Embedding it better...

µTasker Document

µTasker – I^{2}C Support, Demo and Simulation

Table of Contents

1. Introduction

The µTasker project supports the I²C interface in master mode and is designed for simple control of local hardware devices such as EEPROM, RTC, Temperature sensors, etc. It assumes that the device is reliably connected and there is no other master on the bus, since it handles neither bus contention nor error cases. However it offers an easy to use and reliable solution in the many cases where this is adequate.

The µTasker project further supports slave mode operation based on either call-back events or based on queues. This is described in the slave mode section.

2. Example - EEPROM

The µTasker project includes code to configure the I²C interface and read from and write to an I²C EEPROM (24C01). The simulator supports the device, allowing the user to observe the way that the code configures and uses the interface, as well as the simulated device responding to the commands. The methods observed are valid also for various other typical I²C peripheral devices.

The demo code can be activated by first activating the I²C driver support in `config.h` (`#define I2C_INTERFACE`) and then activating the demo use in `application.c` (`#define TEST_I2C` included in `i2c_tests.h`).

2.1. Opening the I²C Interface to the EEPROM

The code first opens the I²C interface by using the `fnOpen()` command – see `fnConfigI2C_Interface()` in `application.c` (included in `i2c_tests.h`). It is suggested to place a break point there in the simulator and the sequence can be stepped through for thorough understand of the code involved and even the hardware interface itself.

```
    I2CTABLE tI2CParameters;
    tI2CParameters.usSpeed = 100;                 // 100k
    tI2CParameters.Rx_tx_sizes.TxQueueSize = 64; // transmit queue size
    tI2CParameters.Rx_tx_sizes.RxQueueSize = 64; // receive queue size
    tI2CParameters.Task_to_wake = 0;              // no wake on transmission
    I2CPortID = fnOpen(TYPE_I2C, FOR_I_O, &tI2CParameters);
```

The configuration parameters are passed in the `I2CTABLE tI2CParameters`. The port is opened as an I²C interface and a handle returned [`QUEUE_HANDLE I2CPortID`] which is later used for all accesses.

2.2. Reading Data from the EEPROM

In the case of the EEPROM it is necessary to first perform a write to the device with the address of the location to be accessed:

```
static const unsigned char ucSetEEPROMAddress0[] = {ADD_EEPROM_WRITE, 0};

fnWrite( I2CPortID,
         (unsigned char *)&ucSetEEPROMAddress0,
         sizeof(ucSetEEPROMAddress0) );
```

In this example, the address of the EEPROM on the I^{2}C bus is written along with the access address. This will cause the device to be addressed on the bus and then the desired read address to be sent. It serves to set the internal pointer in the I^{2}C device for later access.

Immediately following the write, the user can request a read. In the following example 16 bytes are read from the EEPROM, where the internal address starts at 0, as defined by the previous command.

```
static const unsigned char ucReadEEPROM[] = {16, ADD_EEPROM_READ, OWN_TASK};

fnRead( I2CPortID,
       (unsigned char *)&ucReadEEPROM, 0);    // start the read process of 16 bytes
```

The write and read are performed using interrupts at the driver level and can be queued by the user by sending the read immediately after the write. In addition, further commands can also be queued up to the buffer length limit specified in the `I2CTABLE` parameters which were passed to the `fnOpen()` call.

The read specifies the number of bytes to be read from the I^{2}C device, the read address of the device (note that the read address and the write address are specified with the LSB at '1' for a read and '0' for the write, giving 0xa5 and 0xa4 for the 24C01 which is being simulated in the demo) and the task owning the read. The owner task will then be scheduled when the read has terminated – in this case after collecting 16 bytes from the device.

The read length of zero causes the read to be initiated according to the buffer information rather than retrieval of available data from the queue's buffer.

The µTasker project understands the EEPROM type 24C01 (see the file `\WinSim\serial_dev.c` for the internal workings and the devices which are supported on the simulated I^{2}C bus). This means that each interrupt will be processed accordingly and once the complete message has been collected, the application task will be woken. To see this when working with the simulator, set a break point at the following line in `application.c`:

```
while (LengthI2C = fnRead(I2CPortID, ucInputMessage, MEDIUM_MESSAGE)) {
```

Previous to executing this line due to the task being scheduled by an interrupt event from the I²C driver, the input queue contents were checked using:

```
fnMsgs(I2CPortID);
```

This returns the number of received *messages*, which will be 1 after all 16 defined *bytes* have been read. *It doesn't return the number of bytes in the message since this could cause the input buffer to be incorrectly read before the reception has completed.*

Generally the user knows what to expect when reading since the read and its length was also commanded by the reading task. In this example, all available bytes are read from the input buffer, with the available length being returned into `LengthI2C`. The demo displays these by writing them to the debug output (serial port, USB-CDC or Telnet, depending on configuration and availability of these interfaces).

2.3. Writing data to the I2C EEPROM

To demonstrate writing data to the EEPROM two writes are queued. The first writes the byte 0x05 to the EEPROM address 3 and the second writes several bytes from the EEPROM address 5. The following shows the write of 8 bytes to the address 5 and subsequent addresses (the address pointer is automatically incremented in the I²C EEPROM device and this represents a burst write):

```
static const unsigned char ucSetWriteEEPROM1[] = {ADD_EEPROM_WRITE, 3, 5};
static const unsigned char ucSetWriteEEPROM2[] = {ADD_EEPROM_WRITE, 5,
                3,4,5,6,7,8,9,0xa}; // prepare write of multiple bytes to address 5

fnWrite( I2CPortID,
        (unsigned char *)&ucSetWriteEEPROM1,
         sizeof(ucSetWriteEEPROM1));            // start single byte write
fnWrite( I2CPortID,
        (unsigned char *)&ucSetWriteEEPROM2,
         sizeof(ucSetWriteEEPROM2));            // add multiple byte write
```

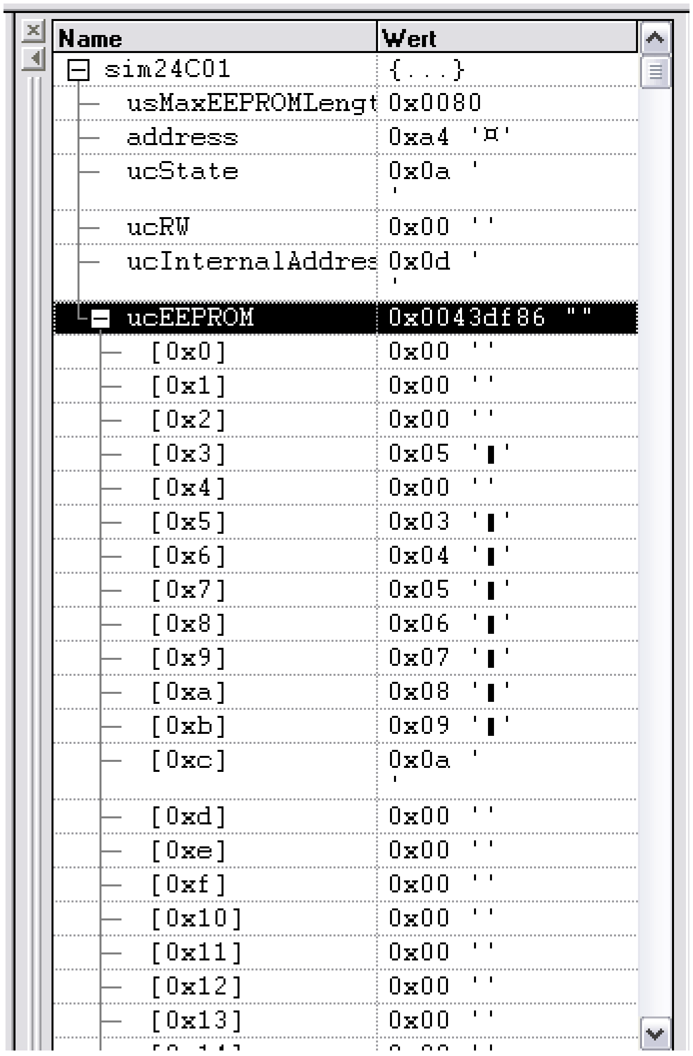
There is no acknowledgement after the writes have been completed (it is assumed that no writes ever fail due to missing or defective hardware) although a task can be optionally scheduled on termination by specifying it in `I2CTABLE` when opening the interface.

2.4. Verifying the contents of the simulated I2C EEPROM

The simulated EEPROM device can be viewed as follows from within VisualStudio (µTasker simulator):

1. Open the file `serial_dev.c` and search for the structure with the name `sim24C01`.
2. Double click on the structure and then drag it to a watch window.
3. Expand the structure in the watch window to view its control elements and more importantly the EEPROM content (`ucEEPROM`) – expanding this after the write has been performed shows that the contents are as expected. Subsequent reads from the EEPROM would then read the present values as is the case of the real device.

This allows user programs to work with (reading, writing) such a device and perform initial verification that the program is writing the correct data to the correct locations, and even correctly reacting to the read contents. Once this has been verified, the program can be run on the real hardware with the knowledge that it has already been checked for correct functionality.



Name	Wert
sim24C01	{...}
usMaxEEPROMLengt	0x0080
address	0xa4 '¤'
ucState	0x0a ' '
ucRW	0x00 ''
ucInternalAddres	0x0d ' '
ucEEPROM	0x0043df86 ""
[0x0]	0x00 ''
[0x1]	0x00 ''
[0x2]	0x00 ''
[0x3]	0x05 '▮'
[0x4]	0x00 ''
[0x5]	0x03 '▮'
[0x6]	0x04 '▮'
[0x7]	0x05 '▮'
[0x8]	0x06 '▮'
[0x9]	0x07 '▮'
[0xa]	0x08 '▮'
[0xb]	0x09 '▮'
[0xc]	0x0a ' '
[0xd]	0x00 ''
[0xe]	0x00 ''
[0xf]	0x00 ''
[0x10]	0x00 ''
[0x11]	0x00 ''
[0x12]	0x00 ''
[0x13]	0x00 ''

Figure 1: Screen shot of the EEPROM contents displayed in a VisualStudio watch window. Note that the contents are as expected after the two writes in the demo program.

3. Example of controlling a RTC via I²C bus

A well known I²C based RTC (Real Time Clock) is the Dallas DS1307. The demo project has been extended to support such a device (from 10.9.2007 - check whether your version includes the define `TEST_DS1307` in `application.c` and check newer service packs if this is not the case).

By activating the define `TEST_DS1307` in `application.c` (rather than `TEST_I2C`) the DS1307 is initialised to start, if not already active, and to generate a 1Hz output signal. A read of the internal time structure is then initiated (see `fnGetRTCTime()` in `application.c`). This reads 7 bytes of data from the RTC and copies the present date and time to a locally formatted structure (`stPresentTime`).

The 1Hz signal from the RTC is used as a 1Hz interrupt to increment the local time without need for new accesses to the RTC, whereby the date and time is requested once every 24 hours to ensure that the data is correctly synchronised – this avoids having to calculate such things as the number of days in a month and leap years.

Normally an application would also support a method of setting a new time and date to the RTC (eg. by synchronising a local PC time via web server) but such functions can be quite easily extended by using the I²C driver interface to send the correctly formatted data.

The DS1307 is also included in the I²C device simulator so that its operation can be tested without the need for such a device connected to the real hardware. When the µTasker simulator starts, its time and data is set to match the values read from the local PC.

4. Transmitter Buffer Space Checking

In some applications where the use of the I^{2}C is intensive it may be important to check that an application task is not writing faster to an output buffer than the buffer can be emptied by sending the data to the I^{2}C bus. The driver was therefore extended as from releases dated later than 1st December 2007 with a check of the output queue space. The following is an example of it in use:

```
if (fnWrite(I2CPortID, 0, sizeof(i2c_Msg)) > 0) {       // check for room in output
    fnWrite(I2CPortID, i2c_testMsg, sizeof(i2c_Msg));
}
```

The first write with a null-pointer instead of data causes the driver to return the amount of space left in the output buffer (plus 1) after a message with the defined length were to be inserted. As long as the call doesn't return 0 it means that there is enough space to accept the advertised message. It is very important to avoid writing data to the I^{2}C interface if it cannot fit into the output buffer since the buffer contains some formatting (additional information is entered) which can cause the driver to fail if the formatting gets corrupted due to content loss.

It is also important to remember that when I^{2}C reads are queued they also occupy transmit queue space. A read requires also transmission of the I^{2}C device address before the data is returned and the queue stores this address plus the amount of data to be read (from 1..255 bytes) as well as the owner task's name. This means that a read also inserts 3 bytes of data into the I^{2}C output buffer. A read thus also can justify a check of the buffer space if the I^{2}C interface is being used intensively. The following is an example of how the same type of check could be performed before queuing a read sequence:

```
if (fnWrite(I2CPortID, 0, 3) > 0) {      // check for adequate room in output queue
    fnRead(I2CPortID, (unsigned char *)&ucReadEEPROM, 0);       // command the read
}
```

5. I^{2}C Bus Deadlock Recovery

I^{2}C slaves are don't usually have a reset line and so there is a constant risk that a slave device can be left in an active state in case the master is reset during bus utilisation. The result is that the slave may be driving the bus (returning an ACK) when the master attempts to start new activity. This condition is recognised by the fact that the bus is immediately detected as busy and it is important to be able to clear this condition in order to restart activity, rather than the I^{2}C bus being unusable until a power cycle is performed (which will finally reset the slave devices).

The method used is to not immediately configure the I^{2}C for this purpose but instead to initialise the pins involved as general purpose I/Os. On first bus utilisation the state of the SDA line is checked and, if found to be '0', the SCL line is set as an output and toggled at a rate of 100kHz (generating clocks) until SDA is subsequently detected as '1'. The I^{2}C pins are then set to their I^{2}C mode so that operation can continue normally.

6. Slave Mode

Slave mode support is enabled by the define I2C_SLAVE_MODE in addition to I2C_INTERFACE.

In order to configure for slave mode of operation the speed is set to 0. An additional slave address (eg. 0xd0 [0xd0 is write address and 0xd1 is read address]) is entered so that the slave responds only to its address.

```
    I2CTABLE tI2CParameters;
    tI2CParameters.usSpeed = 0;                       // slave mode of operation
    tI2CParameters.ucSlaveAddress = OUR_SLAVE_ADDRESS; // slave address
    tI2CParameters.fnI2C_SlaveCallback = fnI2C_SlaveCallback;
                           // the I2C slave interrupt callback on reception
    tI2CParameters.Rx_tx_sizes.TxQueueSize = 64; // transmit queue size
    tI2CParameters.Rx_tx_sizes.RxQueueSize = 64; // receive queue size
    tI2CParameters.Task_to_wake = OWN_TASK;
                    // wake application task when slave reception is ready
    I2CPortID = fnOpen(TYPE_I2C, FOR_I_O, &tI2CParameters);
```

An optional slave callback handler can be entered (or 0 if not used). The input and output queue sizes can be left at 0 if the buffers are not used; for example, the callback takes care of receiving each byte individually and also each transmission byte, as is shown in the subsequent EEPROM emulation.

I²C Slave EEPROM Emulation

The following show how simple it is to emulate an EEPROM slave based on callbacks that are executed on various I²C slave events.

```
// The function is called during I2C slave interrupt handling so that the application can
// immediately respond in order to realise an I2C slave device
//
static int fnI2C_SlaveCallback(int iChannel, unsigned char *ptrDataByte, int iType)
{
    #define I2C_RAM_IDLE              0          // RAM pointer states
    #define SET_ADDRESS_POINTER       1
    static unsigned char usRAM[256] = {0};       // RAM buffer, initially zeroed
    static unsigned char ucState = I2C_RAM_IDLE; // initially idle
    static unsigned char ucAddress = 0;          // RAM address pointer is reset to zero
    switch (iType) {                             // the interrupt callback type
    case I2C_SLAVE_ADDRESSED_FOR_READ:           // the slave is being addressed for reading from
    case I2C_SLAVE_READ:                         // further reads
        *ptrDataByte = usRAM[ucAddress++];       // return the data and increment the address pointer
        ucState = I2C_RAM_IDLE;                  // return to the idle state
        return I2C_SLAVE_TX_PREPARED;            // the prepared byte is to to be sent
    case I2C_SLAVE_READ_COMPLETE:                // complete read is complete
        break;
    case I2C_SLAVE_ADDRESSED_FOR_WRITE:          // the slave is being addressed to write to
        // *ptrDataByte is our address
        //
        ucState = SET_ADDRESS_POINTER;           // a write is being received and we expect the
                                                    address followed by a number of data bytes
        return I2C_SLAVE_RX_CONSUMED;            // the byte has been consumed and nothing is to be
                                                    put in the queue buffer
    case I2C_SLAVE_WRITE:                        // data byte received
        // *ptrDataByte is the data received
        //
        if (ucState == SET_ADDRESS_POINTER) {     // we are expecting the address to be received
            ucAddress = *ptrDataByte;             // set the single-byte address
            ucState = I2C_RAM_IDLE;               // return to data reception
        }
        else {
            usRAM[ucAddress++] = *ptrDataByte;    // save the data and increment the address pointer
        }
        return I2C_SLAVE_RX_CONSUMED;             // the byte has been consumed and nothing is to be
                                                     put in the queue buffer
    case I2C_SLAVE_WRITE_COMPLETE:                // the write has terminated
        // ptrDataByte is 0 and so should not be used
        //
        return I2C_SLAVE_RX_CONSUMED;
    }
    return I2C_SLAVE_BUFFER;                      // use the buffer for this transaction
}
```

Each time the callback is executed it is passed an `iType` value to indicate which event has taken place. The operation of the emulated EEPROM can be described as follows:

1. When a master addresses the I²C slave as read the callback type `I2C_SLAVE_ADDRESSED_FOR_READ` is handled. This occurs after the master has sent a start condition (or a repeated start) and the slave address matches. It is up to the callback to decide what the next byte to be read from the slave will be. In the case of emulated EEPROM the next byte from the EEPROM array is written to the address pointed to by `ptrDataByte` and the value `I2C_SLAVE_TX_PREPARED` returned. The next byte read by the master will be this value.

2. When the master reads subsequent bytes the type `I2C_SLAVE_READ` is valid. In the case of EEPROM emulation the handling is identical to the first byte read.

3. When the master has read the final byte (this is recognised by the fact that the master doesn't acknowledge the previous byte read from the slave) `I2C_SLAVE_READ_COMPLETE` is valid so that the slave knows that no further data is to be prepared. In the case of the EEPROM emulation this event doesn't need to be handled. Note that there is no event for a stop condition after a master read since this is not necessary.

4. `I2C_SLAVE_ADDRESSED_FOR_WRITE` means that master has addressed the slave in order to write data to it. This occurs after a start condition (or repeated start) has been detected, followed by the matching address of the slave being received. The emulated EEPROM prepares to receive the next byte which will be recognised as its address pointer.

5. `I2C_SLAVE_WRITE` is received for each byte (pointed to by `ptrDataByte`) that is subsequently received, whereby the first one is interpreted as the address of the EEPROM's internal memory pointer and following ones as data to be set to this address (the emulated EEPROM's internal address pointer increments after each new byte written or read).

6. `I2C_SLAVE_WRITE_COMPLETE` is received either when the stop condition is detected after the master has completed the write, or when the slave is subsequently addressed for read or write again following a repeated start. This event is consumed by the EEPROM emulator (returns `I2C_SLAVE_RX_CONSUMED`) but otherwise causes no specific handling.

This has shown how slave device emulation can be achieved simply with a small amount of code to handle the slave callback events.

I²C Slave Output Buffer

Although not of interest to the EEPROM emulation, other communication implementations may benefit from making use of the slave mode's output buffer. An example would be to return a message back to the master after it has written a certain command. This would most likely be placed in the handling of the `I2C_SLAVE_WRITE_COMPLETE` and prepare the response by performing a write.

```
case I2C_SLAVE_WRITE_COMPLETE:                         // the write has terminated
    {
        unsigned char message[10] = {9, 1,2,3,4,5,6,7,8,9};
        fnWrite(I2CPortID, message, sizeof(message));
    }
    return I2C_SLAVE_RX_CONSUMED;
```

The write sets the message content to the I²C slave's output buffer and will be returned to the master when it subsequently reads from the slave. This makes it simple to prepare responses without requiring further callback code to manage its specific transmission details. When the events `I2C_SLAVE_ADDRESSED_FOR_READ` and `I2C_SLAVE_READ` are received the callback returns `I2C_SLAVE_BUFFER`, which informs the callback driver that it should not send a byte set by the callback (which is indicated by the return `I2C_SLAVE_TX_PREPARED`) but instead take it from the output buffer.

The master should read the appropriate number of bytes, which may be controlled by the length to be read either being negotiated previously, being fixed, or being reported in the first byte in the output buffer, for example. Should the master not read all of the bytes from the buffer the remaining ones will be read when it reads further data. Once the buffer has been emptied by the master reads the I²C slave will send 0xff should additional reads be made.

I^{2}C Slave Simulation

The operation of the I^{2}C slave can be exercised by using a script file (menu **`Port sim | Open Script`** or **`Port sim | Repeat last script`**). The following are examples of script content to inject various sequences from a bus master.

After a delay of 50ms inject a write on I^{2}C channel 0 to address 0xd0 with 4 bytes of data (0x55, 0xaa, 0x55, 0xaa) and terminate with a stop condition:

```
+50 I2C-0 d0 09 55 aa 55 aa
```

After a delay of 50ms inject a read on I^{2}C channel 0 to address 0xd1, followed by reading 7 bytes of data and terminate with a stop condition:

```
+50 I2C-0 d1 07
```

After a delay of 100ms inject a write on I^{2}C channel 0 to address 0xd0 with 1 byte of data (0x0a). This is NOT terminated by a stop condition but instead the following message will be performed with a repeated start instead:

```
+100 I2CR-0 d0 0a
```

Immediately following the previous command, inject a read on I^{2}C channel 0 from address 0x0d1, followed by a read of 5 bytes and terminate with a stop condition:

```
+0  I2C-0 d1 05
```

By combining these simple script commands it is possible to create scripts that allow extensive testing of the I^{2}C slave operation, exercising complete driver and callback code.

Script commands relevant to I^{2}C:

`I2C-n` – inject start condition, followed by the address and data in the data list, terminated with a stop condition. I^{2}C channel n.

`I2CR-n` – inject start condition, followed by the address and data in the data list. No stop condition is sent, thus allowing repeated starts to be tested. I^{2}C channel n.

7. Conclusion

This document has illustrated the use of the µTasker I^{2}C driver master interface which allow queuing of I^{2}C write and read sequences. The µTasker project contains code to show two common I^{2}C devices in use: an I^{2}C EEPROM and an I^{2}C RTC. Both of these devices are simulated in the µTasker simulator to allow users to comfortably verify their own code before moving on to final tests with the real hardware devices.

The document has further described the I^{2}C slave mode of operation which is based on event callbacks to allow simple emulation of slave devices. The use of the I^{2}C slave mode's output buffer also makes it easy to prepare messages to be read by a master. Slave simulation, based on script files, has been discussed.

Modifications:

V0.01 14.1.2007:
- Initial draft version for the V1.3 project

V0.02 24.12.2007:
- Addition of Real Time Clock example and transmission buffer space checking

V0.03 14.3.2009:
- Reformat document with header and table of contents.

V0.04 23.12.2016:
- Update IIC to I2C to correspond with newer code. Add bus deadlock recovery. Add slave mode description. Add double-buffered Kinetis I^{2}C controller details.

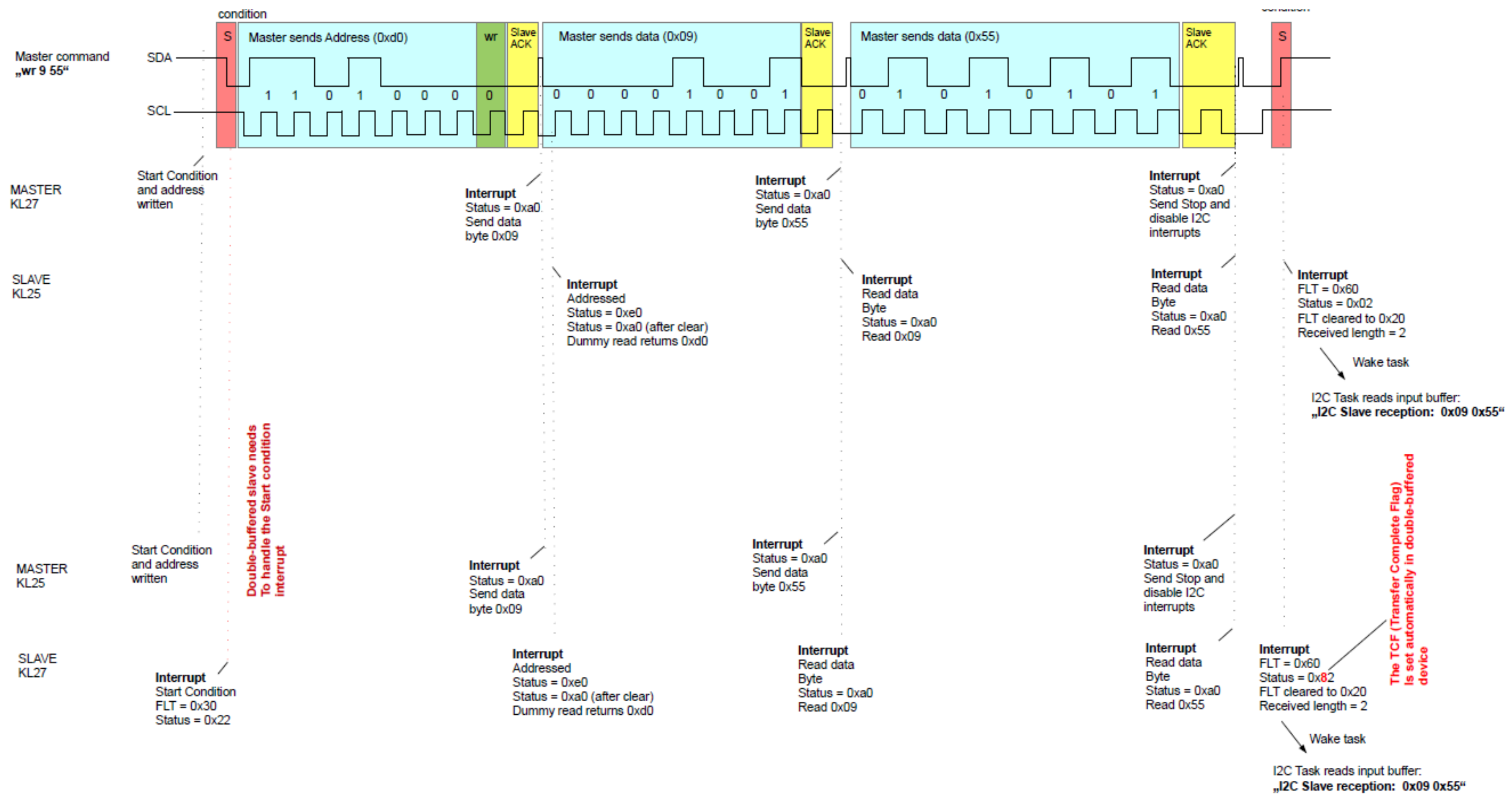
V0.05 20.05.2017:
- Correct EEROM code references.

V0.06 29.12.2017:
- New appendix about adding slave devices to simulator.

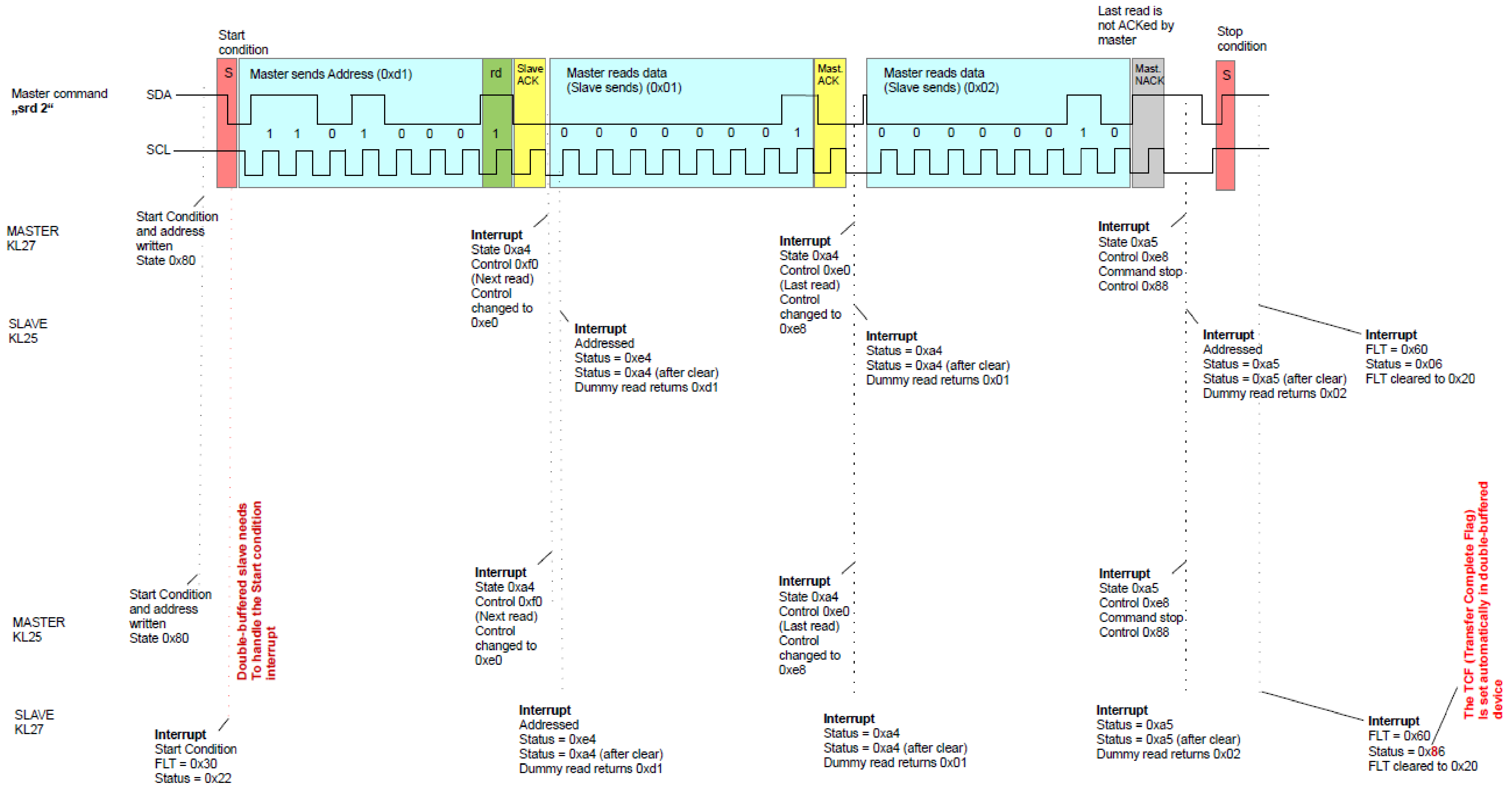
Appendix A – Kinetis Double-Buffered Operation

Some newer Kinetis processors have an I^2C controller that has been modified with double buffered features in order to be able to obtain higher throughput. Such controller types however have various errors in the design since they don't correctly control the clock line when slaves want to hold the bus. Furthermore various undocumented behaviour needs to be worked around in order for full operation to be achieved.

The following illustrates typically state and interrupt behaviour that has been observed in practice and results in a successful and compatible operation. For the tests KL25 (single-buffered design) and KL27 (double-buffered design) parts were connected together in alternating Master and Slave modes.



condition
Master command „wr 9 55“
SDA
SCL
S
Master sends Address (0xd0)
1 1 0 1 0 0 0
wr
0
Slave ACK
Master sends data (0x09)
0 0 0 0 1 0 0 1
Slave ACK
Master sends data (0x55)
0 1 0 1 0 1 0 1
Slave ACK
S
MASTER KL27
Start Condition and address written
Interrupt Status = 0xa0 Send data byte 0x09
Interrupt Status = 0xa0 Send data byte 0x55
Interrupt Status = 0xa0 Send Stop and disable I2C interrupts
SLAVE KL25
Interrupt Addressed Status = 0xe0 Status = 0xa0 (after clear) Dummy read returns 0xd0
Interrupt Read data Byte Status = 0xa0 Read 0x09
Interrupt Read data Byte Status = 0xa0 Read 0x55
Interrupt FLT = 0x60 Status = 0x02 FLT cleared to 0x20 Received length = 2
Wake task
I2C Task reads input buffer: „I2C Slave reception: 0x09 0x55“
Double-buffered slave needs To handle the Start condition interrupt
MASTER KL25
Start Condition and address written
Interrupt Status = 0xa0 Send data byte 0x09
Interrupt Status = 0xa0 Send data byte 0x55
Interrupt Status = 0xa0 Send Stop and disable I2C interrupts
SLAVE KL27
Interrupt Start Condition FLT = 0x30 Status = 0x22
Interrupt Addressed Status = 0xe0 Status = 0xa0 (after clear) Dummy read returns 0xd0
Interrupt Read data Byte Status = 0xa0 Read 0x09
Interrupt Read data Byte Status = 0xa0 Read 0x55
Interrupt FLT = 0x60 Status = 0x82 FLT cleared to 0x20 Received length = 2
The TCF (Transfer Complete Flag) is set automatically in double-buffered device
Wake task
I2C Task reads input buffer: „I2C Slave reception: 0x09 0x55“



MASTER KL27

Start Condition and address written
State 0x80

Interrupt
State 0xa4
Control 0xf0
(Next read)
Control changed to 0xe0

Interrupt
State 0xa4
Control 0xe0
(Last read)
Control changed to 0xe8

Interrupt
State 0xa5
Control 0xe8
Command stop
Control 0x88

SLAVE KL25

Interrupt
Addressed
Status = 0xe4
Status = 0xa4 (after clear)
Dummy read returns 0xd1

Interrupt
Status = 0xa4
Status = 0xa4 (after clear)
Dummy read returns 0x01

Interrupt
Addressed
Status = 0xa5
Status = 0xa5 (after clear)
Dummy read returns 0x02

Interrupt
FLT = 0x60
Status = 0x06
FLT cleared to 0x20

Double-buffered slave needs To handle the Start condition interrupt

MASTER KL25

Start Condition and address written
State 0x80

Interrupt
State 0xa4
Control 0xf0
(Next read)
Control changed to 0xe0

Interrupt
State 0xa4
Control 0xe0
(Last read)
Control changed to 0xe8

Interrupt
State 0xa5
Control 0xe8
Command stop
Control 0x88

SLAVE KL27

Interrupt
Start Condition
FLT = 0x30
Status = 0x22

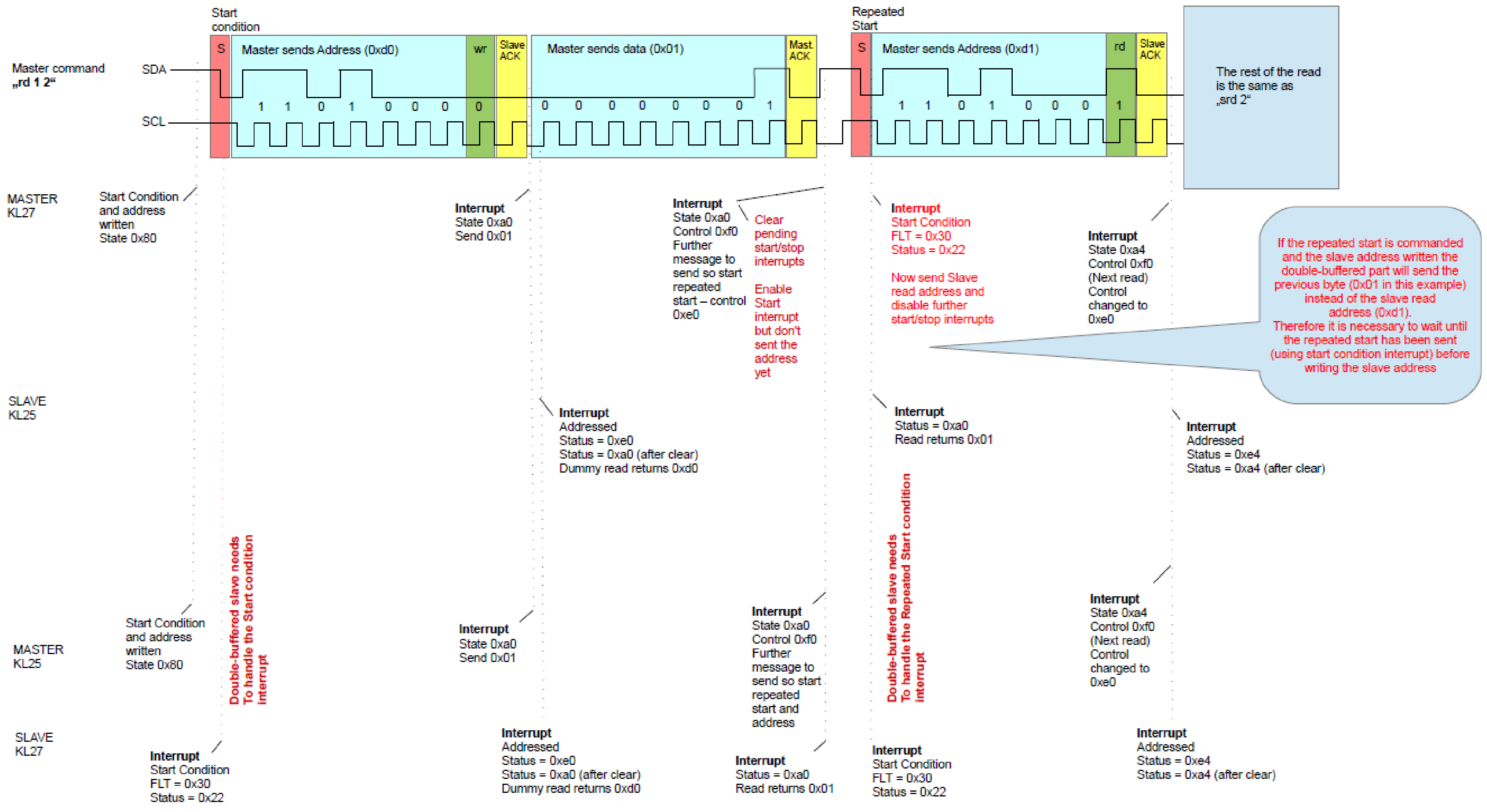
Interrupt
Addressed
Status = 0xe4
Status = 0xa4 (after clear)
Dummy read returns 0xd1

Interrupt
Status = 0xa4
Status = 0xa4 (after clear)
Dummy read returns 0x01

Interrupt
Status = 0xa5
Status = 0xa5 (after clear)
Dummy read returns 0x02

Interrupt
FLT = 0x60
Status = 0x86
FLT cleared to 0x20

The TCF (Transfer Complete Flag) is set automatically in double-buffered device


Start condition
Repeated Start
Master command „rd 1 2“
SDA
SCL
S
Master sends Address (0xd0)
wr
Slave ACK
Master sends data (0x01)
Mast. ACK
S
Master sends Address (0xd1)
rd
Slave ACK
1 1 0 1 0 0 0 0
0 0 0 0 0 0 0 1
1 1 0 1 0 0 0 1
The rest of the read is the same as „srd 2“
MASTER KL27
Start Condition and address written State 0x80
Interrupt State 0xa0 Send 0x01
Interrupt State 0xa0 Control 0xf0 Further message to send so start repeated start – control 0xe0
Clear pending start/stop interrupts
Enable Start interrupt but don't sent the address yet
Interrupt Start Condition FLT = 0x30 Status = 0x22
Now send Slave read address and disable further start/stop interrupts
Interrupt State 0xa4 Control 0xf0 (Next read) Control changed to 0xe0
If the repeated start is commanded and the slave address written the double-buffered part will send the previous byte (0x01 in this example) instead of the slave read address (0xd1). Therefore it is necessary to wait until the repeated start has been sent (using start condition interrupt) before writing the slave address
SLAVE KL25
Interrupt Addressed Status = 0xe0 Status = 0xa0 (after clear) Dummy read returns 0xd0
Interrupt Status = 0xa0 Read returns 0x01
Interrupt Addressed Status = 0xe4 Status = 0xa4 (after clear)
Double-bufferd slave needs To handle the Start condition interrupt
Double-buffered slave needs To handle the Repeated Start condition interrupt
MASTER KL25
Start Condition and address written State 0x80
Interrupt State 0xa0 Send 0x01
Interrupt State 0xa0 Control 0xf0 Further message to send so start repeated start and address
Interrupt State 0xa4 Control 0xf0 (Next read) Control changed to 0xe0
SLAVE KL27
Interrupt Start Condition FLT = 0x30 Status = 0x22
Interrupt Addressed Status = 0xe0 Status = 0xa0 (after clear) Dummy read returns 0xd0
Interrupt Status = 0xa0 Read returns 0x01
Interrupt Start Condition FLT = 0x30 Status = 0x22
Interrupt Addressed Status = 0xe4 Status = 0xa4 (after clear)

The results show that additional start/stop interrupt handling is usually required, plus the fact that repeated starts are only possible when the slave address is not sent until after the repeated start condition has physically taken place on the bus.

The I^{2}C drivers in the µTasker project automatically respect these requirements in order to allow all Kinetis parts to be used reliably and in a compatible manner from the user perspective.

Appendix B – Adding an I^{2}C Slave Device to the Simulator

This appendix explains how a new slave device is added to the simulator by illustrating the steps required to add a MAX6956 so that it can be worked with and visualised.

The MAX6956 is an I^{2}C slave that can act as either a 20-port I/O expander or 28-port LED display driver with a partially selectable address of 0x80..0x9e (write) and 0x81..0x9f (read); in this example the HW pin address selection is assumed to be 0x80 (write) / 0x81 (read).

As is typical for many I^{2}C slave devices a set of command registers exist that are addressed in addition to the slave address. For example, selecting the register address 0x09, allows writing to this register in order to configure P4, P5, P6 and P7. Other registers allow reading ports and setting output data to the ports. This register set needs to be emulated - as does the devices bus address, which requires it to be added to the simulation c-file `serial_dev.c`.

0x7f registers (some not used) are defined and so the MAX6956 instance can be added quite simply with

```
/**************************************************************************/
/*                  Maxim MAX6956 port-expander/LED driver                */
/**************************************************************************/

// Address 0x80
//
typedef struct stMAX6956
{
    unsigned char  address;
    unsigned char  ucState;
    unsigned char  ucRW;
    unsigned char  ucCommand;
    unsigned char  ucRegs[0x7f];
} MAX6956;

static MAX6956 simMAX6956 = {0x80, 0, 0, 0,
                           {0}
};
```

In case any register values need to be initialised to specific values when the simulation starts, the states can be added to the local routine `fnInitialiseI2CDevices()`.

Since the MAX6956 should only respond when addressed and be passive when not its state can be cleared each time another device is addressed by adding

```
    if (ucAddress != simMAX6956.address) {
        simMAX6956.ucState = 0;
    }
```

to the local routine `fnResetOthers()`.

The actual active operation is then controlled in the local routine:

```
unsigned char fnSimI2C_devices(unsigned char ucType, unsigned char ucData);
```

whereby the extent and accuracy of the simulation depends on this entry – in case just a subset of the functionality is used only that needs to be added but of course a complete simulation (when feasible) is always preferred.

There are a small number of events that need to be handled, whereby the most basic is the address event:

```
case I2C_ADDRESS:
```

which is typically handled with

```
    else if ((ucData & ~0x01) == simMAX6956.address) {  // being addressed
        simMAX6956.ucState = 1;
        simMAX6956.ucRW = (ucData & 0x01);              // mark whether read or write
    }
```

Data writes to the addressed slave are then handled by the event:

```
case I2C_TX_DATA:
```

and data reads by the event:

```
case I2C_RX_DATA:
```

Should there be a special operation taking place after the bus transfer terminates the events

```
    case I2C_RX_COMPLETE:
    case I2C_TX_COMPLETE:
```

can also be handled, although these tend to just de-select the previously addressed device.

Specifically for devices of this type the first byte written after its write address is the command byte. Following bytes may control specific functions that are then dependent on the command. When reads follow a command the meaning of the read may also be depending on it too. Generally multiple reads and writes cause data to be read/written to subsequent addresses due to the fact that the internal address pointer is incremented for each byte transfer.

To program the slave device simulation thus requires knowledge of its command set, which is also a good exercise to do before programming an actual application based on the chip since it ensures that its behaviour is indeed appreciated. Should there be a difference in the operation between the simulation and hardware this can be investigated and adjusted accordingly, once the reason for the initial misinterpretation is also understood.

In order to illustrate the basic simulation programming the operation of the 28 LED ports will be analysed in an application driving all of them as individual LEDs. This requires each GPIO to be configured as output with a specific drive strength and then certain turned on and possibly others off. First we look at the *application* code that is used for basic configuration:

```
static const unsigned char config_leds[] = { MAX6956_WRITE_ADDRESS, PORT_CONFIG_P7_P6_P5_P4,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
fnWrite(I2CPortID, (unsigned char *)config_leds, sizeof(config_leds));
```

which writes the command 0x09 (the first command of 7 register addresses for the complete port configuration) followed by 7 0x00 bytes, making use of the auto-increment command address so that 7 configuration registers can be written with a single address and transfer.

The simplified write simulation of the command register and register content is as follows, which is typical code for such operations:

```
    else if (simMAX6956.ucState == 1) {           // MAX6956 is being written to
        simMAX6956.ucCommand = (ucData & 0x7f);
        simMAX6956.ucState++;
    }
    else if (simMAX6956.ucState > 1) {            // date being written
        unsigned char *ptr = (unsigned char *)&simMAX6956.ucRegs;
        ptr += simMAX6956.ucCommand;
        if (simMAX6956.ucCommand < 0x7f) {        // increment the command register
                                                     after each write
            simMAX6956.ucCommand++;
        }
        *ptr = ucData;                            // set the data
    }
```

The simulation saves the written byte to the appropriate register and auto-increments the internal command address up to a maximum value of 0x7f, which is the behaviour described in the MAX6956's data sheet.

If these register were to be read back again the following *application* code could be used:

```
static const unsigned char check_leds[] = { 7, MAX6956_READ_ADDRESS, OWN_TASK };
fnWrite(I2CPortID, (unsigned char *)config_leds, 2);  // the address to be read from
fnRead(I2CPortID, (unsigned char *)check_leds, 0);    // start the read of 7 bytes
```

The read simulation code to allow this is similar to the write case (after the read event) and also increments the command register pointer after each access:

```
    else if ((simMAX6956.ucRW != 0) && (simMAX6956.ucState == 1)) {
                                                 // MAX6956 is being read from
        unsigned char *ptr = (unsigned char *)&simMAX6956.ucRegs;
        ptr += simMAX6956.ucCommand;
        if (simMAX6956.ucCommand < 0x7f) {       // increment the command register
                                                    after each read
            simMAX6956.ucCommand++;
        }
        return (*ptr);
    }
```

Once the basic register reads and writes are implemented any special behaviour that it controller or triggered by particular actions can be handled to suit. An example of the behaviour of the MAX6956 is that its GPIOs or LED outputs can be written/read at multiple registers; one can write 8 bits at a time (P11..P4) by writing to command address 0x44, or one can write just P4 by writing to control register 0x24.Writes to one register thus also influence the content of other registers, which the simulation must handle on a register to register basis to achieve correct behaviour. *The actual state of all of the output can however simply be monitored by watching the single register 0x44*.

In the case of the example use of the MAX6956 GPIOs as LED drivers it is very useful to be able to hook them up to the simulation too, which can be performed as follows:

1. In `app_hw_xxx.h` (where xxx is the processor family being used – eg. `app_hw_kinetis.h`) the define

```
#define _EXTERNAL_PORT_COUNT 1           // 1 external 28 bit port based on a port expander
```

is added, which informs the simulator that it should extend ports by a further (external) port in addition to the processor's GPIO ports.

2. When the project I built it will now complain about there being two routines missing:
`fnGetExtPortDirection()`
and
`fnGetExtPortState()`

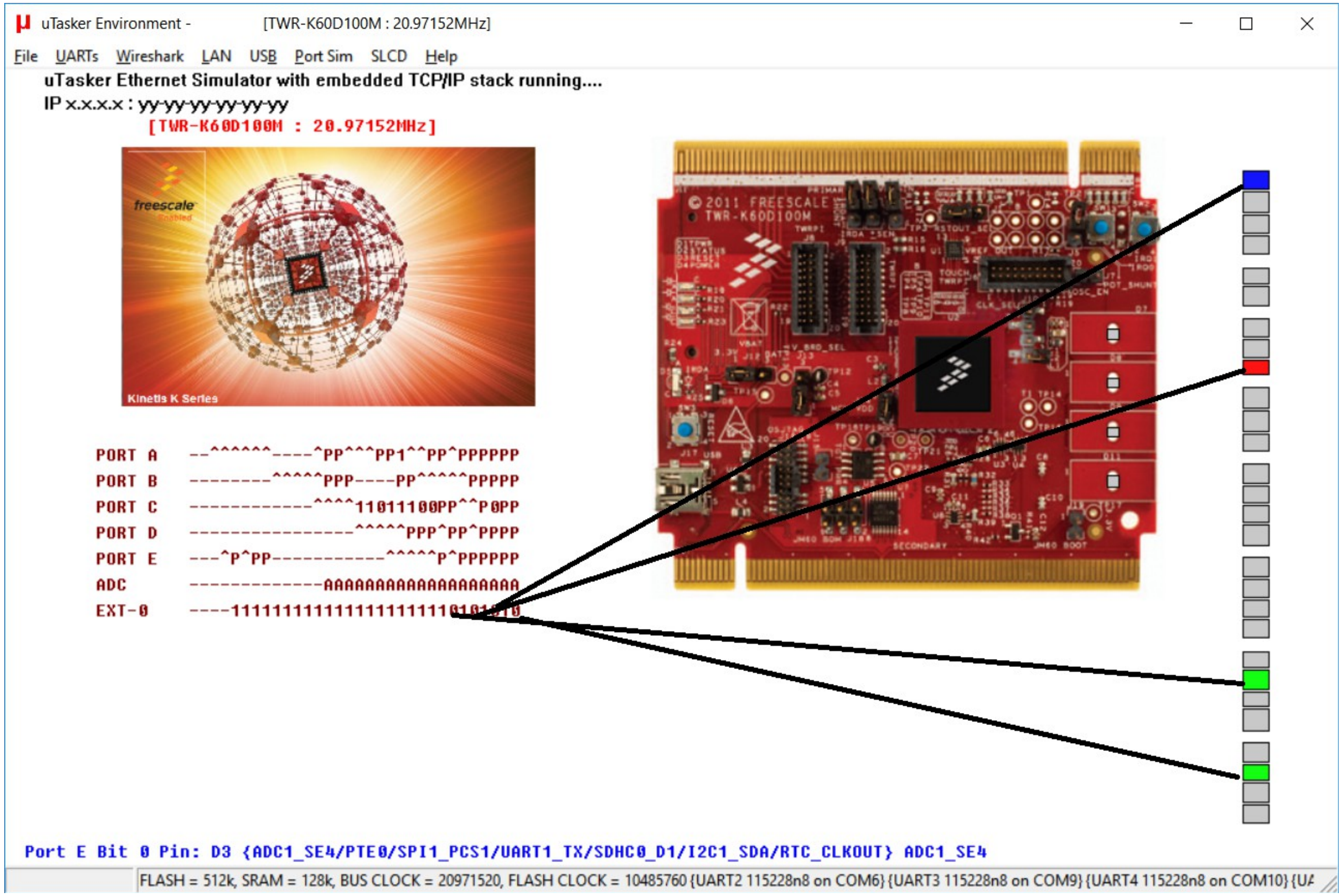
These can also be added to the I^{2}C simulator for the part in question so that the present data direction of the ports and its present logical state can be returned. The simulator will use these to display the details and also to drive LED images if configured. For information about configuring output visualisation see this video: https://youtu.be/x0oe4kscIDI whereby `_PORT_EXP_0` can be used as first external port reference instead of `_PORTA` or other internal port references. A reference simulation is shown on the following page, where '0' on the extended outputs means that the output is driving open-drain low (LED on).

The complete set of LED definitions for this simulation is:

```
                            // '0'       '1'    input state   center (x,   y)   0 = circle, radius, controlling port, controlling pin

#define KEYPAD_LED_DEFINITIONS  {RGB(255,75,0), RGB(200,200,200), 1, {352, 148, 362, 157 }, _PORTA, DEMO_LED_1}, \
                                {RGB(255,128,0),RGB(200,200,200), 1, {352, 186, 362, 194 }, _PORTA, DEMO_LED_2}, \
                                {RGB(0,255,0),  RGB(200,200,200), 1, {352, 224, 362, 232 }, _PORTA, DEMO_LED_3}, \
                                {RGB(20,20,255),RGB(200,200,200), 1, {352, 261, 362, 271 }, _PORTA, DEMO_LED_4}, \
                                {RGB(20,20,255),RGB(200,200,200), 1, {457, 26,  478, 41  }, _PORT_EXP_0, PORTA_BIT4}, \
                                {RGB(20,20,255),RGB(200,200,200), 1, {457, 42,  478, 59  }, _PORT_EXP_0, PORTA_BIT8}, \
                                {RGB(20,255,20),RGB(200,200,200), 1, {457, 60,  478, 75  }, _PORT_EXP_0, PORTA_BIT9}, \
                                {RGB(255,20,20),RGB(200,200,200), 1, {457, 76,  478, 91  }, _PORT_EXP_0, PORTA_BIT5}, \
                                {RGB(20, 255, 20), RGB(200, 200, 200), 1, { 457, 101,  478, 114 }, _PORT_EXP_0, PORTA_BIT19}, \
                                {RGB(255, 20, 20), RGB(200, 200, 200), 1, { 457, 115,  478, 131 }, _PORT_EXP_0, PORTA_BIT18}, \
                                {RGB(20, 255, 20), RGB(200, 200, 200), 1, { 457, 140,  478, 155 }, _PORT_EXP_0, PORTA_BIT7}, \
                                {RGB(20, 255, 20), RGB(200, 200, 200), 1, { 457, 156,  478, 171 }, _PORT_EXP_0, PORTA_BIT10}, \
                                {RGB(255, 20, 20), RGB(200, 200, 200), 1, { 457, 172,  478, 184 }, _PORT_EXP_0, PORTA_BIT6}, \
                                {RGB(20, 255, 20), RGB(200, 200, 200), 1, { 457, 193,  478, 210 }, _PORT_EXP_0, PORTA_BIT13}, \
                                {RGB(20, 255, 20), RGB(200, 200, 200), 1, { 457, 211,  478, 228 }, _PORT_EXP_0, PORTA_BIT12}, \
                                {RGB(255, 20, 20), RGB(200, 200, 200), 1, { 457, 229,  478, 243 }, _PORT_EXP_0, PORTA_BIT11}, \
                                {RGB(20, 20, 255), RGB(200, 200, 200), 1, { 457, 252,  478, 268 }, _PORT_EXP_0, PORTA_BIT14}, \
                                {RGB(20, 255, 20), RGB(200, 200, 200), 1, { 457, 269,  478, 283 }, _PORT_EXP_0, PORTA_BIT17}, \
                                {RGB(20, 255, 20), RGB(200, 200, 200), 1, { 457, 284,  478, 298 }, _PORT_EXP_0, PORTA_BIT16}, \
                                {RGB(255, 20, 20), RGB(200, 200, 200), 1, { 457, 299,  478, 316 }, _PORT_EXP_0, PORTA_BIT15}, \
                                {RGB(20, 20, 255), RGB(200, 200, 200), 1, { 457, 324,  478, 340 }, _PORT_EXP_0, PORTA_BIT20}, \
                                {RGB(20, 255, 20), RGB(200, 200, 200), 1, { 457, 341,  478, 356 }, _PORT_EXP_0, PORTA_BIT23}, \
                                {RGB(20, 255, 20), RGB(200, 200, 200), 1, { 457, 357,  478, 371 }, _PORT_EXP_0, PORTA_BIT22}, \
                                {RGB(255, 20, 20), RGB(200, 200, 200), 1, { 457, 372,  478, 387 }, _PORT_EXP_0, PORTA_BIT21}, \
                                {RGB(20, 20, 255), RGB(200, 200, 200), 1, { 457, 397,  478, 410 }, _PORT_EXP_0, PORTA_BIT24}, \
                                {RGB(20, 255, 20), RGB(200, 200, 200), 1, { 457, 411,  478, 427 }, _PORT_EXP_0, PORTA_BIT2}, \
                                {RGB(20, 255, 20), RGB(200, 200, 200), 1, { 457, 428,  478, 440 }, _PORT_EXP_0, PORTA_BIT25}, \
                                {RGB(255, 20, 20), RGB(200, 200, 200), 1, { 457, 441,  478, 459 }, _PORT_EXP_0, PORTA_BIT3}, \
                                {RGB(20, 20, 255), RGB(200, 200, 200), 1, { 457, 467,  478, 483 }, _PORT_EXP_0, PORTA_BIT26}, \
                                {RGB(20, 255, 20), RGB(200, 200, 200), 1, { 457, 484,  478, 497 }, _PORT_EXP_0, PORTA_BIT0}, \
                                {RGB(20, 255, 20), RGB(200, 200, 200), 1, { 457, 498,  478, 514 }, _PORT_EXP_0, PORTA_BIT27}, \
                                {RGB(255, 20, 20), RGB(200, 200, 200), 1, { 457, 515,  478, 530 }, _PORT_EXP_0, PORTA_BIT1}
```

It is advised to take a look at some existing device simulations that already exist in the `serial_dev.c` file, including the state of the MAX6956 one, to see their specific solution in full detail. These should give adequate inspiration to start and complete further ones as required.


uTasker Environment - [TWR-K60D100M : 20.97152MHz]
File UARTs Wireshark LAN USB Port Sim SLCD Help

uTasker Ethernet Simulator with embedded TCP/IP stack running....
IP x.x.x.x : yy-yy-yy-yy-yy-yy
[TWR-K60D100M : 20.97152MHz]

```
PORT A    --^^^^^^----^PP^^^PP1^^PP^PPPPPP
PORT B    --------^^^^^PPP----PP^^^^^PPPPP
PORT C    ------------^^^^11011100PP^^P0PP
PORT D    ----------------^^^^^PPP^PP^PPPP
PORT E    ---^P^PP-----------^^^^^P^PPPPPP
ADC       -------------AAAAAAAAAAAAAAAAAAA
EXT-0     ----1111111111111111111111010100
```

Port E Bit 0 Pin: D3 {ADC1_SE4/PTE0/SPI1_PCS1/UART1_TX/SDHC0_D1/I2C1_SDA/RTC_CLKOUT} ADC1_SE4

FLASH = 512k, SRAM = 128k, BUS CLOCK = 20971520, FLASH CLOCK = 10485760 {UART2 115228n8 on COM6} {UART3 115228n8 on COM9} {UART4 115228n8 on COM10} {UA

Simulation example with MAX6956 controlled extended outputs driving an array of LEDs.