Micromega Corporation

Using uM-FPU64 with Parallax Propeller

Introduction

This document describes how to use the uM-FPU64 floating point coprocessor (FPU) with the Parallax Propeller microcontroller. For a full description of the uM-FPU64 chip, please refer to the *uM-FPU64 Datasheet* and *uM-FPU64 Instruction Reference*. Application notes are also available on the Micromega website.

The uM-FPU64 provides advanced fixed point and floating point math capabilities, that can be used to augment the capabilities of the Propeller microcontroller. Many GPS navigational calculations and data transformation for MEMS-based sensors require 64-bit precision to achieve the desired amount of accuracy. The uM-FPU64 coprocessor supports both 32-bit and 64-bit floating point numbers which provides the added precision needed for these demanding applications, and can free up the resources of the Propeller microcontroller for other tasks.

The original Propeller object for uM-FPU64 was developed by Istvan Kovesdi and is published in the Math section of the Parallax Propeller Object Exchange, along with numerous application examples. A new object, *Fpu64Driver.Spin,* has been developed to minimize transfer times and significantly improve execution times. It's compatible with the uM-FPU64 IDE (Integrated Development Environment) which makes it easy to create, debug and test code for the uM-FPU64. Code written for the IDE's compiler or assembler can be compiled to generate Spin code targeted for the Propellor microcontroller, or FPU functions stored in FPU Flash memory. The IDE provides support for editing code, compiling, tracing code execution, setting breakpoints, examining registers and programming user-defined functions in Flash memory.

The features of the new *Fpu64Driver.Spin* object include:

- SPI routines use COG counters to transmit data at 10 MHz.
- Spin call overhead is kept to a minimum.
- all FPU timing specifications are optimized
- Spin setup time for the next call is overlapped with the execution time of the current call.
- Spin code, Cog code and uM-FPU64 code can all run simultaneously, taking advantage of the multiple processors.

uM-FPU64 Floating Point Coprocessor Features

64-bit and 32-bit Floating Point
A comprehensive set of 64-bit and 32-bit floating point operations are provided. See the uM-FPU64 datasheet for details.

64-bit and 32-bit Integer
A comprehensive set of 64-bit and 32-bit integer operations are provided. See the uM-FPU64 datasheet for details.

Local Device Support
Local peripheral device support includes: RAM, 1-Wire, I^{2}C, SPI, UART, counter, servo controller, LCD, and SD card FAT16/FAT32 devices. The uM-FPU64 can act as a complete subsystem controller for GPS, sensor networks, robotic subsystems, IMUs, and other applications. Local devices are assigned to digital I/O pins at run-time, and controlled with the DEVIO instruction.

User-defined Functions
User-defined functions can be stored in Flash memory. Flash functions are programmed through the SERIN/SEROUT pins using the uM-FPU64 IDE. A high level language is supported, including control statements and conditional execution.

Matrix Operations
A matrix can be defined as any set of sequential registers. The MOP instruction provides scalar operations, element-wise operations, matrix multiply, inverse, determinant, count, sum, average, min, max, copy and set operations.

FFT Instruction
Provides support for Fast Fourier Transforms. Used as a single instruction for data sets that fit in the available registers, or as a multi-pass instruction for working with larger data sets.

Serial Input / Output
When not required for debugging, the SERIN and SEROUT pins can be used for serial I/O. A second asynchronous serial port, with hardware flow control, is available as a local device using the DEVIO instruction.

NMEA Sentence Parsing
The serial input can be set to scan for valid NMEA sentences with optional checksum. Multiple sentences can be buffered for further processing.

String Handling
String instructions are provided to insert and append substrings, search for fields and substrings, convert from floating point or long integer to a substring, or convert from a substring to floating point or long integer. For example, the string instructions could be used to parse a GPS NMEA sentence, or format multiple numbers in an output string.

Table Lookup Instructions
Instructions are provided to load 32-bit values from a table or find the index of a floating point or long integer table entry that matches a specified condition.

A/D Conversion
Multiple 12-bit A/D channels are provided (six on 28-pin device, nine on 44-pin device). The A/D conversion can be triggered manually, through an external input, or from a built-in timer. The A/D values can be read as raw values or automatically scaled to a floating point value. Data rates of up to 10,000 samples per second are supported.

Real-Time Clock and Timers
A built-in real-time clock is provided, for scheduling events or creating date/time stamps. Timers can be used to trigger the A/D conversion, or to track elapsed time.

Foreground/Background Processing
Event driven foreground/background processing can be used to provide independent monitoring of local peripherals. The microcontroller communicates with the foreground, while background processes can be used to monitor local device activity.

Low Power Modes
When the uM-FPU64 chip is not busy it automatically enters a power saving mode. It can also be configured to enter a sleep mode which turns the device off while preserving register contents. In sleep mode the uM-FPU64 chip consumes negligible power.

Firmware Upgrades
When updates become available, the uM-FPU64 firmware can be upgraded in the field using the uM-FPU64 IDE software.

Installing uM-FPU64 Support Software for Propeller

The support software for using the uM-FPU64 chip with the Parallax Propeller microcontroller is contained in a file called *uM-FPU64-Propeller.zip*. Download and unzip the file. Two Spin files, *Fpu64Driver.spin* and *Fpu64_Opcodes.spin* provide the uM-FPU64 interface. Some sample code files and this document are also included.

The *Fpu64Driver.spin* object provides all of the support methods for interfacing to the uM-FPU64 chip.

The *Fpu64_Opcodes.spin* object provides symbol definitions for all uM-FPU64 opcodes, and additional symbols for some of the action codes and options required by various FPU instructions.

Connecting the uM-FPU64 chip the Propeller

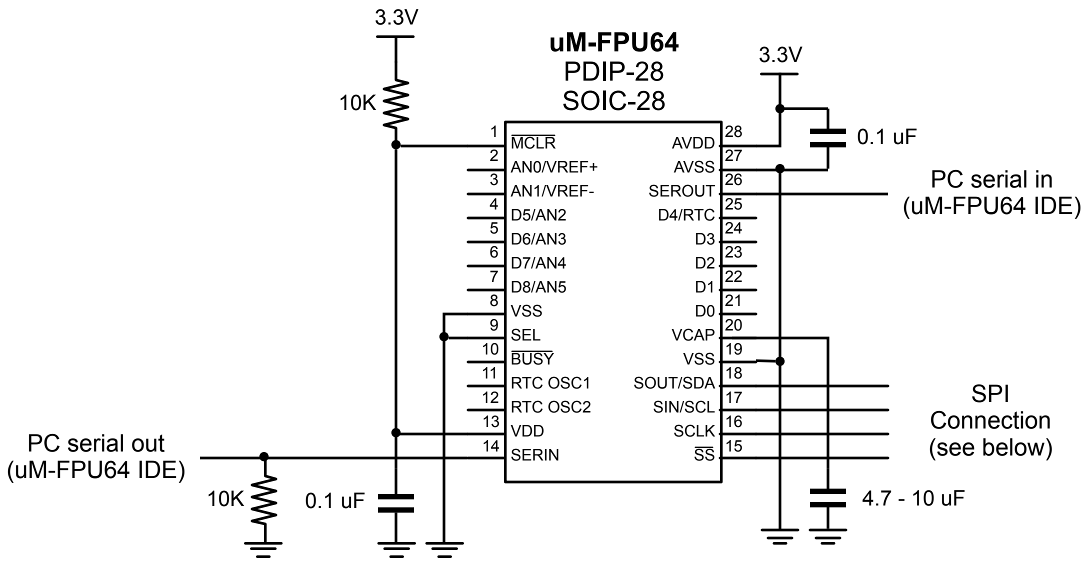
The uM-FPU64 is powered at 3.3V, and is connected to Propeller using an SPI interface. The pins used on the Propeller to interface with the FPU are as follows:

```
SCLKpin SPI clock (connects to FPU SCLK pin)
MISOpin SPI master in, slave out (connects to FPU SOUT pin)
MOSIpin SPI master out, slave in (connects to FPU SIN pin)
```

Any available Propeller pins can be used for the interface, changing the definitions for `SCLKpin`, `MISOpin`, and `MOSIpin`.

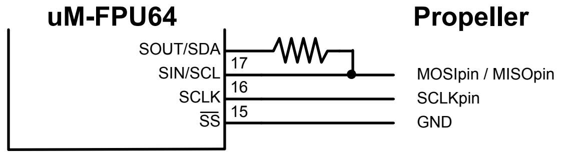
The SPI connection can be 2-wire, 3-wire, or an SPI bus as shown below.

Connecting the uM-FPU64 to the Propeller

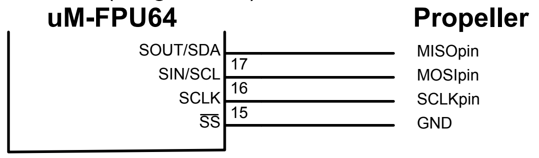


SPI Connection

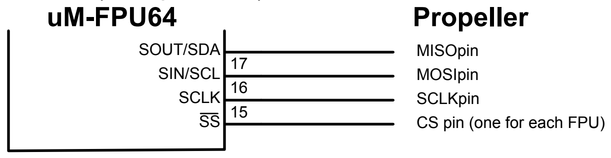
2-wire SPI (Single FPU)



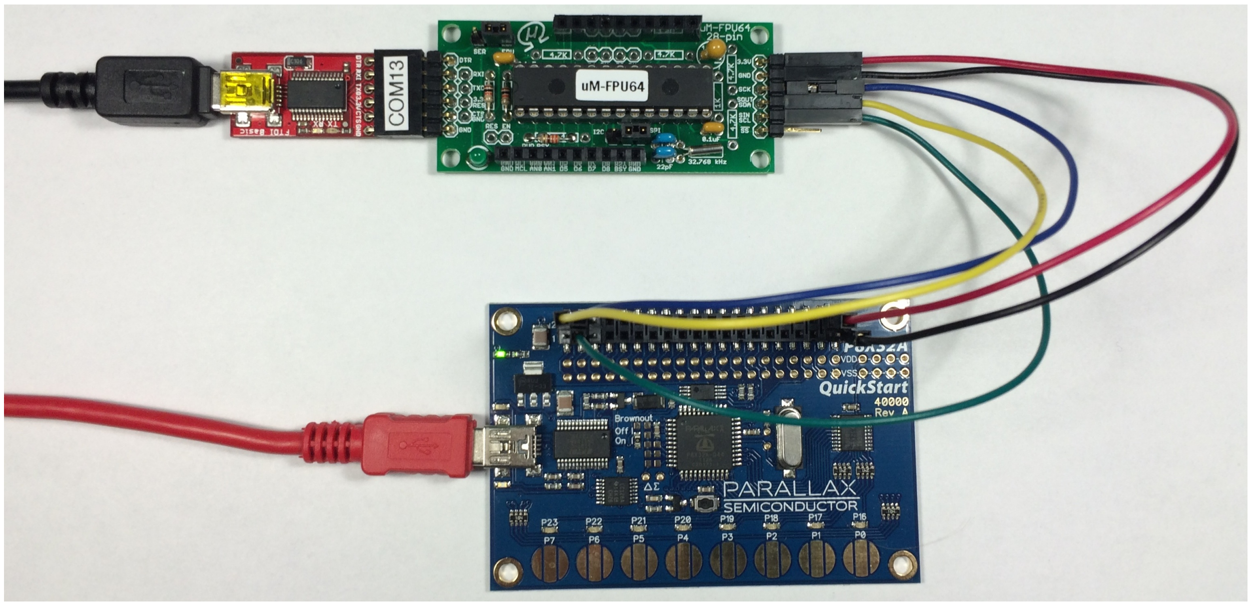
3-wire SPI (Single FPU)



3-wire SPI (Multiple FPU)



Propeller Quickstart Board connected to uM-FPU64 Breakout board



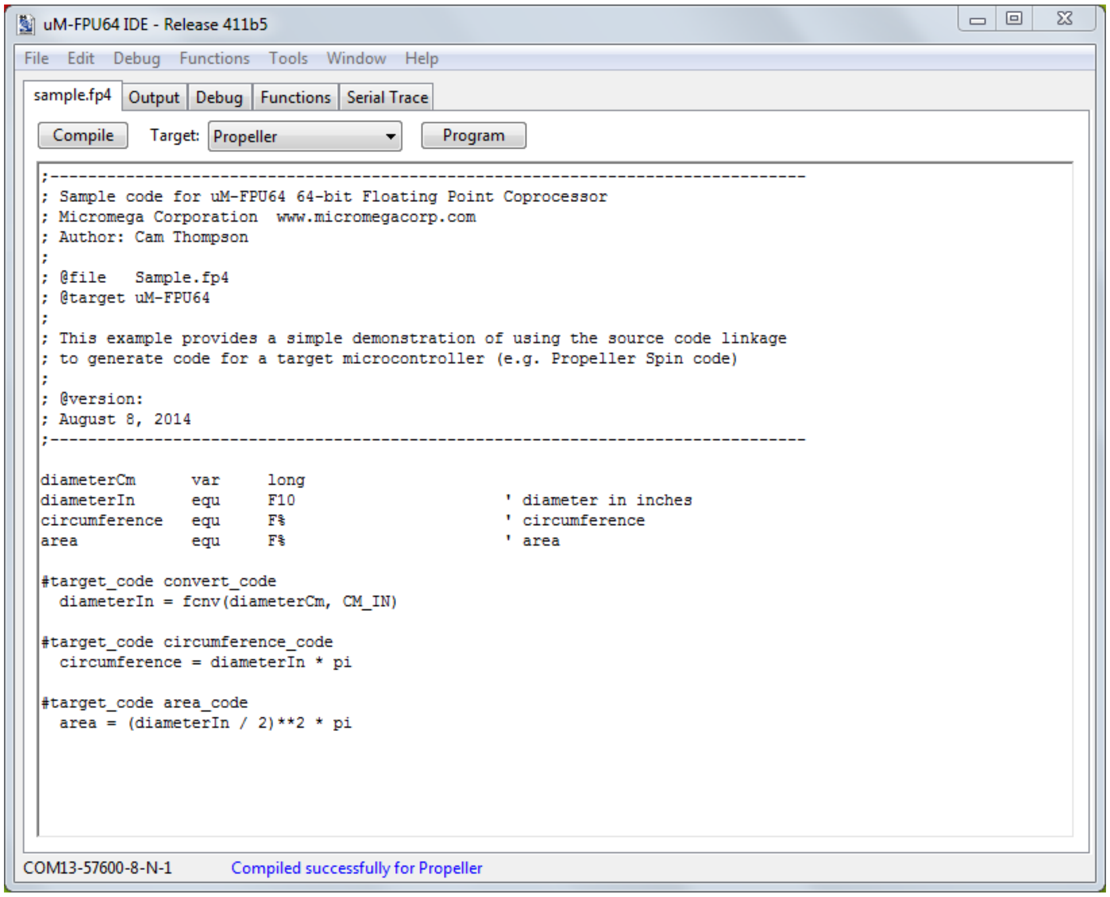
Sample Program

The *Sample.spin* program is provides a simple example of using the *FPU64_Driver* object. It can be also be used as a template for developing your own programs. The example demonstrates the use the source code linkage feature in the uM-FPU64 IDE software to update the FPU code in the Spin program.

The uM-FPU64 code is contained in the file *sample.fp4*.

Running the Sample Program

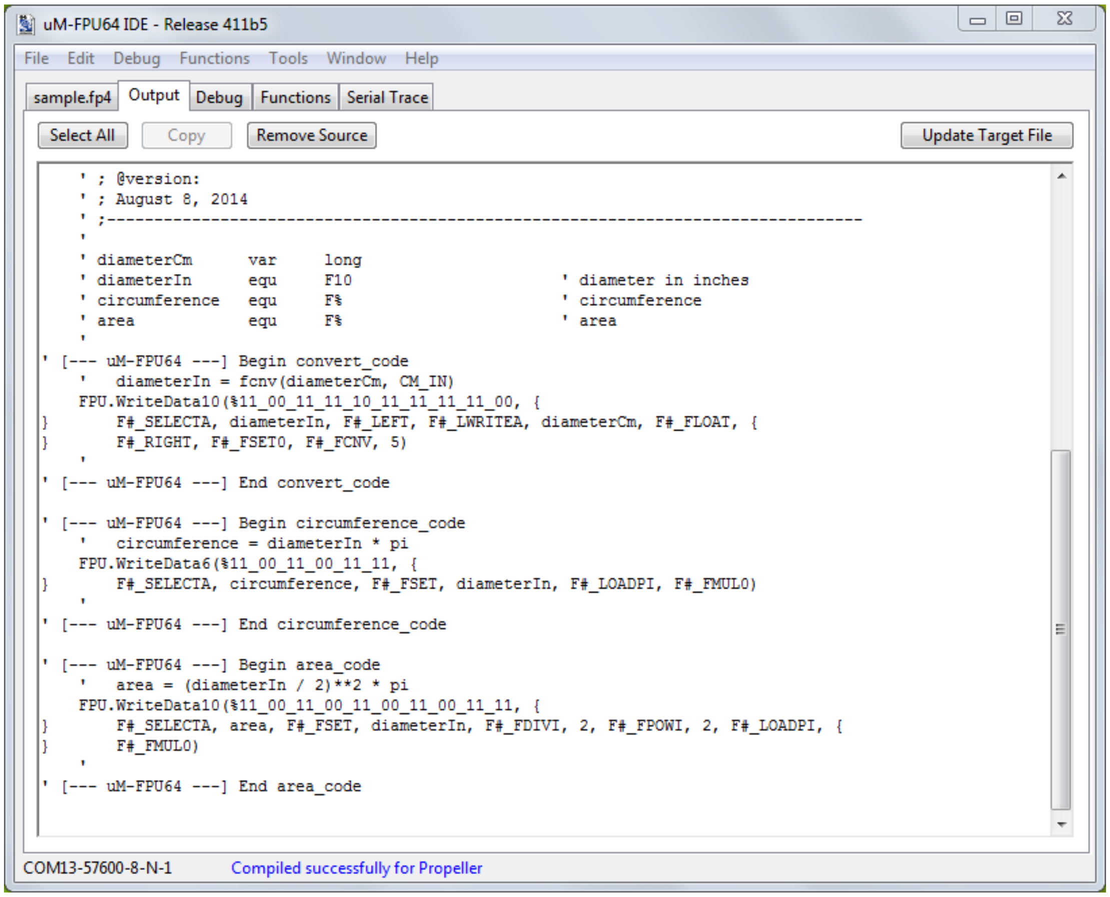
Compiling the FPU code


uM-FPU64 IDE - Release 411b5
File Edit Debug Functions Tools Window Help
sample.fp4
Output
Debug
Functions
Serial Trace
Compile
Target: Propeller
Program
COM13-57600-8-N-1
Compiled successfully for Propeller

```
;-------------------------------------------------------------------------------
; Sample code for uM-FPU64 64-bit Floating Point Coprocessor
; Micromega Corporation  www.micromegacorp.com
; Author: Cam Thompson
;
; @file   Sample.fp4
; @target uM-FPU64
;
; This example provides a simple demonstration of using the source code linkage
; to generate code for a target microcontroller (e.g. Propeller Spin code)
;
; @version:
; August 8, 2014
;-------------------------------------------------------------------------------

diameterCm      var     long
diameterIn      equ     F10                         ' diameter in inches
circumference   equ     F%                          ' circumference
area            equ     F%                          ' area

#target_code convert_code
  diameterIn = fcnv(diameterCm, CM_IN)

#target_code circumference_code
  circumference = diameterIn * pi

#target_code area_code
  area = (diameterIn / 2)**2 * pi
```

- run the *uM-FPU64 IDE* program
- open the file *sample.fp4*
- press the ***Compile*** button

Update the FPU code in the Spin file

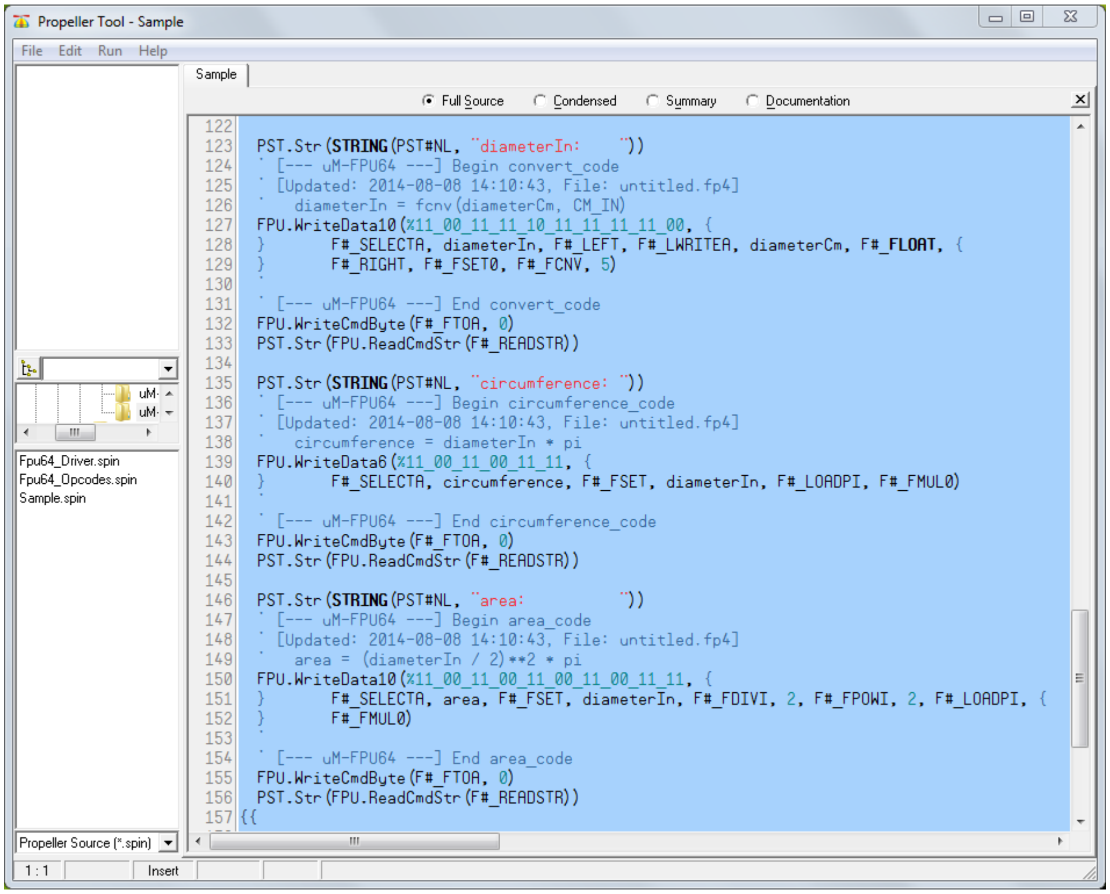


```
    ' ; @version:
    ' ; August 8, 2014
    ' ;-------------------------------------------------------------------------------
    '
    ' diameterCm     var     long
    ' diameterIn     equ     F10                        ' diameter in inches
    ' circumference  equ     F%                         ' circumference
    ' area           equ     F%                         ' area
    '
' [--- uM-FPU64 ---] Begin convert_code
    '   diameterIn = fcnv(diameterCm, CM_IN)
    FPU.WriteData10(%11_00_11_11_10_11_11_11_11_00, {
}       F#_SELECTA, diameterIn, F#_LEFT, F#_LWRITEA, diameterCm, F#_FLOAT, {
}       F#_RIGHT, F#_FSET0, F#_FCNV, 5)
    '
' [--- uM-FPU64 ---] End convert_code

' [--- uM-FPU64 ---] Begin circumference_code
    '   circumference = diameterIn * pi
    FPU.WriteData6(%11_00_11_00_11_11, {
}       F#_SELECTA, circumference, F#_FSET, diameterIn, F#_LOADPI, F#_FMUL0)
    '
' [--- uM-FPU64 ---] End circumference_code

' [--- uM-FPU64 ---] Begin area_code
    '   area = (diameterIn / 2)**2 * pi
    FPU.WriteData10(%11_00_11_00_11_00_11_00_11_11, {
}       F#_SELECTA, area, F#_FSET, diameterIn, F#_FDIVI, 2, F#_FPOWI, 2, F#_LOADPI, {
}       F#_FMUL0)
    '
' [--- uM-FPU64 ---] End area_code
```

COM13-57600-8-N-1   Compiled successfully for Propeller

- select the ***Output*** tab
- press the ***Update Target File*** button
- an open file dialog will be displayed
- select the *Sample.spin* file and press the ***Open*** button
- the links are updated
- a dialog box displays the number of links successfully updated
- a timestamp is added to the linked code

Run Sample.Spin



```
  PST.Str(STRING(PST#NL, "diameterIn:     "))
  ' [--- uM-FPU64 ---] Begin convert_code
  ' [Updated: 2014-08-08 14:10:43, File: untitled.fp4]
  '   diameterIn = fcnv(diameterCm, CM_IN)
  FPU.WriteData10(%11_00_11_11_10_11_11_11_11_00, {
  }        F#_SELECTA, diameterIn, F#_LEFT, F#_LWRITEA, diameterCm, F#_FLOAT, {
  }        F#_RIGHT, F#_FSET0, F#_FCNV, 5)
  '
  ' [--- uM-FPU64 ---] End convert_code
  FPU.WriteCmdByte(F#_FTOA, 0)
  PST.Str(FPU.ReadCmdStr(F#_READSTR))

  PST.Str(STRING(PST#NL, "circumference: "))
  ' [--- uM-FPU64 ---] Begin circumference_code
  ' [Updated: 2014-08-08 14:10:43, File: untitled.fp4]
  '   circumference = diameterIn * pi
  FPU.WriteData6(%11_00_11_00_11_11, {
  }        F#_SELECTA, circumference, F#_FSET, diameterIn, F#_LOADPI, F#_FMUL0)
  '
  ' [--- uM-FPU64 ---] End circumference_code
  FPU.WriteCmdByte(F#_FTOA, 0)
  PST.Str(FPU.ReadCmdStr(F#_READSTR))

  PST.Str(STRING(PST#NL, "area:           "))
  ' [--- uM-FPU64 ---] Begin area_code
  ' [Updated: 2014-08-08 14:10:43, File: untitled.fp4]
  '   area = (diameterIn / 2)**2 * pi
  FPU.WriteData10(%11_00_11_00_11_00_11_00_11_11, {
  }        F#_SELECTA, area, F#_FSET, diameterIn, F#_FDIVI, 2, F#_FPOWI, 2, F#_LOADPI, {
  }        F#_FMUL0)
  '
  ' [--- uM-FPU64 ---] End area_code
  FPU.WriteCmdByte(F#_FTOA, 0)
  PST.Str(FPU.ReadCmdStr(F#_READSTR))
{{
```

- run the Propeller Tool and open *Sample.spin*
- run the program

An example of the output on the Parallax Serial Terminal is as follows:

```
Sample routine for FPU64_Driver
Reset
Sync: 5C
Version: uM-FPU64 r411b5

Sample

diameterCm:     25
diameterIn:     9.84252
circumference:  30.92119
area:           76.0856

Done.
```

Fpu64_Driver Methods

The *Fpu64_Driver* object provides the following methods for using the uM-FPU64 chip with the Propeller microcontroller.

Setup and Special Purpose Methods

Start(SCLKpin, MOSIpin, MISOpin)

Must be called at the start of the main program to allocate the COG used for the FPU driver, assign the pins used to interface with the uM-FPU64 chip, and configure the SPI interface. It must be called before any other *Fpu* methods are called. The Propeller pins used to interface with the FPU chip are as follows:

```
SCLKpin   SPI clock (connects to FPU SCLK pin)
MISOpin   SPI master in, slave out (connects to FPU SOUT pin)
MOSIpin   SPI master out, slave in (connects to FPU SIN pin)
```

Stop

Called at the end of the main program to deallocate the COG used for the FPU driver.

Reset

This method must be called before any other FPU methods are used to initialize the FPU, and ensure synchronizaion.

Sync

Confirms communication with the FPU. It returns `F#SYNC_CHAR (0x5C)` if successful.

e.g.

```
IF (FPU.Sync == F#_SYNC_CHAR)
  PST.Str(STRING("Sync character received"))
```

Wait

The uM-FPU64 chip has a 256 byte instruction buffer which allows it to process in parallel with the Propeller. The *Wait* method will not return until the FPU instruction buffer is empty. This method is not required for data transfers using the *WriteData*, *writeCmd*, *readCmd* methods because the FPU64_Driver keeps track of the buffer size and handles waits automatically.

Break

Executes an FPU `BREAK` instruction. If the uM-FPU64 IDE is enabled, the `BREAK` stops FPU execution and allows the user to check register and memory values, and the step through the FPU code.

Select(cs)

Used to implement chip select when multiple FPUs are interfaced.

Write Commands

WriteByte(b)

Writes the lower 8 bits of *b* to the FPU.

WriteWord(w)

Writes the lower 16 bits of *w* to the FPU as two bytes (MSB first).

WriteLong(m)

Writes the 32 bits of *m* to the FPU as four bytes (MSB first).

WriteStr(s)

Writes a string to the FPU.

WriteNByte(n, b, d)

Writes *n* bytes starting at address *b* to the FPU. An optional delay (*d*) may be required between bytes to avoid buffer overflow for large data transfers.

WriteNWord(n, w, d)

Writes *n* 16-bit words starting at address *w* to the FPU. An optional delay (*d*) may be required between bytes to avoid buffer overflow for large data transfers.

WriteNLong(n, m, d)

Writes *n* 32-bit values starting at address *m* to the FPU. An optional delay (*d*) may be required between bytes to avoid buffer overflow for large data transfers.

WriteData1(t, n1)

WriteData2(t, n1, n2)

WriteData3(t, n1, n2, n3)

WriteData4(t, n1, n2, n3, n4)

WriteData5(t, n1, n2, n3, n4, n5)

WriteData6(t, n1, n2, n3, n4, n5, n6)

WriteData7(t, n1, n2, n3, n4, n5, n6, n7)

WriteData8(t, n1, n2, n3, n4, n5, n6, n7, n8)

WriteData9(t, n1, n2, n3, n4, n5, n6, n7, n8, n9)

WriteData10(t, n1, n2, n3, n4, n5, n6, n7, n8, n9, n10)

WriteData11(t, n1, n2, n3, n4, n5, n6, n7, n8, n9, n10, n11)

WriteData12(t, n1, n2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12)

WriteData13(t, n1, n2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12, n13)

WriteData14(t, n1, n2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12, n13, n14)

When transferring data to the FPU the overhead for each Spin call is much more significant than the actual SPI transfer time. The *WriteData* methods allow for a mix of data types to be combined into a single call which significantly reduces the Spin overhead compared with many separate calls. The uM-FPU64 IDE software will automatically generate *WriteData* calls for compiled code. The type value *t*, contains aa series of 2-bit values specifying the data type for n1 to n14. The value is right justified, with the rightmost 2-bit field corresponding to the rightmost argument.

Bits	**Description**
00	8-bit value
01	16-bit value
10	32-bit value
11	8-bit opcode value

e.g.
The following uM-FPU64 IDE compiler code:

```
val1 = val1 + val2 ** 3
```

would generate the following Propeller code:

```
FPU.WriteData11(%11_00_11_11_00_11_00_11_00_11_11, {
}       F#_SELECTA, val1, F#_LEFT, F#_FSET, val2, F#_FPOWI, 3, F#_FADD, val1, {
}       F#_RIGHT, F#_FSET0)
```

WriteCmd(cmd)

Writes *cmd* command to the FPU.

WriteCmdByte(cmd, b1)

WriteCmdByte2(cmd, b1, b2)

WriteCmdByte3(cmd, b1, b2, b3)

WriteCmdByte4(cmd, b1, b2, b3, b4)

Writes *cmd* opcode to the FPU followed by data 8-bit bytes *b1* through *b4*.

WriteCmdByteWord(cmd, b, w)

Writes *cmd* opcode to the FPU followed by 8-bit byte *b*, and 16-bit word *w*.

WriteCmdByte2Word(cmd, b1, b2, w)

Writes *cmd* opcode to the FPU followed by two 8-bit bytes *b1*, *b2*, and 16-bit word *w*.

WriteCmdByteLong(cmd, b, n)

Writes *cmd* opcode to the FPU followed by 8-bit bytes *b* and 32-bit value *n*.

WriteCmdWord(cmd, w)

Writes *cmd* opcode to the FPU followed by 16-bit word *w*.

WriteCmdLong(b, n)

Writes *cmd* opcode to the FPU followed by 32-bit value *n*.

WriteCmdStr(cmd, s)

Writes an opcode to the FPU followed by zero-terminated string *s*.

WriteCmdByteStr(cmd, b, s)

Writes a command to the FPU followed by 8-bit value *b,* and zero-terminated string *s*.

WriteCmdByte2Str(cmd, b1, b2, s)

Writes an opcode to the FPU followed by 8-bit bytes *b1*, *b2*, and zero-terminated string s.

Read Commands

ReadByte

Reads an 8-bit byte from the FPU and returns the result.

ReadWord

Reads an 16-bit word from the FPU and returns the result.

ReadLong

Reads an 32-bit value from the FPU and returns the result. *ReadLong* can be used to read both floating point and long integer values.

ReadStr

Reads a zero-terminated string from the FPU and stores it in an internal buffer in the Propeller Hub. The address of the string is returned.

ReadNByte(n, m, d)

Reads *n* bytes from the FPU and stores them starting at address *m*. An optional delay (*d*) may be required between bytes to avoid buffer overflow for large data transfers.

ReadNWord(n, m, d)

Reads *n* words from the FPU and stores them starting at address *m*. An optional delay (*d*) may be required between bytes to avoid buffer overflow for large data transfers.

ReadNLong(n, m, d)

Reads *n* 32-bit values from the FPU and stores them starting them at address *m*. An optional delay (*d*) may be required between bytes to avoid buffer overflow for large data transfers. *ReadNLong* can be used to read both floating point and long integer values.

ReadCmdByte(cmd)

Writes *cmd* opcode to the FPU, then reads an 8-bit byte from the FPU and returns the result.

ReadCmdWord(cmd)

Writes *cmd* opcode to the FPU, then reads a 16-bit word from the FPU and returns the result.

ReadCmdLong(cmd)

Writes *cmd* opcode to the FPU, then reads a 32-bit value from the FPU and returns the result. *ReadCmdLong* can be used to read both floating point and long integer values.

ReadCmd2Long(cmd, r)

Writes *cmd* opcode to the FPU followed by the 8-bit value *r*, then reads a 32-bit value from the FPU and returns the result. *ReadCmd2Long* can be used to read both floating point and long integer values.

ReadCmdStr(cmd)

Writes *cmd* opcode to the FPU, then reads a zero-terminated string from the FPU and stores it in an internal buffer in the Propeller Hub. The address of the string is returned.

ReadCmdByte3(cmd, b1, b2, b3)

Writes *cmd* opcode to the FPU followed by the three 8-bit values *b1*, *b2*, *b3*, then reads a 32-bit value from the FPU and returns the result.

Spin-based 32-bit Arithmetic Operations

F32_Add(a, b)

result = a + b

F32_Sub(a, b)

result = a - b

F32_Mul(a, b)

*result = a * b*

F32_Div(a, b)

result = a / b

F32_Mod(a, b)

result = a MOD b

F32_Abs(a)

result = | *a* |

F32_Neg(a)

result = -a

F32_Cmp(a, b, epsilon)

if | *a-b* | *<= epsilon, result = 0*
if a < b, result = -1
if a > b, result = 1

L32_To_F32(a)

result = float(a)

FloatToString(a, format)

result = address of a floating point string, with value *a* displayed according to the *format* specified.

Further Information

The following documents are also available:

uM-FPU64 Datasheet	provides hardware details and specifications
uM-FPU64 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.