uM-FPU Application Note 8

Developing an I^2C Interface for uM-FPU V2

Micromega Corporation

This application note describes a suggested method of developing support software for connecting a microcontroller to the uM-FPU V2 floating point coprocessor using an I^2C interface.

Introduction

Micromega provides support software for many popular microcontrollers, so it's worth checking the Micromega website (http://www.micromegacorp.com) to see if software is already available for your microcontroller, or email info@micromegacorp.com to enquire about any plans for development. If support is not currently available for your microcontroller, you can easily develop your own support software. This application note describes a suggested method for developing an I^2C interface. Since implementation details vary with each different microcontroller, pseudo-code is used to describe the actions of the each routine, which can then be translated into assembly code, C or Basic for your microcontroller.

Additional Documents

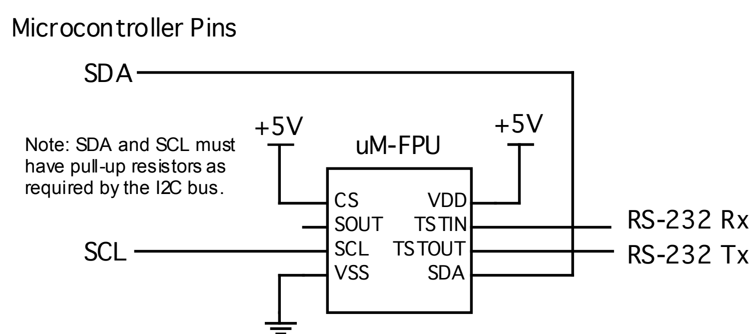
Before getting started it is recommended that you review the *uM-FPU V2 Datasheet*, and the *uM-FPU V2 Instruction Set* documents. It is also recommended that the serial interface for the uM-FPU debug monitor be connected as described in the *uM-FPU V2 Datasheet*. This provides access to valuable debugging information while testing the support routines.

Support software and documentation provided by Micromega for other microcontrollers can also serve as a good example of the code you will need to develop.

I^2C Interface

The I^2C interface uses two bi-directional lines, SCL and SDA, that are connected through a pull-up resistor to the positive supply voltage, and shared by all connected devices. The uM-FPU can handle I^2C data speeds up to 400 kHz.

I^2C Connection



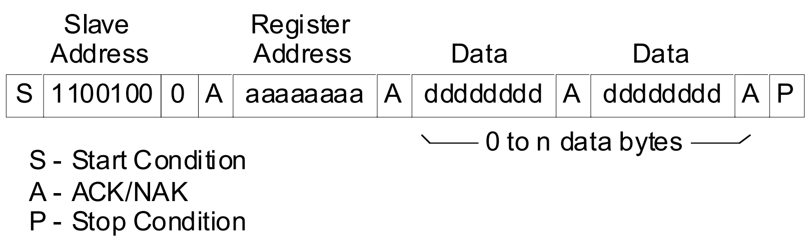
Each connected device must have a unique slave address. The uM-FPU uses the 7-bit address mode. Data is transferred using a protocol that consists of a Start condition, followed by data, and terminated by a Stop condition (or in some cases a new Start conditon). The first byte following the Start condition is the 7-bit uM-FPU slave address, followed by an 8th bit which specifies whether the microcontroller wishes to write data to the uM-FPU (0), or read data from the uM-FPU (1). The default slave address for the uM-FPU is 1100100x (binary).

- expressed as a 7-bit value (no Read/Write bit), the default uM-FPU address is 0x64 (hex), or 100 (decimal)
- expressed as a 8-bit value (Read/Write bit set to zero) the default uM-FPU address is 0xC8 (hex), or 200 (decimal)

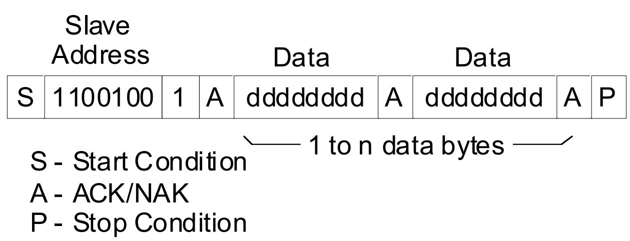
The slave address can be changed to another value and stored in nonvolatile flash memory using the built-in serial debug monitor, as described in the *uM-FPU V2 Datasheet*.

The following diagrams show the write and read data transfers. The write transfer consists of a start condition, slave address, write bit, and register address, followed by 0 to n data bytes and a stop condition. A read transfer is normally preceded by a write transfer to select the register to read from. The read transfer consists of a start condition, slave address, and read bit, followed by 0 to n data bytes and a stop condition. A NAK should be sent on the last byte of a read transfer.

I^{2}C Write Data Transfer



I^{2}C Read Data Transfer



I^{2}C Registers

Register Address	Write	Read
0	Data	Data / Status
1	Reset	Buffer Space

I^{2}C Device Level Support Routines

The I^{2}C interface can be implemented with the following device level support routines:

`i2c_master`	initializes the I^{2}C interface for master mode
`i2c_start`	sends the Start condition
`i2c_stop`	sends the Stop condition
`i2c_write`	writes a byte of data and returns the ACK/NAK value
`i2c_readACK`	reads a byte of data and responds with an ACK
`fpu_readNAK`	reads a byte of data and responds with a NAK

Implementation of the I^{2}C device level support routines is not discussed in this application note. These are common routines for use with any I^{2}C device, and most compilers for microcontrollers provide support for these or similar functions. Basic compilers generally have commands such as `I2CWRITE`, `I2CREAD` that allows writing and reading of multiple bytes to I^2C devices. They handle the start condition, stop condition and address byte. If you need to develop your own routines, there are many examples available to work from.

uM-FPU Device Level Support Routines

The interface with the uM-FPU is described in terms of the I^2C device level support routines listed above.

`fpu_reset`	resets the uM-FPU
`fpu_startWrite`	starts a write transfer
`fpu_startRead`	starts a read transfer
`fpu_readByte`	reads an 8-bit byte from the uM-FPU
`fpu_wait`	waits until the uM-FPU buffer is empty
`fpu_readDelay`	implements the required read delay

The steps required to implement the support routines are as follows:

1. Implement initial version of `fpu_reset`
2. Implement `fpu_startWrite`
3. Implement `fpu_startRead`
4. Implement `fpu_readByte`
5. Add synchronization check to `fpu_reset`
6. Implement `fpu_wait`
7. Implement `fpu_readDelay`
8. Create include file with uM-FPU opcode definitions

Step 1 – Implement initial version of fpu_reset

The uM-FPU must be reset at the start of every program to establish synchronization with the microprocessor. This is the first routine that needs to be implemented. The `fpu_reset` routine sends the reset command, waits for the uM-FPU to complete the reset code, then checks for proper synchronization by sending a `SYNC` opcode (`0xF0`) and reading the response byte. Since we have not yet developed the code to send and read data, we will add the synchronization check later (in step 5).

Parameters: none
Return: sync character
C prototype: `unsigned char fpu_reset(void);`
Pseudo-code:

```
i2c_start                    ; send Start condition
i2c_write(0xC8)              ; send write address (uM-FPU address plus write bit)
i2c_write(1)                 ; select control register (register 1)
i2c_write(0)                 ; write 0 to control register (reset)
i2c_stop                     ; send Stop condition
delay for 8 milliseconds     ; wait for reset to complete
return
```

Debug Monitor:
Whenever a reset occurs , the following message is displayed by the debug monitor:

```
{RESET}
```

Step 2 – Implement fpu_startWrite

This routine starts a write transfer to the uM-FPU, and is called before sending any instructions to the uM-FPU. Data is sent to the uM-FPU using the `i2c_write` routine. The `i2c_stop` function is called to terminate a write transfer.

Parameters: none

Return: none
C prototype: `void fpu_startWrite(void);`
Pseudo-code:

```
i2c_start                    ; send Start condition
i2c_write(0xC8)              ; send write address (uM-FPU address plus write bit)
i2c_write(0)                 ; select data register (register 0)
return
```

Write a test routine to send three bytes to the uM-FPU (e.g. `0x00`, `0xFF` and `0xAA`).

```
fpu_reset
fpu_startWrite
i2c_write(0x00)
i2c_write(0xFF)
i2c_write(0xAA)
i2c_stop(
```

Debug Monitor:
If the instructions are received properly by the uM-FPU, the debug monitor will display the following:

```
00 FF AA
```

Step 3 – Implement fpu_startRead

This routine starts a read transfer, and is called before reading any data from the uM-FPU. Data is read from the uM-FPU using either the `i2c_readACK` or `i2c_readNAK` functions. If multiple bytes are read, `i2c_readACK` should be used for all bytes except the last byte. To ensure that the read transfer is terminated properly, the last byte must be read using `i2c_readNAK`. If only a single byte is read, `i2c_readNAK` should be used. The `i2c_stop` function is called to terminate a read transfer.

Parameters: none
Return: none
C prototype: `void fpu_startRead(void);`
Pseudo-code:

```
fpu_startWrite               ; send Start condition, Write address, and select register 0
i2c_start                    ; send new Start condition
i2c_write(0xC9)              ; send read address (uM-FPU address plus read bit)
return
```

Step 4 – Implement fpu_readByte

This routine can be called after any instruction that returns data from the uM-FPU. It waits for the Read Setup Delay, then starts a read transfer, reads a single byte and terminates the read transfer. The byte read from the uM-FPU is returned.

Parameters: none
Return: 8-bit value
C prototype: `unsigned char fpu_readByte(void);`
Pseudo-code:

```
fpu_readDelay                ; wait for read delay
fpu_startRead                ; send Start condition and Read address
n = i2c_readNAK              ; read byte from uM-FPU
i2c_stop                     ; end the read transfer
return n                     ; return the byte
```

Write a test routine to send the SYNC opcode (0xF0) to the uM-FPU and read the byte that is returned. If SYNC is successful, a 0x5C byte will be returned. The sequence is as follows:

```
fpu_startWrite                    ; start write transfer
i2c_write(0xF0)                   ; send SYNC command
i2c_stop                          ; stop write transfer
n = fpu_readByte                  ; read the return value
```

Debug Monitor:
If the instructions are received properly by the uM-FPU, the debug monitor will display the following:

```
F0:5C
```

Step 5 – Add synchronization check to fpu_reset

Add the synchronization code shown above to the end of the fpu_reset routine. The byte that is read after sending the SYNC opcode is returned by fpu_reset. The calling routine can check if the return value is 0x5C to ensure that the reset and synchronization was successful. Note: In all other situations fpu_wait must be called before sending an opcode that returns data, but fpu_wait is specifically not used in the fpu_reset routine, since an 8 millisecond delay immediately precedes it, and the uM-FPU will always be ready if the reset is successful. If the reset is not successful an fpu_wait could wait indefinitely, so by not including it, the fpu_reset routine will always return with a value.

Step 6 – Implement fpu_wait

The fpu_wait routine is used to ensure that the 32-byte instruction buffer and the debug trace buffer are both empty. The fpu_wait routine must always be called before data is read from the uM-FPU. It should also be called at least once for every 32 bytes of output to ensure that the instruction buffer doesn't overflow. The busy/ready status is returned whenever the data register is read and no data is waiting to be returned. If the uM-FPU is ready, a zero byte is returned. If the uM-FPU is busy, either executing instructions, or because the debug monitor is active, a 0x80 byte is returned. The fpu_wait routine continues to read the busy/ready status until the uM-FPU is ready.

Parameters: none
Return: none
C prototype: `void fpu_wait(void);`
Pseudo-code:

```
loop:
   fpu_startRead
   dataByte = i2c_readNAK           ; read busy/ready status
   i2c_stop                         ; wait until ready
   if dataByte <> 0 then goto loop
return
```

Step 7 – Implement fpu_readDelay

Instructions that read data from the uM-FPU require a minimum 180 microseconds delay after the opcode has been sent before data can be read (or 90 microseconds if debug trace is not enabled). The read delay routine simply delays for 180 microseconds. In some implementations, the fpu_readDelay routine may not be required if sufficient delay occurs due to the overhead associated with I^2C bus protocol. For future compatibility, and when operating at higher speeds, it is a good idea to still implement code using fpu_readDelay before reading data.

Parameters: none
Return: none

C prototype: `void fpu_readDelay(void);`
Pseudo-code:

```
delay for 180 microseconds        ; delay for Read Setup Delay
return
```

Step 8 – Create include file with uM-FPU opcode definitions

To make it easer to write code for the uM-FPU, an include file should be created that contains definitions for all of the uM-FPU opcodes and the sync character. Various include files are available on the Micromega website and can easily be adapted as required.

Examples using the Device Level Support Routines

Calculate y = 5x + 30

```
fpu_startWrite               ; start write transfer
i2c_write(SELECTA+Y)         ; select Y register
i2c_write(LOADBYTE)          ; load 5 to register 0 and convert to floating point
i2c_write(5)
i2c_write(FSET)              ; y = 5.0
i2c_write(FMUL+X)            ; y = y * x
i2c_write(LOADBYTE)
i2c_write(30)                ; load 30 to register 0 and convert to floatint point
i2c_write(FADD)              ; y = y + 30
i2c_stop                     ; end write transfer
```

Read byte from uM-FPU register n

```
fpu_wait                     ; wait until uM-FPU is ready
fpu_startWrite               ; start write transfer
i2c_write(SELECTA+N)         ; select N register (32-bit integer value)
i2c_write(XOP)               ; send READBYTE instruction
i2c_write(READBYTE)
i2c_stop                     ; end the write transfer
n = fpu_readByte             ; read the byte
```

Adding Additional Support Routines

The five device level support routines developed above provide all of the necessary support for using the uM-FPU with your microcontroller. Using these device level routines, additional routines can be developed to provide a higher level of support. Examples might include:

print_float	print formatted floating point number
print_long	print formatted long integer number
print_version	print uM-FPU version number
load_float	load floating point value
load_long	load long integer value
load_floatStr	load floating point string
load_longStr	load long integer string

Further Information

Check the Micromega website at www.micromegacorp.com for up-to-date information.