Digital Sensors

Background:

Microcontrollers are small and cheap enough that several sensors have a microcontroller incorporated into their design. This makes interfacing to these sensors easy with another microcontroller: all data is sent and received via digital communications.

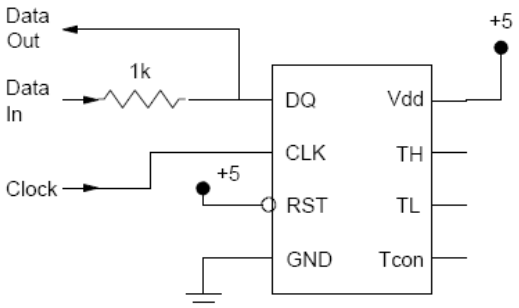
The most common form of communications is Serial Perihperial Interface (SPI). This requires

- A chip select line to let the sensor know you are sending data
- A clock line, to let the sensor know when the data is valid,
- A data out line - where the sensor sends data out to the microcontroller, and
- A data in line - where the microcontroller sends data to the sensor.

For example, the DS1620 is a digital temperature sensor. For this sensor, the data out and data in line are shared (saving one I/O pin). You can split this into the data out and data in using a 1k resistor:

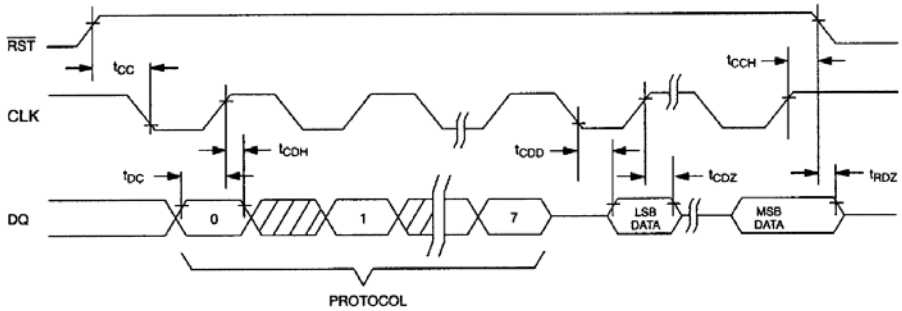
- When DQ is input, DQ is passive and the Data In line sets the voltage at DQ.
- When DQ is output, the DS1620 sets the voltage on the data line, and drives Data Out.

Otherwise, it's a normal SPI interface.



To talk to the DS1620, look for the timing diagrams in the data sheets:

READ DATA TRANSFER Figure 4



This tells you to:

- Pull RST high
- Clock in the data onto the DQ line, with the data valid on the rising edge of the clock, least significant bit first, and
- Pulse RST low then high to end the message.

Next, you need to know what messages to send. Again from the data sheets:

Command	Description
0xAA	Read Temperature
0xEE	Start Temperature Conversion
0x01	Write to TH
0x02	Write to TL
0x0C	Write to CONFIG
0x0C 0x00	Set up for continuous conversion

Finally, you need to know how to interprit the data when it comes back.

To drive a DS1620, let's use bottom-up programming. First, write a routine which writes or reads one byte, LSB first. Assume

- RC0 = RST
- RC1 = CLK
- RC2 = DQ

```
void DS1620_Write(unsigned char DATA) {
   unsigned char i;
   TRISC1 = 0;
   TRISC2 = 0;

   RC1 = 0;                // start with CLK = 0
   RC0 = 1;                // start with RST = 1

   for (i=0; i<8; i++) {
      if (DATA & 1) RC2 = 1; else RC2 = 0;
      RC1 = 1;
      DATA = DATA >> 1;
      RC1 = 0;
      }
    }
```

```
unsigned char DS1620_Read(void) {
   unsigned char RESULT, i;
   TRISC1 = 0;
   TRISC2 = 1;             // DQ = output (input to PIC)
   RESULT = 0;
   for (i=0; i<8; i++) {
      RESULT = RESULT >> 1;
      RC1 = 1;             // read on the rising edge of CLK
      if (RC2) RESULT = RESULT + 0x80;
      RC1 = 0;
      }
   return(RESULT);
   }
```

Next, write a routine which sends the connand {0x0C, 0x00} to set up the DS1620 for continuous conversion

```
void DS1620_Init(void) {
   TRISC0 = 0;
   RC0 = 1;                // start of message
   DS1620_Write(0x0C);
   DS1620_Write(0x00);
   RC0 = 0;
   }
```

Finally, write a routine which reads the temperature

```
float DS1620_Get_T(void)
{
   int DATA;
   float CELCIUS;

   TRISC0 = 0;
   RC0 = 1;                // start of message
   DS1620_Write(0xAA)      // read temperature
   DATA = DS1620_Read();   // high byte first
   DATA = (DATA << 8) + DS1620_Read();
   RC0 = 0;

// convert to Celcius
   if (DATA & 0x0100) DATA = DATA | 0xFF00;  // sign extend
   CELCIUS = 0.5 * DATA;
   return(CELCIUS);        // temperature in C
   }
```

GPS

A GPS sensor is also a digital sensor. It's output is typically 9600 baud asynchronous, however (SCI). Likewise, the SCI module is needed for reading GPS data.

The SparkFun modules are rather convenient: power them up and a red light turns on. When GPS is locked on, the red light blinks. Once per second, a GPS message is sent out, with a typical message looking like the following:

```
$GPGGA,152410.979,4731.42559,N,09233.10091,W,1,10,0.8,436.16,M,-30.59,M,
$GPGSA,A,3,15,05,08,29,27,18,21,26,06,22,,,1.4,0.8,1.1*30
$GPGSV,3,1,12,21,74,292,42,15,68,119,47,18,56,265,45,26,38,053,46*76
$GPGSV,3,2,12,48,25,230,23,29,25,190,39,06,24,310,39,27,18,120,42*7F
$GPGSV,3,3,12,03,15,319,,22,13,258,33,05,11,074,40,08,08,026,33*7D
$GPRMC,152410.979,A,4731.42559,N,09233.10091,W,0002.15,172.05,140312,,
$GPVTG,172.05,T,,M,0002.15,N,00003.98,K,A*08
$GPZDA,152410.979,14,03,2012,00,00*55
```

Each message has fields separated by commas. The GPGGA message has the following data:

Field	Data	Meaning
1	GPRMC	Recommended minimum GPS data
2	152410.979	Time: 15:23:10.979 UTC
3	A	A = OK, V = warning
4,5	4731.42559,N	Latitude: 47d 21.42559'
6,7	09233.10091,W	Longitude: 092d 33.10091'
8	0002.15	Speed (knots)
9	172.05	Direction of motion (degrees)
10	140312	Date: 14:03:12 March 14, 2012

This brings up two problems:

- How to read in the GPS data into a buffer, and
- How to parse the data, and
- How to convert to meters.

Reading GPS data into a buffer:

You pretty much have to use SCI interrupts since you don't know when the data is coming in. An interrupt routine which looks for a carriage return (binary 13) to terminate each message and stores it into an array MSG[] follows:

If you want to check if the message was a GPRMC message, look for an 'R' in the third spot. If found, copy MSG to a buffer for later processing.

```
void interrupt IntServe(void) {
   if (RCIF) {

      RC0 = !RC0;                // debug info.  Should toggle each char
      TEMP = RCREG;
      while(!TRMT); TXREG = TEMP;

      MSG[N] = TEMP;
      N += 1;
      if (N > 80) N = ;
      if (TEMP == 13) {
         if (MSG[3] == 'R') {    // GPRMC message detected
            for (i=0; i<80; i++) GPS[i] = MSG[i];
            FLAG = 1;
            RC1 = !RC1;          // debug info - should toggle 1/sec
            }
         N = 0;
         }

      RCIF = 0;
      }
```

Parse the Data

In the main routine, you can watch for FLAG=1. This means you have GPS data. To parse the data, you need to pull out the various fields. Fortunately, each message and each field is fixed length.

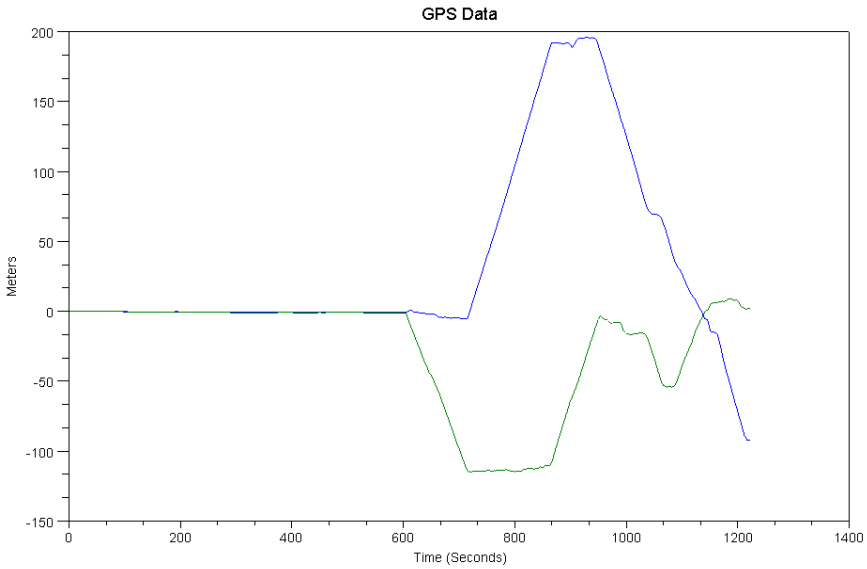
```
while(1) {
      if (FLAG) {

   // Latitude in minutes

         LATITUDE = (GPS[20]-'0')*600 +
                    (GPS[21]-'0')*60  +
                    (GPS[23] - '0')*10 +
                    (GPS[24] - '0')*1 +
                    (GPS[25] - '0')*0.1 +
                    (GPS[26] - '0')*0.01 +
                    (GPS[27] - '0')*0.001;



   // Longitude in minutes

         LONGITUDE = (GPS[33] - '0')*6000 +
                     (GPS[34] - '0')*600  +
                     (GPS[35] - '0')*60  +
                     (GPS[37] - '0')*10 +
                     (GPS[38] - '0')*1 +
                     (GPS[39] - '0')*0.1 +
                     (GPS[40] - '0')*0.01 +
                     (GPS[41] - '0')*0.001;

   // Speed in Knots

        KNOTS      = (GPS[47] - '0')*1000 +
                     (GPS[48] - '0')*100  +
                     (GPS[49] - '0')*10  +
```

```
                    (GPS[50] - '0')*1 +
                    (GPS[51] - '0')*0.1 +
                    (GPS[52] - '0')*0.01 +

     FLAG = 0;

    NORTH = LATITUDE * 1849.12;
    WEST = LONGITUDE * 1334.60;
    SPEED = KNOTS * 0.5144;

     }
```

If you want to convert to normal units, at Fargo, ND, this is

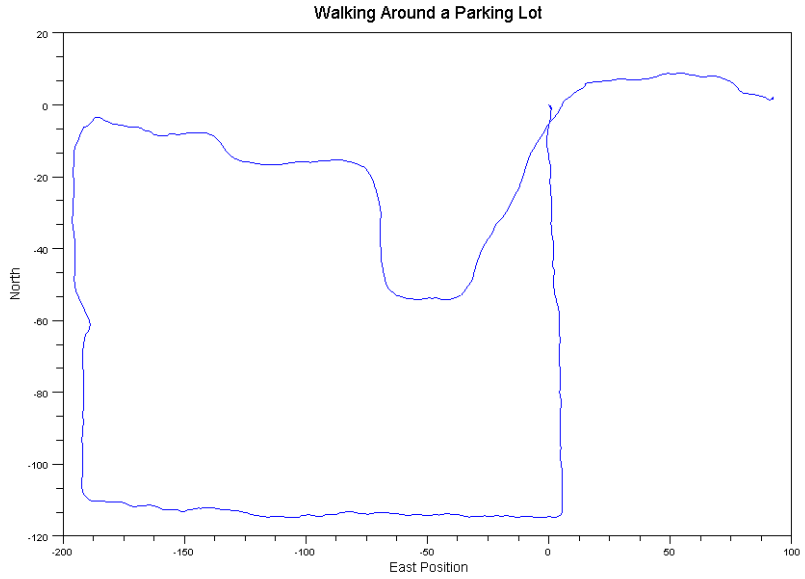
- Polar Radius = 6,356.8 km
  - 1 minute = 1849.12 meters
- Equatorial Radius = 6,378.1 km
  - 1 minute = 1,855.31 meters at the equator
  - 1 minute = 1,334.60 meters at 46 degrees north (Fargo)
- 1 Knot = 0.5144 meters/second

Doing this, you can keep track of where you are with a PIC processor. As an example, walking around a parking lot last March resulted in the following plot:



Relative position from start: N (blue), E (green)

Plotting your position



Note that GPS has some noise. For the first 10 minutes, the GPS receiver sat at one spot on the ground. Zooming in on this region shows that your GPS position drifts about 1 meter over 10 minutes. GPS isn't accurate enough to sink a golf ball in a hole, but good enough to figure out where your dog goes at night.