

# Using an SSN as a Base Station

Joel Jordan

September 17, 2004

## 1 Introduction

Any SSN node can be used as a base station when programmed with the correct firmware. This document explains how to use the SSN debugging board with a node and a PC to create a base station. It concludes with an explanation of the base station firmware on the SSN and debugging board.

## 2 Building a Base Station

A base station can be made by loading the correct firmware onto an SSN node, then plugging in a debugging board to the expansion connector and hooking this up to a PC via a serial port. The document “Programming the SSN” gives details on how to put the base station firmware onto the SSN. This program is designed to receive packets via its radio module, then forward them to the debugging board over an I<sup>2</sup>C link. The debugging board translates these packets into standard RS-232 signals for a PC serial port to read.

The debugging board (schematic shown in Figure 1) also needs the proper firmware loaded onto its PIC18F252 to do the I<sup>2</sup>C-to-RS-232 translation. First, a serial bootloader is put onto the PIC using a normal programmer, such as the Microchip PICSTART or microEngineering Labs EPIC Plus programmer. Then, the debugging board base station firmware can be loaded onto the PIC18F252 with Microchip’s serial bootloader software, available from their website. This program is shown in Figure 2. The stock serial bootloader PIC firmware on the Microchip website needs some modification to run on a PIC18F252. This mostly means removing statements which access registers not available on the PIC18F252. A modified version is provided on the SigArch website.

To program the PIC18F252 with the serial bootloader, first connect using the desired serial port. The program should tell you that it has found a PIC18F252 with firmware version 0.10. Erase the PIC using the button that has a red chip with a green ‘X’ through it. Open the HEX file you wish to write to the PIC using the folder icon on the far left, then write your firmware using the button to the left of the erase button. Finally, push the green arrow button on the far right to put the PIC into “run” mode.



Table 1: Packet format.

Byte offset	Contents
0	Start Code (0xEFBEADDE)
4	Node ID (16 bits)
5	Time Code (32 bits)
9	Light Level
10	Sound Level
11	Checksum (8 bits)

Once both PICs are programmed with the base station firmware and the SSN is connected to the debugging board, JP6 and JP7 on the SSN should be left open, and JP1 on the debugging board should be shunted. This provides 3.3 V to the SSN from the debugging board’s regulator. If a PC is connected to the debugging board with a terminal emulator (38400 baud, 8 bits, 1 stop bit, no parity), the following message should be displayed:

```
Stochastic Sensor Debugger
By Joel Jordan
Commands:
Q - Reset
F - Reset to Firmware
Base Station Online
```

If the last message (“Base Station Online”) is not displayed, this means that the SSN is not communicating with the debugging board. The two commands allow control of the debugging board. Typing Q resets the program. Typing F resets the PIC to the serial bootloader so a new program can be loaded over the serial link.

Due to a design mistake, the debugging board as designed requires a *null-modem* (or crossover) serial cable but has a female connector. A female-to-male gender changer will be necessary on the debugging board end of the cable. The board could also be easily redesigned with a male DB-9 connector.

When the base station has been properly connected, packets received by the SSN base station will be forwarded to a PC. The format of sound-level packets is given in Table 1. Multi-byte quantities are sent least significant byte first. Packets are not guaranteed to any length, so the best way to process them is to read one byte at a time while looking for the start code.

### 3 SSN Base Station Firmware Explanation

The SSN base station firmware program is given at the end of this document. This section explains it section-by-section.

It begins with configuration data for the PIC. The important things here are that the watchdog timer is off and that the oscillator selector is configured to use the internal oscillator.

This is followed by several macros defining the functions of some I/O pins. This is only for convenience in referring to the pins. It does not actually define them as inputs or outputs.

Next, variables are declared. These are separated into sections based on what the variables are used for. Also, several macros are defined for accessing flags. There are two start codes defined. The first, `StartCodeHigh` and `StartCodeLow`, are the raw baseband Manchester-encoded bitstream that the receiver looks for to begin a packet. The second, `StartCode1` and `StartCode2`, are decoded bytes that are used to verify that a valid packet start has been detected. This is explained in more detail below.

The buffer `RFPacket` is the memory where each received packet is stored before being transmitted over the I<sup>2</sup>C link.

Next, the main program code begins. At `0x0000`, the reset vector, an instruction jumps to the initialization code. Locations `0x0008` and `0x0018` are the high- and low-priority interrupt vectors, respectively. The high-priority interrupt vector is used to handle I<sup>2</sup>C requests, while the low-priority interrupt is used to handle serial port (radio) reception.

The radio receive interrupt handler is straightforward. It receives each byte and decodes it into a 4-bit nybble, then receives another byte, decodes it, and combines them. If an error occurs, as indicated by UART flags or invalid encoded symbols, the UART resets itself. This can cause strange errors where all received data is shifted out of place by four bits. It is possible that this could be handled more intelligently.

The high-priority interrupt handler was essentially borrowed from Microchip Application Note AN734. It essentially implements a state machine, where the state is derived from the bits in the synchronous serial port (SSP) flags register. The SSN is an I<sup>2</sup>C slave device, and the debugging board is the master. This means that the debugging board periodically polls the SSN to see if any packets have been received.

Next comes the initialization routine. This sets IO ports to be inputs or outputs as necessary, then displays a “Night Rider” test on the debugging board LEDs with the `LEDDemo` routine. It sets up or turns off peripherals as necessary. The routine `LoadTestString` loads the string “**Base Station Online**” into the I<sup>2</sup>C buffer, then sets the flag `I2CTXFull` so that when polled by the debugging board, the I<sup>2</sup>C interrupt will know to send the contents of the buffer.

A brief delay loop ensures that the radio is kept in the receive state for at least 15 ms, as specified in the TR1004 data sheet, and then the radio is put to sleep.

The main loop of the program consists of a call to `ReceivePacket` and a delay loop to give the debugging board enough time to read the packet. Alternatively, a loop waiting for the I<sup>2</sup>C buffer to be reported empty has been commented out. The reasoning behind this is that the start code detection doesn’t work when interrupts are being serviced. Therefore, the base station

should be given a chance to empty the buffer before `ReceivePacket` is called again. The timeout method was chosen because it does not require the debugging board to be attached for the program to proceed.

The `ReceivePacket` routine is where the program spends most of its time. First, it puts the radio in receive mode. Then, it loads the address of the packet buffer, disables the UART, sets the oscillator at 1MHz, and calls `WaitForStartCode`. When a start code has been detected, it enables the UART, checks for another two bytes of start code, and if these are seen, it proceeds to load the packet into the buffer. It first puts a length byte and its own start tag into the buffer, then reads the rest of the packet. Finally, it disables the UART again and tells the I<sup>2</sup>C handler that the buffer is full.

The next routines are straightforward. `Receive_Byte` waits for the UART to receive a full byte, then returns it. `DecodeEightToFour` decodes a Manchester-encoded byte into a nybble using a lookup table. If an invalid encoded byte is used, `-1` is returned.

The `WaitForStartCode` routine is probably the most difficult to follow. It starts by waiting for pending I<sup>2</sup>C transfers to end. Then, it clears two bytes to use as a buffer for incoming bits. Since the MSb of the 16-bit start code is a one, this prevents the program from seeing a start code before at least 16 bits have been seen. The TR1000 is put into receive mode, just in case it somehow wasn't, and interrupts are enabled.

The numbers in the comments of the `StartCode_Loop` section indicate cycle counts. Each cycle of the loop is exactly one bit period for the transmitting UART. The routine rotates each new bit into its 16-bit buffer, then compares the contents of the buffer to the start code. If an I<sup>2</sup>C event occurs while this loop is running, the loop is restarted after the I<sup>2</sup>C transfer has completed. If an I<sup>2</sup>C event occurs when the buffer is empty, this will still halt the start code detection, so it is possible that the periodic I<sup>2</sup>C polling will prevent a valid start code from being seen. This might be dealt with by disabling the I<sup>2</sup>C interrupt after the buffer has been emptied.

When a valid start code has been detected, the PIC's oscillator is set to 8 MHz and the UART baud rate generator is reconfigured for 19200 baud. The baud rate generator is restarted when a value is written to the `SPBRG` register, so this should improve synchronization.

The start code detected in this routine is the Manchester-encoded value of 0x55 with start and stop bits added. Start bits are zeroes and stop bits are ones. A picture of this can be seen in Table 2. The top row of boxes shows what gets loaded into `StartByteHigh` and `StartByteLow`, and the bottom row shows what each bit corresponds to in the Manchester-encoded, framed data stream. The start code does not include the final stop bit. The idea was that the UART would be enabled in time to see this stop bit, then synchronized itself on the edge of the first start bit it sees. I couldn't verify that this synchronization scheme actually worked this way.

The rest of the program is straightforward and commented. It includes routines for setting up peripherals, changing oscillator speed, and testing the LED display.

Table 2: Twenty-bit start code and framing bits. Bits not in boxes are ignored.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
0	0	1	1	0	0	1	1	0	1	0	0	1	1	0	0	1	1	0	1	1
St ar t	0x5						St op	St ar t	0x5						St op					

## 4 Debug Board Firmware Explanation

The firmware for the debugging board is located at the end of this document. This section explains its operation section-by-section.

The file begins with configuration information, selecting the proper PIC, setting the oscillator type properly, and disabling the watchdog timer.

Next, some constants and macros used by the I<sup>2</sup>C and serial port code are defined. Variables, including transmit and receive buffers for the UART, are also defined.

The main program starts at address 0x0200. The serial bootloader is located at the reset vector, so the debug board firmware must be located elsewhere in memory. The bootloader automatically redirects resets and interrupts to these vectors as appropriate.

The high-priority interrupt, at address 0x0208, is not used in this program. The low-priority interrupt, at address 0x0218, is used for transmit and receive events on the UART. The receive interrupt interprets single-character commands and executes them, while the transmit interrupt sends an entire buffer full of data. The `ReprogramFirmwareMode` routine is executed when the F command is sent. It writes 0xFF to the last location in data EEPROM memory to signal to the bootloader firmware that it should wait to be reprogrammed, then resets.

The initialization routine follows at the label `Main`. This sets up the SSP, UART, and Timer 0. Then, it sends the startup message over the serial port and enters the main loop.

In the main loop, the sensor node is polled, and data is received if present. Then, the timeout loop waits for Timer 0 to roll over before polling again, preventing unnecessarily frequent polling.

The `PollSensorNode` routine sets up an I<sup>2</sup>C receive request and waits for a response from the SSN. If any data is received, it is copied into the serial transmit buffer and sent to the host PC.

The rest of the program consists of routines to set up data transfers. More information about using the peripherals can be found in the PIC18F252 data sheet.

## 5 SSN Base Station Firmware Listing

```
; -----  
; soundbasestation.asm  
; By Joel Jordan  
; University of Illinois at Urbana-Champaign  
;  
; Receives sound level packets over radio  
; intelligent  
; Debug board polls periodically over I2C  
; when polled, send last packet received  
;  
; -----  
  
; -----  
; select target device  
; -----  
    #include "p18f4320.inc"  
    LIST P=18f4320, R=HEX  
  
; -----  
; select configuration bits  
; -----  
  
    __CONFIG    _CONFIG1H, _IESO_OFF_1H & _FSCM_OFF_1H & _INTIO1_OSC_1H  
    __CONFIG    _CONFIG2L, _PWRT_ON_2L & _BOR_OFF_2L & _BORV_20_2L  
    __CONFIG    _CONFIG2H, _WDT_OFF_2H & _WDTPS_256_2H  
    __CONFIG    _CONFIG3H, _MCLRE_ON_3H & _PBAD_DIG_3H & _CCP2MX_C1_3H
```

```

    __CONFIG _CONFIG4L, _DEBUG_OFF_4L & _LVP_OFF_4L & _STVR_OFF_4L

    __CONFIG _CONFIG5L, _CPO_OFF_5L & _CP1_OFF_5L & _CP2_OFF_5L & _CP3_OFF_5L
    __CONFIG _CONFIG5H, _CPB_OFF_5H & _CPD_OFF_5H
    __CONFIG _CONFIG6L, _WRT0_OFF_6L & _WRT1_OFF_6L & _WRT2_OFF_6L & _WRT3_OFF_6L
    __CONFIG _CONFIG6H, _WRTC_OFF_6H & _WRTB_OFF_6H & _WRTD_OFF_6H
    __CONFIG _CONFIG7L, _EBTR0_OFF_7L & _EBTR1_OFF_7L & _EBTR2_OFF_7L & _EBTR3_OFF_7L
    __CONFIG _CONFIG7H, _EBTRB_OFF_7H

; -----
; macros for I/O pins
; changing these does not change pin assignments
; everywhere in the code!
; these are just here for convenience when doing
; single-bit operations
; -----

#define RF_RX                PORTC, 7
#define MIC_POWER            PORTB, 4

#define RF_CTR1              PORTD, 3
#define RF_CTRO              PORTD, 2
#define RF_CTR1_TRIS        TRISD, 3
#define RF_CTRO_TRIS        TRISD, 2

; -----
; declare global variables
; -----

∞      cblock 0x0
        WSave
        BSRSave
        StatusSave
        LoopCounter          ; generic loop counter, free to use
        Flags
        StartByteHigh
        StartByteLow
        DisplayByte
        DecodeNybble
        PacketCount

    endc

```

```

; RF Packet structure
cblock
    RXPacketDest
    RXPacketSrc
endc

; RF Receive Variables
cblock
    RXFlags
    RXByte
    RXByteCounter
    RXTemp
endc

#define NextTestFlag      Flags, 0
#define RXByteReady      RXFlags, 0
#define RXNybble         RXFlags, 1
#define RXError           RXFlags, 2

; High byte includes stop bit, start bit
; which are mandated by UART
#define StartCodeHigh    b'10011010'
#define StartCodeLow     b'01100110'

; all valid packets begin with 0x0137
StartCode1 equ 0x01
StartCode2 equ 0x37

; I2C Variables
cblock
    I2CAddr                ; address associated with last I2C event
    I2CRXBuf:0x10          ; RX buffer for I2C
    I2CRXLength
    I2CTXLength            ; user counter
    I2CTXBufAddrL
    I2CTXBufAddrH
    I2CTXBufAddrU
    I2CTXIndex             ; used from ISR
    I2CState               ; status
    I2CTXFill              ; flag ready to fill TX buffer
    I2CTXFull              ; flag whether TX buf full
    Temp

```

```

        endc

; some stuff needed for I2C state machine
#define IDLE          0x00
#define RECV         0x01
#define XMIT         0x02
#define WAIT_BUF     0x03

#define I2CAddress    0x02

; buffer for received packet data
cblock
        RFPacket:0x0
        RFPacketLength:0x1
        RFPacketData:0x40
    endc

; -----
; program code
; -----

        org          0x00
ResetVector:
        goto        Initialize

        org          0x08
HighPriorityInt:
        goto        HPInt

        org          0x18
LowPriorityInt:
        ; save status
        movwf      WSave
        movff     STATUS, StatusSave
        movff     BSR, BSRSave

        ; see what caused the interrupt
        btfsc     PIR1, RCIF      ; recv interrupt?
        bra      RecvInterrupt

LPIntDone:
        movff     BSRSave, BSR

```

```

        movff    StatusSave, STATUS
        movf     WSave, w

        retfie

RecvInterrupt:
        movff    RCREG, RXTemp
        bcf     PIR1, RCIF

        ; check for framing and overrun errors
        movlw   b'00000110'
        andwf   RCSTA, w
        bnz     RecvError

        movf     RXTemp, W
        btfss   RXNybble, ACCESS
        bra     RXNewByte
        ; low nybble
        rcall   DecodeEightToFour
        iorwf   RXByte, F, ACCESS
        xorlw   0xFF
        bz      RecvError
        bcf     RXNybble, ACCESS
        bsf     RXByteReady, ACCESS
        goto    RecvIntDone

RXNewByte:
        ; high nybble first
        rcall   DecodeEightToFour
        movwf   RXByte, ACCESS
        xorlw   0xFF
        bz      RecvError
        swapf   RXByte, F, ACCESS
        bsf     RXNybble, ACCESS

RecvIntDone:
        goto    LPIntDone

RecvError:
        bcf     RCSTA, CREN
        bsf     RCSTA, CREN
        bsf     RXError
        bcf     RXNybble           ; next time, read new byte
        goto    LPIntDone

```

```

; High Priority Interrupt Handler
; used by I2C
HPInt:
    btfss    PIR1, SSPIF, 0          ; interrupt on I2C event?
    retfie   FAST                    ; break out if not
    call     service_I2C              ; and service I2C event
    retfie   FAST                    ; restore WREG, STATUS and BSR from fast return stack

; -----
; Service routine for I2C interrupts
; adapted from Microchip App note AN734
; -----
service_I2C
    movf     SSPSTAT, w, 0            ; get SSP status
    andlw    b'00101101'             ; and mask out bits not used to determine state
    movwf    Temp

State1
    movlw    b'00001001'             ; Receiving, last byte was an address,
    xorwf    Temp,w                  ; buffer full
    bnz      State2

    movlw    RECV                     ; mark that we're in the recv state
    movwf    I2CState
    clrf     I2CRXLength
    movf     SSPBUF,w,0                ; dummy read to get address
    bsf      SSPCON1,CKP,0             ; release clock
    bcf      PIR1, SSPIF, 0           ; we're done, clear SSPIF
    return

State2
    movlw    b'00101001'             ; Receiving, last byte was data,
    xorwf    Temp,w                  ; buffer is full
    bnz      State3

    ; handle received data here
    ; right now, we're just ignoring it

    bsf      SSPCON1,CKP,0            ; release clock
    bcf      PIR1, SSPIF, 0           ; we're done, clear SSPIF
    return

```

```

State3                                     ; Transmitting, last byte was an
movlw  b'00001100'                         ; address, buffer is empty
xorwf  Temp,w
bnz    State4

movff  RFPacketLength, I2CTXLength

movlw  XMIT                                 ; mark that we're in the xmit state
movwf  I2CState
clrf   I2CTXIndex
lfsr   FSR2, RFPacket
movlw  0x00
btfsc  I2CTXFull, 0
movf   INDF2, w, 0
call   WriteI2C
incf   I2CTXIndex, f, 0
bcf    PIR1, SSPIF, 0                       ; we're done, clear SSPIF
return

State4                                     ; Transmitting, last byte was data,
movlw  b'00101100'                         ; buffer is empty
xorwf  Temp,w
bnz    State5

lfsr   2, RFPacket                          ; load byte currently being transmitted
movf   I2CTXIndex, w
addwf  FSR2L, f
btfsc  STATUS, C
incf   FSR2H, f
movlw  0x00
btfsc  I2CTXFull, 0
movf   INDF2, w, 0
call   WriteI2C
incf   I2CTXIndex, F
movf   I2CTXIndex, w
andlw  0x3f                                 ; test buffer overflow
btfsc  STATUS, Z
clrf   I2CTXIndex
bcf    PIR1, SSPIF, 0                       ; we're done, clear SSPIF
return

```

```

State5                                ; NACK received
    movlw    b'00101000'
    xorwf    Temp,w
    bnz      I2CError
    movlw    IDLE                        ; mark that we're in the idle state
    movwf    I2CState
    bcf      I2CTXFull, 0
    bcf      PIR1, SSPIF, 0             ; we're done, clear SSPIF
    return

```

```

I2CError                                ; otherwise, hang and wait for WDT
    return

```

```

; -----
; WriteI2C
; -----

```

```

WriteI2C
    btfsc    SSPSTAT,BF,0               ; buffer full?
    goto     $-2                        ; wait for it to empty

```

```

write
    bcf      SSPCON1,WCOL,0
    movwf    SSPBUF,0
    btfsc    SSPCON1,WCOL,0           ; write collision?
    goto     write
    bsf      SSPCON1,CKP,0             ; release clock

    return

```

14

```

; -----
; main routine
; -----

```

```

Initialize:
    call     SetupIOPorts
    clrf    PacketCount

    bsf     LATB, 0
    bsf     RCON, IPEN                 ; enable interrupt priorities

    ; default to 8MHz oscillator speed

```

```

call    SetOSC8000
call    LEDDemo                ; test LEDs

; turn off unnecessary peripherals
call    DisableTimers
call    DisableCCP

movlw   b'11000111'           ; enable TMR0
movwf   TOCON                  ; with 1:256 prescale

; most other stuff should be disabled at power-on

call    SetupI2C
call    SetupUART

; make a fun initial packet
call    LoadTestString
bsf     I2CTXFull,0

; have to wait about 15ms before putting radio into
; sleep mode
clrf    LoopCounter
radio_on_delay
decfsz  LoopCounter
goto    radio_on_delay

; put radio into sleep mode
bcf     RF_CTR1
bcf     RF_CTR0

15
MainLoop:
call    ReceivePacket

movf    PacketCount, W
call    LEDDisplayByte
incf    PacketCount, F

; wait for the packet to get read
; btfsc  I2CTXFull, 0
; bra    $-2

bcf     INTCON, TMR0IF

```

```

        clrf    TMR0L           ; clear timer0
TimeoutLoop
        btfss  INTCON, TMR0IF
        bra    TimeoutLoop
        bra    MainLoop

; -----
; ReceivePacket: Wait for a full packet to be received
; -----

ReceivePacket:
        bsf    RF_CTR1
        bsf    RF_CTR0
Receive_WaitForStartCode:
        lfsr   FSR0, RFPacket

        rcall  DisableUART
        rcall  SetOSC1000
        rcall  WaitForStartCode      ; sets back to 8MHz before return
        rcall  EnableUART

        rcall  Receive_Byte
        btfsc  RXError
        bra    Receive_HandleError
        xorlw  StartCode1
        bnz    Receive_HandleError

        rcall  Receive_Byte
        btfsc  RXError
        bra    Receive_HandleError
        xorlw  StartCode2
        bnz    Receive_HandleError

        movlw  d'11'                ; sound-level packets are 9 bytes long
        movwf  LoopCounter
        movwf  INDF0
        movlw  0x5                    ; add serial start code length + 1
        addwf  POSTINC0, F            ; since I2C wants length + 1 for some reason
        movlw  0xDE
        movwf  POSTINC0
        movlw  0xAD
        movwf  POSTINC0

```

```

        movlw    0xBE
        movwf    POSTINCO
        movlw    0xEF
        movwf    POSTINCO
ReceiveLoop1:
        rcall    Receive_Byte
        btfsc    RXError
        bra      Receive_HandleError
        movwf    POSTINCO
        decfsz   LoopCounter
        bra      ReceiveLoop1
; ignore end of message markers for now
Receive_Done:
        call     DisableUART

; just send everything over I2C
        bsf      I2CTXFull, 0
        bsf      PIE1, SSPIE, 0          ; re-enable I2C interrupts

        return

Receive_HandleError:
        bcf      RXError
        bsf      PIE1, SSPIE, 0          ; re-enable I2C interrupts
        bra      Receive_WaitForStartCode

; -----
; Waits to receive one byte.
; Returns result in W
; -----

Receive_Byte:
        btfsc    RXError
        return
        btfss    RXByteReady
        bra      Receive_Byte
        movf     RXByte, W
        bcf      RXByteReady
        return

DecodeEightToFour:
        ; return -1 if the input is invalid

```

```

; table lookup used because speed is more important
; than code size here.
movwf    DecodeNybble
movlw    UPPER(EightToFourTable)
movwf    TBLPTRU
movlw    HIGH(EightToFourTable)
movwf    TBLPTRH
movlw    LOW(EightToFourTable)
movwf    TBLPTL
movf     DecodeNybble, W
bra     DoLookup

DoLookup:
    addwf    TBLPTL, F
    movlw    0
    addwfc   TBLPTRH, F
    addwfc   TBLPTRU, F
    tblrd*
    movf     TABLAT, W
    return

; Poll RX pin waiting for a start code
; Runs at 1MHz
; -----
WaitForStartCode:
    ; 1MHz at 19200 baud means is 13 cycles per bit

    ; wait for pending I2C transfers to end
    ; since they mess up our timing
StartCodeWaitI2C:
    movf     I2CState, F
    bnz     StartCodeWaitI2C

    clrf     StartByteHigh
    clrf     StartByteLow                ; first bit of start code is 1, so all 8
                                          ; zeros must be shifted out before compare
                                          ; can find valid start code

    movlw    b'00001100'                ; put TR1000 in receive mode
    iorwf    LATD, F                    ;

    bsf     INTCON, GIEL

```

```

        bsf      INTCON, GIEH          ; enable interrupts

StartCode_Loop
    bsf      STATUS, C                ; 11
    btfss   RF_RX                     ; 12
    bcf      STATUS, C                ; 0      Grab the latest bit
    rlc     StartByteLow, F           ; 1
    rlc     StartByteHigh, F         ; 2
    movlw   StartCodeHigh             ; 3  check against the start code
    xorwf   StartByteHigh, W         ; 4
    bz      StartCode_CheckHighByte ; 5  keep going until start code found
    movf    I2CState, F               ; 6  sets zero flag if I2C is idle
    bnz     StartCodeWaitI2C         ; 7
    nop                                           ; 8
    bra     StartCode_Loop           ; 9,10

        ; check the high byte
StartCode_CheckHighByte:
    movlw   StartCodeLow              ; 7
    xorwf   StartByteLow, W           ; 8
    bnz     StartCode_Loop           ; 9,10

StartCode_End:
    rcall   SetOSC8000
    movlw   d'25'                     ; 19200 bps 8MHz
    movwf   SPBRG                     ; restarts baud rate generator

        ; disable I2C while receiving a packet
        ; since the buffer is being overwritten
    bcf     PIE1, SSPIE

    return

DisableTimers:
    bcf     TOCON, TMR0ON
    bcf     T1CON, T1OSCEN
    bcf     T1CON, TMR1ON
    bcf     T2CON, TMR2ON
    bcf     T3CON, TMR3ON
    return

DisableCCP:

```

```

    movlw    0xf0
    andwf    CCP1CON, F
    andwf    CCP2CON, F
    return

```

```

; -----
; set up serial port
; -----

```

#### SetupUART:

```

    bcf      IPR1, RCIP           ; set low interrupt priority
    bcf      IPR1, TXIP          ; set low interrupt priority
    bcf      TRISC, 6             ; TX output
    bsf      TRISC, 7             ; RX input
    movlw    d'25'                ; 19200 bps 8MHz
    movwf    SPBRG                ;
    movlw    b'00000100'          ; 8-bit, async high speed, initially disabled
    movwf    TXSTA
    movlw    b'00000000'          ; serial port disabled, 8-bit, receiver disabled
    movwf    RCSTA
    return

```

#### EnableUART:

```

    bsf      RCSTA, SPEN
    bsf      RCSTA, CREN
    bsf      PIE1, RCIE
    bcf      RXByteReady
    return

```

#### DisableUART:

```

    bcf      RCSTA, CREN
    bcf      RCSTA, SPEN
    bcf      PIE1, RCIE
    return

```

#### SetupIOPorts:

```

; set all pins as digital outputs for lowest power consumption
    movlw    b'00001101'
    movwf    ADCON1                ; PORTA<0:1> analog, the rest digital
    movlw    b'00010010'
    movwf    TRISA                ; PORTA<1,4> inputs, the rest outputs

```

```

    clrf    TRISB                ; PORTB outputs (saves power)
    movlw  b'10000000'          ; PORTC<7> input, the rest outputs
    movwf  TRISC
    clrf    TRISD                ; PORTD all outputs

; set everything low but LEDs PORTD<0:1,4:7> and PORTC<1:2> (active low)
; and radio module PORTD<2:3>

    clrf    LATA                ; all values initially 0 (saves power)
    clrf    LATB
    clrf    LATC

    movlw  b'00001100'          ; radio must be started in receive mode
    movwf  LATD

    return

```

#### SetupI2C:

```

    clrf    I2CState
    clrf    I2CTXFill
    movlw  b'00011000'          ; set TRISC<3:4> to be inputs
    iorwf  TRISC, f, 0
    movlw  b'00110110'          ; set slave mode, enable MSSP,
    movwf  SSPCON1, 0
    bsf    SSPCON2, SEN, 0      ; enable clock stretching on send/recv
    movlw  I2CAddress
    movwf  I2CAddr
    movff  I2CAddr, SSPADD      ; set address
    clrf    SSPSTAT, 0
    clrf    PIR1, 0             ; clear current interrupt status
    bsf    PIE1, SSPIE, 0       ; enable I2C interrupts
    bsf    INTCON, GIEL, 0      ; enable global interrupts
    bsf    INTCON, GIEH, 0      ; (i2c is high priority by default)

    return

```

#### SetOSC31:

```

    movf    OSCCON, W
    andlw  b'10001111'
    movwf  OSCCON
    return

```

SetOSC1000:

```
    movf    OSCCON, W
    andlw  b'10001111'
    iorlw  b'01000000'
    movwf  OSCCON
    return
```

SetOSC2000:

```
    movf    OSCCON, W
    andlw  b'10001111'
    iorlw  b'01010000'
    movwf  OSCCON
    return
```

SetOSC4000:

```
    movf    OSCCON, W
    andlw  b'10001111'
    iorlw  b'01100000'
    movwf  OSCCON
    return
```

SetOSC8000:

```
    ; setup internal oscillator to run at 8MHz
    movf    OSCCON, W
    iorlw  b'01110000'
    movwf  OSCCON
    return
```

LEDDisplayByte

```
    ; Byte to show:    0 1          2 3 4567
    ; Display is PORTC<1:2>, PORTD<0:1,4:7>
    xorlw  0xFF          ; negate w

    movwf  DisplayByte
    movlw  0xF9
    andwf  LATC, F          ; mask out PORTC<1:2>

    rlncl  DisplayByte, W ; shift the last two bits
    andlw  0x06          ; mask everything else out
    iorwf  LATC, F          ; or it into the right place

    movlw  0x0C
```

```

        andwf    LATD, F                ; mask out PORTD<0:1,4:7>
        movlw   0xF0                    ; handle PORTD<4:7>
        andwf   DisplayByte, W
        iorwf   LATD, F

        rrrncf  DisplayByte, F ; finally, PORTD<0:1>
        rrrncf  DisplayByte, W
        andlw   0x03
        iorwf   LATD, F

        return

LEDDemo:
        clrfsz  Temp
        movlw   0x00
        bsf     STATUS, C
        movlw   b'11010111'           ; configure 1:256 prescale for TMR0
        movwf   TOCON

DemoLeftLoop:
        rlcf    Temp, F
        btfsc   STATUS, C
        bra     DemoRightLoop
        movf    Temp, W
        rcall   LEDDisplayByte
        rcall   DemoWait
        bra     DemoLeftLoop

DemoRightLoop:
        rrcf    Temp, F
        btfsc   STATUS, C
        bra     DemoEnd
        movf    Temp, W
        rcall   LEDDisplayByte
        rcall   DemoWait
        bra     DemoRightLoop

DemoEnd:
        bcf     TOCON, TMR0ON
        return

DemoWait:
        bcf     INTCON, TMR0IF

DemoWaitLoop:
        btfsz   INTCON, TMR0IF

```

```

        bra    $-2
        return

; -----
; LoadTestString: Loads a sample string to test
; the I2C interface
; -----
LoadTestString:
    movlw    UPPER(TestString)
    movwf    TBLPTRU
    movlw    HIGH(TestString)
    movwf    TBLPTRH
    movlw    LOW(TestString)
    movwf    TBLPTL
    movlw    TestStringLength
    movwf    Temp
    movwf    RFPacketLength
    lfsr     FSR0, RFPacketData
LoadTestString_Loop:
    tblrd*+
    movff    TABLAT, POSTINCO
    decfsz   Temp
    goto     LoadTestString_Loop
    return

EightToFourTable:
    ;      0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
db      -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1      ; 0
db      -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1      ; 1
db      -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1      ; 2
db      -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1      ; 3
db      -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1      ; 4
db      -1, -1, -1, -1, -1, 0x0, 0x1, -1, -1, 0x2, 0x3, -1, -1, -1, -1, -1      ; 5
db      -1, -1, -1, -1, -1, 0x4, 0x5, -1, -1, 0x6, 0x7, -1, -1, -1, -1, -1      ; 6
db      -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1      ; 7
db      -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1      ; 8
db      -1, -1, -1, -1, -1, 0x8, 0x9, -1, -1, 0xA, 0xB, -1, -1, -1, -1, -1      ; 9
db      -1, -1, -1, -1, -1, 0xC, 0xD, -1, -1, 0xE, 0xF, -1, -1, -1, -1, -1      ; a
db      -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1      ; b
db      -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1      ; c
db      -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1      ; d
db      -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1      ; e

```

```

        db      -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1      ; f

TestString
        db      "Base Station Online", 0xD, 0xA
TestStringEnd
        db      0x00
TestStringLength equ TestStringEnd - TestString

        end

```

## 6 Debug Board Firmware Listing

```

; debugboard.asm
; By Joel Jordan, UIUC
;
; 19200 baud, no parity, no flow-control, 1 stop bit

```

```

list p=18F252
#include      <P18F252.INC>

__CONFIG    _CONFIG1H, _OSCS_OFF_1H & _ECIO_OSC_1H
__CONFIG    _CONFIG2L, _BOR_OFF_2L & _BORV_20_2L & _PWRT_OFF_2L
__CONFIG    _CONFIG2H, _WDT_OFF_2H & _WDTPS_128_2H
__CONFIG    _CONFIG3H, _CCP2MX_ON_3H
__CONFIG    _CONFIG4L, _STVR_ON_4L & _LVP_OFF_4L & _DEBUG_OFF_4L
__CONFIG    _CONFIG5L, _CPO_OFF_5L & _CP1_OFF_5L & _CP2_OFF_5L & _CP3_OFF_5L
__CONFIG    _CONFIG5H, _CPB_ON_5H & _CPD_OFF_5H
__CONFIG    _CONFIG6L, _WRT0_OFF_6L & _WRT1_OFF_6L & _WRT2_OFF_6L & _WRT3_OFF_6L
__CONFIG    _CONFIG6H, _WRTC_OFF_6H & _WRTB_OFF_6H & _WRID_OFF_6H
__CONFIG    _CONFIG7L, _EBTR0_OFF_7L & _EBTR1_OFF_7L & _EBTR2_OFF_7L & _EBTR3_OFF_7L
__CONFIG    _CONFIG7H, _EBTRB_OFF_7H

```

```

;*****
; constants

```

```

#define I2CBaudSel      0x3F          ; baud rate = some random number
#define TestAddr        0x02          ; test slave address

#define RXDoneFlag      SerialFlags, 0 ; 1 if not transferring
#define TXDoneFlag      SerialFlags, 1 ; 1 if not transferring
#define I2CRecvError    SerialFlags, 2 ; 1 if error
#define I2CRecvZero     SerialFlags, 3 ; 1 if zero bytes received

```

```

SensorNodeAddr equ          0x2

;*****
; variables

    cblock 0x000
        I2CAddr          ; address associated with last I2C event
        I2CBuf:0x40      ; data buffer for I2C
        I2CBufLen        ; length of I2C data buffer contents
        I2CCount         ; generic counter
        DelayCount       ; delay counter
        SerialRecvBufLen ; length of serial buffer contents
        SerialTransBufLen ; length of serial xmit buffer contents
        SerialFlags
        Temp             ; scratch var
        WSave           ; interrupt context save vars
        StatusSave
        BSRSave
        RCByte          ; used in interrupt handler to free up RCREG
    endc

    cblock 0x100
        SerialTransBuf:0x100 ; data buffer for serial responses
    endc

    cblock 0x200
        SerialRecvBuf:0x100 ; data buffer for serial commands
    endc

;*****
;program code starts here
    org          0x200          ; Beginning of program EPROM
Start
    goto        Main

    org          0x208          ; Interrupt Vector
Interrupt1
    retfie     FAST

    org          0x218          ; low priority interrupt vector
Interrupt2

```

```

; handle serial port interrupts
MOVWF    WSave
MOVFF    STATUS, StatusSave
MOVFF    BSR, BSRSave

    btfsc    PIR1, TXIF                ; xmit buffer interrupt?
    goto    XmitInterrupt
    btfsc    PIR1, RCIF                ; recv interrupt?
    goto    RecvInterrupt

Interrupt2Done
    MOVFF    BSRSave, BSR
    MOVF     WSave, W
    MOVFF    StatusSave, STATUS
    RETFIE

RecvInterrupt
    movff    RCREG, RCByte
    bcf      PIR1, RCIF
; check for framing and overrun errors
    movlw   b'00000110'
    andwf   RCSTA, w
    bnz     RecvError
    movlw   b'11011111'                ; make all incoming chars uppercase
    andwf   RCByte, F
    movlw   "F"                        ; F - set to firmware reprogram mode
    xorwf   RCByte, w
    bz      ReprogramFirmwareMode

    movlw   "Q"                        ; Q - reset the PIC
    xorwf   RCByte, w
    bz      ResetPIC

RecvError
    bcf     RCSTA, CREN
    bsf     RCSTA, CREN
    goto    Interrupt2Done

ReprogramFirmwareMode
; Sets the bootloader enable condition and resets the PIC
    setf    EEDATA                    ; write FF
    setf    EEADR                      ; to location FF

```

```

        clrf    EECON1
        bsf     EECON1, WREN                ; enable write mode

        bcf     INTCON, GIEH                ; disable interrupts
        movlw   0x55                        ; required write sequence
        movwf   EECON2                      ; to prevent accidental writes
        movlw   0xAA                        ; see datasheet for details
        movwf   EECON2
        bsf     EECON1, WR                  ; set write bit

        bsf     INTCON, GIEH                ; enable interrupts

        btfsc   EECON1, WR                  ; wait for write to complete
        bra     $ - 2
        bcf     EECON1, WREN                ; disable write mode
ResetPIC
        reset

```

```

XmitInterrupt
; Note: While transmits are in progress, FSR2 is reserved for use by
; the serial port code

```

```

        BTFSC   TXDoneFlag
        GOTO    XmitDisable

        MOVFF   POSTINC2, TXREG
        DCFSNZ  SerialTransBufLen, F

```

```

        BSF     TXDoneFlag
        GOTO    Interrupt2Done
XmitDisable
        BCF     TXSTA, TXEN                ; disable xmitter
        GOTO    Interrupt2Done

```

```

; -----
; Main Program Routine
; -----

```

```

Main
        BSF     RCON, IPEN                ; enable interrupt priorities

```

```

        MOVLW    b'11000111'        ; enable TMRO
        MOVWF   TOCON                ; with 1:256 prescale

        CALL    UART_Setup
        CALL    I2C_master_setup
        BSF     INTCON, GIEH         ; enable high priority interrupts
        BSF     INTCON, GIEL         ; enable serial interrupts

        CALL    SendStartupMessage

Loop    BCF     INTCON, TMR0IF

        CLRF    TMR0L                ; clear timer0
        CALL    PollSensorNode

        BCF     INTCON, TMR0IF

TimeoutLoop
        BTFSS   INTCON, TMR0IF
        bra     TimeoutLoop
        bra     Loop

; -----
; Check for new data from sensor node
; -----

PollSensorNode
        movlw   SensorNodeAddr
        movwf   I2CAddr
        call    I2C_master_recv

        btfsc   I2CRecvError
        return
        btfsc   I2CRecvZero
        return

ForwardSensorData
        ; wait for pending transfers to finish
        btfss   TXDoneFlag
        goto    ForwardSensorData

        ; copy sensor data into serial buffer
        movff   I2CBufLen, Temp

```

```

        lfsr    FSR0, I2CBuf
        lfsr    FSR1, SerialTransBuf
CopySensorDataLoop
        movff   POSTINC0, POSTINC1
        decfsz  Temp
        goto    CopySensorDataLoop

        movff   I2CBufLen, SerialTransBufLen
        call    UART_SendBuffer
        return

; -----
; set up serial port, 38400 baud
; Assumes 8MHz oscillator
; -----

UART_Setup
        BCF     IPR1, RCIP           ; set low interrupt priority
        BCF     IPR1, TXIP          ; set low interrupt priority
        BCF     TRISC, 6             ; TX output
        BSF     TRISC, 7             ; TX input
        MOVLW   d'12'                ;
        MOVWF   SPBRG                ; 38400 bps
        MOVLW   b'00000100'          ; 8-bit, async high speed, initially disabled
        MOVWF   TXSTA
        MOVLW   b'10010000'          ; enabled, 8-bit, receiver enabled
        MOVWF   RCSTA
        BSF     PIE1, RCIE           ; turn on receive interrupt
        BSF     PIE1, TXIE           ; enable serial xmit interrupt
        BSF     TXDoneFlag           ; set up xmitter free flag

        RETURN

; -----
; UART send routine
; -----
UART_SendBuffer
        BCF     TXDoneFlag
        LFSR    FSR2, SerialTransBuf
        MOVFF   POSTINC2, TXREG      ; load first char
        DCFSNZ  SerialTransBufLen, F ; decrement counter

```

```

        BSF      TXDoneFlag
        BSF      TXSTA, TXEN          ; enable xmitter
        RETURN

; -----
; Send data over I2C bus in master mode
;   I2CAddr    - address to send to; 0x00 = broadcast
;   I2CBuf     - data to send
;   I2CBufLen  - number of data bytes to send
; -----
I2C_master_send
    movff      I2CBufLen, I2CCount    ; load counter with buf length
    lfsr       FSR0, I2CBuf          ; get pointer to I2CBuf

send_start
    bsf        SSPCON2, SEN, 0        ; create START condition
    btfsc     SSPCON2, SEN, 0        ; wait until START completes
    goto      $-2

    movf      I2CAddr, w              ; clear bit 0 (R/W#)
    andwf    0xFE
    movwf     SSPBUF, 0               ; send out slave address
    btfsc     SSPSTAT, R_W, 0        ; wait until addr sent
    goto      $-2

    btfsc     SSPCON2, ACKSTAT, 0    ; ack received? (0 = yes)
    goto      stop_send

send_byte
    movff     POSTINC0, SSPBUF        ; send byte
    btfsc     SSPSTAT, R_W, 0        ; wait until byte sent
    goto      $-2

    btfsc     SSPCON2, ACKSTAT, 0    ; ack received? (0 = yes)
    goto      stop_send

    decfsz   I2CBufLen                ; continue if bytes remain
    goto     send_byte

stop_send
    bsf        SSPCON2, PEN, 0        ; create STOP condition
    btfsc     SSPCON2, PEN, 0        ; wait for STOP to complete

```

```

        goto    $-2

        return

; -----
; Receive from an I2C slave
;   I2CAddr    - slave to receive from
;   I2CBuf     - data from slave
;   I2CBufLen  - length of data from slave
; -----
I2C_master_recv
    bcf        I2CRecvError
    bcf        I2CRecvZero

    andlw     0x00
    movwf     I2CBufLen                ; zero buffer length
    lfsr      0, I2CBuf                ; get pointer to I2CBuf

recv_start
    bsf        SSPCON2, SEN, 0          ; create START condition
    btfsc     SSPCON2, SEN, 0          ; wait until START completes
    goto      $-2

    movf      I2CAddr, w                ; set bit 0 (R/W#)

    iorlw     0x01
    movwf     SSPBUF, 0                ; send out slave address
    btfsc     SSPSTAT, R_W, 0          ; wait until addr sent
    goto      $-2

    btfss     SSPCON2, ACKSTAT, 0      ; ack received? (0 = yes)
    bra       recv_length

    bsf        I2CRecvError            ; set error flag
    bra       stop_recv

recv_length
    bsf        SSPCON2, RCEN, 0        ; first receive length byte
    bsf        SSPCON2, RCEN, 0        ; enable receive
    btfsc     SSPCON2, RCEN, 0        ; wait until receive done
    goto      $-2
    movf      SSPBUF, w, 0              ; get length byte
    btfsc     SSPSTAT, R_W, 0          ; wait until length received

```

```

        goto    $-2
        andlw   0x3F                                ; cap length at 63 bytes
        movwf   I2CBufLen
        movwf   I2CCount
        movlw   0x00                                ; send STOP if 0 bytes indicated
        cpfsgt  I2CBufLen                            ; (0 bytes = not ready)
        bra     recv_zero
        bcf     SSPCON2, ACKDT, 0                    ; send ACK
        bsf     SSPCON2, ACKEN, 0                    ; enable acknowledge
        btfsc   SSPCON2, ACKEN, 0
        goto    $-2

recv_byte
        bsf     SSPCON2, RCEN, 0                      ; enable receive
        btfsc   SSPCON2, RCEN, 0                      ; wait until receive done
        goto    $-2
        movff   SSPBUF, POSTINC0
        btfsc   SSPSTAT, R_W, 0                       ; wait until data received
        goto    $-2
        dcfsnz  I2CCount
        goto    recv_done
        bcf     SSPCON2, ACKDT, 0                    ; send ACK
        bsf     SSPCON2, ACKEN, 0                    ; enable acknowledge
        btfsc   SSPCON2, ACKEN, 0
        goto    $-2
        goto    recv_byte

recv_done
        bsf     SSPCON2, ACKDT, 0                    ; send NACK
        bsf     SSPCON2, ACKEN, 0                    ; enable acknowledge
        btfsc   SSPCON2, ACKEN, 0
        goto    $-2

stop_recv
        bsf     SSPCON2, PEN, 0                       ; create STOP condition
        btfsc   SSPCON2, PEN, 0                       ; wait for STOP to complete
        goto    $-2

return

recv_zero
        bsf     I2CRecvZero

```

```

        bra      recv_done

; -----
; Set up I2C master mode
;   I2CBaudSel - baud rate divisor
; -----
I2C_master_setup
    movlw      I2CBaudSel          ; select baud rate
    movwf      SSPADD, 0          ; set I2C baud rate
    andlw      0x00                ; enable slew rate control
    movwf      SSPSTAT, 0

    movlw      b'00011000'        ; set SCL and SDA to be inputs
    iorwf      TRISC, f, 0
    movlw      b'00111000'        ; set I2C master mode, enable serial
    movwf      SSPCON1, 0

    return

; -----
; sends startup message to PC
; -----
SendStartupMessage
    bcf        TXDoneFlag
    movlw      StartupStringEnd - StartupString
    movwf      Temp
    movwf      SerialTransBufLen
    lfsr       FSRO, SerialTransBuf
    movlw      (StartupString>>8)&0xFF    ; load base offset of StringTable
    movwf      TBLPTRH              ; high order
    movlw      StartupString&0xFF        ; low byte
    movwf      TBLPTRL

StartupMsgFillBuffer
    tblrd*+          ; load string byte
    movff      TABLAT, POSTINCO        ; store it in xmit buffer
    decfsz    Temp, F
    goto      StartupMsgFillBuffer

    call      UART_SendBuffer
    return

```

```

StringTable      equ          0x800
                 org          StringTable
StartupString
db  0xD, 0xA
db   "Stochastic Sensor Debugger", 0xD, 0xA
db  "By Joel Jordan", 0xD, 0xA
db   "Commands:", 0xD, 0xA
db   "Q - Reset", 0xD, 0xA
db   "F - Reset to Firmware", 0xD, 0xA
StartupStringEnd
db   0x00

if StartupStringEnd-StartupString > 0x100
error "Startup string too large"
endif

END                ; directive indicates end of code

```