures that the sequence of transferred data words is respected. Unlike Buffer Mode, the FIFO Mode does not have a state machine but rather, it is a simple firmware wrapper around the FIFO hardware. Any received data is captured by the hardware buffer and can be retrieved on a first-in-first-out basis. Any data that is loaded in the transmit FIFO begins transmission immediately. Therefore, FIFO Mode supports variable number of data bytes in a frame. Figure 8 is a diagram that represents the flow of data in FIFO Mode, and what layers are responsible for handling portions of the data flow.

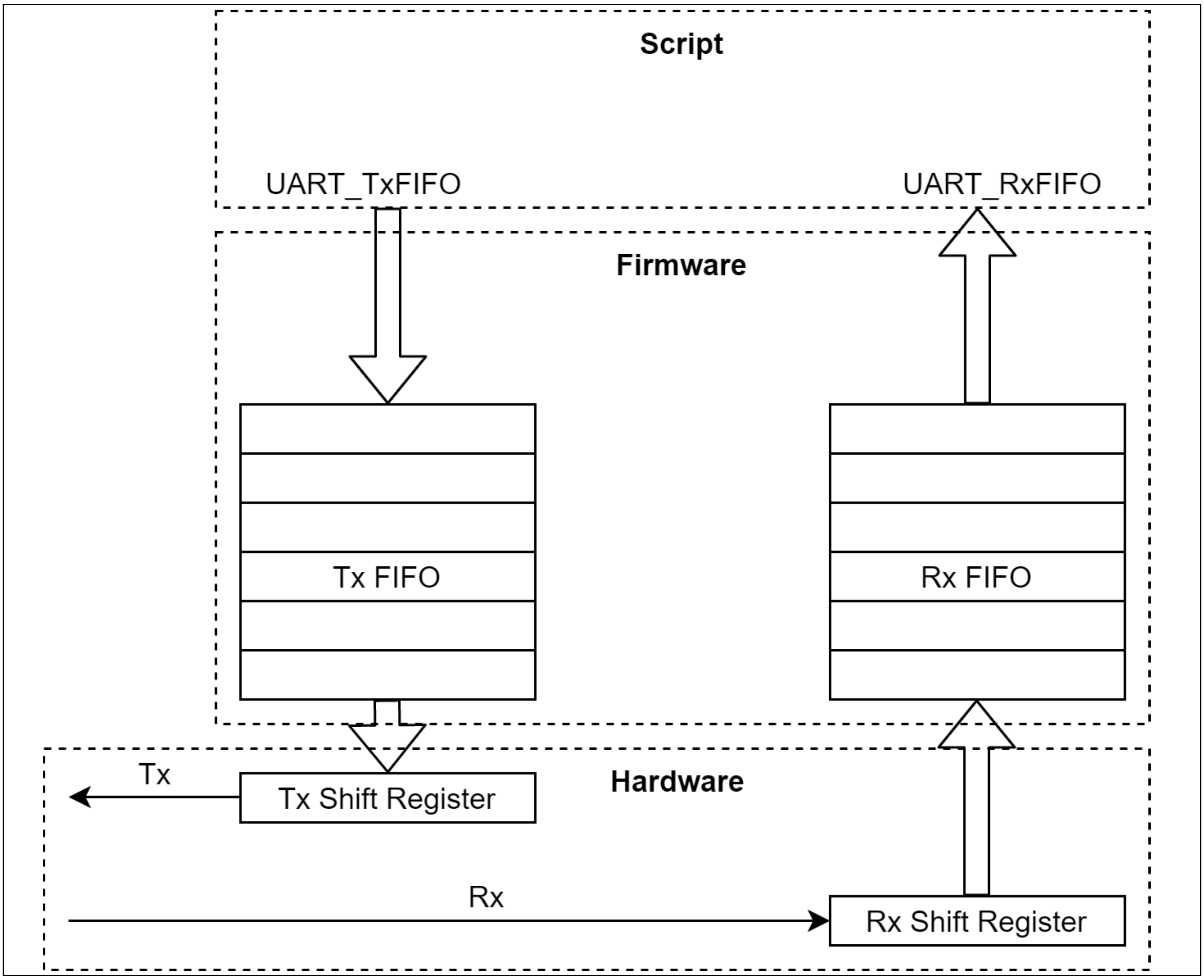

Figure 8 FIFO Mode Diagram

The advantage of FIFO Mode is that it has much more flexibility and doesn't have as much associated underlying firmware overhead. The disadvantage is that it is not as simple to use as Buffer Mode. With FIFO mode, data can only be sent and received on a first-in-first-out basis and the timing requirement associated with the network layer must be implemented using scripting by the user.

3.2 FIFO Mode Custom Protocol Example

Let's implement the same protocol as described in section 2.2.2.

3.2.1 Initializing FIFO Mode

Nothing about the UART driver initialization or the Script settings needs to change. All we need to do is initialize our FIFO by setting the size of our receive and transmit data frames respectively. We do this by setting the rxFifoSize and txFifoSize to 7 using `UART_FifoInit()`. This sets the receive and transmit FIFO sizes to 7 bytes each.

The following Code Listing 6 shows the initialization code for the MCEWizard/MCEDesigner.

Code Listing 6 Driver and FIFO Initialization for MCEWizard/MCEDesigner

```
/*****Script Settings****************************************/
/*Script version value should be 255.255*/
#SET SCRIPT_USER_VERSION (1.02)
/*Script execution time for Task1 in 10mS, maximum value
65535*/
#SET SCRIPT_TASK1_EXECUTION_PERIOD (2)
/* Start command, Task0: Bit0, Task1: Bit1; if bit is set,
script executes after init */
#SET SCRIPT_START_COMMAND (0x3)
/* Script Task1 step, this defines the number of lines to be
executed every 10mS*/
#SET SCRIPT_TASK1_EXECUTION_STEP (200)
/**************************************************************/

Script_Task1_init()
{
  UART_DriverInit(
                1,      /* channel */
                0,      /* rxInvert */
                0,      /* txInvert */
                115200, /* baudrate */
                8,      /* dataBits */
                0,      /* parity */
                1       /* stopBits */
                );
  UART_FifoInit(
              7, /* rxFifoSize */
              7  /* txFifoSize */
              );
}
```

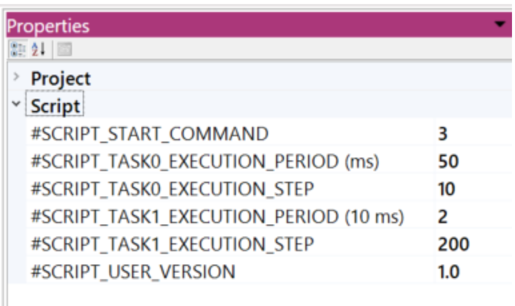

Figure 9 The execution period and step of Task1 for iSD

FIFO Mode

The following Code Listing 7 shows the initialization code for the iSD.

Code Listing 7 Driver and FIFO Initialization for iSD (Script_Task1.mcs)

```
/*Task1 init function*/
Script_Task1_init()
{
  UART_DriverInit(
                1,      /* channel */
                0,      /* rxInvert */
                0,      /* txInvert */
                115200, /* baudrate */
                8,      /* dataBits */
                0,      /* parity */
                1       /* stopBits */
                );
  UART_FifoInit(
                7, /* rxFifoSize */
                7  /* txFifoSize */
                );
}
```

Other than the initialization of FIFO Mode, nothing else about our protocol needs to change. Please refer to 2.2.2, 2.2.3, and 2.2.4 on the structure of the receive, transmit, and error data frames. We can go straight to Script code implementation.

3.2.2 Protocol Implementation using FIFO Mode

In `Script_Task1()`, using `UART_GetStatus()`, we poll for the `isRxFifoFull` bit. Polling for this bit lets us know that we have a received one frame that has filled the size of the FIFO.

Once we have received a data frame we store all the bytes from the frame byte by byte in first-in-first-out order using `UART_RxFifo()`. We then clear the receive FIFO by setting `ClrRxFIFO` bit using `UART_Control()`.

Next, we calculate the checksum and compare it against the checksum from the received data frame. If it's correct, we execute one of the commands based on the Command byte. If it's not, we send an error frame with the correct checksum.

Finally, we send a transmit frame byte by byte using `UART_TxFIFO()`, keeping in mind the first byte in the FIFO is the first byte transmitted over the line.

The following Code Listing 8 shows the FIFO Mode Code Implementation for the MCEWizard/MCEDesigner.

Code Listing 8 FIFO Mode Code Implementation for MCEWizard/MCEDesigner

```
/*Task1 function*/
Script_Task1()
{
  const int START_RX_BYTE = 0xA5;
  const int START_TX_BYTE = 0x5A;
  const int ER_CODE_BYTE = 0xEE;
  const int LOW_BYTE_MASK = 0xFF;
  const int MCE_CMD_MOTOR_STOP = 0;
  const int MCE_CMD_MOTOR_START = 1;
  const int MCE_CMD_PFC_STOP = 0;
```

Code Listing 8 FIFO Mode Code Implementation for MCEWizard/MCEDesigner

```
const int MCE_CMD_PFC_START = 1;
const int PROC_CMD_MOTOR_START = 1;
const int PROC_CMD_MOTOR_STOP = 2;
const int PROC_CMD_GET_STATUS = 3;
const int UART_STATUS_RX_FIFO_FULL = 0x0002;
const int UART_CONTROL_CLEAR_RX_FIFO = 0x0002;

int rx_status;
int rx_start_byte;
int proc_cmd;
int speed_l;
int speed_h;
int checksum_pc;
int checksum_calc;
int tmp;

/* Get Config UART Status */
rx_status = UART_GetStatus();

/* IsRxFIFOFull */
if(rx_status & UART_STATUS_RX_FIFO_FULL)
  {
    rx_start_byte = UART_RxFifo(); /* (1) rx start byte */
    proc_cmd = UART_RxFifo();      /* (2) proc_cmd byte */
    speed_l = UART_RxFifo();       /* (3) speed low byte */
    speed_h = UART_RxFifo();       /* (4) speed high byte */
    tmp = UART_RxFifo();           /* (5) null data */
    tmp = UART_RxFifo();           /* (6) null data */
    checksum_pc = UART_RxFifo();   /* (7) checksum byte */
    UART_Control(UART_CONTROL_CLEAR_RX_FIFO);
    checksum_calc = -(rx_start_byte + proc_cmd + speed_l + speed_h) & LOW_BYTE_MASK;

    if(checksum_pc == checksum_calc)
      {
        /* Set Speed, Start motor, Start PFC */
        if(proc_cmd == UART_CMD_MOTOR_START)
          {
            TargetSpeed = speed_l | (speed_h << 8);
            PFC_Command = MCE_CMD_PFC_START;
            Command = MCE_CMD_MOTOR_START;
            checksum_calc = -(START_TX_BYTE + (TargetSpeed >> 8) + (TargetSpeed & LOW_BYTE_MASK)+ (Command & LOW_BYTE_MASK) + (PFC_Command & LOW_BYTE_MASK)) & LOW_BYTE_MASK;
            UART_TxFifo(START_TX_BYTE);
            UART_TxFifo(TargetSpeed >> 8);
            UART_TxFifo(TargetSpeed & LOW_BYTE_MASK);
            UART_TxFifo(Command & LOW_BYTE_MASK);
            UART_TxFifo(PFC_Command & LOW_BYTE_MASK);
            UART_TxFifo(0x00);
            UART_TxFifo(checksum_calc);
          }
        /* Set speed to min speed, Stop motor, stop PFC */
        if(proc_cmd == UART_CMD_MOTOR_STOP)
          {
            Command = MCE_CMD_MOTOR_STOP;
            PFC_Command = MCE_CMD_PFC_STOP;
            TargetSpeed = MinSpd;
```

Code Listing 8 FIFO Mode Code Implementation for MCEWizard/MCEDesigner

```
              checksum_calc = -(START_TX_BYTE + (TargetSpeed >> 8) + (TargetSpeed & LOW_BYTE_MASK) + (Command & LOW_BYTE_MASK) + (PFC_Command & LOW_BYTE_MASK)) & LOW_BYTE_MASK;
              UART_TxFifo(START_TX_BYTE);
              UART_TxFifo(TargetSpeed >> 8);
              UART_TxFifo(TargetSpeed & LOW_BYTE_MASK);
              UART_TxFifo(Command & LOW_BYTE_MASK);
              UART_TxFifo(PFC_Command & LOW_BYTE_MASK);
              UART_TxFifo(0x00);
              UART_TxFifo(checksum_calc);
            }
          /* Get status information */
          if(proc_cmd == 3)
            {
              checksum_calc = -( START_TX_BYTE + (TargetSpeed >> 8) + (TargetSpeed & LOW_BYTE_MASK) + (Command & LOW_BYTE_MASK) + (PFC_Command & LOW_BYTE_MASK) ) & LOW_BYTE_MASK;
              UART_TxFifo(START_TX_BYTE);
              UART_TxFifo(TargetSpeed >> 8);
              UART_TxFifo(TargetSpeed & LOW_BYTE_MASK);
              UART_TxFifo(Command & LOW_BYTE_MASK);
              UART_TxFifo(PFC_Command & LOW_BYTE_MASK);
              UART_TxFifo(0x00);
              UART_TxFifo(checksum_calc);
            }
        }
      /* incorrect checksum received, send correct checksum */
      else
        {
          UART_TxFifo(START_TX_BYTE);
          UART_TxFifo(checksum_calc);
          UART_TxFifo(ER_CODE_BYTE);
          UART_TxFifo(ER_CODE_BYTE);
          UART_TxFifo(checksum_calc);
          UART_TxFifo(0x00);
          UART_TxFifo(0x00);
        }
    }
}
```

The following Code Listing 9 shows the FIFO Mode Code Implementation for the iSD.

Code Listing 9 FIFO Mode Code Implementation for iSD (Script_Task1.mcs)

```
/*************************************************************/
/*Task1 function*/
Script_Task1()
{
  const int START_RX_BYTE = 0xA5;
  const int START_TX_BYTE = 0x5A;
  const int ER_CODE_BYTE = 0xEE;
  const int LOW_BYTE_MASK = 0xFF;
  const int MCE_CMD_MOTOR_STOP = 0;
  const int MCE_CMD_MOTOR_START = 1;
  const int MCE_CMD_PFC_STOP = 0;
  const int MCE_CMD_PFC_START = 1;
```

Code Listing 9 FIFO Mode Code Implementation for iSD (Script_Task1.mcs)

```
const int UART_CMD_MOTOR_START = 1;
const int UART_CMD_MOTOR_STOP = 2;
const int UART_CMD_GET_STATUS = 3;
const int UART_STATUS_RX_FIFO_FULL = 0x0002;
const int UART_CONTROL_CLEAR_RX_FIFO = 0x0002;

int rx_status;
int rx_start_byte;
int proc_cmd;
int speed_l;
int speed_h;
int checksum_pc;
int checksum_calc;
int tmp;

/* Get Config UART Status */
rx_status = UART_GetStatus();

/* IsRxFIFOFull */
if(rx_status & UART_STATUS_RX_FIFO_FULL)
  {
    rx_start_byte = UART_RxFifo(); /* (1) rx start byte */
    proc_cmd = UART_RxFifo();      /* (2) proc_cmd byte */
    speed_l = UART_RxFifo();       /* (3) speed low byte */
    speed_h = UART_RxFifo();       /* (4) speed high byte */
    tmp = UART_RxFifo();           /* (5) null data */
    tmp = UART_RxFifo();           /* (6) null data */
    checksum_pc = UART_RxFifo();   /* (7) checksum byte */
    UART_Control(UART_CONTROL_CLEAR_RX_FIFO);
    checksum_calc = (-(
                      rx_start_byte
                      + proc_cmd
                      + speed_l
                      + speed_h
                      )
                    & LOW_BYTE_MASK);

    if(checksum_pc == checksum_calc)
      {
        /* Set Speed, Start motor, Start PFC */
        if(proc_cmd == UART_CMD_MOTOR_START)
          {
            APP_MOTOR0.TargetSpeed = (wr_tar_sped_l |
                                      (wr_tar_sped_h << 8));
            APP_MOTOR0.Command = MCE_CMD_MOTOR_START;
            APP_PFC.Command = MCE_CMD_PFC_START;
            checksum_calc = (-(
                              START_TX_BYTE
                              + (APP_MOTOR0.TargetSpeed >>
   8)
                              + (APP_MOTOR0.TargetSpeed &
   LOW_BYTE_MASK)
                              + (APP_MOTOR0.Command &
   LOW_BYTE_MASK)
```

Code Listing 9 FIFO Mode Code Implementation for iSD (Script_Task1.mcs)

```
                                            + (APP_PFC.Command &
   LOW_BYTE_MASK)
                                            )
                                          & LOW_BYTE_MASK);
                       UART_TxFifo(START_TX_BYTE);
                       UART_TxFifo(APP_MOTOR0.TargetSpeed >> 8);
                       UART_TxFifo(APP_MOTOR0.TargetSpeed &
   LOW_BYTE_MASK);
                       UART_TxFifo(APP_MOTOR0.Command & LOW_BYTE_MASK);
                       UART_TxFifo(APP_PFC.Command & LOW_BYTE_MASK);
                       UART_TxFifo(0x00);
                       UART_TxFifo(checksum_calc);
                     }
                   /* Set speed to min speed, Stop motor, stop PFC */
                   if(proc_cmd == UART_CMD_MOTOR_STOP)
                     {
                       APP_MOTOR0.Command = MCE_CMD_MOTOR_STOP;
                       APP_PFC.Command = MCE_CMD_PFC_STOP;
                       APP_MOTOR0.TargetSpeed = APP_MOTOR0.MinSpd;
                       checksum_calc = (-(
                                            START_TX_BYTE
                                            + (APP_MOTOR0.TargetSpeed >>
   8)
                                            + (APP_MOTOR0.TargetSpeed &
   LOW_BYTE_MASK)
                                            + (APP_MOTOR0.Command &
   LOW_BYTE_MASK)
                                            + (APP_PFC.Command &
   LOW_BYTE_MASK)
                                            )
                                          & LOW_BYTE_MASK);
                       UART_TxFifo(START_TX_BYTE);
                       UART_TxFifo(APP_MOTOR0.TargetSpeed >> 8);
                       UART_TxFifo(APP_MOTOR0.TargetSpeed &
   LOW_BYTE_MASK);
                       UART_TxFifo(APP_MOTOR0.Command & LOW_BYTE_MASK);
                       UART_TxFifo(APP_PFC.Command & LOW_BYTE_MASK);
                       UART_TxFifo(0x00);
                       UART_TxFifo(checksum_calc);
                     }
                   /* Get status information */
                   if(proc_cmd == UART_CMD_GET_STATUS)
                     {
                       checksum_calc = (-(
                                            START_TX_BYTE
                                            + (APP_MOTOR0.TargetSpeed >>
   8)
                                            + (APP_MOTOR0.TargetSpeed &
   LOW_BYTE_MASK)
                                            + (APP_MOTOR0.Command &
   LOW_BYTE_MASK)
                                            + (APP_PFC.Command &
   LOW_BYTE_MASK)
                                            )
```

Code Listing 9 FIFO Mode Code Implementation for iSD (Script_Task1.mcs)

```
                                            & LOW_BYTE_MASK);
                        UART_TxFifo(START_TX_BYTE);
                        UART_TxFifo(APP_MOTOR0.TargetSpeed >> 8);
                        UART_TxFifo(APP_MOTOR0.TargetSpeed &
   LOW_BYTE_MASK);
                        UART_TxFifo(APP_MOTOR0.Command & LOW_BYTE_MASK);
                        UART_TxFifo(APP_PFC.Command & LOW_BYTE_MASK);
                        UART_TxFifo(0x00);
                        UART_TxFifo(checksum_calc);
                      }
                  }
                /* incorrect checksum received, send correct checksum */
                else
                  {
                    UART_TxFifo(START_TX_BYTE);
                    UART_TxFifo(checksum_calc);
                    UART_TxFifo(ER_CODE_BYTE);
                    UART_TxFifo(ER_CODE_BYTE);
                    UART_TxFifo(checksum_calc);
                    UART_TxFifo(0x00);
                    UART_TxFifo(0x00);
                  }
              }
          }
```

3.2.3 Performance Evaluation

Given a 10 kHz inverter frequency, and 33 kHz PFC frequency while both motor and PFC are running, the following metrics were taken:

- When a data frame is received, the script takes less than 1 ms to run and transmit a data frame.
- Sending data frames at an interval of 150 ms for an extended period of time, the MCE slave was able to respond correctly, within 20 ms, and with no issues.
- The CPU load average was 69 % and CPU load peak was 72 %.
  - Script_Task1 does not affect CPU load as it gets what is left of the CPU bandwidth.
- The script required 8 RAM variables and 13 constants (although one could do without the constants).
- With class B software enabled CPU load average was 75 % and CPU load peak was 77 %.
  - Average CPU load value would swing from 68 % to 75 %.

4 Guidelines & Limitations

4.1 Buffer Mode vs FIFO Mode

For a given application it may be difficult to decide whether a user should use Buffer or FIFO mode in implementing their custom protocol. The below comparison table should be used to help make this decision.

Table 7 Buffer vs FIFO Mode Comparison

Features	Buffer	FIFO
Maximum frame size supported	9 bytes[3]	31 bytes
Implements part of the network layer?	Yes	No
Random access?[1]	Yes	No
Maximum Baudrate supported	115,200 bps[4]	230,400 bps
Supports half duplex?	Yes	Yes
Additional CPU load[2]	Yes	No

1. *Random access: The ability to select specific bytes in a data structure. FIFO Mode is not a random access data structure, whereas Buffer Mode is.*
2. *Additional CPU load: Incurs because of the underlying firmware associated with each mode. Buffer Mode contains a state machine in firmware, whereas FIFO Mode does not.*
3. *When using the Maximum Frame Size to 9 bytes (8 buffer frame), set the Baudrate to 19,200 bps or less.*
4. *When setting the Maximum Baudrate to 115,200 bps, set the frame size to 7 bytes (6 buffer frame) or less.*

As mentioned in section 2.2.6 and 3.2.3, FIFO mode requires more RAM to be used in the Script code implementation. Because we cannot randomly access the data in the receive FIFO, we have to place the data in a variable and use it later on in the script. Whereas in Buffer Mode a user can randomly access the data from the buffer using `UART_TxBuffer()` API and specifying an index for the byte that is desired.

4.2 Limitations

Please refer to Table 8 and Table 9 for limitations of the Buffer and FIFO modes.

Table 8 Limitations of Buffer Mode

Limitation	Explanation
Maximum frame size	The maximum frame size for Buffer Mode configuration is 8 bytes (not including rxFlag/txFlag).
Maximum baudrate	The maximum baudrate supported by Buffer Mode is 115,200 baud per second.
Class B issue	When Class B safety tests are enabled the Buffer Mode UART eventually enters failsafe mode due to the stack overflow test failing. If one desires customized UART protocol while enabling Class B safety tests, it is advised to use FIFO Mode instead.
Task 0	We do not recommend using Task 0 when also using Buffer mode as this may interfere with the Buffer Mode state machine. If Task 0 must be used, it must only be used for time critical operations.
CPU limit	When CPU limit approaches 90% one may see issues with Buffer Mode as CPU bandwidth becomes scarce.

Table 9 Limitations of FIFO Mode

Limitation	Explanation
Maximum frame length	The maximum frame length supported by FIFO mode is 31 bytes.
Maximum baudrate	The maximum baudrate support by FIFO mode is 230,400 baud per second.
Task 0	Task 0 may affect CPU loading, which in turn could affect one's communication protocol. If Task 0 must be used, it must only be used for time critical operations.
Class B issue	Depending on the complexity of the script, motor frequency, and PFC frequency one may run into issues with Class B software entering failsafe mode. This is due to excessive stack consumption triggering Class B software to enter failsafe mode.

4.3 Guidelines

Here are some general guidelines for determing if the Configurable UART can implement one's desired protocol:

1. The maximum data frame supported by FIFO Mode is 31 bytes and is the maximum data frame supported by the Configurable UART.
2. If an application requires motor, PFC, Class B safety tests, and Configurable UART, then a user must be wary of CPU load issues and issues with script complexity.
   a. Depending on the PFC and inverter frequency one may run into limits of the CPU. The drive may enter failsafe mode if CPU load goes above 95%.
   b. The drive may also enter failsafe mode depending on the complexity of the Script. This is mainly due to constraints on the stack, and how a user writes their Script code may consume more stack than necessary. Please refer to [3] on how to reduce stack consumption in Script code.
3. It is possible to implement half duplex communication but only with Buffer Mode.
   a. Buffer Mode can directly support this through `BufferInit()` API by setting the halfDuplex parameter.
4. The Configurable UART natively supports packet-at-a-time receiving on fixed length packets.
   a. This means that a user has to only poll for when the FIFO or Buffer is full to detect an entire packet.
   b. The Configurable UART only uses fixed length packets to detect an entire frame, whereas other protocols may use different schemes to detect a packet.
5. FIFO Mode supports variable length transmit frame sizes.
6. The Configurable UART does not have any API to signify the completion of a transmission.
7. "UART link break fault" is only applicable to a use case where User UART Protocol is selected. When using Configurable UART function, it is imperative to disable "UART link break fault" option in the "System Protection" section in iSD as shown in Figure 10 to avoid triggering that fault. For more information regarding the "UART link break fault", please refer to [1] and [2].

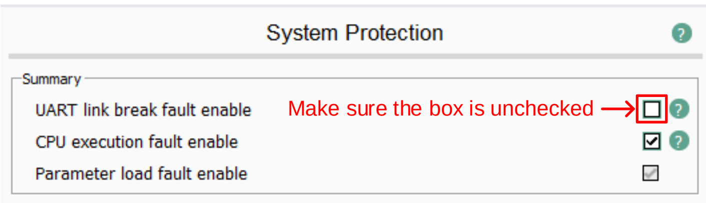

Figure 10 "UART link break fault" in the "System Protection" section in iSD

5 References

[1] iMOTION™ Motor Control Engine Software Reference Manual (REV 1.34).

[2] iMOTION™ Motor Control Engine Functional Reference Manual (REV 1.01).

[3] How to use iMOTION™ script language (REV 1.2).

Revision history

Document version	Date of release	Description of changes
V 1.0	2021-08-11	Initial Release
V 2.0	2023-06-14	Software Reference Manual was updated to the latest version. Added Functional Reference Manual for the reference. Example code for iSD were added in Section 2.2.1, 2.2.5, 3.2.1and 3.2.2. The protocol (The number of the data frame) was changed from 9 to 7. Added the Buffer mode constraint in Section 4.1. Added the precaution statement for the UART link break fault.

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2023-06-14

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Email: erratum@infineon.com

Document reference
AN2021-08