How to use iMOTION™ Configurable I2C Interface

About this document

Scope and purpose

This application note provides an explanation and script examples of how to use the configurable I2C interface provided by iMOTION™ Motion Control Engine (MCE) to communicate with external I2C devices EEPROM eliminating the need for a secondary microcontroller to interpret I2C messages. The I2C driver mentioned in this application note is available when using an SDPack newer than version 5.2.

Intended audience

This application note is intended for customers who want to understand how to use the iMOTION™ configurable I2C driver provided by iMOTION™ Motion Control Engine (MCE) SW package.

Table of contents

1 Introduction

iMOTION™ Motor Control Engine (MCE) has an I2C driver which allows devices to send and receive information and commands using the I2C protocol configured through scripting. Use this application note as a reference for writing code, understanding I2C protocol implementations, and predicting performance impact of using the I2C Interface.

1.1 Overview

I2C, or Inter-Integrated Circuit, is a communication protocol used by controllers found in many sensors, EEPROMs, I/O expanders, Displays, and NFC receivers. It is also used to communicate simple parameters inside VGA and HDMI connectors. This protocol was created in 1982 to improve popular protocols of the time [1]. One popular protocol at the time, UART, only uses two wires but can only be used between two devices. Another I2C precursor, SPI, can have multiple slaves however it can only have one master and uses 3 or 4 wires. I2C combines the benefits of UART and SPI. It uses only two wires and can have multiple slaves. Some other benefits of the protocol are: it can have multiple masters, it is synchronous which means the data rate does not need to be known by the slaves before communication starts, and it has low requirements for computational complexity.

I2C's connections consist of two lines: the serial data line (SDA) and the serial clock line (SCL). No matter how many targets there are, only two lines are required for the entire I2C network. The target is selected by the master by sending its address. Each bit transmitted on the data line must be accompanied by a clock pulse on the clock line. On each line at any given moment, the two possible states are logic high and logic low.

I2C communicates in messages. A message is made up of frames, a start condition, a stop condition, and ACK/NACK bits. Data inside of a frame is sent with the Most Significant Bit (MSB) first. Figure 1 shows each part of an I2C message in the order that they must occur.

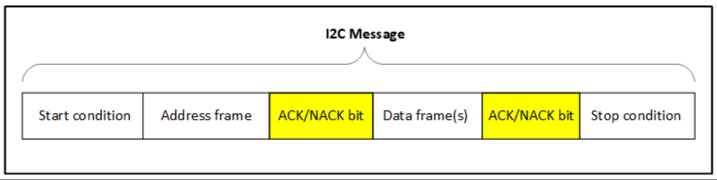

Figure 1 Parts of I2C message

The start and stop conditions are instructions to start and stop communication on the I2C bus and are consistent between all devices. The start condition informs all slave devices that communication is about to begin and makes the I2C bus busy. The stop condition informs all slaves that communication has finished, making the I2C bus free. These two conditions are completed by changing the state of the SDA bus while the SCL bus is high. This differs from other data as normally the SDA bus will change at the same time that the SCL bus is changing from high to low. A start condition consists of the SDA bus going from high to low while the SCL bus is high. The iMOTION™ I2C driver combines the start condition with the address frame (which will be detailed in the latter part of this section) to be a part of the same API, I2C_MasterStart(x). A stop condition consists of the SDA bus going from low to high while the SCL bus is high. A stop condition can be sent using the API I2C_MasterStop(x). See Figure 2 for a visual representation of these two conditions.

There are times when the master needs to send a repeated start condition after communication has already begun. This is done when communication has begun already but it's desired to switch the value of the R/$\overline{W}$ bit. The protocol's allowance of a second start condition while communication is already occurring prevents the need for a stop condition, freeing the I2C bus in the middle of communication. A repeated start condition can be sent using the API I2C_MasterRepeatedStart(x).

Highlighted in yellow, Figure 1 mentions "ACK/NACK" bits in every signal combination, except for a stop condition. These two bits are called "acknowledge" and "not acknowledge" bits. ACK/NACK bits are used after every frame. "Acknowledge" bits can be sent by the master or slave. They indicate that the previous frame was received successfully and that the next frame may be sent. The reason this is highlighted in yellow is to indicate that this bit is being received, not sent. "Not acknowledge" bits can be interpreted in different ways; this either means that there was a problem receiving the frame, or that no more data is desired to be sent. The column on the righthand side of figure 2 demonstrates what the ACK/NACK bits look like and show that two APIs can control the use of them.

The address is where an address select code is sent to all slave devices in order to establish a connection with only one slave. Depending on the slave device, the address frame may consist of either 8 bits, or 11 bits and is only sent once per transmission. The last bit specifies whether the master is about to read or write to the target. This bit is usually referred to as the R/$\overline{W}$ bit as a 1 specifies that a read instruction is about to take place and a 0 specifies a write instruction. iMOTION™'s I2C driver combines the start condition with the address frame to be a part of the same API, I2C_MasterStart(x).

Once the address frame is sent to select the slave device, data frames may follow. Each data frame consists of one byte. The data frame is where the main payload resides. Consult the slave's datasheet, or read on to the use cases in Section 2, to properly format the data inside the data frames. A data frame can be sent using the API I2C_Transmit(x).

For a list of all I2C related APIs with detailed descriptions refer to [2].

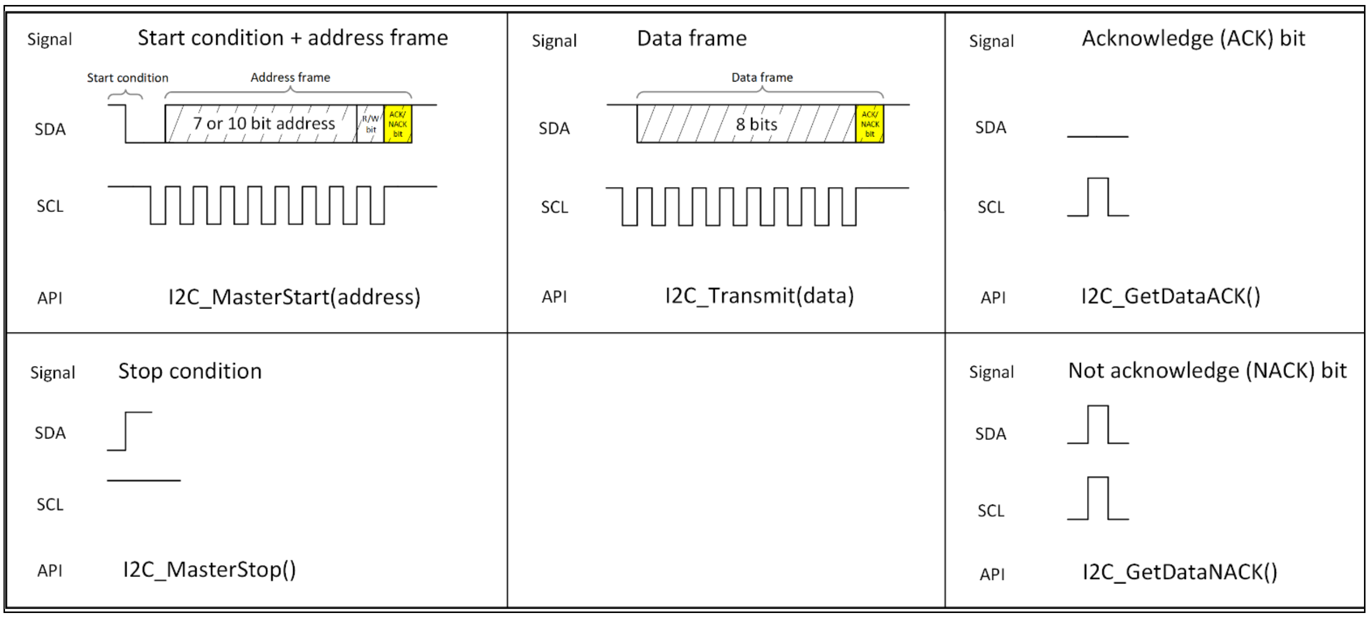

Figure 2 I2C signals

1.2 iMOTION™ protocol implementation

iMOTION™ MCE implements the I2C protocol in firmware and provides a series of API calls enabled by the MCE script engine for users to accommodate I2C communication sequences. The iMOTION™ MCE script engine is used for customizing system level functionalities which runs in parallel with the motor and PFC control algorithm. This script engine is most commonly used to read or modify MCE parameters, to make use of spare analog inputs or digital GPIO resources, to support customized UART protocol implementation, or to support I2C communication. Once the pinout section of your device's datasheet is checked and the proper pins are connected to the SDA/SCL lines, a script can be created to communicate with any I2C devices by making use of I2C APIs.

Writing and debugging script code is supported by iMOTION™ Solution Designer (iSD) script editor. Install iSD online found in the "Design Support" section of our website [4] and then by following the directions found in the "Getting Started with iMOTION™ Solution Designer" guide [5] which will give you a walkthrough of creating a configuration file, a script, and then programming and debugging the device. For a list of all I2C APIs refer to the Functional Reference Manual [2], and for an example of APIs being used to read and write data from an EEPROM, continue to Section 2.

2 Use Case Example: EEPROM

iMOTION™ MCE's I2C interface allows a high level of customization as it transmits one byte of data at a time. An example of EEPROM read / write operation is provided in this section to demonstrate how to customize MCE's I2C interface to support EEPROM communication.

An EEPROM has two types of instructions: read and write, and each of these instructions is made up of 6 and 5 API functions respectively. There are a few reasons why you may find the need to interface with an EEPROM. You may find that some devices such as an NFC interface IC has built-in EEPROM that may store commands. This would allow the ability to change variables or request parameter values wirelessly in the field.

This section explains how to configure the data inside of multiple data frames in order to interact with data from a desired memory address inside an EEPROM that has a unique address select code. The way that interacting with EEPROM differs from other I2C enabled devices mostly has to do with what's inside the data frames. Lastly, EEPROM's slow write time may require a mechanism to be introduced which is described in section 2.1.1.

One fundamental mechanism of EEPROM to understand is an internal counter that keeps track of the memory address being used, which will be referred to as the memory counter. This exists so that multiple addresses can be written to, or read from, with the address being specified only one time. The way it works is that the counter increases by one after an action is done to a memory address. Every write instruction and most read instructions start by setting the memory counter inside of the EEPROM. The most common size of a memory counter is two bytes, which allows addressing up to 65,536 bytes of memory space. The first two data frames in the following Figure 3 and Figure 4 demonstrate the memory counter being set. Check the EEPROM's datasheet to ensure that its memory address is neither larger nor smaller or else adjust the amount of data frames accordingly.

2.1 EEPROM Write Design Flow

After understanding how the I2C communication protocol works, the pieces can now be put together to write data to an EEPROM. This section explains how to write to an EEPROM.

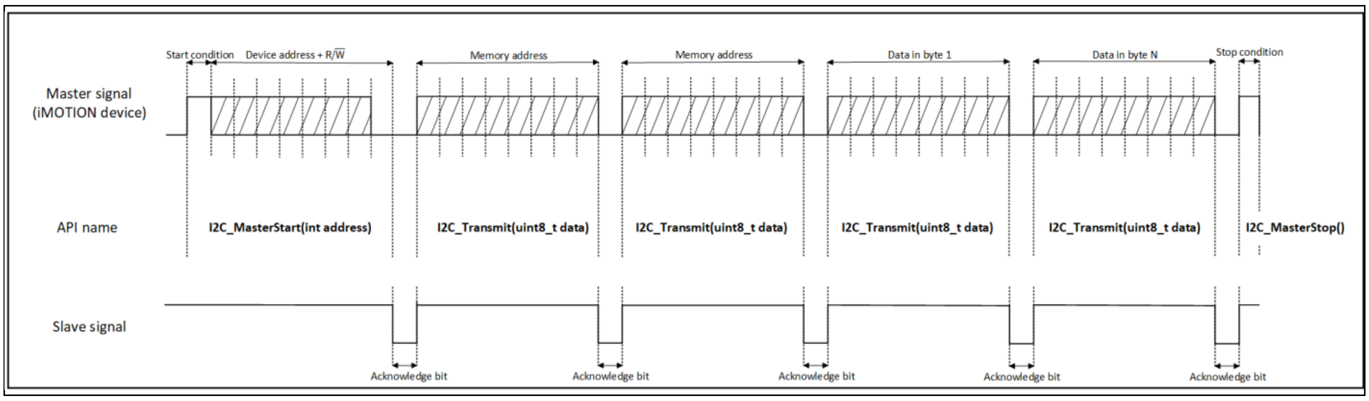

Figure 3 EEPROM write instruction signals

Figure 3 follows the same pattern that every I2C message follows which was previously described in Section 1.1 and illustrated in Figure 1. The only difference is that Figure 3 contains four data frames rather than one, and instead of having the slave signals highlighted in yellow, it is separated. If you took every API in this figure and put it in a script it would write 2 bytes to 2 EEPROM memory addresses. The four data frames are made up of two memory address bytes and two bytes to be stored.

The last bit of the address frame, or the R/$\overline{W}$ bit, is logic LOW, indicating a write instruction, just as described in Section 1.1. This is achieved by making the least significant bit inside the "I2C_MasterStart()" argument equal to 0.

The words "Data in byte N" in Figure 3 demonstrate that you may continually add more data frames to store more data. Please note that there will be a limit to how much data can be stored in one message. The EEPROM's datasheet must be checked for the maximum limit. Look for mention of a "page write" and do not exceed the specification. Read on to Section 2.3.1 for more info about a page write.

In your EEPROM's datasheet also look for how many memory address locations there are, or look for the size of the memory counter. In Figure 1 the size of the memory counter is two bytes. If your EEPROM is small it may only require one byte for the memory address. Or if it is bigger than 65,536 bytes it may require more than two bytes to set the memory counter.

Figure 1 does not include all the APIs for a successful write instruction. I2C_DriverInit() API will also need to be called. The reason this API is not pictured is because it is not part of an instruction. It should be run upon the device starting up. Refer to line 13 of code listing 1 for an example of how it is used and to the Functional Reference Manual [2] for more information on using this function and its arguments.

Keeping in mind factors that can vary, this section gives you all the building blocks needed to build a write instruction from scratch.

2.1.1 Pages and byte writes

Be cautious when writing multiple bytes that an EEPROM write cycle does not occur in the middle of a message. EEPROM write cycles commonly take 5ms [3] which is substantial compared to the 20us it takes to send one data frame in fast mode. This mistake can easily be made if using multiple byte writes when a page write is intended. Pages are a type of EEPROM architecture which are a subdivision of the memory. The reason manufacturers have these smaller sections of memory is to allow multiple bytes to be written in one single write cycle. However, EEPROM with this architecture allows for a different kind of write operation called a byte write. Users may prefer multiple byte writes over a page write if they want to protect a section of a page from constantly being written to. The drawback of a byte write is that a write cycle occurs after every single byte, introducing a large 5ms delay.

If only page writes are desired and you want to avoid accidentally sending byte writes, read your EEPROM's pinout description and ensure there is no dedicated pin, such as a $\overline{WC}$ pin, that determines whether a page write or byte write should occur [3]. If this pin is logic LOW, multiple write cycles can occur in a single message. If the pin is logic HIGH or if there is no $\overline{WC}$ pin then there will only be one write cycle per message.

When using a page write you must understand the address that the page will end; if you continue writing past the end of a page a "roll-over" occurs, i.e. the bytes exceeding the page end are written on the same page, from location 0 [3]. If you wish to write to two pages you must send two write instructions. If page one ends at address 63 and you wish to write to the next page, a stop instruction should always occur after writing to address 63, followed by a new message sets the address to 64.

If you are writing multiple bytes and intend for byte writes instead of a single page write you must anticipate a delay before the EEPROM is ready to receive more data. Code listing 3 shows how to introduce a delay in between each instruction using a parameter called RunTimeCounter which increases by one every millisecond. This code writes data to five different memory addresses, waiting 5ms in between every data frame. It does this after the state of a GPIO changes. After the main bulk of code is run once, the EEPROM's addresses 0 through 4 should have the value of 2.

Following the code, the I2C driver is initialized using the "I2C_DriverInit()" API. For the duration that an iMOTION™ device is powered on, driver initialization is only needed once and is therefore located in the Script_Task1_init() loop. The API arguments specify that the address is 7 bits and the data rate should be 400kHz. Inside the Script_Task1 loop on line 12, this code waits for the state of a GPIO input to change and checks that 5ms have passed since the last write cycle before proceeding. It does this by subtracting the current RunTimeCounter value by the time that was stored when the last write cycle started. It then sends a start condition followed by an address frame that contains "A0". As explained in Section 1.1, write instructions always start with the value of 0 for the least significant bit (R/$\overline{W}$ bit). Next, the memory counter is set to 0 on lines 17-18, followed by 5 data frames nested inside of 5 if statements. After each data frame, line 50-51 are run which stores the value of the current time in the variable "startTime" and increases the value of "loop" by one.

Code listing 1 Sending one message with 5 byte writes

```
001        #SET SCRIPT_TASK1_EXECUTION_PERIOD(5)
002        #SET SCRIPT_TASK1_EXECUTION_STEP(50)
003        Script_Task1_init()
004        {
005              int loop;
006              int startTime;
007              int finished;
008              I2C_DriverInit(1,0);
009        }
```

```
Script_Task1()
{
    if(finished == 0 & FB_GPIO.GPIO_Status.GPIO10 == 1 &
MCEOS.RunTimeCounter - startTime > 4)
    {
        if(loop == 0)
        {
            I2C_MasterStart(0xA0);    //device select
code
            I2C_Transmit(0x00);       //memory address
            I2C_Transmit(0x00);       //memory address
            I2C_Transmit(2);//write to memory
        }
        else
        {
            if(loop == 1)
            {
                I2C_Transmit(2);//write to memory
            }
            else
            {
                if(loop == 2)
                {
                    I2C_Transmit(2);//write to
memory
                }
                else
                {
                    if(loop == 3)
                    {
                        I2C_Transmit(2);//write
                    }
                    else
                    {
                        if(loop == 4)
                        {
I2C_Transmit(2);//write
                            finished = 1;
                        }
                    }
                }
            }
        }
        startTime = MCEOS.RunTimeCounter;
        loop = loop + 1;
    }
}
```

2.1.2 Script Implementation

Code listing 1 shows all the building blocks for a write instruction put together into code that can be run inside of an iMOTION device. The script uses I2C to write the values of 3 MCE parameters into the first 5 addresses of an EEPROM once there's a voltage on a GPIO.

The script initializes the I2C driver inside of the "Script_Task1_init()" function, on line 11. Section 2.6.3 of the Functional Reference Manual [2]states that the initialization functions are called once during start-up. The "I2C_DriverInit()" API is called within this function because the driver only needs to be initialized during start-up. The arguments inside of the function specify that the address size is 7 bits. Note that the "I2C_MasterStart()" API requires 8 bits as the argument; 7 bits are considered the address and the 8th bit is the R/$\overline{W}$ bit which is was defined above in section 1.1. The next two lines of code after the "I2C_MasterStart()" are lines 21 and 22 which are two data frames specifying the memory address. In this case the memory address is set to 0. The next five lines of code (23-27) are where the values of three MCE parameters are sent to be stored in the EEPROM. The order of these five lines of code mean that the lower byte of the MinSpd parameter will be stored in address 0. The upper byte in address 1. The lower byte of TargetSpeed in address 2. The upper byte in 3. And the entirety of PwmDeadtimeR in address 4.

One useful practice is to use an OR operator inside of the "start" API to easily see whether the EEPROM is being written to or read from. Code listing 1 below shows an example of this on line 20. Another good practice on lines 23 and 24 of code listing 1 ensures that the argument of the transmit function is only one byte. On line 24, the lower byte of the memory address variable is sent to the EEPROM. An AND operator and bit-shifting ensure that only one byte of a 2-byte variable is sent at a time. The same method of transmitting a variable over two API functions is employed again in the lines immediately after, 25 and 26 of code listing 1 on an MCE parameter. There are a small handful MCE parameters that are only one byte in size, however. Line 27 of Code listing 1 shows that no operator nor bit-shifting is required to write the value of "PwmDeadtimeR" because it is only one byte. For information on parameter size, refer to the Parameter Reference Manual.

Code listing 2 Write example

```
001         #SET SCRIPT_TASK1_EXECUTION_PERIOD(1)
002         #SET SCRIPT_TASK1_EXECUTION_STEP(50)
003         /*Global variables*/
004         const int deviceSelect = 0xA0;
005         const int I2C_READ_CMD = 1;
006         const int I2C_WRITE_CMD = 0;
007         int writeComplete;
008
009         /*************************************************************
   *******/
010         /*Task0 init function*/
011         Script_Task1_init()
012         {
013                 I2C_DriverInit(1, 0);
014         }
015
016         /*************************************************************
   *******/
017         /*Task0 script function*/
018         Script_Task1()
019         {
020                 if (FB_GPIO.GPIO_Status.GPIO10 == 1 & writeComplete ==
   0)
021                 {
```

```
                I2C_MasterStart(deviceSelect|I2C_WRITE_CMD);
                I2C_Transmit(0x00);
                I2C_Transmit(0x00);
                I2C_Transmit(APP_MOTOR0.MinSpd >> 8);
                I2C_Transmit(APP_MOTOR0.MinSpd & 0xFF);
                I2C_Transmit(APP_MOTOR0.TargetSpeed >> 8);
                I2C_Transmit(APP_MOTOR0.TargetSpeed & 0xFF);
                I2C_Transmit(APP_MOTOR0.PwmDeadtimeR);
                I2C_MasterStop();
                writeComplete = 1;
        }
}
```

2.1.3 Test Result

In order to find out how long an I2C message will take just take the inverse of the data rate (100kHz for normal mode or 400kHz for fast mode) and multiply it by the amount of bits that are within the message. Code listing 2 in the previous section contains 8 bytes that are being sent as there are eight one-byte values in the arguments of the APIs. Along with the eight bytes there are eight ACK/NACK bits, one after every byte, there is one start bit, and one stop bit. Using the equation: $\frac{1}{400\text{kHz}} * \left(8 \text{ bytes} * \frac{8 \text{ bits}}{\text{byte}} + 8 \text{ ACK/NACK bits} + 1 \text{ start bit} + 1 \text{ stop bit}\right) = 185\mu\text{s}$ and the equation: $\frac{1}{100\text{kHz}} * \left(8 \text{ bytes} * \frac{8 \text{ bits}}{\text{byte}} + 8 \text{ ACK/NACK bits} + 1 \text{ start bit} + 1 \text{ stop bit}\right) = 740\mu\text{s}$ we can determine that the I2C message will take 740µs in normal mode and 185µs fast mode. This data is useful for those who have other operations going on in the script that are time sensitive.

The script in code listing 2 was tested while a sensorless motor control function was running and it was observed that the CPU load increases by 3.1% while the I2C driver writes.

2.2 EEPROM Read Design Flow

Follow along to learn how to use the data in any EEPROM that uses I2C within a script. Figure 4 shows an example of an EEPROM read instruction. If the four APIs in figure 4 were put into a script, it would return two bytes stored at an address and, as described in Section 1.1, the address used would be determined by the value of the memory counter, and the memory counter will increase by one after a byte is sent. In the address frame the R/$\overline{W}$ bit is logic high indicating a read instruction. The EEPROM then immediately sends back the data. The "I2C_GetDataACK()" API is used next to both utilize the data from the EEPROM and send an acknowledge bit. The value returned will always be one byte. As mentioned in Section 1.1, "acknowledge" bits indicate that the previous frame was received successfully and that the next frame may be sent. Therefore, the data stored in the next address is sent after this "acknowledge" bit. The words "Data out byte N" suggest that you may continually add more "acknowledge" bytes to receive more data. "I2C_GetDataNack()" should be used when no more data is needed. This API sends a "not acknowledge" bit, indicating that no more data is needed. Lastly, a stop condition ends communication on the data line.

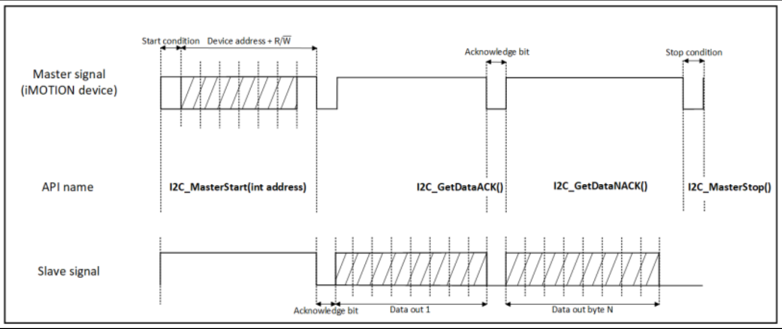

Figure 4 Read instruction

If you wish to change the memory counter's value before reading, additional steps are required. Figure 5 adds a start condition, address frame, and two data frames to the previous example.

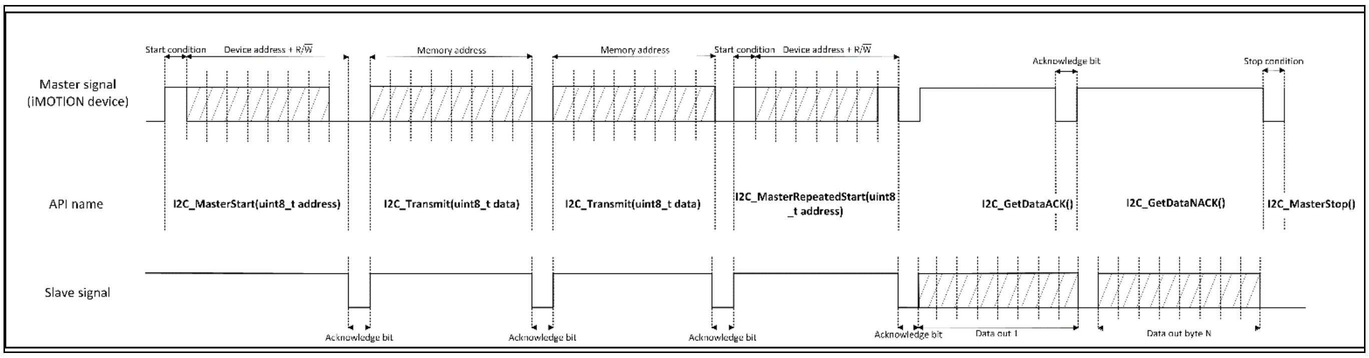

Figure 5 Read Instruction with memory counter setting

Looking more closely at the new start condition, address frame, and two data frames, notice that Figure 5's new address frame has a logic low as its R/$\overline{W}$ bit, meaning a write instruction is beginning. The reason it appears that a write instruction is beginning rather than a read instruction is because the memory counter needs to be written to. No data in the EEPROM will be overwritten. After the address frame there are two data frames. This contains the two-byte value of the memory counter. Check your EEPROM's datasheet for how many memory address locations there are, or look for the size of the memory counter. If your EEPROM is small it may only require one byte for the memory address. Or if it is bigger than 65,536 bytes it may require more than two bytes to set the memory counter. After the memory counter is set there is another start condition and address frame, this time with the R/$\overline{W}$ bit being high, switching the instruction to a read instruction. A start condition and address frame is always required to switch from a write instruction to a read instruction, or vice versa. This means that a start condition does not always indicate a new message; it may be used in the middle of a message.

Along with the new start condition, address frame, and two data frames, Figure 5 changes one other thing from the previous example. There's a new API name called "I2C_MasterRepeatedStart()". A new API is needed to control an additional SCL pulse before the start condition which prevents an accidental stop condition.

After the repeated start condition, Figure 5 is identical to the previous example. Three more APIs are used to: return the data of two bytes, send an ACK which indicates another byte is needed, send a NACK byte which indicates no more bytes are needed, then a stop condition.

2.2.1 Script Implementation

See Code listing 2 for an example of reading from EEPROM. The code reads from 4 EEPROM memory locations and applies the values to 2 parameters, MinSpd and TargetSpeed. The entire process is done within the Script_Task1_init() function which is convenient for users who want to immediately apply previously stored values upon startup. The example writes zero to the memory counter so that the memory locations read are 0-4.

As described in section 2.2, the last bit of the first address frame must be a 0 when you want to set the value of the memory counter before reading. You can see this demonstrated in line 12, followed by two bytes with the value of zero. This sets the value of the memory counter to zero. Lines 12 and 15 use an OR operator to control the last bit, the R/$\overline{W}$ bit. This is done so you can easily see what type of instruction you're using when glancing at the script. Lines 16 and 17 use bit-shifting and an OR operator to combine two bytes. In this example address 0 contains the desired value of the lower byte of MinSpd and address 1 contains the desired value of the upper byte of MinSpd. To achieve this, the value that the first "I2C_GetDataACK()" API returns is shifted by 8 bits to the left and combined with the value of another "I2C_GetDataACK()" API via an OR operator. Line 17 does the same thing except using an "I2C_GetDataNACK()" API this time to indicate that no more data is needed.

Code listing 3 Read example with memory counter setting

```
034        #SET SCRIPT_TASK1_EXECUTION_PERIOD(100)
035        #SET SCRIPT_TASK1_EXECUTION_STEP(50)
036        const int deviceSelect = 0xA0;
037        const int I2C_READ_CMD = 1;
038        const int I2C_WRITE_CMD = 0;
039        /*************************************************************
   *******/
040        /*Task1 init function*/
041        Script_Task1_init()
042        {
043              I2C_DriverInit(1, 0);
044
045              I2C_MasterStart(deviceSelect | I2C_WRITE_CMD);
046              I2C_Transmit(0x00);
047              I2C_Transmit(0x00);
048              I2C_MasterRepeatedStart(deviceSelect | I2C_READ_CMD);
049              APP_MOTOR0.MinSpd = I2C_GetDataACK() << 8 |
   I2C_GetDataACK();
050              APP_MOTOR0.TargetSpeed = I2C_GetDataACK() << 8 |
   I2C_GetDataNACK();
051              I2C_MasterStop();
```

```
        }

        /**************************************************************
   *******/
        /*Task1 script function*/
        Script_Task1()
        {
        }
```

2.2.2 Test Result

In order to find out how long an I2C message will take just take the inverse of the data rate (100kHz or 400kHz) and multiply it by the amount of bits that are within the message. Code listing 3 in the previous section contains 8 bytes that are being sent and received. Four of the APIs contain a one-byte argument and the following four APIs return one byte. Along with the eight bytes there are eight ACK/NACK bits (one after every byte), there is also one start bit, and one stop bit. Using the equation: $\frac{1}{400\text{kHz}} * \left(8\text{ bytes} * \frac{8\text{ bits}}{\text{byte}} + 2\text{ bits}\right) = 165\mu\text{s}$ and the equation: $\frac{1}{100\text{kHz}} * \left(8\text{ bytes} * \frac{8\text{ bits}}{\text{byte}} + 2\text{ bits}\right) = 660\mu\text{s}$ we can determine that the I2C message will take 660µs in normal mode and 165µs in fast mode.

The script in code listing 3 was tested while a sensorless motor control function was running and it was observed that the CPU load increases by 3.1% while the I2C driver reads.

3 Hardware Design

According to the I2C specification, the SDA and SCL lines must be connected to a positive supply voltage via pull-up resistors so that when the lines are free they are high [3]. Refer to iMOTION™ device datasheets to find out which pin corresponds to the SDA pin and SCL pin. Note its SDA/SCL pins can usually be used for other purposes which will no longer be available. Look for pinout diagrams in the datasheet and most often you will see other functionalities next to the SDA or SCL text. Once the I2C driver is initialized, the other functionalities will be disabled.

There is, however, one case where one pin may be used for more than one functionality. This is the rare case when RXD0 or TXD0 are multiplexed with SCL/SDA. RXD0 and TXD0 are the only pins that can be used to program devices. UART1 cannot be used to program devices, only for communication after programming has occurred. If the "Solution Designer COMM port" parameter is set to "UART1 " you still may not use UART1 to program; UART0 only. Because of this fact precautionary measures must be taken to prevent UART commands from being sent on the I2C lines or I2C messages from being sent to UART devices.

If developing on or referencing the EVAL-M7-111T schematic, resistors R110 and R111 must be removed in order for I2C communication to function properly. This is because the SDA/SCL pins of the EVAL board are connected to the TX and RX pins of an XMC4200 MCU that deals with debugging UART messages, and it may respond to signals that are unrecognized or misinterpreted. If you neglect to disconnect the two resistors and send an I2C command on the SDA line while monitoring the bus with an oscilloscope, you will notice undesirable signals being sent by the XMC4200 MCU. These signals will likely introduce unexpected behavior to any device connected to the I2C lines. If desired, one may use the micro-USB connector on the EVAL board to program the device prior to removing the two resistors. Doing so would allow one to lock in a final configuration before permanently disconnecting the "debugger" XMC4200 MCU from the I2C lines. If continued development of configuration files and additional programming is needed, an iMOTION link is a useful tool in this case as it can easily be connected and disconnected to the I2C lines.

Along with disconnecting UART devices from the SDA/SCL lines while I2C is being used, this should also be avoided when UART signals are being transmitted as this can create unexpected behavior within I2C devices such as EEPROM memory being altered, or unexpected I2C messages being sent. Achieving this is trickier than plugging in/unplugging an iMOTION link. This will require hardware to be introduced to disconnect the I2C lines from the rest of the system. Figure 6 shows a circuit with switches that disconnect an EEPROM from the rest of the system. This would allow users to open the connection, then utilize UART to program the device, then close the connection so that I2C can communicate. This may seem like a lot of work, however don't forget that this is only important during development; UART0 is only needed for programming which will not be occurring after development. UART1 can take care of all other UART needs and so no UART-enabled devices will be connected to the pins in the field. Check the device’s datasheet to ensure that UART0 can be used at the same time as I2C.

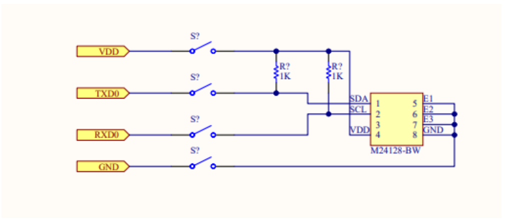

Figure 6 Breaking connection to I2C bus

4 Performance Evaluation

4.1 Timing impact of APIs

The first step to count how many bits are being sent in a message and multiply by the inverse of the selected data rate and using equation 1.

$$\frac{1}{\text{data rate (Hz)}} * \text{bits} = \text{seconds for a message to complete}$$

Equation 1 Message duration equation

To see how long each API will take, count how many bits an API transmits or receives, plus the ACK/NACK bit being sent back from another device. Refer back to Figure 2 if you forget how many bits each API transmits or receives. The "I2C_MasterStart" and "I2C_MasterRepeatedStart" will send either 9 or 12 bits. They either send a 7-bit address or 10-bit address (depending on what was selected during initialization) plus one r/w bit, plus one ACK/NACK bit. Data frames, on the other hand, always contain one byte plus one ACK/NACK bit. Along with Figure 2, the Function Reference Manual [2] shows that the APIs that deal with data frames (I2C_Transmit(),I2C_GetDataACK(), and I2C_GetDataNACK()) only ever send 8 bits or receive 8 bits.
Expect the message to take 810ns longer than what is calculated since that is how long it takes for the start bit to show up on the SDA line after the I2C_MasterStart() API is called.

If the target device does not respond after having sent a read instruction, the script will wait for a maximum of 250µs. After that time or a response is received, the script will continue on. If no response is sent back from the target for the whole 250µs an error flag will be set.

5 Guidelines

1. We recommend using Task 1 when creating a script that utilizes I2C. I2C does not affect the motor control and PFC functions as it gets what is left of the CPU bandwidth.
2. Functionalities (other than UART0) that are multiplexed with SDA/SCL pins cannot be used after the I2C driver is initialized.
3. If an I2C error occurs, a flag will be set that is visible using the "I2C_GetStatus()" API. Handling error flags can prevent execution faults from occurring. A "No response" flag will be set if the slave device does not respond. For more information on flags read the Functional Reference Manual [2]. You should not allow there to be 4 "get data" APIs executed while the target device is not responding. Executing 4 I2C_GetDataACK()/I2C_GetDataNACK() APIs while there is a no response error will cause an execution fault to occur. Utilize the I2C_GetStatus() and the I2C_Control() APIs to check and clear the no response flag. As described in the Functional Reference Manual [2], bit 26 of the data that I2C_GetStatus() returns contains if a no response error has occurred and using I2C_Control() can clear the flag.

6 References

[1] UM10204, “I2C-bus specification and user manual”; NXP Semiconductors, Revision 06 April 4, 2014

[2] iMOTION™ Motion Control Engine Functional Reference Manual.

[3] STMicroelectronics, “M24128-BW M24128-BR M24128-BF M24128-DF,” Jan 2010 [Revised May 2022].

[4] iMOTION™ Integrated Motor Control Solutions, https://www.infineon.com/cms/en/product/power/motor-control-ics/imotion-integrated

[5] Getting Started with iMOTION™ Solution Designer

Revision history

Document revision	Date	Description of changes
1.0	2023-10-25	Initial release

Edition 2023-10-25

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Email: erratum@infineon.com

Document reference

AN2024-01