Embedding it better...

µTasker Document

µTasker – I2C Support, Demo and Simulation

Table of Contents

1. Introduction

The µTasker 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.

2. Demo Example - EEPROM

The µTasker demo project includes code to configure the I²C interface and read and write from / 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 IIC_INTERFACE`) and then activating the demo use in `application.c` (`#define TEST_IIC`).

2.1. Opening the I²C Interface to the EEPROM

The code first opens the I²C interface by using the `fnOpen()` command – see `fnConfigEEPROM()` in `application.c`. 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.

```
IICPortID = fnOpen( TYPE_IIC, FOR_I_O, &tIICParameters );
```

The configuration parameters are passed in the `IICTABLE tIICParameters`. The port is opened as an I²C interface and a handle returned [`QUEUE_HANDLE IICPortID`] 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( IICPortID,
         (unsigned char *)&ucSetEEPROMAddress0,
         sizeof(ucSetEEPROMAddress0) );
```

In this example, the address of the EEPROM on the I²C 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 I²C 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( IICPortID,
       (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 `IICTABLE` parameters which were passed to the `fnOpen()` call.

The read specifies the number of bytes to be read from the I²C 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 woken 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 `\Hardware\IIC_devices\ IIC_dev.c` for the internal workings and the devices which are supported on the simulated I²C 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 (LengthIIC = fnRead( IICPortID, ucInputMessage, MEDIUM_MESSAGE)) {
```

Previous to executing this line due to the task being woken by an interrupt event from the I^2C driver, the input queue contents were checked using:

```
fnMsgs(IICPortID);
```

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 `LengthIIC`. The demo displays these by writing them to the debug output (serial port if activated, or Telnet if enabled).

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 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 I^2C 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( IICPortID,
        (unsigned char *)&ucSetWriteEEPROM1,
         sizeof(ucSetWriteEEPROM1));            // start single byte write
fnWrite( IICPortID,
        (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 woken on termination by specifying it in `IICTABLE` 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 `IIC_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 basically tested for correct functionality.

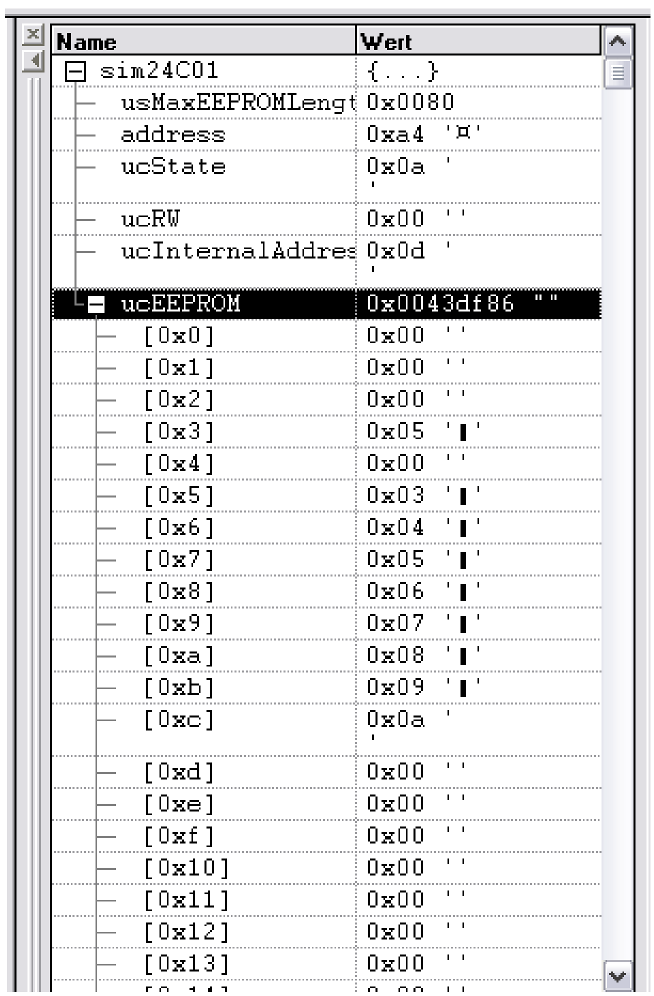

Figure 1: Screen shot of the EEPROM contents displayed in a VisualStudio watch window. Note that the contents are as expected after the two writes in the demo program.

3. Example of controlling a RTC via I2C bus

A well known I^2C 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_IIC)` 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 I^2C driver interface to send the correctly formatted data.

The DS1307 is also included in the I^2C 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 I²C 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 I²C 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(IICPortID, 0, sizeof(iic_Msg)) > 0) {       // check for room in output
    fnWrite(IICPortID, iic_testMsg, sizeof(iic_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 I²C 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 I²C reads are queued they also occupy transmit queue space. A read requires also transmission of the I²C 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 I²C output buffer. A read thus also can justify a check of the buffer space if the I²C 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(IICPortID, 0, 3) > 0) {      // check for adequate room in output queue
    fnRead(IICPortID, (unsigned char *)&ucReadEEPROM, 0);       // command the read
}
```

5. Conclusion

This document has illustrated the use of the µTasker I²C driver interface which allow queuing of I²C write and read sequences. The µTasker demo project contains code to show two common I²C devices in use: an I²C EEPROM and an I²C 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 devices and hardware.

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.