Micromega Corporation

Application Note 101
uM-FPU64

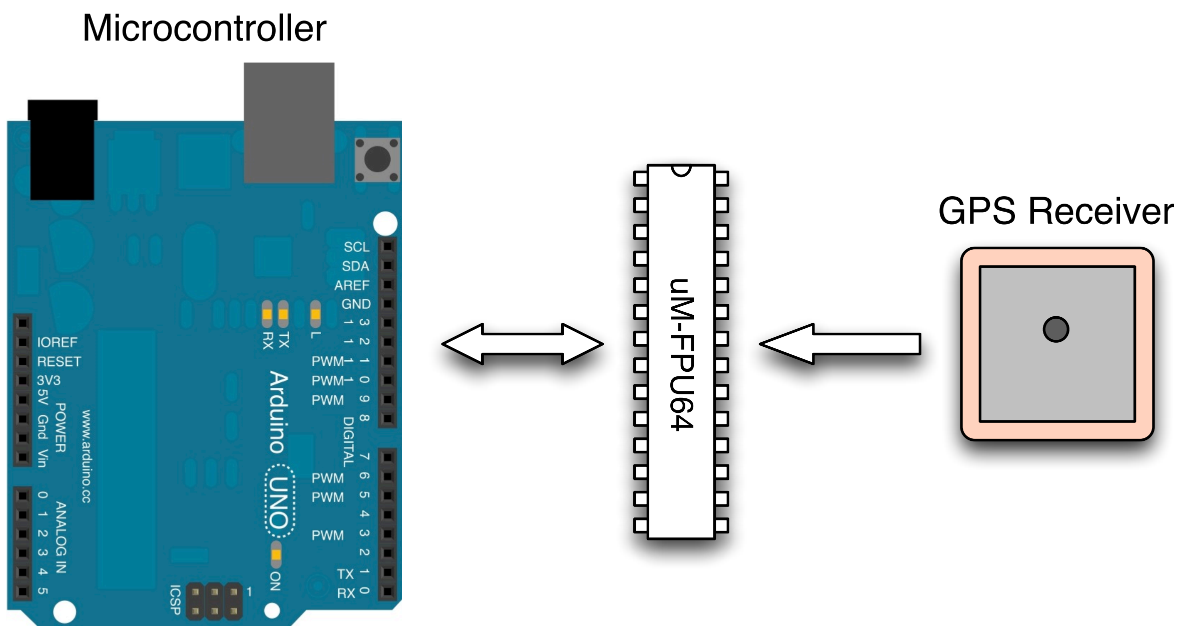
Reading GPS Data

Introduction

GPS data is used in a wide range of embedded systems applications. Adding a GPS device to an application can consume significant resources on a microcontroller to handle the serial interface, and to parse and process the GPS data. The uM-FPU64 64-bit floating point coprocessor has a number of capabilities designed to make handling GPS data very straightforward. To utilize the maximum accuracy available from the GPS device, and to accurately perform navigational calculations, 64-bit floating point support is required – a feature that is not readily available with most commonly used microcontrollers.

This application note describes how to use the uM-FPU64 to read and process data from a GPS device. The GPS device can be interfaced directly to the uM-FPU64, and all of the GPS processing offloaded from the main microcontroller. For some applications the uM-FPU64 can be configured for stand-along operation, with no additional microcontroller required.

Using uM-FPU64 as coprocessor for processing GPS data.



GPS devices can be interfaced to any available uM-FPU64 digital input pins, or the `SERIN` pin. Built-in support for NMEA sentence parsing and a full set of string instructions are available to facilitate reading the GPS data fields. The *uM-FPU64 IDE* software can be used to create user-defined functions for handling GPS data, and these functions can be stored in Flash memory on the FPU. Several code examples showing how to process GPS data are available on the Micromega website.

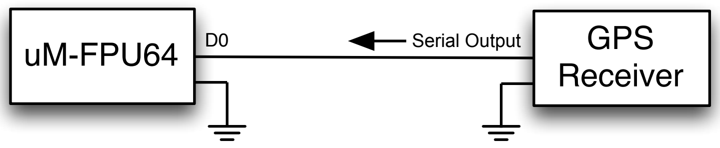
Connecting the GPS device to the uM-FPU64 chip

Most GPS devices send out serial data continuously at a regular interval using the NMEA-0183 serial data interface. The serial data is then processed to obtain the required information. Two serial interfaces are available on the uM-FPU64. One serial interface uses the `SERIN` and `SEROUT` pins which are also used by the built-in debug monitor and *uM-FPU64 IDE* software for debugging and programming the FPU. When the debug monitor is not being used, the `SERIN` and `SEROUT` pins can be used for general purpose serial I/O. The second serial interface can be assigned to any of the digital input pins using the `DEVIO,ASYNC` instruction. The preferred method of connecting the GPS device is to use one of the digital input pins, since the `SERIN` and `SEROUT` pins are then available for debugging.

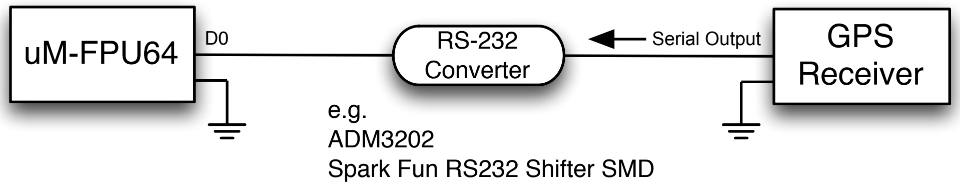
If data is only received from the GPS device, only a single serial input is required. Applications that require configuring the GPS device, or use proprietary interfaces may require both serial input and serial output and may also require hardware flow control. These options are readily available using the `DEVIO,ASYNC` instructions.

The uM-FPU64 requires a serial interface of 3.3V. Many of the GPS devices used for embedded applications provide TTL level output of 0V to ~2.5V. These can be interfaced directly to any of the FPU digital input pins. GPS devices that use a TTL level output of 0V to 5V, can be connected directly to one of the FPU digital input pins that is 5V tolerant (e.g. `D0`, `D1`). If the GPS device uses RS-232 output, a converter is required to convert the signal to the 3.3V logic levels used by the uM-FPU64 chip.

Connecting GPS with TTL output.



Connecting GPS with RS-232 output.



Brief Overview of the NMEA-0183 Data Interface

The NMEA-0183 data interface is commonly used for marine instruments and GPS devices. This interface usually provides continuous serial data at a regular sample interval of one or two seconds, although GPS devices can often be configured for different sample rates. The data is readable ASCII text, consisting of a series of NMEA sentences. Each sentence begins with a dollar sign ($) and ends with a carriage return and linefeed. An optional checksum is usually included, consisting of an asterisk (*) and two checksum digits at the end of the sentence. Each field is separated by a comma. The first field indicates the sentence type, and the remaining fields contain the data. If the data for a field is unavailable, the field is empty.

Sample NMEA-0183 Output from GPS device

```
$GPRMC,185847,A,5105.2567,N,11520.0118,W,29.6,288.2,250313,17.4,E,D*0A
$GPRMB,A,,,,,,,,,,,,V,D*19
$GPGGA,185847,5105.2567,N,11520.0118,W,2,07,1.2,1357.0,M,-17.4,M,,*47
$GPGSA,A,3,01,11,,14,,20,22,,25,,32,,0.0,1.2,1.2*32
$GPGSV,3,1,12,01,47,256,40,11,29,232,33,12,13,034,00,14,39,067,37*7B
$GPGSV,3,2,12,17,08,322,22,20,35,296,41,22,15,122,25,23,07,253,36*7F
$GPGSV,3,3,12,25,20,065,37,31,59,148,44,32,64,290,37,51,31,169,46*7D
$GPGLL,5105.2567,N,11520.0118,W,185847,A,D*5F
$GPBOD,,T,,M,,*47
$GPVTG,288.2,T,270.8,M,29.6,N,54.8,K,D*2F
$PGRME,5.4,M,8.8,M,10.4,M*1A
$PGRMZ,4452,f,*1F
$PGRMM,WGS 84*06
$HCHDG,270.8,,,17.4,E*16
$GPRTE,1,1,c,*37
```

Most applications only use a portion of the data that is received from the GPS device. The data is scanned until particular NMEA sentence types are recognized, then the sentences are parsed to extract the data fields of interest.

NMEA sentence parsing using the uM-FPU64 chip

The `SERIN` instruction has a special input mode for NMEA sentence parsing. Using this input mode, the uM-FPU64 will scan for valid NMEA sentences, strip the $, checksum, carriage return and linefeed characters, confirm the checksum, then set the status byte and store the remaining string in a 256 byte NMEA sentence buffer. The buffer can store multiple sentences, allowing one sentence to be processed while other sentences are being received. The `SERIN` instruction reads NMEA sentence from the buffer, sets the status byte, and moves the sentence to the string buffer. The sentences can then be parsed, and the data fields extracted using various string instructions.

The following compiler statements enable digital pin `D0` as a serial input, sets the baud rate to 4800 baud, and enables NMEA input mode.

```
devio(ASYNC, ENABLE, D0, RX+BAUD_4800)
serial(ENABLE_NMEA+ASYNC)
```

Once NMEA input mode has been enabled, the serial input is scanned for the beginning of the next sentence ($). The sentence is then read, the checksum is confirmed, and the sentence is stored in a 256 byte NMEA buffer with the leading $, checksum, carriage return, and linefeed removed. The `serial(READ_NMEA+ASYNC)` statement is used to read a sentence from the buffer. This statement waits for the next available sentence, then transfers the sentence to the string buffer and selects the first field (sentence type). A string comparison can then be done to determine if the sentence will be processed further, or ignored.

If you need to do other things while you wait for a sentence to be available, the `serial(STATUS_NMEA +ASYNC)` instruction can be used. It checks the NMEA buffer and sets the status byte to zero (Z) if the buffer is empty, or non-zero (NZ) if a sentence is available or an error occurred.

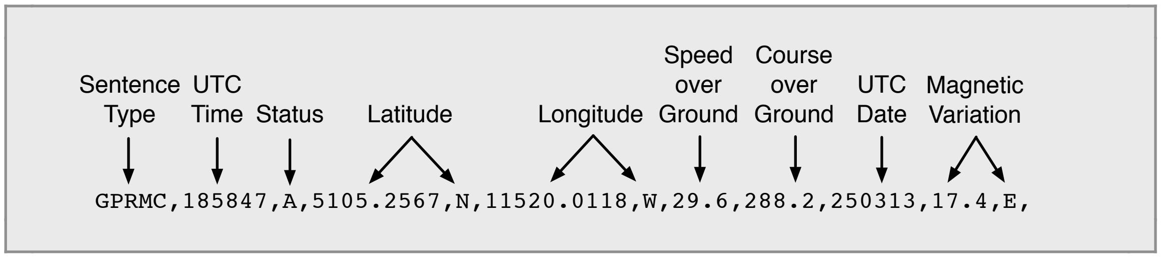
In this example, we'll use the `serial(READ_NMEA+ASYNC)` statement to wait until the next sentence is available. Since we're only interested in the GPRMC sentence, we simply loop until the sentence type matches.

```
do
  serial(READ_NMEA+ASYNC)
  if strfield() = "GPRMC" then exit
loop
```

Reading Latitude and Longitude from the GPRMC sentence

The GPRMC sentence is the "Recommended Minimum" sentence, and is the most common sentence transmitted by GPS devices. It contains latitude, longitude, speed, bearing, satellite-derived time, fix status and magnetic variation.

The following diagram shows an example of the contents of the string buffer after receiving a GPRMC sentence.



We're interested in the latitude and longitude fields. Latitude is stored in field 4 as DDMM.MMMM, where DD is degrees and MM.MMMM is minutes. Field 5 is the North/South indicator (N for North, S for South). Longitude is stored in field 6 as DDDMM.MMMM, where DDD is degrees and MM.MMMM is minutes. Field 7 is the East/West indicator (E for East, W for West). Navigation calculations require the latitude and longitude values expressed as decimal degrees. The values are converted as follows:

Latitude = DD + (MM.MMMM / 60)
Longitude = DDD + (MM.MMMM / 60)

e.g.
`5105.2567,N` is converted to 51.087611666 degrees latitude.
`11520.0118,W` is converted to -115.33353 degrees longitude.

The following function converts latitude or longitude fields to decimal degrees. The field number is passed as a function argument. To maintain full precision, 64-bit floating point values are used for latitude and longitude.

```
;---------------------------------------------------------------------------
#function NMEA_to_degrees(long) float64
;
; Convert NMEA latitude and longitude fields to decimal degrees.
;
; input:    arg1    field of latitude or longitude value
; output:   result  decimal degrees
;---------------------------------------------------------------------------
field       equ L2                      ; field
deg         equ F129                    ; degrees
mm          equ F130                    ; minutes

  field = arg1                          ; get local copy of parameter
  strfield(field)                       ; select the field
  deg = strlong() / 100                 ; get degrees
  mm = strfloat() % 100                 ; get minutes
  deg = deg + mm / 60                   ; convert to decimal degrees

  field = field + 1                     ; check for N/S or E/W indicator
  strfield(field)
  if strfield() = "S" or strfield() = "W" then deg = -deg

  return deg                            ; return decimal degrees

#end
```

The function is called as follows:

```
  latitude = NMEA_to_degrees(4)         ; get latitude in decimal degrees
  longitude = NMEA_to_degrees(6)        ; get longitude in decimal degrees
```

For navigational calculations, the latitude and longitude values need to be converted to radians since the uM-FPU64 and most math libraries use radians for trigonometric calculations. Decimal degrees can be converted to radians with the radians() functions.

```
  latitude = radians(NMEA_to_degrees(4))   ; get latitude in radians
  longitude = radians(NMEA_to_degrees(6))  ; get longitude in radians
```

A simple example of reading the current latitude and longitude is shown below. It enables NMEA input, waits for the first GPRMC sentence, reads latitude and longitude, disables the NMEA input, and returns.

```
#function read_GPRMC

  devio(ASYNC, ENABLE, GPS_PIN, RX+BAUD_4800)
  serial(ENABLE_NMEA+ASYNC)             ; enable NMEA input mode

  do
    serial(READ_NMEA+ASYNC)             ; read next NMEA sentence
    if strfield() = "GPRMC" then exit   ; wait for GPRMC sentence
  loop

  latitude = NMEA_to_degrees(4)         ; get latitude and longitude
  longitude = NMEA_to_degrees(6)

  serial(DISABLE+ASYNC)                 ; disable serial input

#end
```

Waiting for GPS Data

The `serial(READ_NMEA+ASYNC)` statement will only wait for an NMEA sentence if the instruction buffer is empty. This provides a way for the microcontroller to abort the instruction (e.g. by sending a `NOP` instruction). To ensure that the instruction buffer is empty when the `serial(READ_NMEA+ASYNC)` statement is executed, the microcontroller should wait for the Ready status after any `serial(READ_NMEA+ASYNC)` statement, or any call to a user-defined function that uses the `serial(READ_NMEA+ASYNC)` statement.

Error Checking

The `serial(STATUS_NMEA+ASYNC)` and `serial(READ_NMEA+ASYNC)` statements set the status byte for the next available sentence. If no sentence is available, the status byte is set to zero (`Z`). If the sentence is valid, the status byte is set to greater-than (`GT`). If an error occurred, the status byte is set to less-than (`LT`), and bit 4 (`NMEA_OVERRUN`) is set if an overrun error occurred, and bit 5 (`NMEA_CHECKSUM`) is set if a CRC error occurred.

Overrun Errors

The NMEA sentence buffer is 256 bytes in length. The average NMEA sentence is about 60 to 70 bytes, so the buffer can hold three or four sentences of average length, or more if there are smaller sentences. The sentences should be read at a rate that is fast enough to avoid overrun. For example, if the GPS is sending data at 4800 baud, and the average sentence length is 60 bytes, then sentences arrive about every 125 milliseconds and must be processed at a faster rate to avoid overrun. Sending data to a serial port on the microcontroller at a baud rate that's equal to or less than the baud rate for the incoming GPS data is time consuming, and is a common source of overrun errors. To recover from an overrun error, sentences can be flushed from the buffer, or the serial input can be disabled and re-enabled to start with an empty buffer. If GPS data is only required periodically, the best solution is to disable the input between readings.

Infrequent readings

If GPS data is not being read continuously, the `serial(DISABLE+ASYNC)` statement should be used to disable serial input when not required, and the `serial(ENABLE_NMEA+ASYNC)` instruction should be used to enable NMEA input before taking a reading. This eliminates the serial input interrupt overhead when not required, and avoids overrun errors.

Debugging Tips

Since NMEA sentences are processed from the string buffer, an effective method for debugging is to simulate the GPS data by loading the string buffer with a known NMEA string and then executing the code to parse the data. Sample data can easily be captured from a GPS using a terminal emulator, and sample sentences can be copied to your test application for processing.

Comment out the code that enables NMEA input mode.

```
; devio(ASYNC, ENABLE, GPS_PIN, RX+BAUD_4800)
; serial(ENABLE_NMEA+ASYNC)
```

Comment out the `serial(READ_NMEA+ASYNC)` statement.
Add a `strset` statement to set the string buffer and a `strfield` statement to select the first field.

```
; serial(READ_NMEA+ASYNC)
strset("GPRMC,185847,A,5105.2567,N,11520.0118,W,29.6,288.2,250313,17.4,E,")
strfield(1)
```

The debugger can then be used to develop and test the parsing code without having to have an active GPS device (e.g. indoors). Once the parsing code is tested, remove the `strset` and `strfield` debug statements, reenable the original statements, and you're ready to go with live data.

Sample Code

Sample code for reading GPS data is available on the Micromega website. The files related to this application note are as follows:

`read_GPRMC.ino`
Arduino test program for demo1 and demo2.

`read_GPRMC-demo1.fp4`
FPU code to read latitude and longitude from a single GPRMC sentence.

`read_GPRMC-demo2.fp4`
FPU code to read latitude and longitude from a single GPRMC sentence with error checking.

`read_GPRMC2.ino`
Arduino test program for demo3.

`read_GPRMC-demo3.fp4`
FPU code to read latitude, longitude and time from multiple GPRMC sentences.

Further Information

See the Micromega website (http://www.micromegacorp.com) for additional information regarding the uM-FPU64 floating point coprocessor, including:

uM-FPU64 Datasheet
uM-FPU64 Instruction Set
uM-FPU64 IDE User Manual
uM-FPU64 IDE Compiler Manual