Column #85 May 2002 by Jon Williams:

I2C Fun For Everyone

For those of you who are old enough to do it, do you remember the last time you bought a new car? It feels great, doesn't it? Then, as you hit the open road, proud of your shiny new machine and loving that new-car smell ... you notice that just about every second human on the planet is driving the exact same model....

I went through that recently, but not with a car. I did it with the I^2C (Inter-integrated Circuit) bus (I've also just recently discovered how useful crock pots are ... but I'll talk about that when I connect a BASIC Stamp to one).

Honestly, I don't know what I was thinking. I^2C has been around for over 20 years and I was certainly aware of it, I just never paid much attention. Silly me. Since I can't turn back the clock I've spent the last couple of weeks making up for lost time and I have to say, I'm having a blast. And with my growing interest in robotics, I^2C is a fantastic way to expand the Stamp's capabilities without chewing up a bunch of pins.

There are a couple pieces of great news concerning I^2C for us Stamp users: (1) There are literally hundreds of I^2C parts available that we can connect to and, (2) The protocol is simple enough to implement on any Stamp – I've even heard of Stampers implementing it on the BS1!

As you know, the BS2p has built-in I^2C capability with its **I2COUT** and **I2CIN** commands. We've covered those commands in a couple past articles so this month we're going to give I^2C to the rest of the BS2 family.

I^2C Basics

The I^2C-bus is a two-wire, synchronous bus that uses a Master-Slave relationship between components. The Master initiates communication with the Slave and is responsible for generating the clock signal. If requested to do so, the Slave can send data back to the Master. This means the data pin (SDA) is bi-directional and the clock pin (SCL) is [usually] controlled only by the Master.

The transfer of data between the Master and Slave works like this:

Master sending data
Master initiates transfer
Master addresses Slave
Master sends data to Slave
Master terminates transfer

Master receiving data
Master initiates transfer
Master addresses Slave
Master receives data from Slave
Master terminates transfer

The I^2C specification actually allows for multiple Masters to exist on a common bus and provides a method for arbitrating between them. That's a bit beyond the scope of what we need to do so we're going to keep things simple. In our setup, the BS2 (or BS2e or BS2sx) will be the Master and anything connected to it will be a Slave.

You'll notice in I^2C schematics that the SDA and SCL lines are pulled up to Vdd (usually through 4.7K). The specification calls for device bus pins to be open drain. To put a high on either line, the associated bus pin is made an input (floats) and the pull-up takes the line to Vdd. To make a line low, the bus pin pulls it to Vss (ground).

This scheme is designed to protect devices on the bus from a short to ground. Since neither line is driven high, there is no danger. We're going to cheat a bit. Instead of writing code to pull a line low or release it (certainly possible – I did it), we're going to use **SHIFTOUT** and **SHIFTIN** to move data back and forth. Using **SHIFTOUT** and **SHIFTIN** is faster and saves precious code space. If you're concerned about a bus short damaging the Stamp's SDA or SCL pins during

SHIFTOUT and **SHIFTIN**, you can protect each of them with a 220 ohm resistor. I've been careful with my wiring and code and haven't found this necessary.

Low Level I²C Code

At its lowest level, the I²C Master needs to do four things:

Generate a Start condition
Transmit 8-bit data to the Slave
Receive 8-bit data from Slave – with or without Acknowledge
Generate Stop condition

A Start condition is defined as a HIGH to LOW transition on the SDA line while the SCL line is HIGH. All transmissions begin with a Start condition. A Stop condition is defined as a LOW to HIGH transition of the SDA line while the clock line is HIGH. A Stop condition terminates a transfer and can be used to abort it as well.

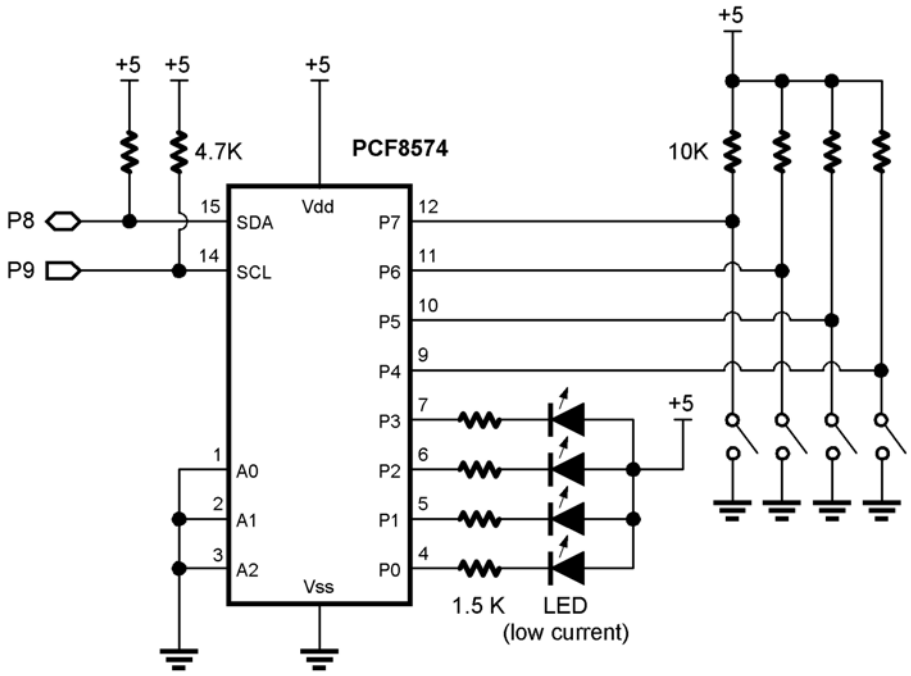
There is a brief period when the Slave can take control of the SCL line. If a Slave is not ready to transmit or receive data, it can hold the SCL line low after the Start condition. The Master can monitor this to wait for the Slave to be ready. At the speed of the BS2, monitoring the clock line usually isn't necessary but I've built the clock-hold test into the I2C_Start subroutine just to be safe.

Data is transferred eight bits at a time, sending the MSB first. After each byte, the I²C specification calls for the receiving device to acknowledge the transmission by bringing the bus low for the ninth clock. The exception to this is when the Master is the receiver and is receiving the final byte from the Slave. In this case, there is no Acknowledge bit sent from Master to Slave.

Sending and receiving data from a specific slave always requires a Start condition, sending the Slave address and finally, the Stop condition. What happens between the Slave address and the Stop are dependent on the device and what we're doing.

What you'll need to do is get the data sheet for the I²C device you want to connect to. I have found, without exception, that data sheets for I²C-compatible parts have very clear protocol definitions – usually in graphic form – that makes implementing our low-level I²C routines very simple.

Figure 85.1: PCF8574 to BASIC Stamp 2 Schematic



Let's Make It Work

Now, I'd love to have you all believe that I'm the sharpest knife in the drawer ... but we all know that isn't the case and I've just admitted to being a Jonny-come-lately as far as I^2C is concerned. So let me tell you that the working I^2C code I'm presenting here is my version of similar code that I have obtained from several sources. A quick web search will turn up many sites that use I^2C devices with the BS2, BS2e and BS2sx.

To demonstrate the use of I^2C we'll work with two components, one very simple and the other a little more sophisticated, but no more difficult to use. Both are useful in robotics projects.

The first is the Philips PCF8574 I/O port expander. I've used it with the BS2p and bring it up again because of its utility and how easy it is to communicate with. The PCF8574 has eight I/O pins that can be used either as input, outputs or in combination. The spec sheet calls for the pins –

whether inputs or outputs – to be active low. Inputs, then, should be pulled up to Vdd and taken to Vss when active. For outputs, the device will sink current – but not very much. Only three milliamps per pin, actually. So use low current LEDs or a buffer if you need more current from a PCF8574 output.

The PCF8574 has no data direction register and we must always write or read the full eight bits. When using it only for outputs or only for inputs, this isn't a problem. But when mixing I/O, things get just a bit tricky. You see, if we write a zero to a pin that is being used as an input, the next read cycle can read back that zero and make it look like a false input. The way to avoid this problem is to mask any input pins (with a one) when we do a write.

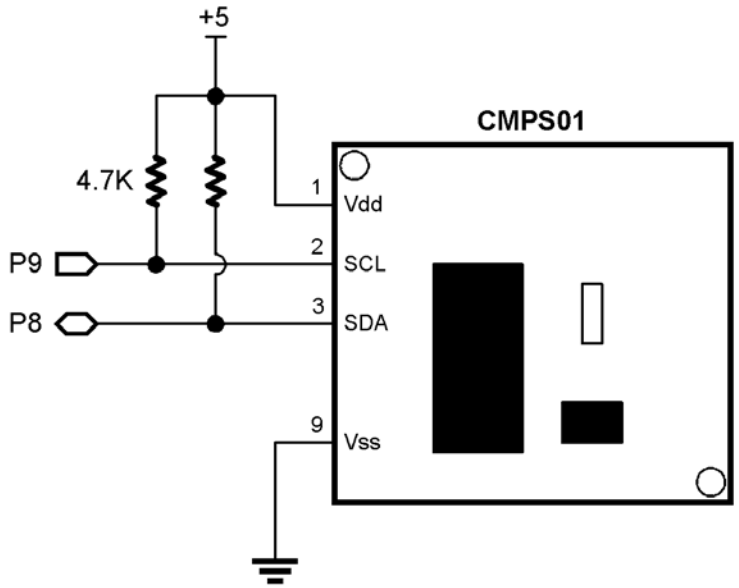
Program Listing 85.1 is the program that goes along with the schematic in Figure 85.1. In this program we will use a single PCF8574 to display a four-bit counter and read back four switch inputs. To prevent the counter write cycle from creating false inputs, the counter (inverted for active-low outputs) is ORed with a constant called MixDDR. In this constant, a one represents an input pin, zero an output since inputs are pulled-up and outputs are active low.

The subroutine called Write_PCF8574 takes care of the details and, as you can see, it is very straightforward. First, the Start condition is generated. The next step is to send the Slave address. The upper four bits of the Slave address define the device type. Bits two, three and four are the physical device address. With three address bits, we can have up to eight PCF8574 chips on the same bus, giving us up to 64 bits of I/O (you'll need to make the Slave address a variable to do this). Bit zero of the Slave address defines write (when zero) or read (when one).

I2C_TX_Byte is used to send eight bits to the Slave device and to read back the acknowledge bit. Notice how simple this is using **SHIFTOUT** and **SHIFTIN**. There may be times when you'll want to check the received ACK bit to make sure everything is working. The PCF8574 is a simple device, so this won't be necessary. One other thing that I should point out is that both **SHIFTOUT** and **SHIFTIN** take care of setting the specified data and clock pins as required, so it doesn't matter that we enter into I2C_TX_Byte with the SCL line set as an input.

You may wonder why the data byte (masked counter) is transmitted twice. The reason is that the PCF8574 behaves like a shift register. The first write places data into an internal holding register and subsequent writes force the holding register to the output pins. For programs that may not be refreshing the PCF8574 as frequently as we are here, writing twice ensures that the outputs reflect their proper state. After the second write, we must generate a Stop condition to terminate the transfer and free the bus.

Figure 85.2: CMPS01 Electronic Compass



Reading data back from the PCF8574 is just as straightforward. Read_ PCF8574 generates a Start, transmits the Slave address (this time with bit zero set to one for read), reads from the device then generates a Stop. Remember what I mentioned earlier, that the final read, when the Master is receiver, does not send an ACK (low) bit. Since we're only reading one byte, we'll use I2C_RX_Byte_Nak.

Notice that this is really just as entry point for I2C_RX and sets the *i2cAck* variable to a one. I2C_RX does the work by shifting in eight bits, starting with the MSB. **SHIFTOUT** is used to send *i2cAck*. In this case it's a one, so the bus is high (NAK) during the ninth clock pulse.

Read_ PCF8574 returns data to the main code in a variable called *i2cData*. To make things easy, the variable called *switches* is aliased to the upper four bits of *i2cData* (since the inputs on the PCF8574 are P4 .. P7). A few **DEBUG** statements are used in the main body to update the display. All-in-all, this one is pretty easy, and demonstrates the utility of the PCF8574.

Next up is another neat device from those cool guys across the pond at Devantech. This one is the CMPS01 compass module (available from Acroname). The CMPS01 is an electronic compass that will give us readings in Brads (0 to 255) or in tenths of Degrees (0.0 to 359.9). To make it compatible with other I^2C devices, the engineers at Devantech designed the CMPS01 to behave like a typical memory device.

This program (Program Listing 85.2), like the PCF8574, is very easy so I'm not going to cover it line-by-line. I just want to go over a couple of the high-level subroutines because they demonstrate techniques that will be used in many other I^2C devices.

The first subroutine to look at is called Write_Word. This routine writes a 16-bit variable to the CMPS01, starting at the locations specified by *i2cReg*. Notice that after the high byte is written, the low byte is written without worry of the register number. The reason is that the register number is automatically incremented after each write. This allows us to send a stream of contiguous bytes to the device. For the CMPS01, we only need to send two bytes, but for other devices (like an RTC), it might convenient to write several bytes without having to set the address for each.

Now we'll look at the routines for reading from the CMPSO1. To read from a location we will actually begin what looks like a write cycle. We need to do this to set the register address. Once the register address is sent, another Start condition is generated. This is what actually sets the register number and then allows us to do the read from it. As with writes, the register number is incremented with each read cycle. This allows us to read a 16-bit variable (as in Read_Word) by specifying the address of the of high byte.

I really like the CMPS01 and have it designed into a little robot I'm working on to collect empty soda cans as part of the Dallas Personal Robotics Group's Roborama contest.

Mixing And Matching ... Sort Of

As you've seen, it's pretty simple to write code for I^2C devices with the core subroutines developed in our demo programs. One thing that I haven't yet discussed is the use of a variable for the SDA pin. The reason is this: You will find devices that have no internal addresses (PCF8574), some with less than 256 locations so they use a single address byte (CMPS01) and some with enough locations to require two address bytes (24LC32 EEPROM).

I made the SDA line a variable because it is possible that a particular application will require more than one SDA line to prevent devices from stepping on each other. I spent a frustrating day swapping out RTC chips that I thought were bad only to find that an EEPROM was stepping on the RTC's transmissions. The SCL line can be shared with all devices since we can only talk to

one device at a time. If your own project uses just one device, or devices are compatible, you can simplify the code a bit by using the SDA constant in the low-level I2C routines. Otherwise, set the value of *i2cSDA* to the bus pin you want to use, then call the routines.

You'll find the code and schematic for that RTC (PCF8583) in the ZIP file that goes along with this article. There's also code for a couple of EEPROMs and the PCF8591 four-channel A2D.

BS2p Update

For those of you using I^2C with the BS2p, an upgrade will be available shortly. The upgrade does two things: (1) It extends the clock-hold timeout period for so that intelligent devices (like the Devantech compass) have time to do their internal processing, and (2) With the Version 1.33 compiler, you no longer have to specify an internal address byte for devices that don't need them (like the PCF8574 and PCF8591).

See the Parallax web site for details on getting your BS2p module upgraded.

Next Month...

Believe it or night, I'm actually thinking ahead for a change. So, what's up? Well, next month we'll *spear* our embedded control problems with the new Javelin Stamp. Can you say Object Oriented Programming in a BS2-sized module? If you haven't heard the news yet, this new module from Parallax is physically and electrically identical to the BS2sx and has these features:

- Programs in (a subset of) Sun's Java language
- Has 32K of flat program space
- Has 32K of RAM (space not used by program is available for variables)
- Can run up to six background processes (virtual peripherals) concurrent with main program

The background processes are particularly exciting. With the Javelin you can receive or send serial data, control servos or motors with PWM, measure an analog voltage or even spit one out, have precise timer functions – all without affecting the foreground program.

To be fair, the Javelin is a very sophisticated beast but once you get used to it, it's a heck of a lot of fun. If you want to get a jump on next month's article, be sure to visit the Javelin web site and download the documentation.

Until then, have fun with I^2C devices and Happy Stamping.

```
' =============================================================================
'
'   Program Listing 85.1
'   File..... PCF8574.BS2
'   Purpose... PCF8574 control via I2C
'   Author.... Jon Williams
'   E-mail.... jonwms@aol.com
'   Started... 20 MAR 2002
'   Updated... 29 MAR 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' This program demonstrates essential I2C routines and communication with the
' Philips PCF8574 port expander.  The expander is a quasi-bidirectional device;
' you can write to outputs or read from inputs no data direction register.
'
' Inputs and outputs are active low.  When writing to the device, a "1"
' should be written to any pin that is used an input.


' -----------------------------------------------------------------------------
' Revision History
' -----------------------------------------------------------------------------


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------
'
SDA             CON     8                       ' I2C serial data line
SCL             CON     9                       ' I2C serial clock line


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

DevType         CON     %0100 << 4              ' device type
DevAddr         CON     %000 << 1               ' address = %000 -> %111
Wr8574          CON     DevType | DevAddr | 0   ' write to PCF8574
Rd8574          CON     DevType | DevAddr | 1   ' read from PCF8574

ACK             CON     0                       ' acknowledge bit
NAK             CON     1                       ' no ack bit

MixDDR          CON     %11110000               ' 1 = input for mixed I/O

Yes             CON     0
No              CON     1
```

```
CrsrXY          CON     2                       ' DEBUG Position Control


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

i2cSDA          VAR     Nib                     ' I2C serial data pin
i2cData         VAR     Byte                    ' data to/from I2C device
i2cWork         VAR     Byte                    ' work byte for I2C TX code
i2cAck          VAR     Bit                     ' ACK bit from device

counter         VAR     Nib
switches        VAR     i2cData.HighNib         ' from PCF8574


' -----------------------------------------------------------------------------
' EEPROM Data
' -----------------------------------------------------------------------------


' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  PAUSE 250                                     ' let DEBUG open
  DEBUG CLS, "PCF8574 Demo"
  DEBUG CrsrXY, 0, 2, "Counter:  ", BIN4 counter
  DEBUG CrsrXY, 0, 3, "Switches: ", BIN4 switches

  i2cSDA = SDA                                  ' define SDA pin
  i2cData = %11111111                           ' clear outputs
  GOSUB Write_PCF8574
  IF (i2cAck = ACK) THEN Main                   ' device is present

  DEBUG CLS, "Error: No ACK from PCF8574"
  END


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  FOR counter = 0 TO 15
    DEBUG CrsrXY, 10, 2, BIN4 counter           ' display counter on screen
    i2cData = MixDDR | ~counter                 ' mask inputs
    GOSUB Write_PCF8574                         ' display counter on LEDs
    GOSUB Read_PCF8574                          ' get data from PCF8574
    DEBUG CrsrXY, 10, 3, BIN4 switches          ' display switch inputs
    PAUSE 100
  NEXT
  GOTO Main

  END
```

```
' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

' Data to be sent is passed in i2cData

Write_PCF8574:
  GOSUB I2C_Start                               ' send Start
  i2cWork = Wr8574                              ' send address
  GOSUB I2C_TX_Byte
  i2cWork = i2cData
  GOSUB I2C_TX_Byte                             ' send i2cData to device
  GOSUB I2C_TX_Byte                             ' force to pins
  GOSUB I2C_Stop                                ' send Stop
  RETURN


' Data received is returned in i2cData

Read_PCF8574:
  GOSUB I2C_Start                               ' send Start
  i2cWork = Rd8574                              ' send address
  GOSUB I2C_TX_Byte
  GOSUB I2C_RX_Byte_Nak                         ' get byte from device
  i2cData = i2cWork
  GOSUB I2C_Stop                                ' send Stop
  RETURN


' -----------------------------------------------------------------------------
' Low Level I2C Subroutines
' -----------------------------------------------------------------------------

' --- Start ---

I2C_Start:                                      ' I2C start bit sequence
  INPUT i2cSDA
  INPUT SCL
  LOW i2cSDA                                    ' SDA -> low while SCL high

Clock_Hold:
  IF (Ins.LowBit(SCL) = 0) THEN Clock_Hold      ' device ready?
  RETURN


' --- Transmit ---

I2C_TX_Byte:
  SHIFTOUT i2cSDA,SCL,MSBFIRST,[i2cWork\8]      ' send byte to device
  SHIFTIN i2cSDA,SCL,MSBPRE,[i2cAck\1]          ' get acknowledge bit
  RETURN


' --- Receive ---

I2C_RX_Byte_Nak:
  i2cAck = NAK                                  ' no ACK = high
```

```
  GOTO I2C_RX

I2C_RX_Byte:
  i2cAck = ACK                                  ' ACK = low

I2C_RX:
  SHIFTIN i2cSDA,SCL,MSBPRE,[i2cWork\8]         ' get byte from device
  SHIFTOUT i2cSDA,SCL,LSBFIRST,[i2cAck\1]       ' send ack or nak
  RETURN


' --- Stop ---

I2C_Stop:                                       ' I2C stop bit sequence
  LOW i2cSDA
  INPUT SCL
  INPUT i2cSDA                                  ' SDA --> high while SCL high
  RETURN
```

```
' =============================================================================
'
'   File...... CMPS01.BS2
'   Purpose... Daventech CMPS01 Electronic Compass Demo
'   Author.... Jon Williams
'   E-mail.... jonwms@aol.com
'   Started... 10 MAR 2002
'   Updated... 29 MAR 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' This program demonstrates essential I2C routines and communication with the
' Daventech CMPS01 electronic compass.  The Daventech compass behaves very
' like a typical I2C memory device and the routines to read from and write to
' it are identical to those used with EEPROMs.


' -----------------------------------------------------------------------------
' Revision History
' -----------------------------------------------------------------------------


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

SDA             CON     8                       ' I2C serial data line
SCL             CON     9                       ' I2C serial clock line


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

WrCMPS01        CON     $C0                     ' write to compass
RdCMPS01        CON     $C1                     ' read from compass

Ack             CON     0                       ' acknowledge bit
Nak             CON     1                       ' no ack bit

' Compass registers
'
CMPS01_Rev      CON      0
```

```
CMPS01_Brads    CON      1                      ' bearing, 0 - 255
CMPS01_DegHi    CON      2                      ' degrees, high byte
CMPS01_DegLo    CON      3                      ' degrees, low byte
CMPS01_S1THi    CON      4                      ' sensor 1 test, high
CMPS01_S1TLo    CON      5                      ' sensor 1 test, low
CMPS01_S2THi    CON      6                      ' sensor 2 test, high
CMPS01_S2TLo    CON      7                      ' sensor 2 test, low
CMPS01_S1CHi    CON      8                      ' sensor 1 cal, high
CMPS01_S1CLo    CON      9                      ' sensor 1 cal, low
CMPS01_S2CHi    CON     10                      ' sensor 2 cal, high
CMPS01_S2CLo    CON     11                      ' sensor 2 cal, low
CMPS01_X1       CON     12                      ' not used
CMPS01_X2       CON     13                      ' not used
CMPS01_CalDone  CON     14                      ' calibration done flag
CMPS01_CalCmd   CON     15                      ' calibration cmd register

CrsrXY          CON     2                       ' DEBUG Position Control


' -----[ Variables ]-------------------------------------------------------
' Variables
' -------------------------------------------------------------------------

i2cSDA          VAR     Nib                     ' I2C serial data pin
i2cData         VAR     Word                    ' data to/from device
i2cReg          VAR     Byte                    ' register address
i2cWork         VAR     Byte                    ' work byte for TX routine
i2cAck          VAR     Bit                     ' Ack bit from device

temp            VAR     Word                    ' for rj printing
digits          VAR     Nib
width           VAR     Nib


' -----[ EEPROM Data ]-----------------------------------------------------
' EEPROM Data
' -------------------------------------------------------------------------


' -----[ Initialization ]--------------------------------------------------
' Initialization
' -------------------------------------------------------------------------

Init:
  PAUSE 250
  DEBUG CLS
  DEBUG CrsrXY, 0, 0, "Devantech CMPS01 Compass Demo"
  DEBUG CrsrXY, 0, 1, "-----------------------------"

  i2cSDA = SDA                                  ' define SDA pin
  i2cReg = CMPS01_Rev                           ' compass revision number
```

```
  GOSUB Read_Byte
  DEBUG CrsrXY, 0, 3, "Rev Num... "
  temp = i2cData
  width = 3
  GOSUB RJ Print

  DEBUG CrsrXY, 0, 5, "Brads..... "
  DEBUG CrsrXY, 0, 6, "Degrees... "


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  i2cReg = CMPS01 Brads                         ' get brads, 0 - 255
  GOSUB Read Byte
  DEBUG CrsrXY, 11, 5
  temp = i2cData
  GOSUB RJ Print

  i2cReg = CMPS01 DegHi                         ' get degrees, 0.0 - 359.9
  GOSUB Read Word
  DEBUG CrsrXY, 11, 6
  temp = i2cData / 10
  GOSUB RJ Print
  DEBUG ".", DEC1 i2cData, "  "

  PAUSE 250
  GOTO Main
  END


' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

RJ Print:                                       ' right justify
  digits = width
  LOOKDOWN temp, <[0,10,100,1000,65535], digits
  DEBUG REP " "\(width - digits), DEC temp
  RETURN


' -----------------------------------------------------------------------------
' Compass Access Subroutines
' -----------------------------------------------------------------------------

' Writes low byte of i2cData to i2cReg

Write_Byte:
```

```
  GOSUB I2C_Start
  i2cWork = WrCMPS01
  GOSUB I2C TX Byte                          ' send device address
  i2cWork = i2cReg
  GOSUB I2C TX Byte                          ' send register number
  i2cWork = i2cData.LowByte
  GOSUB I2C_TX_Byte                          ' send the data
  GOSUB I2C Stop
  RETURN


' Writes i2cData to i2cReg

Write Word:
  GOSUB I2C Start
  i2cWork = WrCMPS01
  GOSUB I2C TX Byte                          ' send device address
  i2cWork = i2cReg
  GOSUB I2C_TX_Byte                          ' send register number
  i2cWork = i2cData.HighByte
  GOSUB I2C TX Byte                          ' send the data - high byte
  i2cWork = i2cData.LowByte
  GOSUB I2C TX Byte                          ' send the data - low byte
  GOSUB I2C_Stop
  RETURN


' Read i2cData (8 bits) from i2cReg

Read_Byte:
  GOSUB I2C_Start
  i2cWork = WrCMPS01
  GOSUB I2C TX Byte                          ' send compass address
  i2cWork = i2cReg
  GOSUB I2C TX Byte                          ' send register number
  GOSUB I2C_Start                            ' repeat start (sets register)
  i2cWork = RdCMPS01
  GOSUB I2C TX Byte                          ' send read command
  GOSUB I2C RX Byte Nak
  GOSUB I2C Stop
  i2cData = i2cWork                          ' return the data
  RETURN


' Read i2cData (16 bits) from i2cReg

Read Word:
  GOSUB I2C_Start
  i2cWork = WrCMPS01
  GOSUB I2C TX Byte                          ' send compass address
  i2cWork = i2cReg
```

```
  GOSUB I2C_TX_Byte                             ' send register number
  GOSUB I2C Start                               ' repeat start (sets register)
  i2cWork = RdCMPS01
  GOSUB I2C TX Byte                             ' send read command
  GOSUB I2C RX Byte
  i2cData.HighByte = i2cWork                    ' read high byte of data
  GOSUB I2C_RX_Byte_Nak
  GOSUB I2C Stop
  i2cData.LowByte = i2cWork                     ' read low byte of data
  RETURN


' -----------------------------------------------------------------------------
' Low Level I2C Subroutines
' -----------------------------------------------------------------------------

' --- Start ---

I2C_Start:                                      ' I2C start bit sequence
  INPUT i2cSDA
  INPUT SCL
  LOW i2cSDA                                    ' SDA -> low while SCL high

Clock_Hold:
  IF (Ins.LowBit(SCL) = 0) THEN Clock_Hold      ' device ready?
  RETURN


' --- Transmit ---

I2C_TX_Byte:
  SHIFTOUT i2cSDA,SCL,MSBFIRST,[i2cWork\8]      ' send byte to device
  SHIFTIN i2cSDA,SCL,MSBPRE,[i2cAck\1]          ' get acknowledge bit
  RETURN


' --- Receive ---

I2C RX Byte Nak:
  i2cAck = Nak                                  ' no Ack = high
  GOTO I2C RX

I2C_RX_Byte:
  i2cAck = Ack                                  ' Ack = low

I2C RX:
  SHIFTIN i2cSDA,SCL,MSBPRE,[i2cWork\8]         ' get byte from device
  SHIFTOUT i2cSDA,SCL,LSBFIRST,[i2cAck\1]       ' send ack or nak
  RETURN
```

```
' --- Stop ---

I2C Stop:                                       ' I2C stop bit sequence
  LOW i2cSDA
  INPUT SCL
  INPUT i2cSDA                                  ' SDA --> high while SCL high
  RETURN
```