Micromega Corporation

Using uM-FPU V3.1 with the WinAVR™ Compiler

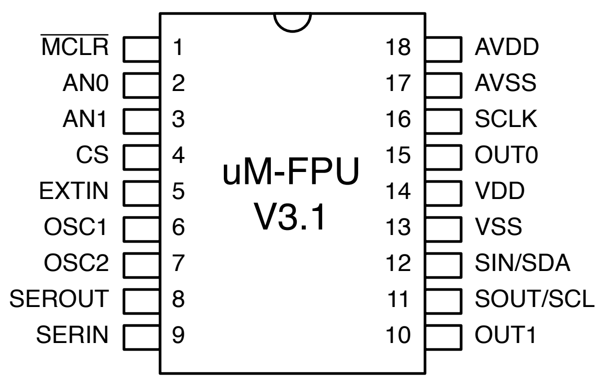
Introduction

The uM-FPU V3.1 chip is a 32-bit floating point coprocessor that can be easily interfaced with Atmel AVR® microcontrollers, and programmed using the WinAVR™ Compiler, to provide support for 32-bit IEEE 754 floating point operations and 32-bit long integer operations. The uM-FPU V3.1 chip supports both I^2C and SPI connections.

This document describes how to use the uM-FPU V3.1 chip with the WinAVR compiler. For a full description of the uM-FPU V3.1 chip, please refer to the *uM-FPU V3.1 Datasheet* and *uM-FPU V3.1 Instruction Reference*. Application notes are also available on the Micromega website.

uM-FPU V3.1 Pin Diagram and Pin Description

DIP-18, SOIC-18



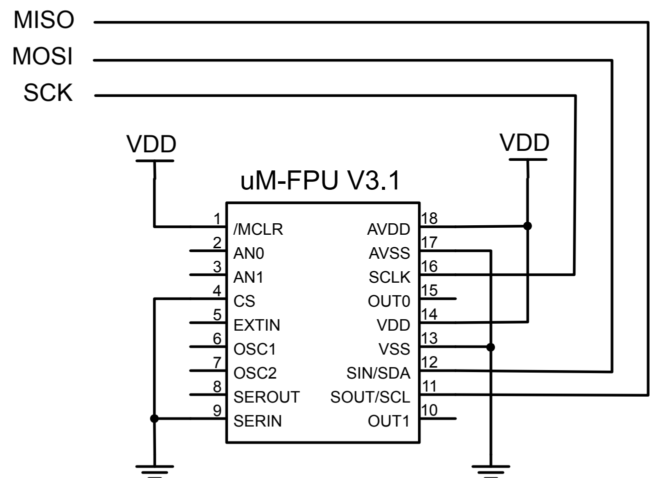
Pin	Name	Type	Description
1	/MCLR	Input	Master Clear (Reset)
2	AN0	Input	Analog Input 0
3	AN1	Input	Analog Input 1
4	CS	Input	Chip Select, Interface Select
5	EXTIN	Input	External Input
6	OSC1	Input	Oscillator Crystal (optional)
7	OSC2	Output	Oscillator Crystal (optional)
8	SEROUT	Output	Serial Output, Debug Monitor - Tx
9	SERIN	Input	Serial Input, Debug Monitor - Rx
10	OUT1	Output	Digital Output 1
11	SOUT SCL	Output Input	SPI Output, Busy/Ready Status I^2C Clock

12	SIN SDA	Input In/Out	SPI Input I^2C Data
13	VSS	Power	Digital Ground
14	VDD	Power	Digital Supply Voltage
15	OUT0	Output	Digital Output 0
16	SCLK	Input	SPI Clock
17	AVSS	Power	Analog Ground
18	AVDD	Power	Analog Supply Voltage

Connecting the Atmel AVR using 3-wire SPI

If the uM-FPU V3.1 chip is the only chip connected to the SPI port, only three pins are required for interfacing the Atmel AVR to the uM-FPU V3.1 chip using a 3-wire SPI interface. The communication uses a clock pin, an input data pin, and an output data pin. The SPI pin definitions are included in the *fpu_spi.h* file.

Atmel AVR Pins

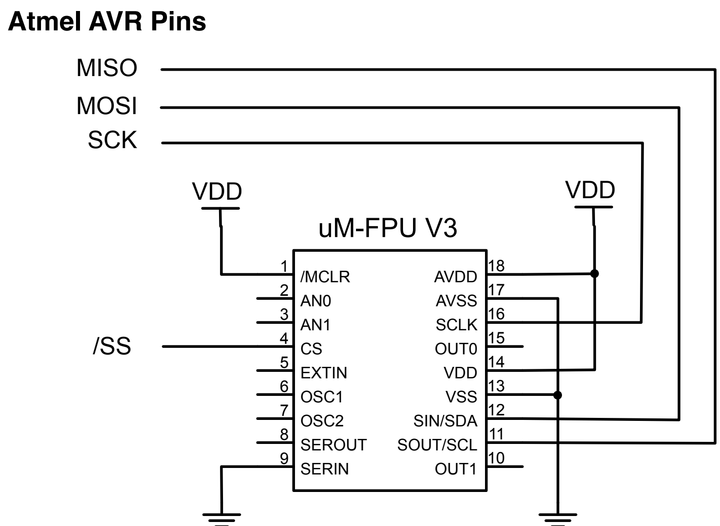


SPI Pins for various Atmel AVR microcontrollers

Pin Name	ATmega48 ATmega88 ATmega168 ATmega8	ATmega16 ATmega32 ATmega164P ATmega324P ATmega644P ATmega162	ATmega64 ATmega128
SCLK	PB5	PB7	PB1
MOSI	PB3	PB5	PB2
MISO	PB4	PB6	PB3

Connecting the Atmel AVR using an SPI Bus Interface

If the uM-FPU V3.1 chip will be connected to the SPI bus with multiple devices, the `CS` pin on the FPU must be enabled as a chip select. The procedure for enabling the `CS` pin is described on the next page. The SPI pin definitions are included in the *fpu_spi.h* file. The symbol `FPU_CS_ENABLED` must be defined to enable the chip select code in the support software. This can be done by adding a definition to the *fpu_spi.h* header file, or by adding the custom compilation option `-DFPU_CS_ENABLED=1` to the project.



The clock signal is idle low and data is read on the rising edge of the clock (often referred to as SPI Mode 0).

SPI Pins for various Atmel AVR microcontrollers

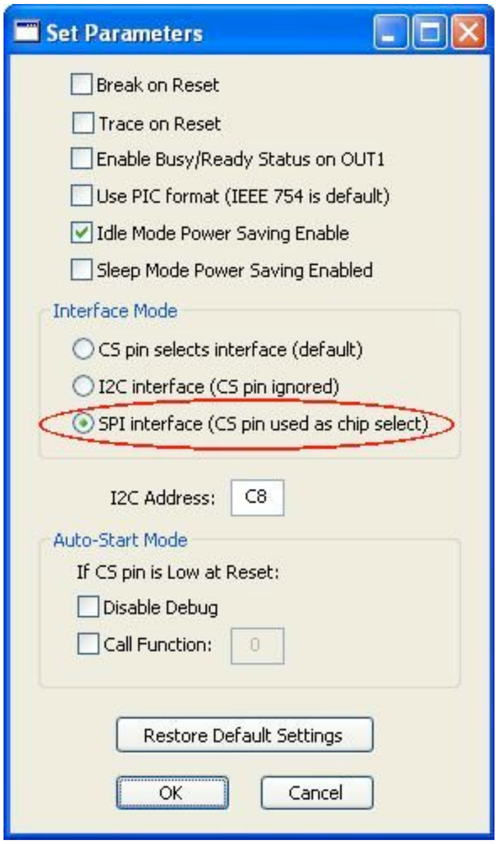
Pin Name	ATmega48 ATmega88 ATmega168 ATmega8	ATmega16 ATmega32 ATmega164P ATmega324P ATmega644P ATmega162	ATmega64 ATmega128
/SS	PB2	PB4	PB0
SCLK	PB5	PB7	PB1
MOSI	PB3	PB5	PB2
MISO	PB4	PB6	PB3

Enabling the CS pin as a chip select

The uM-FPU V3.1 `CS` pin is enabled as a chip select by setting bits 1:0 of mode parameter byte 0 to 11. The mode parameter bytes are stored in Flash memory on the uM-FPU V3.1 chip, and programmed using the built-in debug monitor (see *uM-FPU V3.1 Datasheet* for details). Since the parameter bytes are stored in Flash memory, these bits only need to be set once.

> Note: For production runs, the uM-FPU V3.1 can be ordered directly from Micromega with the parameter byte set for *SPI interface (CS pin used as chip select)*.

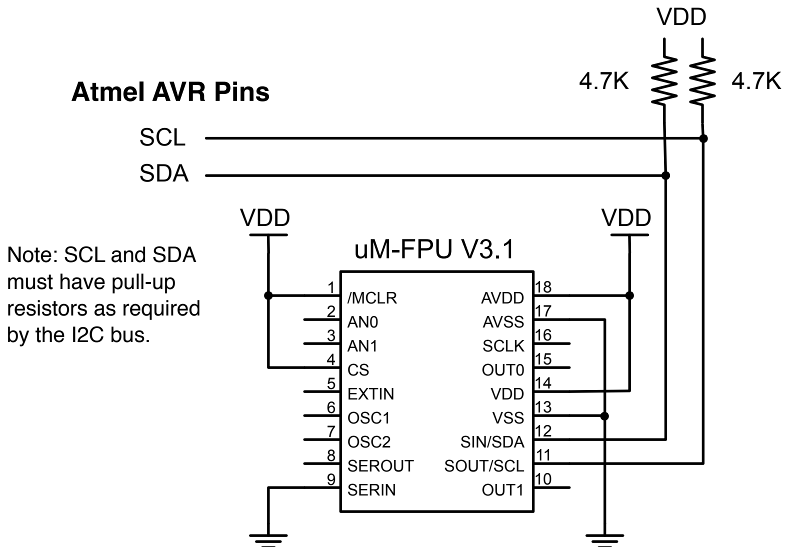
To set the interface mode, the uM-FPU V3 IDE (Integrated Development Environment) software can be used. The *Set Parameters*... command in the *Functions* menu displays the dialog shown below. Select the *SPI interface (CS pin used as chip select)* interface mode.



When this mode is selected, the SPI interface is automatically selected at Reset, and the `CS` pin is enabled as an active low chip select. The `SOUT` pin is high impedance when the uM-FPU V3.1 chip is not selected.

Connecting the Atmel AVR using an I^2C Bus Interface

The uM-FPU V3.1 can be connected using an I^2C interface. The Two-wire Serial Interface (TWI) is used on the Atmel AVR. The default slave ID for the uM-FPU chip is $C8.



I^2C Pins for various Atmel AVR microcontrollers

Pin Name	ATmega48 ATmega88 ATmega168 ATmega8	ATmega16 ATmega32 ATmega164P ATmega324P ATmega644P	ATmega64 ATmega128
SCL	PC4	PC0	PD0
SDA	PC5	PC1	PD1

Brief Overview of the uM-FPU V3.1 Floating Point Coprocessor

For a full description of the uM-FPU V3.1 chip, please refer to the *uM-FPU V3.1 Datasheet*, *uM-FPU V3.1 Instruction Reference*. Application notes are also available on the Micromega website.

The uM-FPU V3.1 chip is a separate coprocessor with its own set of registers and instructions designed to provide microcontrollers with 32-bit floating point and long integer capabilities. The Atmel AVR communicates with the FPU using an SPI or I^2C interface. Instructions and data are sent to the FPU, and the FPU performs the calculations. The Atmel AVR is free to do other tasks while the FPU performs calculations. Results can be read back to the Atmel AVR or stored on the FPU for later use. The uM-FPU V3.1 chip has 128 registers, numbered 0 through 127, that can hold 32-bit floating point or long integer values. Register 0 is often used as a temporary register and is modified by some of the uM-FPU V3.1 instructions. Registers 1 through 127 are available for general use.

The `SELECTA` instruction is used to select any one of the 128 registers as register A. Register A can be regarded as an accumulator or working register. Arithmetic instructions use the value in register A as an operand and store the result of the operation in register A. If an instruction requires more than one operand, the additional operand is specified by the instruction. The following example selects register 2 as register A and adds register 5 to it:

```
SELECTA, 2          select register 2 as register A
FADD, 5             register[A] = register[A] + register[5]
```

Sending Instructions to the uM-FPU

Appendix A contains a table that gives a summary of each uM-FPU V3.1 instruction, with enough information to follow the examples in this document. For a detailed description of each instruction, refer to the *uM-FPU V3.1 Instruction Reference*.

The `fpu_write` routine is used to send instructions and data to the FPU. There are four versions of the `fpu_write` routine depending on the number of bytes being sent:

```
void fpu_write(char bval1);
void fpu_write2(char bval1, char bval2);
void fpu_write3(char bval1, char bval2, char bval3);
void fpu_write4(char bval1, char bval2, char bval3, char bval4);
```

An example of sending an instruction to the FPU is as follows:

```
fpu_write2(FADD, 5);
```

All instructions have an opcode that tells the FPU which operation to perform. The following example calculates the square root of register A:

```
fpu_write(SQRT);
```

Some instructions require additional operands or data which are specified in the bytes following the opcode. The following example adds register 5 to register A.

```
fpu_write2(FADD, 5);
```

Some instructions return data. This example reads the lower 8 bits of register A:

```
fpu_wait();
fpu_write(LREADBYTE);
dataByte = fpu_read();
```

The following example adds the value in register 5 to the value in register 2.

```
fpu_write4(SELECTA, 2, FADD, 5);
```

It's a good idea to use constant definitions to provide meaningful names for the registers. This makes your program easier to read and understand. The same example using constant definitions would be:

```
#define Total 2         // total amount  (uM-FPU register)
#define Count 5         // current count (uM-FPU register)

fpu_write4(SELECTA, Total, FADD, Count);
```

Tutorial Examples

Now that we've introduced some of the basic concepts of sending instructions to the uM-FPU chip, let's go through a tutorial example to get a better understanding of how it all ties together. This example takes a temperature reading from a DS1620 digital thermometer and converts it to Celsius and Fahrenheit.

Most of the data read from devices connected to the Atmel AVR will return some type of integer value. In this example, the interface routine for the DS1620 reads a 9-bit value and stores it in a variable on the Atmel AVR called `rawTemp`. The value returned by the DS1620 is the temperature in units of 1/2 degrees Celsius. The following instructions load the `rawTemp` value to the FPU, convert it to floating point, then divide it by 2 to get degrees in Celsius.

```
fpu_write3(SELECTA, DegC, LOADWORD);
fpu_writeWord(rawTemp);
fpu_write3(FSET0, FDIVI, 2);
```

Description:

`SELECTA, DegC`	select DegC as register A
`LOADWORD, rawTemp`	load rawTemp to register 0 and convert to floating point
`FSET0`	DegC = register[0] (i.e. the floating point value of rawTemp)
`FDIVI, 2`	divide by the floating point value 2.0

To get the degrees in Fahrenheit we use the formula F = C * 1.8 + 32. Since 1.8 is a constant value, it would normally be loaded once in the initialization section of the program and used later in the main program. The value 1.8 can be loaded to register `F1_8` using the `fpu_writeFloatReg` as follows:

```
fpu_writeFloatReg(1.8, F1_8);
```

We calculate the degrees in Fahrenheit (F = C * 1.8 + 32) as follows:

```
fpu_write4(SELECTA, DegF, FSET, DegC);
fpu_write4(FMUL, F1_8, FADDI, 32);
```

Description:

`SELECTA, DegF`	select DegF as register A
`FSET, DegC`	DegF = DegC
`FMUL, F1_8`	DegF = DegF * 1.8
`FADDI, 32`	DegF = DegF + 32.0

Note: this tutorial example is intended to show how to perform a familiar calculation, but the `FCNV` instruction could be used to perform unit conversions in one step. See the *uM-FPU V3.1 Instruction Reference* for a full list of conversions.

There are support routines provided for sending floating point and long integer strings to the serial port. We use `print_float` to send a formatted floating point string. The `format` parameter selects the desired format, with the tens digit specifying the total number of characters to display, and the ones digit specifying the number of digits after the decimal point. The DS1620 has a maximum temperature of 125° Celsius and one decimal point of precision, so we'll use a format of 51. Before calling the `print_float` routine the FPU register is selected. The following example sends the temperature in degrees Celsius and Fahrenheit to the serial port.

```
fpu_write2(SELECTA, DegC);
print_float(51);
```

```
fpu_write2(SELECTA, DegF);
print_float(51);
```

Sample code for this tutorial and a wiring diagram for the DS1620 are shown at the end of this document. The file *demo1.c* is also included with the support software. There is a second file called *demo2.c* that extends this demo to include minimum and maximum temperature calculations. If you have a DS1620 you can wire up the circuit and try out the demos.

uM-FPU V3.1 Support Software

Several files are provided for interfacing the uM-FPU V3.1 chip with the Atmel AVR microcontroller. Support is provided for both the I^2C and SPI interface. The routines are compatible with the WinAVR calling conventions.

fpu.h
This header file includes the file *fops.h* which defines all of the uM-FPU V3.1 opcodes, matrix operations and FFT operations. It also defines the FPU status bits and provides function prototypes for all of the C callable routines in *fpu_spi.S, fpu_i2c.S* and *fpuPrint.c*. It should be included in any WinAVR program that calls the FPU support routines.

fpu_spi.S
This file contains all of the low-level support routines for interfacing with the uM-FPU V3.1 chip using the master SPI serial interface on the Atmel AVR microprocessor. Descriptions of all of the C callable routines are provided below.

fpu_spi.h
This header file contains the pin assignments for the SPI interface and some definitions used by *fpu_spi.S* file.

fpu_i2c.S
This file contains all of the low-level support routines for interfacing with the uM-FPU V3.1 chip using the two-wire interface on the Atmel AVR microprocessor. Descriptions of all of the C callable routines are provided below.

fpu_i2c.h
This header file contains the pin assignments for the I^2C interface and some definitions used by *fpu_i2c.S* file.

fpuPrint.c
This file contains various print utility routines. Descriptions of all of the C callable routines are provided below.

uart.c
Contains support for used a serial port for *stdin* and *stdout*.

uart.h
This include file contains the provides function prototypes for all of the C callable routines in *uart.c*.

A program template files (called *template.c*) and various sample programs are provided with the support routines. The template file provides an example of the initializing the FPU, and can be used as a starting point for new programs.

fpu_reset

```
char fpu_reset(void);
```

To ensure that the Atmel AVR and the FPU are synchronized, a reset call must be done at the start of every program. The `fpu_reset` routine resets the FPU, confirms communications, and returns the sync character (0x5C) if the reset is successful. A sample reset call is included in the *template.c* file.

fpu_wait

```
void fpu_wait(void);
```

The FPU must have completed all instructions in the instruction buffer, and be ready to return data, before sending an instruction to read data from the FPU. The `fpu_wait` routine checks the ready status of the FPU and waits until it is ready. The print routines check the ready status, so calling `fpu_wait` before calling a print routine isn't required, but if your program reads directly from the FPU using one of the `fpu_write` functions, a call to `fpu_wait` is required prior to sending the read instruction. An example of reading a byte value is as follows:

```
fpu_wait();
fpu_write(LREADBYTE);
dataByte = fpu_readByte();
```

Description:

- wait for the FPU to be ready
- send the LREADBYTE instruction
- wait for the read setup delay
- read a byte value and store it in the variable dataByte

The uM-FPU V3.1 chip has a 256 byte instruction buffer. In most cases, data will be read back before 256 bytes have been sent to the FPU. If a long calculation is done that requires more than 256 bytes to be sent to the FPU, an `Fpu_Wait` call should be made at least every 256 bytes to ensure that the instruction buffer doesn't overflow.

fpu_write

```
void fpu_write(char bval1);
void fpu_write2(char bval1, char bval2);
void fpu_write3(char bval1, char bval2, char bval3);
void fpu_write4(char bval1, char bval2, char bval3, char bval4);
```

These routines are used to send instructions and data to the FPU. Each parameter specifies an 8-bit value to be sent to the FPU.

fpu_writeWord

```
void fpu_writeWord(int wval);
```

This routine is used to send a 16-bit value to the FPU.

fpu_writeFloat

```
void fpu_writeFloat(float fval);
```

This routine is used to send a 32-bit floating point value to the FPU

fpu_writeLong

```
void fpu_writeLong(long lval);
```

This routine is used to send a 32-bit long integer value to the FPU.

fpu_writeFloatReg

```
void fpu_writeFloatReg(float lval, char reg);
```

This routine is used to write a 32-bit floating point value to one of the FPU registers. The `reg` parameter specifies the register to write as follows:

0 to 127	register 0 to 127
-1	register A
-2	register X

fpu_writeLongReg

```
void fpu_writeLongReg(long lval, char reg);
```

This routine is used to write a 32-bit long integer value to one of the FPU registers. The `reg` parameter specifies the register to write as follows:

0 to 127	register 0 to 127
-1	register A
-2	register X

fpu_wrblk

```
void fpu_wrblk(char cnt, float *ptr);
```

This routine writes multiple values from a floating point array to the FPU registers. Register X is used to sequentially address the FPU registers and should be set prior to calling `fpu_wrblk`. The `fpu_wrblk` routine sends the `WRBLK` instruction, then sequentially writes the floating point values to the the FPU using the pointer specified. The valid range for `cnt` is 1 to 128.

fpu_writeString

```
void fpu_writeString(char *s);
```

This routine is used to write a zero-terminated string to the FPU.

fpu_read

```
char fpu_read(void);
```

This routine reads an 8-bit value from the FPU.

fpu_readWord

```
int fpu_readWord(void);
```

This routine reads an 16-bit value from the FPU.

fpu_readFloat

```
float fpu_readFloat(void);
```

This routine reads an 32-bit floating point value from the FPU.

fpu_readLong

```
long fpu_readLong(void);
```

This routine reads an 32-bit long integer value from the FPU.

fpu_readFloatReg

```
float fpu_readFloatReg(char reg);
```

This routine is used to read a 32-bit floating point value from one of the FPU registers. The `reg` parameter specifies the register to read as follows:

0 to 127	register 0 to 127
-1	register A
-2	register X

An `fpu_wait` call is done internally, before the read instruction is sent.

fpu_readLongReg

```
long fpu_readLongReg(char reg);
```

This routine is used to read a 32-bit long integer value from one of the FPU registers. The `reg` parameter specifies the register to read as follows:

0 to 127	register 0 to 127
-1	register A
-2	register X

An `fpu_wait` call is done internally, before the read instruction is sent.

fpu_rdblk

```
void fpu_rdblk(char cnt, float *ptr);
```

This routine reads multiple FPU register values into a floating point array. Register X is used to sequentially address the FPU registers and should be set prior to calling `fpu_rdblk`. The `fpu_rdblk` routine waits for the FPU to be ready, sends the `RDBLK` instruction, then reads the FPU registers sequentially and stores the floating point values using the pointer specified.

fpu_readStatus

```
char fpu_readStatus(void);
```

This routine reads the status byte from the FPU. An `fpu_wait` call is done internally, before the `READSTATUS` instruction is sent.

fpu_readString

```
void fpu_readString(char *s);
```

This routine is used to read a zero-terminated string from the FPU.

fpu_readDelay

```
void fpu_readDelay(void);
```

After a read instruction is sent, and before the first data is read, a setup delay is required to ensure that the FPU is ready to send data. Note: All of the `fpu_read` routines include an `fpu_readDelay` call, so this function is not not usually called directly an application program.

fpu_fcall

```
void fpu_fcall(char func);
```

This routine calls a user-defined function stored in Flash memory on the FPU. The function number is specified by the `func` parameter.

print_version

```
void print_version(void);
```

This routine sends the FPU version string to the serial port.

print_float

```
void print_float(char format);
```

The value in register A is sent to the serial port as a floating point string. The `format` parameter is used to specify the desired format. If the `format` parameter is zero, the length of the displayed value is variable and can be from 3 to 12 characters in length. Up to eight significant digits will be displayed if required, and very large or very small numbers are displayed in exponential notation. The special cases of NaN (Not a Number), +Infinity, -Infinity, and -0.0 are handled. Examples of the display format are as follows:

```
1.0          NaN          0.0
1.5e20       Infinity     -0.0
3.1415927    -Infinity    1.0
-52.333334   -3.5e-5      0.01
```

If the `format` parameter is non-zero, it determines the display format. The tens digit specifies the total number of characters to display and the ones digit specifies the number of digits after the decimal point. If the value is too large for the format specified, then asterisks will be displayed. If the number of digits after the decimal points is zero, no decimal point will be displayed. Examples of the display format are as follows:

Value in A register	format	Display format
123.567	61 (6.1)	123.6
123.567	62 (6.2)	123.57

```
123.567          42 (4.2)         *.**
0.9999           20 (2.0)          1
0.9999           31 (3.1)         1.0
```

print_long

```
void print_float(char format);
```

The value in register A is sent to the serial port as a signed long integer string. The `format` parameter is used to specify the desired format. If the `format` parameter is zero, the length of the displayed value is variable and the displayed value can range from 1 to 11 characters in length. Examples of the display format are as follows:

```
1
500000
-3598390
```

If the `format` parameter is non-zero, it determines the display format. A value between 0 and 15 specifies the width of the display field for a signed long integer. The number is displayed right justified. If 100 is added to the format value the value is displayed as an unsigned long integer. If the value is larger than the specified width, asterisks will be displayed. If the width is specified as zero, the length will be variable. Examples of the display format are as follows:

Value in register A	format	Display format
-1	10 (signed 10)	-1
-1	110 (unsigned 10)	4294967295
-1	4 (signed 4)	-1
-1	104 (unsigned 4)	****
0	4 (signed 4)	0
0	0 (unformatted)	0
1000	6 (signed 6)	1000

print_fpuString

```
void print_fpuString(char opcode);
```

This routine sends the contents of the FPU string buffer to the serial port.

print_CRLF

```
void print_CRLF(void);
```

This routine sends a carriage return and linefeed to the serial port.

uart_init

```
void uart_init(void);
```

This routine initializes the serial port and must be called at the start of the main program. A definition for the uart I/O stream must also be added to the main program. See the WinAVR standard library documentation for more details. e.g.

```
FILE uart_stream = FDEV_SETUP_STREM(uart_putchar, uart_getchar, _FDEV_SETUP_RW);
```

uart_putchar

```
int uart_putchar(char c, FILE *stream));
```

This routine provides *putchar* support for *stdout* which enables C standard output to be used for serial output.

uart_getchar

```
int uart_getchar(FILE *stream));
```

This routine provides *getchar* support for *stdin* which enables C standard input to be used for serial input.

Writing Data Values to the FPU

Most of the data read from devices connected to the Atmel AVR will return some type of integer value. There are several ways to load integer values to the FPU and convert them to 32-bit floating point or long integer values.

8-bit Integer to Floating Point

The FSETI, FADDI, FSUBI, FSUBRI, FMULI, FDIVI, FDIVRI, FPOWI, and FCMPI instructions read the byte following the opcode as an 8-bit signed integer, convert the value to floating point, and then perform the operation. It's a convenient way to work with constants or data values that are signed 8-bit values. The following example stores the lower 8 bits of variable dataByte to the Result register on the FPU.

```
fpu_write4(SELECTA, Result, FSETI, dataByte);
```

The LOADBYTE instruction reads the byte following the opcode as an 8-bit signed integer, converts the value to floating point, and stores the result in register 0.

The LOADUBYTE instruction reads the byte following the opcode as an 8-bit unsigned integer, converts the value to floating point, and stores the result in register 0.

16-bit Integer to Floating Point

The LOADWORD instruction reads the two bytes following the opcode as a 16-bit signed integer (MSB first), converts the value to floating point, and stores the result in register 0. The following example adds the lower 16 bits of variable dataWord to the Result register on the FPU.

```
fpu_write3(SELECTA, Result, LOADWORD);
fpu_writeWord(dataWord);
fpu_write(FADD0);
```

The LOADUWORD instruction reads the two bytes following the opcode as a 16-bit unsigned integer (MSB first), converts the value to floating point, and stores the result in register 0.

32-bit Floating Point to Floating point

The FWRITE, FWRITEA, FWRITEX, and FWRITE0 instructions interpret the four bytes following the opcode as a 32-bit floating point value and stores the value in the specified register. The fpu_writeFloat routine sends the four bytes of a floating point value. The following example sets register Angle to the value 20.0.

```
fpu_writeFloat2(FWRITE, Angle);
fpu_writeFloat(20.0);
```

The fpu_writeFloatReg routine can also be used to write a floating point value to a register. This routine sends the required FWRITE, FWRITEA, FWRITEX, or FWRITE0 instruction, then sends the floating point value.

```
fpu_writeFloatReg(20.0, Angle);
```

The fpu_wrblk routine writes multiple values from a floating point array to the FPU registers. Register X is used to sequentially address the FPU registers and should be set prior to calling fpu_wrblk. The fpu_wrblk routine sends the WRBLK instruction, then sequentially writes the floating point values to the the FPU using the pointer specified. The following example writes 16 floating values from array2 on the Atmel AVR to the FPU starting at register Array1.

```
fpu_write(SELECTX, Array1);
fpu_wrblk(16, array2);
```

ASCII string to Floating Point

The ATOF instruction is used to convert zero-terminated strings to floating point values. The instruction reads the bytes following the opcode (until a zero terminator is read), converts the string to floating point, and stores the result in register 0. The following example sets the register Angle to 1.5885.

```
fpu_write3(SELECTA, Angle, ATOF);
fpu_writeString("1.5885");
fpu_write(FSET0);
```

Note: The fpu_writeFloatReg routine described above is a better way to set a register to a floating point value.

8-bit Integer to Long Integer

The LSETI, LADDI, LSUBI, LMULI, LDIVI, LCMPI, LUDIVI, LUCMPI, and LTSTI instructions read the byte following the opcode as an 8-bit signed integer, convert the value to long integer, and then perform the operation. It's a convenient way to work with constants or data values that are signed 8-bit values. The following example adds the lower 8 bits of variable dataByte to the Total register on the FPU.

```
fpu_write4(SELECTA, Total, LADDI, dataByte);
```

The LONGBYTE instruction reads the byte following the opcode as an 8-bit signed integer, converts the value to long integer, and stores the result in register 0.

The LONGUBYTE instruction reads the byte following the opcode as an 8-bit unsigned integer, converts the value to long integer, and stores the result in register 0.

16-bit Integer to Long Integer

The LONGWORD instruction reads the two bytes following the opcode as a 16-bit signed integer (MSB first), converts the value to long integer, and stores the result in register 0. The following example adds the lower 16 bits of variable dataWord to the Total register on the FPU.

```
fpu_write3(SELECTA, Total, LOADWORD);
fpu_writeWord(dataWord);
fpu_write(LADD0);
```

The LONGUWORD instruction reads the two bytes following the opcode as a 16-bit unsigned integer (MSB first), converts the value to long integer, and stores the result in register 0.

32-bit integer to Long Integer

The LWRITE, LWRITEA, LWRITEX, and LWRITE0 instructions interpret the four bytes following the opcode as a 32-bit long integer value and stores the value in the specified register. The fpu_writeLong routine sends the four bytes of a long integer value. The following example sets register Total to the value 500000.

```
fpu_write(LWRITE, Total);
fpu_writeLong(500000);
```

The fpu_writeLongReg routine can also be used to write a long integer value to a register. This routine sends the required LWRITE, LWRITEA, LWRITEX, or LWRITE0 instruction, then sends the long integer value.

```
fpu_writeLongReg(500000, Angle);
```

ASCII string to Long Integer

The ATOL instruction is used to convert strings to long integer values. The instruction reads the bytes following the opcode (until a zero terminator is read), converts the string to long integer, and stores the result in register 0. The following example sets the register Total to 500000.

```
fpu_write3(SELECTA, Total, ATOL);
fpu_writeString("500000");
fpu_write(FSET0);
```

Note: The fpu_writeLongReg routine described above is a better way to set a register to a long integer value.

The fastest operations occur when the FPU registers are already loaded with values. In time critical portions of code floating point constants should be loaded beforehand to maximize the processing speed in the critical section. With 128 registers available on the FPU, it's often possible to pre-load all of the required constants. In non-critical sections of code, data and constants can be loaded as required.

Reading Data Values from the FPU

The uM-FPU V3.1 chip has a 256 byte instruction buffer which allows data transmission to continue while previous instructions are being executed. Before reading data, you must check to ensure that the previous commands have completed, and the FPU is ready to send data. The `fpu_wait` routine is used to wait until the FPU is ready, then a read command is sent and one of the read routines is called to read the data from the FPU.

8-bit Integer

The `LREADBYTE` instruction reads the lower 8 bits from register A. The following example stores the lower 8 bits of register A in variable `dataByte`.

```
fpu_wait();
fpu_write(LREADBYTE);
dataByte = fpu_read();
```

16-bit Integer

The `LREADWORD` instruction reads the lower 16 bits from register A. The following example stores the lower 16 bits of register A in variable `dataWord.`

```
fpu_wait();
fpu_write(LREADWORD);
dataWord = fpu_readWord();
```

32-bit Integer

The `LREAD`, `LREADA`, `LREADX`, and `LREAD0` instructions return a 32-bit long integer value from the specified register. The `fpu_readLong` routine reads a 32-bit long integer value. The following example reads the value in register `Total`.

```
fpu_wait();
fpu_write(LREAD, Total);
tmp = fpu_readLong();
```

The `fpu_readLongReg` routine can also be used to read a long integer value from a register. This routine waits for the FPU to be ready, sends the required `LREAD`, `LREADA`, `LREADX`, or `LREAD0` instruction, then reads the long integer value.

```
tmp = fpu_readLongReg(Total);
```

Long Integer to ASCII string

The `LTOA` instruction can be used to convert long integer values to an ASCII string. The `print_long` routine uses this instruction to read the value from register A and send the long integer string to the debug port.

The `fpu_readString` routine can be used to read a string from the FPU and store it at the pointer location specified.

```
fpu_write2(LTOA, 0);
fpu_wait();
fpu_write(READSTR);
fpu_readString(&strbuf);
```

Floating Point

The FREAD, FREADA, FREADX, and FREAD0 instructions return a 32-bit floating point value from the specified register. The fpu_readFloat routine reads a 32-bit floating point value. The following example reads the value in register Angle.

```
fpu_wait();
fpu_write(LREAD, Angle);
tmp = fpu_readFloat();
```

The fpu_readFloatReg routine can also be used to read a floating point value from a register. This routine waits for the FPU to be ready, sends the required FREAD, FREADA, FREADX, or FREAD0 instruction, then reads the floating point value.

```
tmp = fpu_readFloatReg(Angle);
```

The fpu_rdblk routine reads multiple FPU register values into a floating point array. Register X is used to sequentially address the FPU registers and should be set prior to calling fpu_rdblk. The fpu_rdblk routine waits for the FPU to be ready, sends the RDBLK instruction, then reads the FPU registers sequentially and stores the floating point values using the pointer specified. The following example reads 16 floating values, starting at FPU register Array1, and stores them in array2 on the Atmel AVR.

```
fpu_write(SELECTX, Array1);
fpu_rdblk(16, array2);
```

Floating Point to ASCII string

The FTOA instruction can be used to convert floating point values to an ASCII string. The print_float routine uses this instruction to read the value from register A and send the floating point string to the debug port.

The fpu_readString routine can be used to read a string from the FPU and store it at the pointer location specified.

```
fpu_write2(FTOA, 0);
fpu_wait();
fpu_write(READSTR);
fpu_readString(&strbuf);
```

Comparing and Testing Floating Point Values

Floating point values can be zero, positive, negative, infinite, or Not a Number (which occurs if an invalid operation is performed on a floating point value). The status byte is read using the `Fpu_ReadStatus` routine. It waits for the FPU to be ready before sending the `READSTATUS` instruction and reading the status byte. Bit definitions are provided for the FPU status bits as follows:

`STATUS_ZERO`	Zero status bit (0-not zero, 1-zero)
`STATUS_SIGN`	Sign status bit (0-positive, 1-negative)
`STATUS_NAN`	Not a Number status bit (0-valid number, 1-NaN)
`STATUS_INF`	Infinity status bit (0-not infinite, 1-infinite)

The `FSTATUS` and `FSTATUSA` instructions are used to set the status byte to the floating point status of the selected register. The following example checks the floating point status of register A:

```
fpu_write(FSTATUSA);
status = fpu_readStatus();
if (status & STATUS_SIGN) printf("Result is negative");
if (status & STATUS_ZERO) printf("Result is zero");
```

The `FCMP`, `FCMP0`, and `FCMPI` instructions are used to compare two floating point values. The status bits are set for the result of register A minus the operand (the selected registers are not modified). For example, to compare register A to the value 10.0:

```
fpu_write2(FCMPI, 10);
status = fpu_readStatus();
if (status & STATUS_ZERO)
  printf("Value1 = Value2");
else if (status & STATUS_SIGN)
  printf("Value1 < Value2");
else
  printf("Value1 > Value2");
```

The `FCMP2` instruction compares two floating point registers. The status bits are set for the result of the first register minus the second register (the selected registers are not modified). For example, to compare registers `Value1` and `Value2`:

```
fpu_write2(FCMP2, Value1, Value2);
status = fpu_readStatus();
```

Comparing and Testing Long Integer Values

A long integer value can be zero, positive, or negative. The status byte is read using the `fpu_readStatus` routine. It waits for the FPU to be ready before sending the `READSTATUS` instruction and reading the status byte. Bit definitions are provided for the FPU status bits as follows:

`STATUS_ZERO`	Zero status bit (0-not zero, 1-zero)
`STATUS_SIGN`	Sign status bit (0-positive, 1-negative)

The `LSTATUS` and `LSTATUSA` instructions are used to set the status byte to the long integer status of the selected register. The following example checks the long integer status of register A:

```
fpu_write(LSTATUSA);
status = fpu_readStatus();
if (status & STATUS_SIGN) printf("Result is negative");
if (status & STATUS_ZERO) printf("Result is zero");
```

The `LCMP`, `LCMP0`, and `LCMPI` instructions are used to do a signed comparison of two long integer values. The status bits are set for the result of register A minus the operand (the selected registers are not modified). For example, to compare register A to the value 10:

```
fpu_write2(LCMPI, 10);
status = fpu_readStatus();
if (status & STATUS_ZERO)
  printf("Value1 = Value2");
else if (status & STATUS_SIGN)
  printf("Value1 < Value2");
else
  printf("Value1 > Value2");
```

The `LCMP2` instruction does a signed compare of two long integer registers. The status bits are set for the result of the first register minus the second register (the selected registers are not modified). For example, to compare registers `Value1` and `Value2`:

```
fpu_write2(LCMP2, Value1, Value2);
status = fpu_readStatus();
```

The `LUCMP`, `LUCMP0`, and `LUCMPI` instructions are used to do an unsigned comparison of two long integer values. The status bits are set for the result of register A minus the operand (the selected registers are not modified).

The `LUCMP2` instruction does an unsigned compare of two long integer registers. The status bits are set for the result of the first register minus the second register (the selected registers are not modified).

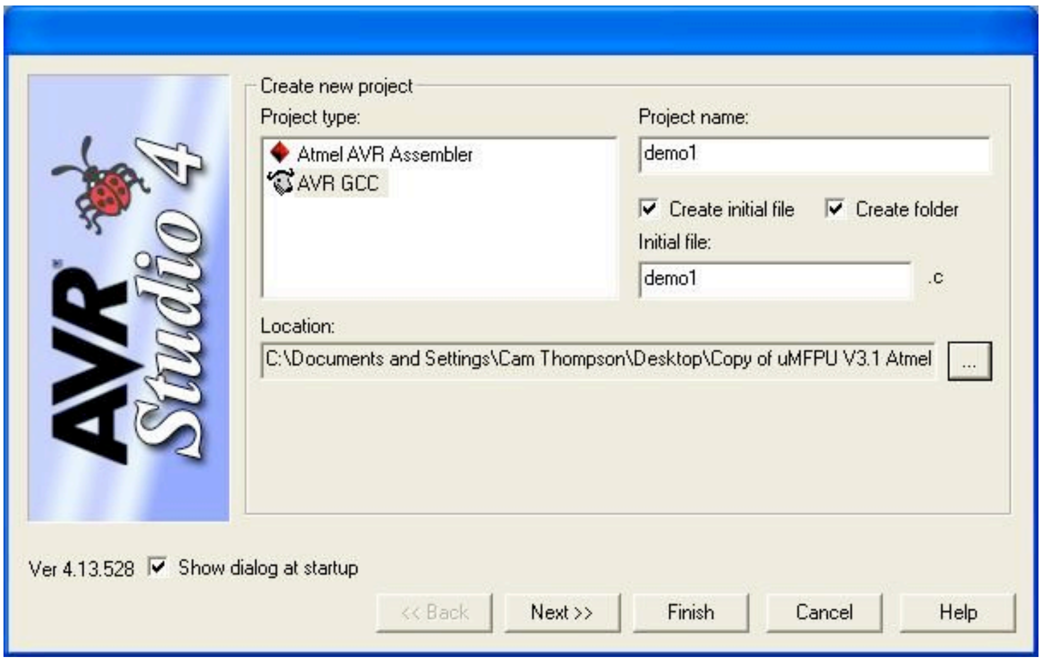
The `LTST`, `LTST0` and `LTSTI` instructions are used to do a bit-wise compare of two long integer values. The status bits are set for the logical `AND` of register A and the operand (the selected registers are not modified).

Creating a WinAVR project for uM-FPU 3.1

This section describes how to use the Atmel AVR Studio 4 to create a WinAVR project for the tutorial example (*demo1.c*). The major steps for configuring the project are described below. Once the project is configured, the program can be compiled, linked, and programmed into the Atmel AVR chip. Use Step 1A to create a new AVR GCC project, or step 1B to open an existing AVR GCC project. Several sample project files are included with the support software.

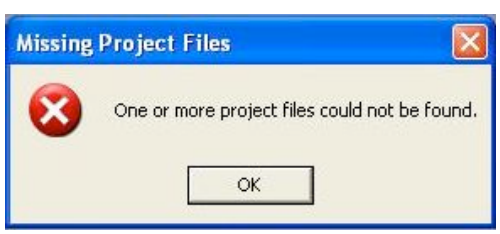
Step 1A - Create AVR GCC Project with AVR Studio 4

To create a new WinAVR project, select the *New Project* button from the AVR Studio 4 startup screen, or select *New Project* from the *Project* menu. The dialog shown below will appear. Specify the Location, Project name and Initial file for the project, then click the *Finish* button.



Step 1B - Open an existing AVR GCC Project with AVR Studio 4

To open an existing WinAVR project file, select the *Open* button from the AVR Studio 4 startup screen, or select *Open Project* from the *Project* menu, and select the WINAVR project file. The first time you open the sample project files included with the uM-FPU V3.1 support software, you will likely see the following dialog.



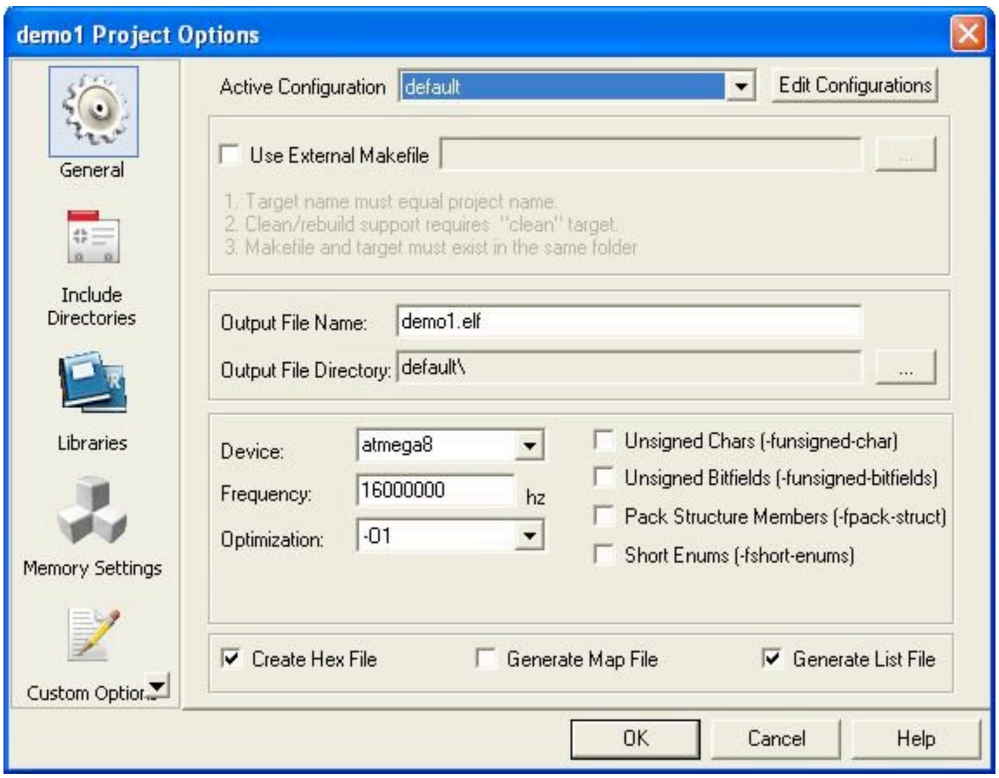
Click OK, and look at the Source Files in the panel on the left side of the AVR Studio window (see diagram in Step 3). For each Source File that has a red slash in the file icon, right-click the file name to get a pop-up menu, and select *Locate File...* to locate the file on your system. For the sample project files, additional source files are normally located in the directory one level up.

Step 2 - Set Device and Frequency

The device and operating frequency must be specified. These are used to determine timing parameters required by the FPU support routines and the UART. They are specified as follows:

- Select *Configuration Options* from the *Project* menu.
- Specify the Output File Name and Output File Directory.
- Specify the Device
- Specify the Frequency
- Specify the Optimization (-O1 is recommended)
- Specify other settings as desired

Note: the default baud rate for uart.c is 19200 baud. If you would like to use a different baud rate you can use Custom Options to set the value of the BAUD_RATE symbol (e.g. -DBAUD_RATE=9600)



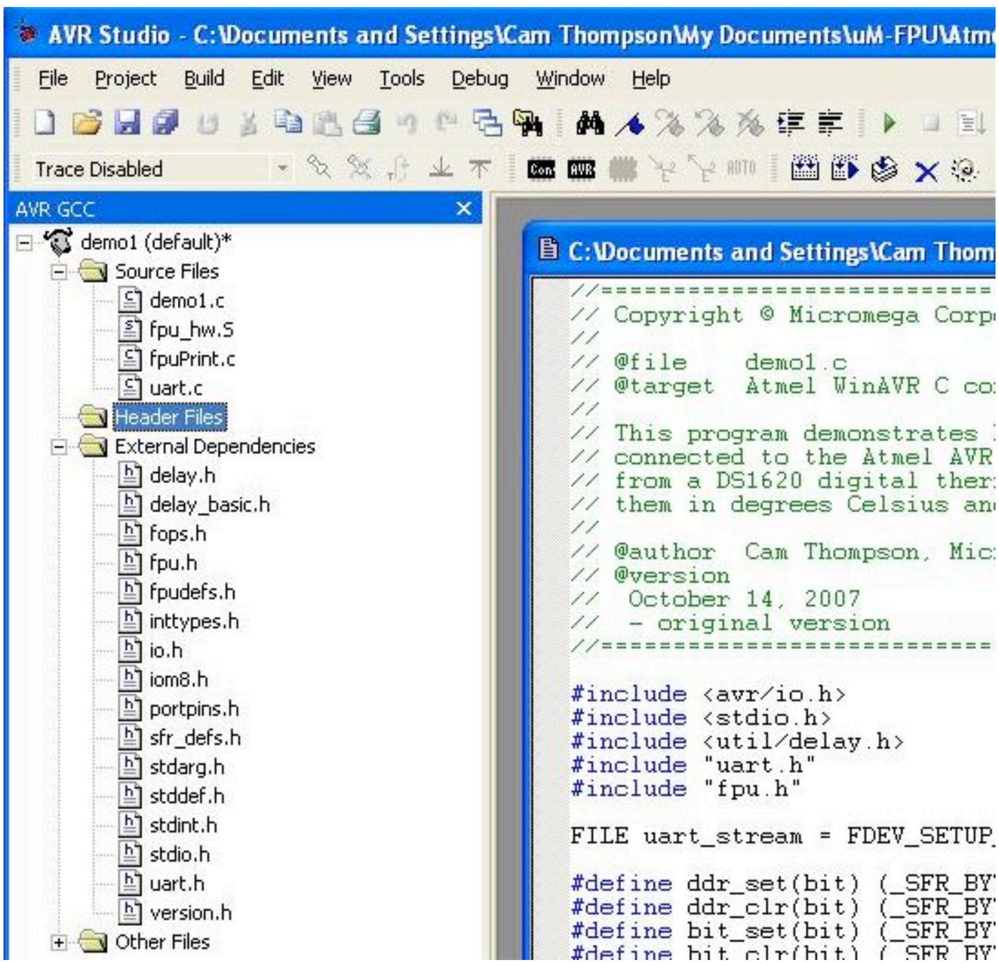
Note: Always confirm that the Device and Frequency are set correctly for the Atmel AVR microcontroller you are using.

Step 3 - Add Files to Project

This step is only required if you're creating a new project. Use the panel on the left side of the AVR Studio window to select the files needed for the project. For example:

demo1.c	the main routine
fpu_spi.S	uM-FPU V3.1 SPI support routines (or *fpu_i2c.S* for I^2C support routines)
fpuPrint.S	uM-FPU print routines
uart.c	support routines for using the UART as stdout

The files are added by right-clicking on the *Source Files* folder in the panel on the left side of the AVR Studio window to get a pop-up menu. Select *Add Existing Source File(s)…* or *Create New Source File* from the pop-up menu to add files. An example is shown below.



The following header files and include files are used by the FPU support software and must be in the search path:

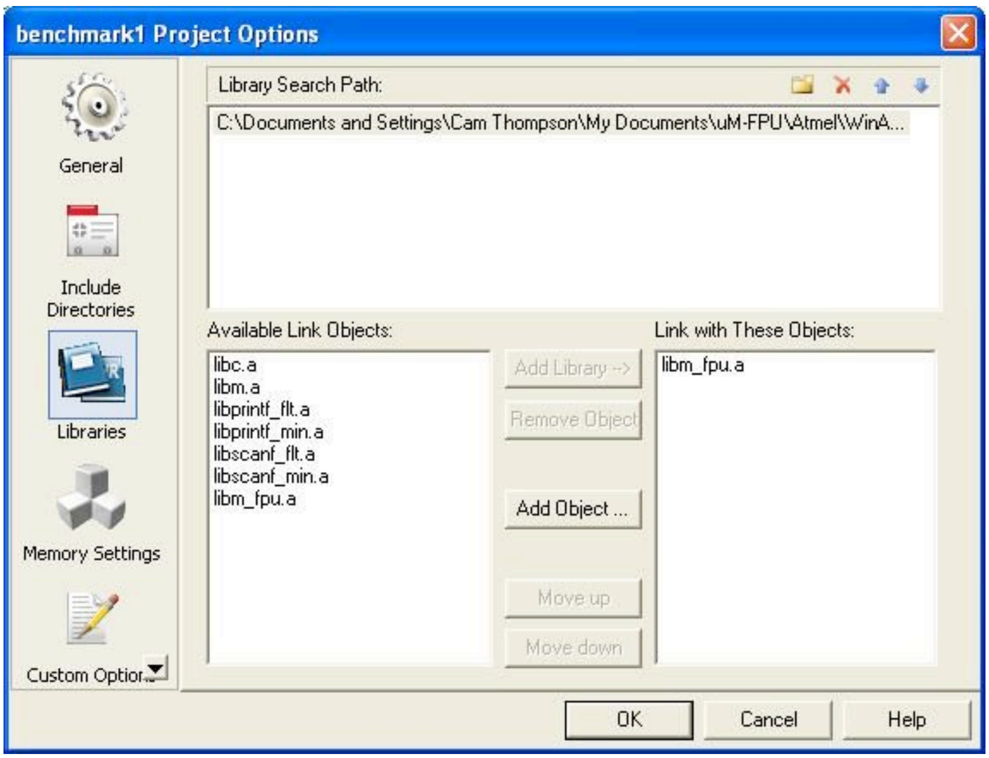
ctoasm.inc	WinAVR include file
fops.h	defines FPU opcodes, matrix operation codes and FFT operation codes
fp32def.h	WinAVR include file
fpu.h	defines function prototypes for the FPU support routines, includes fop.h
fpu_spi.h	defines the SPI pins used by the FPU support routines (or *fpu_i2c.h* if using an I^2C interface)
gasava.inc	WinAVR include file
macros.inc	WinAVR include file
uart.h	definitions for using UART as stdio

Step 4 - Replace the math library (optional)

The standard math routines provided by the WinAVR C compiler are contained in the *libm.a* library file. The *libm_fpu.a* library file provides replacement routines that use the uM-FPU V3.1 chip. The *libm_fpu.a* routines automatically send data to the FPU, perform the operation, and read the results back to the Atmel AVR. For simple operations (e.g. add, subtract, multiply and divide) the FPU routines may take slightly longer due to the data transfers required, but all of the more complex operations (e.g. sin, cos, tan, exp, log, etc.) are significantly faster. The FPU routines also use considerably less code space. To use the *libm_fpu.a* library,

- select *Configuration Options* under the *Project* menu,
- specify the Library Search path for the *libm_fpu.a* file
- add the *libm_fpu.a* file to the Link with These Objects panel
- click the *OK* button

An example is shown below.



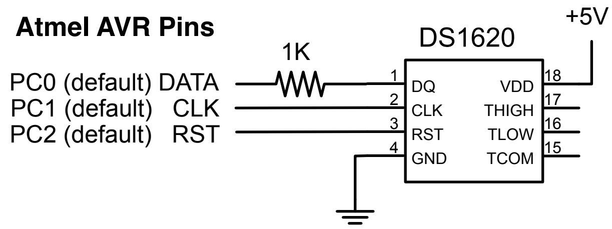
Further Information

The following documents are also available:

uM-FPU V3.1 Datasheet	provides hardware details and specifications
uM-FPU V3.1 Instruction Reference	provides detailed descriptions of each instruction
uM-FPU Application Notes	various application notes and examples

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

DS1620 Connections for Demo 1



Sample Code for Tutorial (demo1.c)

```
// This program demonstrates how to use the uM-FPU V3.1 floating point coprocessor
// connected to the Atmel AVR over an SPI interface. It takes temperature readings
// from a DS1620 digital thermometer, converts them to floating point and displays
// them in degrees Celsius and degrees Fahrenheit.

#include <avr/io.h>
#include <stdio.h>
#include <util/delay.h>
#include "uart.h"
#include "fpu.h"

FILE uart_stream = FDEV_SETUP_STREAM(uart_putchar, NULL, _FDEV_SETUP_WRITE);

#define ddr_set(bit) (_SFR_BYTE(DDRC) = _SFR_BYTE(DDRC) | _BV(bit))
#define ddr_clr(bit) (_SFR_BYTE(DDRC) = _SFR_BYTE(DDRC) & ~_BV(bit))
#define bit_set(bit) (_SFR_BYTE(PORTC) = _SFR_BYTE(PORTC) | _BV(bit))
#define bit_clr(bit) (_SFR_BYTE(PORTC) = _SFR_BYTE(PORTC) & ~_BV(bit))
#define bit_read(bit) (_SFR_BYTE(PINC) & _BV(bit))

#define DS_RST      0                          // DS1620 reset
#define DS_CLK      1                          // DS1620 clock
#define DS_DATA     2                          // DS1620 data

//------------------ local prototypes -----------------------------------------

void init_DS1620(void);
void write_DS1620(char bval);
int read_DS1620(void);
void delay(int msec);

//-------------------- uM-FPU register definitions ----------------------------

#define DegC  1                                // degrees Celsius
#define DegF  2                                // degrees Fahrenheit
#define F1_8  3                                // constant 1.8
```

```
//------------------ main ------------------------------------------------------

int main(void) {
    int rawTemp;

       // initialize the UART and set stdout
       uart_init();
       stdout = &uart_stream;

       printf("\r\n\r\nDemo1\r\n");

       // reset FPU and check synchronization
       if (fpu_reset() == SYNC_CHAR) {
              print_version();
              print_CRLF();
       }
       else {
              printf("uM-FPU not detected");
              return(0);
       }

       // initizlize the DS1620 chip
       init_DS1620();

       // store constant value (1.8)
       fpu_writeFloatReg(1.8, F1_8);

       // main sample loop
       while (1) {
              // get temperature reading from DS1620
              rawTemp = read_DS1620();

              // send to FPU and convert to floating point
              // divide by 2 to get degrees Celsius
              fpu_write3(SELECTA, DegC, LOADWORD);
              fpu_writeWord(rawTemp);
              fpu_write3(FSET0, FDIVI, 2);

              // degF = degC * 1.8 + 32
              fpu_write4(SELECTA, DegF, FSET, DegC);
              fpu_write4(FMUL, F1_8, FADDI, 32);

              // display degrees Celsius
              printf("\r\nDegrees C: ");
              fpu_write2(SELECTA, DegC);
              print_float(51);

              // display degrees Fahrenheit
              printf("\r\nDegrees F: ");
              fpu_write2(SELECTA, DegF);
              print_float(51);
              print_CRLF();

              // delay 2 seconds, then get the next reading
              delay(2000);
       }
}
```

```
//-------------------- init_DS1620 -------------------------------------------

void init_DS1620(void) {

	ddr_set(DS_RST);
	ddr_set(DS_CLK);
	ddr_set(DS_DATA);

	bit_clr(DS_RST);
	bit_set(DS_CLK);
	delay(100);

	bit_set(DS_RST);
	write_DS1620(0x0C);
	write_DS1620(0x02);
	bit_clr(DS_RST);
	delay(100);

	bit_set(DS_RST);
	write_DS1620(0xEE);
	bit_clr(DS_RST);
	delay(1000);
}

//-------------------- write_DS1620 ------------------------------------------

void write_DS1620(char bval) {
	int i;

	for (i = 0; i < 8; i++) {
		bit_clr(DS_CLK);
		if (bval & 0x01)
			bit_set(DS_DATA);
		else
			bit_clr(DS_DATA);
		bval >>= 1;
		bit_set(DS_CLK);
	}
}

//-------------------- read_DS1620 -------------------------------------------

int read_DS1620(void) {
	int tmp, i;

	bit_set(DS_RST);
	write_DS1620(0xAA);

	ddr_clr(DS_DATA);
	tmp = 0;
	for (i = 0; i < 9; i++) {
		bit_clr(DS_CLK);
		tmp >>= 1;
		if (bit_read(DS_DATA)) tmp |= 0x0100;
		bit_set(DS_CLK);
	}
	bit_clr(DS_CLK);
	ddr_set(DS_DATA);
```

```
	bit_clr(DS_RST);
	if (tmp & 0x0100) tmp |= 0xFF00;
	return tmp;
}

//-------------------- delay --------------------------------------------------

void delay(int msec) {
	int i;

	for (i = 0; i < msec; i += 10) {
		_delay_ms(10);
	}
}
```

Appendix A
uM-FPU V3.1 Instruction Summary

Instruction	Opcode	Arguments	Returns	Description
NOP	00			No Operation
SELECTA	01	nn		Select register A
SELECTX	02	nn		Select register X
CLR	03	nn		reg[nn] = 0
CLRA	04			reg[A] = 0
CLRX	05			reg[X] = 0, X = X + 1
CLR0	06			reg[nn] = 0
COPY	07	mm,nn		reg[nn] = reg[mm]
COPYA	08	nn		reg[nn] = reg[A]
COPYX	09	nn		reg[nn] = reg[X], X = X + 1
LOAD	0A	nn		reg[0] = reg[nn]
LOADA	0B			reg[0] = reg[A]
LOADX	0C			reg[0] = reg[X], X = X + 1
ALOADX	0D			reg[A] = reg[X], X = X + 1
XSAVE	0E	nn		reg[X] = reg[nn], X = X + 1
XSAVEA	0F			reg[X] = reg[A], X = X + 1
COPY0	10	nn		reg[nn] = reg[0]
COPYI	11	bb,nn		reg[nn] = long(unsigned byte bb)
SWAP	12	nn,mm		Swap reg[nn] and reg[mm]
SWAPA	13	nn		Swap reg[A] and reg[nn]
LEFT	14			Left parenthesis
RIGHT	15			Right parenthesis
FWRITE	16	nn,b1,b2,b3,b4		Write 32-bit floating point to reg[nn]
FWRITEA	17	b1,b2,b3,b4		Write 32-bit floating point to reg[A]
FWRITEX	18	b1,b2,b3,b4		Write 32-bit floating point to reg[X]
FWRITE0	19	b1,b2,b3,b4		Write 32-bit floating point to reg[0]
FREAD	1A	nn	b1,b2,b3,b4	Read 32-bit floating point from reg[nn]
FREADA	1B		b1,b2,b3,b4	Read 32-bit floating point from reg[A]
FREADX	1C		b1,b2,b3,b4	Read 32-bit floating point from reg[X]
FREAD0	1C		b1,b2,b3,b4	Read 32-bit floating point from reg[0]
ATOF	1E	aa…00		Convert ASCII to floating point
FTOA	1F	bb		Convert floating point to ASCII
FSET	20	nn		reg[A] = reg[nn]
FADD	21	nn		reg[A] = reg[A] + reg[nn]
FSUB	22	nn		reg[A] = reg[A] - reg[nn]
FSUBR	23	nn		reg[A] = reg[nn] - reg[A]
FMUL	24	nn		reg[A] = reg[A] * reg[nn]
FDIV	25	nn		reg[A] = reg[A] / reg[nn]
FDIVR	26	nn		reg[A] = reg[nn] / reg[A]
FPOW	27	nn		reg[A] = reg[A] ** reg[nn]
FCMP	28	nn		Compare reg[A], reg[nn], Set floating point status
FSET0	29			reg[A] = reg[0]
FADD0	2A			reg[A] = reg[A] + reg[0]

FSUB0	2B			reg[A] = reg[A] - reg[0]
FSUBR0	2C			reg[A] = reg[0] - reg[A]
FMUL0	2D			reg[A] = reg[A] * reg[0]
FDIV0	2E			reg[A] = reg[A] / reg[0]
FDIVR0	2F			reg[A] = reg[0] / reg[A]
FPOW0	30			reg[A] = reg[A] ** reg[0]
FCMP0	31			Compare reg[A], reg[0], Set floating point status
FSETI	32	bb		reg[A] = float(bb)
FADDI	33	bb		reg[A] = reg[A] - float(bb)
FSUBI	34	bb		reg[A] = reg[A] - float(bb)
FSUBRI	35	bb		reg[A] = float(bb) - reg[A]
FMULI	36	bb		reg[A] = reg[A] * float(bb)
FDIVI	37	bb		reg[A] = reg[A] / float(bb)
FDIVRI	38	bb		reg[A] = float(bb) / reg[A]
FPOWI	39	bb		reg[A] = reg[A] ** bb
FCMPI	3A	bb		Compare reg[A], float(bb), Set floating point status
FSTATUS	3B	nn		Set floating point status for reg[nn]
FSTATUSA	3C			Set floating point status for reg[A]
FCMP2	3D	nn,mm		Compare reg[nn], reg[mm] Set floating point status
FNEG	3E			reg[A] = -reg[A]
FABS	3F			reg[A] = I reg[A] I
FINV	40			reg[A] = 1 / reg[A]
SQRT	41			reg[A] = sqrt(reg[A])
ROOT	42	nn		reg[A] = root(reg[A], reg[nn])
LOG	43			reg[A] = log(reg[A])
LOG10	44			reg[A] = log10(reg[A])
EXP	45			reg[A] = exp(reg[A])
EXP10	46			reg[A] = exp10(reg[A])
SIN	47			reg[A] = sin(reg[A])
COS	48			reg[A] = cos(reg[A])
TAN	49			reg[A] = tan(reg[A])
ASIN	4A			reg[A] = asin(reg[A])
ACOS	4B			reg[A] = acos(reg[A])
ATAN	4C			reg[A] = atan(reg[A])
ATAN2	4D	nn		reg[A] = atan2(reg[A], reg[nn])
DEGREES	4E			reg[A] = degrees(reg[A])
RADIANS	4F			reg[A] = radians(reg[A])
FMOD	50	nn		reg[A] = reg[A] MOD reg[nn]
FLOOR	51			reg[A] = floor(reg[A])
CEIL	52			reg[A] = ceil(reg[A])
ROUND	53			reg[A] = round(reg[A])
FMIN	54	nn		reg[A] = min(reg[A], reg[nn])
FMAX	55	nn		reg[A] = max(reg[A], reg[nn])
FCNV	56	bb		reg[A] = conversion(bb, reg[A])
FMAC	57	nn,mm		reg[A] = reg[A] + (reg[nn] * reg[mm])
FMSC	58	nn,mm		reg[A] = reg[A] - (reg[nn] * reg[mm])

LOADBYTE	59	bb		reg[0] = float(signed bb)
LOADUBYTE	5A	bb		reg[0] = float(unsigned byte)
LOADWORD	5B	b1,b2		reg[0] = float(signed b1*256 + b2)
LOADUWORD	5C	b1,b2		reg[0] = float(unsigned b1*256 + b2)
LOADE	5D			reg[0] = 2.7182818
LOADPI	5E			reg[0] = 3.1415927
LOADCON	5F	bb		reg[0] = float constant(bb)
FLOAT	60			reg[A] = float(reg[A])
FIX	61			reg[A] = fix(reg[A])
FIXR	62			reg[A] = fix(round(reg[A]))
FRAC	63			reg[A] = fraction(reg[A])
FSPLIT	64			reg[A] = integer(reg[A]), reg[0] = fraction(reg[A])
SELECTMA	65	nn,b1,b2		Select matrix A
SELECTMB	66	nn,b1,b2		Select matrix B
SELECTMC	67	nn,b1,b2		Select matrix C
LOADMA	68	b1,b2		reg[0] = Matrix A[bb, bb]
LOADMB	69	b1,b2		reg[0] = Matrix B[bb, bb]
LOADMC	6A	b1,b2		reg[0] = Matrix C[bb, bb]
SAVEMA	6B	b1,b2		Matrix A[bb, bb] = reg[A]
SAVEMB	6C	b1,b2		Matrix B[bb, bb] = reg[A]
SAVEMC	6D	b1,b2		Matrix C[bb, bb] = reg[A]
MOP	6E	bb		Matrix/Vector operation
FFT	6F	bb		Fast Fourier Transform
WRBLK	70	tc t1…tn		Write multiple 32-bit values
RDBLK	71	tc	t1…tn	Read multiple 32-bit values
LOADIND	7A	nn		reg[0] = reg[reg[nn]]
SAVEIND	7B	nn		reg[reg[nn]] = reg[A]
INDA	7C	nn		Select register A using value in reg[nn]
INDX	7D	nn		Select register X using value in reg[nn]
FCALL	7E	bb		Call user-defined function in Flash
EECALL	7F	bb		Call user-defined function in EEPROM
RET	80			Return from user-defined function
BRA	81	bb		Unconditional branch
BRA	82	cc, bb		Conditional branch
JMP	83	b1, b2		Unconditional jump
JMP	84	cc, b1, b2		Conditional jump
TABLE	85	tc,t0…tn		Table lookup
FTABLE	86	cc,tc,t0…tn		Floating point reverse table lookup
LTABLE	87	cc,tc,t0…tn		Long integer reverse table lookup
POLY	88	tc,t0…tn		reg[A] = nth order polynomial
GOTO	89	nn		Computed GOTO
RET	8A	cc		Conditional return from user-defined function
LWRITE	90	nn,b1,b2,b3,b4		Write 32-bit long integer to reg[nn]
LWRITEA	91	b1,b2,b3,b4		Write 32-bit long integer to reg[A]
LWRITEX	92	b1,b2,b3,b4		Write 32-bit long integer to reg[X], X = X + 1
LWRITE0	93	b1,b2,b3,b4		Write 32-bit long integer to reg[0]

LREAD	94	nn	b1,b2,b3,b4	Read 32-bit long integer from reg[nn]
LREADA	95		b1,b2,b3,b4	Read 32-bit long value from reg[A]
LREADX	96		b1,b2,b3,b4	Read 32-bit long integer from reg[X], X = X + 1
LREAD0	97		b1,b2,b3,b4	Read 32-bit long integer from reg[0]
LREADBYTE	98		bb	Read lower 8 bits of reg[A]
LREADWORD	99		b1,b2	Read lower 16 bits reg[A]
ATOL	9A	aa…00		Convert ASCII to long integer
LTOA	9B	bb		Convert long integer to ASCII
LSET	9C	nn		reg[A] = reg[nn]
LADD	9D	nn		reg[A] = reg[A] + reg[nn]
LSUB	9E	nn		reg[A] = reg[A] - reg[nn]
LMUL	9F	nn		reg[A] = reg[A] * reg[nn]
LDIV	A0	nn		reg[A] = reg[A] / reg[nn] reg[0] = remainder
LCMP	A1	nn		Signed compare reg[A] and reg[nn], Set long integer status
LUDIV	A2	nn		reg[A] = reg[A] / reg[nn] reg[0] = remainder
LUCMP	A3	nn		Unsigned compare reg[A] and reg[nn], Set long integer status
LTST	A4	nn		Test reg[A] AND reg[nn], Set long integer status
LSET0	A5			reg[A] = reg[0]
LADD0	A6			reg[A] = reg[A] + reg[0]
LSUB0	A7			reg[A] = reg[A] - reg[0]
LMUL0	A8			reg[A] = reg[A] * reg[0]
LDIV0	A9			reg[A] = reg[A] / reg[0] reg[0] = remainder
LCMP0	AA			Signed compare reg[A] and reg[0], set long integer status
LUDIV0	AB			reg[A] = reg[A] / reg[0] reg[0] = remainder
LUCMP0	AC			Unsigned compare reg[A] and reg[0], Set long integer status
LTST0	AD			Test reg[A] AND reg[0], Set long integer status
LSETI	AE	bb		reg[A] = long(bb)
LADDI	AF	bb		reg[A] = reg[A] + long(bb)
LSUBI	B0	bb		reg[A] = reg[A] - long(bb)
LMULI	B1	bb		reg[A] = reg[A] * long(bb)
LDIVI	B2	bb		reg[A] = reg[A] / long(bb) reg[0] = remainder
LCMPI	B3	bb		Signed compare reg[A] - long(bb), Set long integer status
LUDIVI	B4	bb		reg[A] = reg[A] / unsigned long(bb) reg[0] = remainder
LUCMPI	B5	bb		Unsigned compare reg[A] and long(bb), Set long integer status

LTSTI	B6	bb		Test reg[A] AND long(bb), Set long integer status
LSTATUS	B7	nn		Set long integer status for reg[nn]
LSTATUSA	B8			Set long integer status for reg[A]
LCMP2	B9	nn,mm		Signed long compare reg[nn], reg[mm] Set long integer status
LUCMP2	BA	nn,mm		Unsigned long compare reg[nn], reg[mm] Set long integer status
LNEG	BB			reg[A] = -reg[A]
LABS	BC			reg[A] = I reg[A] I
LINC	BD	nn		reg[nn] = reg[nn] + 1, set status
LDEC	BE	nn		reg[nn] = reg[nn] - 1, set status
LNOT	BF			reg[A] = NOT reg[A]
LAND	C0	nn		reg[A] = reg[A] AND reg[nn]
LOR	C1	nn		reg[A] = reg[A] OR reg[nn]
LXOR	C2	nn		reg[A] = reg[A] XOR reg[nn]
LSHIFT	C3	nn		reg[A] = reg[A] shift reg[nn]
LMIN	C4	nn		reg[A] = min(reg[A], reg[nn])
LMAX	C5	nn		reg[A] = max(reg[A], reg[nn])
LONGBYTE	C6	bb		reg[0] = long(signed byte bb)
LONGUBYTE	C7	bb		reg[0] = long(unsigned byte bb)
LONGWORD	C8	b1,b2		reg[0] = long(signed b1*256 + b2)
LONGUWORD	C9	b1,b2		reg[0] = long(unsigned b1*256 + b2)
SETSTATUS	CD	ss		Set status byte
SEROUT	CE	bb bb bd bb aa…00		Serial output
SERIN	CF	bb		Serial input
SETOUT	D0	bb		Set OUT1 and OUT2 output pins
ADCMODE	D1	bb		Set A/D trigger mode
ADCTRIG	D2			A/D manual trigger
ADCSCALE	D3	ch		ADCscale[ch] = B
ADCLONG	D4	ch		reg[0] = ADCvalue[ch]
ADCLOAD	D5	ch		reg[0] = float(ADCvalue[ch]) * ADCscale[ch]
ADCWAIT	D6			wait for next A/D sample
TIMESET	D7			time = reg[0]
TIMELONG	D8			reg[0] = time (long integer)
TICKLONG	D9			reg[0] = ticks (long integer)
EESAVE	DA	mm,nn		EEPROM[nn] = reg[mm]
EESAVEA	DB	nn		EEPROM[nn] = reg[A]
EELOAD	DC	mm,nn		reg[mm] = EEPROM[nn]
EELOADA	DD	nn		reg[A] = EEPROM[nn]
EEWRITE	DE	nn,bc,b1…bn		Store bytes in EEPROM
EXTSET	E0			external input count = reg[0]
EXTLONG	E1			reg[0] = external input counter
EXTWAIT	E2			wait for next external input
STRSET	E3	aa…00		Copy string to string buffer
STRSEL	E4	bb, bb		Set selection point
STRINS	E5	aa…00		Insert string at selection point

STRCMP	E6	aa…00		Compare string with string buffer
STRFIND	E7	aa…00		Find string and set selection point
STRFCHR	E8	aa…00		Set field separators
STRFIELD	E9	bb		Find field and set selection point
STRTOF	EA			Convert selected string to floating point
STRTOL	EB			Convert selected string to long integer
READSEL	EC		aa…00	Read selected string
STRBYTE	ED	bb		Insert byte at selection point
STRINC	EE			Increment string selection point
STRDEC	EF			Decrement string selection point
SYNC	F0		5C	Get synchronization byte
READSTATUS	F1		ss	Read status byte
READSTR	F2		aa…00	Read string from string buffer
VERSION	F3			Copy version string to string buffer
IEEEMODE	F4			Set IEEE mode (default)
PICMODE	F5			Set PIC mode
CHECKSUM	F6			Calculate checksum for uM-FPU code
BREAK	F7			Debug breakpoint
TRACEOFF	F8			Turn debug trace off
TRACEON	F9			Turn debug trace on
TRACESTR	FA	aa…00		Send string to debug trace buffer
TRACEREG	FB	nn		Send register value to trace buffer
READVAR	FC	nn		Read internal register value
RESET	FF			Reset (9 consecutive FF bytes cause a reset, otherwise it is a NOP)

Notes:

Opcode	Instruction opcode in hexadecimal
Arguments	Additional data required by instruction
Returns	Data returned by instruction
nn	register number (0-127)
mm	register number (0-127)
fn	function number (0-63)
bb	8-bit value
b1,b2	16-bit value (b1 is MSB)
b1,b2,b3,b4	32-bit value (b1 is MSB)
b1…bn	string of 8-bit bytes
ss	Status byte
bd	baud rate and debug mode
cc	Condition code
ee	EEPROM address slot (0-255)
ch	A/D channel number
bc	Byte count
tc	32-bit value count
t1…tn	String of 32-bit values
aa…00	Zero terminated ASCII string

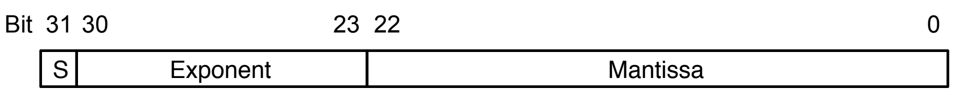
Appendix B
Floating Point Numbers

Floating point numbers can store both very large and very small values by “floating” the window of precision to fit the scale of the number. Fixed point numbers can’t handle very large or very small numbers and are prone to loss of precision when numbers are divided. The representation of floating point numbers used by the uM-FPU V3.1 is defined by the 32-bit IEEE 754 standard. The number of significant digits for a 32-bit floating point number is slightly more than 7 digits, and the range of values that can be handled is approximately $\pm 10^{38.53}$.

32-bit IEEE 754 Floating Point Representation

IEEE 754 floating point numbers have three components: a sign, exponent, the mantissa. The sign indicates whether the number is positive or negative. The exponent has an implied base of two and a bias value. The mantissa represents the fractional part of the number.

The 32-bit IEEE 754 representation is as follows:



Sign Bit (bit 31)
The sign bit is 0 for a positive number and 1 for a negative number.

Exponent (bits 30-23)
The exponent field is an 8-bit field that stores the value of the exponent with a bias of 127 that allows it to represent both positive and negative exponents. For example, if the exponent field is 128, it represents an exponent of one (128 – 127 = 1). An exponent field of all zeroes is used for denormalized numbers and an exponent field of all ones is used for the special numbers +infinity, -infinity and Not-a-Number (described below).

Mantissa (bits 30-23)
The mantissa is a 23-bit field that stores the precision bits of the number. For normalized numbers there is an implied leading bit equal to one.

Special Values

Zero

A zero value is represented by an exponent of zero and a mantissa of zero. Note that +0 and –0 are distinct values although they compare as equal.

Denormalized

If an exponent is all zeros, but the mantissa is non-zero the value is a denormalized number. Denormalized numbers are used to represent very small numbers and provide for an extended range and a graceful transition towards zero on underflows. Note: The uM-FPU does not support operations using denormalized numbers.

Infinity

The values +infinity and –infinity are denoted with an exponent of all ones and a fraction of all zeroes. The sign bit distinguishes between +infinity and –infinity. This allows operations to continue past an overflow. A nonzero number divided by zero will result in an infinity value.

Not A Number (NaN)

The value NaN is used to represent a value that does not represent a real number. An operation such as zero divided by zero will result in a value of NaN. The NaN value will flow through any mathematical operation. Note: The uM-FPU initializes all of its registers to NaN at reset, therefore any operation that uses a register that has not been previously set with a value will produce a result of NaN.

Some examples of 32-bit IEEE 754 floating point values displayed as displayed as 32-bit hexadecimal constants are as follows:

```
0x00000000     // 0.0
0x3DCCCCCD     // 0.1
0x3F000000     // 0.5
0x3F400000     // 0.75
0x3F7FF972     // 0.9999
0x3F800000     // 1.0
0x40000000     // 2.0
0x402DF854     // 2.7182818 (e)
0x40490FDB     // 3.1415927 (pi)
0x41200000     // 10.0
0x42C80000     // 100.0
0x447A0000     // 1000.0
0x449A522B     // 1234.5678
0x49742400     // 1000000.0
0x80000000     // -0.0
0xBF800000     // -1.0
0xC1200000     // -10.0
0xC2C80000     // -100.0
0x7FC00000     // NaN (Not-a-Number)
0x7F800000     // +inf
0xFF800000     // -inf
```