Firmware Development for bootloader-enabled SensorStick with PIC32MX microcontroller

Part I: Introduction

This document is a guide to firmware development for bootloader-enabled SensorStick with the PIC32MX microcontroller. By studying this guide you will learn about what is involved in the development of real life products that use sensors and microntrollers. Hopefully it is a fun and rewarding way to learn that stimulates creativity.

The current version of SensorStick contains 6 types of sensor, five of which connect to 10 analog input pins of the PIC32 and one of which (the compass) connects to the IC2 pin. There are five light emitting diodes (LEDs) connecting to five general purpose input-output (GPIO) pins and a switch that connects to an external interrupt pin. These are the components you can control while developing your firmware skills. It is recommended that you ignore the compass while learning firmware programming. The compass used by SensorStick (including I^2C bus and IC2 pin) is hard to control and extremely sensitive to timing constraints. Any changes to the code that queries the sensors may change the timing and affect the ability to read the compass rendering it non-functional.

The microcontroller chip used in SensorStick is the Microchip PIC32MX440F256H. To understand the operation of the microcontroller in greater detail you may wish to refer to PIC32MX Flash Programming Specification at
http://ww1.microchip.com/downloads/en/DeviceDoc/61145G.pdf
and/or

http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en534168.

Real-time and embedded systems programming differs significantly from desktop application programming. Differences include: (i) strict timing constraints for real-time systems; (ii) hardware connections and interaction; (iii) sensitivity to correctness and robust operation for real-time systems; (iv) sensitivity to cost, weight, or power consumption for embedded devices. This significantly different skill-set is why good embedded programmers generally have no problem finding a job! You really have to understand the hardware and be able to read through data sheets to figure out how to make things work. Further, every processor or microcontroller and device is different so you have to treat each of them independently and carefully consider how they work together.

Part 2: Development Environment setup (Windows)

2.1. Download and install MPLAB Integrated Development Environment (IDE)

http://www.sensorstick.net/web_documents/MPLAB_IDE_v8_56.zip

The default installation path is C:\Program Files (x86)\Microchip. It is advisable to install it in the default location.

After IDE installation, you will be asked if want to install HI-TECH C Compiler. You could skip it. It is irrelevant here.

2.2. Download and install Microchip Application Library.

http://www.sensorstick.net/web_documents/MCHP_App_Lib_v2010_08_04_Installer.zip

The default installation path is C:\Microchip Solutions v2010-08-04. It is advisable to install it in the default location. During installation, you might see a dialog box requesting to install the newest version of JDE. If you encounter this dialog, just go ahead and install the latest JDE.

If you installed the above two elements somewhere other than the default locations, make sure you can find them. Also, make sure you reboot your system before proceeding to step 3.

Note that the version of the above two components may be different and the default locations may be different as well. Adjust the following directories accordingly. Copy the whole microchip application library folder at "C:\Microchip Solutions v2010-08-04\microchip\" to the parent folder of your project folder. For example: if your project is located at C:\Projects\SensorStick, paste a copy of microchip application library folder there and make sure the name of the folder is 'microchip'.

2.3. Copy or download the folder "SensorStick skeleton firmware project"

Copy or download the folder

http://www.sensorstick.net/web_documents/Project/sensorstick_fw_basic.zip

and place in your project directory. Using a project directory like C:\Projects\SensorStick is advisable.

2.4 Set up build option (including file look-up directory) for a firmware project.

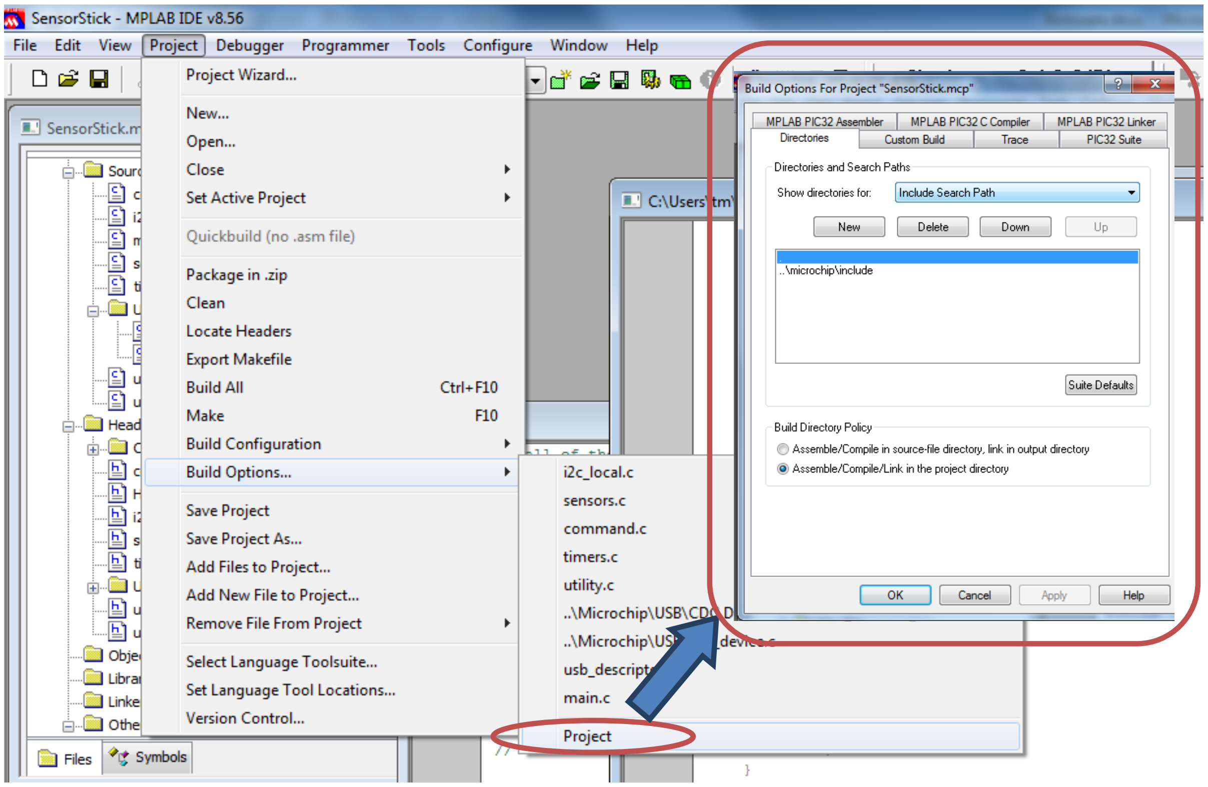
Start MPLAB by double-clicking the desktop icon or from the start menu. Once it has started, use the Project->Open menu to open the provided project (SensorStick.mcp). (You can also open the project by double-clicking the project file SensorStick.mcp) Once it has opened successfully, you will need to modify the project build options to match the setup on your computer. To do this, select the Project->Build Options...->Project menu item to open the build options for the project (See Fig.1).

Once the build options dialog is open, select the ‘Directories’ tab. Using the drop-down list, set the following options (these assume that the default locations were used to install MPLAB):

> Output Directory: hex
> Intermediary Directory: obj
> Assembler Include Search Path:
> Include search path: add "." and “..\microchip\include”
> Library search path: add "C:\Program Files (x86)\Microchip\MPLAB C32 Suite\lib" and "C:\Program Files (x86)\Microchip\MPLAB C32 Suite\pic32mx\lib"

The assembler include search path can be left blank as we won’t be doing any assembly language programming in this course.

If your version is the same as is listed here, the above setting is the default setting of the provided SensorStick firmware project so you won’t need to change them.



Show directories for: Include Search Path

..\microchip\include

( ) Assemble/Compile in source-file directory, link in output directory
(●) Assemble/Compile/Link in the project directory

Figure 1: Screenshot of MPLAB when one clicks the Project->Build Options->Project

Next, check to verify that the correct tool suite has been selected. To verify the tool suite, select the Project->Select Language Toolsuite... menu item to open the dialog box. Make sure that the Microchip PIC32 C-Compiler Toolsuite is selected and that the location of the tools is in the correct MPLAB installation directory. If it is not, browse to the correct location where MPLAB was installed. You are looking for … \Microchip\MPLAB C32 Suite\bin\pic32-gcc.exe. You can also check the memory usage of the firmware by clicking “View->Memory Usage Gauge”

2.5. Build the project and create .hex file

Build the project: you can use the menu item Project->Build All or the key combination Ctrl+F10. If successful, you will find a .hex file in the hex folder. This is your firmware file stored in Intel hex format. It is ASCII text that is the hexadecimal representation of the binary code that is programmed into the microcontroller. In some cases, the MPLAB may ask you to choose a compiler when you click the build the first time (this shouldn’t happen if you configured the tool suite). Just choose the default one.

2.6. Find bootloader program: HIDBootLoader.exe

Find the program at "C:\Microchip Solutions v2010-08-04\USB Device - Bootloaders\HID - Bootloader\HIDBootLoader.exe". It is included as part of the Microchip Application Library. You will use this program to burn your firmware into the PIC32 microcontroller chip via the USB cable.

2.7. Start the "HIDBootLoader.exe" program.

After you start the program, hold the button on the SensorStick and plug it into a USB slot. Since it is configured as a human interface device (HID), Windows will recognize and install the device automatically the first time you plug it in. It may ask you to restart the computer. If so, just click “No”. You will find that the first and second LED flash alternatively and the other two LEDs are turned on. Release your hold on the SensorStick button.

The program will recognize the device saying "Device attached". Then click "Open Hex File" and choose a hex file you want to burn to your SensorStick. Then choose "Program/Verify". Wait a while, when the process is finished, you can unplug the SensorStick and plug it in again. The four LEDs will flash a pattern and then turn off. Now you can test your firmware.

Part 3: Programming

3.1. Introduction

For more information on details you are referred to the PIC32 reference manual. Only the information required to achieve the necessary results is provided as part of this class. The full reference manual can be downloaded from the Microchip web site.

Before reading and writing to any I/O port, the desired pin(s) should be properly configured for the application. Each I/O pin has three registers (TRIS, LAT and PORT) that are directly associated with their operation.

TRIS is a data direction or tri-state control register that determines whether a digital pin is an input or an output. Setting a TRISx register bit = 1 configures the corresponding I/O pin as an input; setting a TRISx register bit = 0 configures the corresponding I/O pin as an output. All port I/O pins are defined as inputs after a device Reset. Certain I/O pins are shared with analog peripherals and default to analog inputs after a device Reset.

PORT is a register used to read the current state of the signal applied to the port I/O pins. Writing to a PORTx register performs a write to the port's latch, LATx register, latching the data to the port's I/O pins.

LAT is a register used to write data to the port I/O pins. The LATx latch register holds the data written to either the LATx or PORTx registers. Reading the LATx latch register reads the last value written to the corresponding port or latch register.

Microchip offers a number of built-in functions to control these registers, and they offer a lot of project examples. The best way to learn is read through these examples. Our SensorStick project is a great one with which to start.

3.2. SensorStick firmware project file introduction

The files you will need to study are listed along with a brief description of each file. Detailed information can be found in the file.

common.h: header file for things that are common to all files including LED/Timer control functions and data-type definitions.

utility.h+utility.c: pin initialization functions and utility functions such as checksum, reset_sensor, initialize header, etc.

time.h+time.c: functions associated with system time and timers. We are using timer2 to poll the sensors and timer3 is not used. The period for timer2 is 10 ms (100 Hz).

sensors.h+sensors.c: Source file that contains the functions associated with the sensors (accelerometer, temperature, humidity, pressure and photodetectors) for the SensorStick. It also has functions that work with specific blocks of the microcontroller such as the Analog-to-Digital Converter (ADC) that are associated with the sensors. You may wish to review the data sheets for these devices to gain a better understanding of how they work. **NOTE:** The code is compiler sensitive. When changing the sensor polling code, you may get unexpected behavior because of the black box nature of the compiler.

command.c: Source file that contains the functions associated with processing command packets.

main.c: initialization and main program loop. For most firmware, the main function includes an infinite loop. ProcessIO(void) is the actual function running in the loop. It is responsible for the packet communication between device and host.

3.3. Timing

The internal CPU clock rate on the SensorStick is 40 MHz. The PIC32 is capable of clock rates up to 80 MHz. Any operation in the firmware takes a certain amount of time that can be measured in clock cycles. For the SensorStick with the PIC32 running at 40 MHz, each clock cycle is 25ns (1 / 40000000 = 0.000000025).

Analog sampling consists of two steps: acquisition and conversion. During acquisition the analog input pin is connected to the Sample and Hold Amplifier (SHA). After the pin has been sampled for a sufficient period, the sample voltage is equivalent to the input. Then the pin is disconnected from the SHA to provide a stable input voltage for the conversion process. The conversion process converts the analog voltage to a binary representation. It typically requires 2.4 us. Thus, the Analog-to-Digital Converter (ADC) has a maximum conversion speed around 400 k samples/second (≈1000/2.4) in this implementation. Since we have 10 channels, the maximum sampling rate is around 40 kS/s (=400/10). In reality, this number will be even smaller because it takes time to manipulate data in the microcontroller. Figure 2 shows the conversion process in the PIC32.

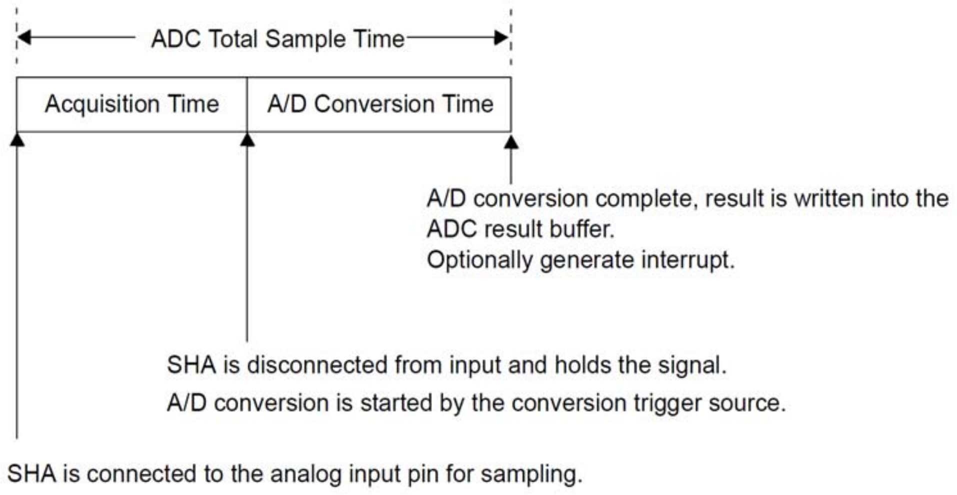

ADC Total Sample Time
Acquisition Time
A/D Conversion Time
A/D conversion complete, result is written into the ADC result buffer.
Optionally generate interrupt.
SHA is disconnected from input and holds the signal.
A/D conversion is started by the conversion trigger source.
SHA is connected to the analog input pin for sampling.

Figure 2: Timing diagram of the ADC

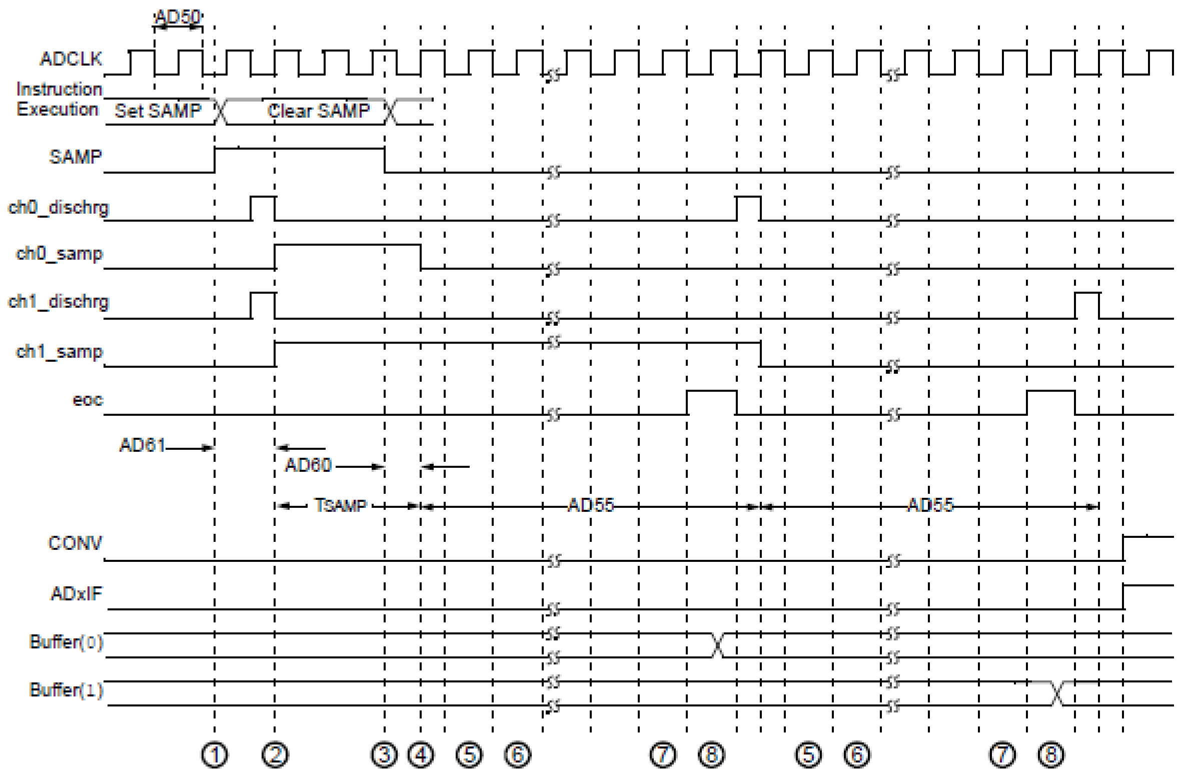
Before we continue, it's worth taking some time to understand timing as it relates to embedded devices and systems. To understand the timing characteristics of a given microcontroller or device, you have to be able to read the data sheet for the device. Generally, the timing information is given in the form of timing diagrams and tables. The diagrams show the timing characteristics as it relates to the clock for a device and the tables provide minimum, maximum and typical timing information.

Figure 3 shows the ADC conversion characteristics from the PIC32 data sheet. As can be seen, there is a lot of information packed into a single diagram. This diagram shows a manual conversion process which is done by setting a bit in a control register (the SAMP bit [1] in the ADC control register). This is shown as step 1 in the diagram. Step 2 shows the sampling of the voltage on the input pin. While the sampling is being done, the software clears the bit in the control register to start the conversion (step 3 in the diagram). This is done while the sampling is taking place. On the next clock cycle (the ADCLK waveform that is at the top of the diagram) after the bit has been cleared, the conversion of the data starts. This is step 4 in the diagram. One bit of the 10-bit output value is converted on each clock cycle. The conversion proceeds until all of the bits have been converted (step 7). It takes one additional clock cycle to complete the conversion process (step 8).

Using this diagram, we can see that the sampling takes 3 full clock cycles (look at the clock waveform from step 2 to step 4). If we have an ADC clock that is running at 4MHz, then it will take 750ns to sample the voltage on the input pin. We can determine this because the clock period is 250ns (1 / 4000000 = 0.00000025 or 250ns). The conversion takes 11 cycles (1 cycle per output bit plus 1 cycle to end the conversion) which is 2.75us (11 * 250ns = 2.75us). Adding the 2 times together gives us 3.5us per conversion (750ns + 2.75us = 3.5us). Assuming that everything else takes no time, it should be possible to perform almost 286k samples per second. Of course, time is required to read the data from the ADC data register, store it into a value or an array and then start the next conversion. It is reasonable to assume that an extra 1.25us (50 instruction clock cycles for our 40MHz clock) is required for each conversion (you would have to actually look at the compiler generated assembly code to determine the exact amount of time required). We are now up to almost 5us per conversion. If we are converting 10 ADC channels (3 for the accelerometer, 4 for the photodetectors, 1 for temperature, 1 for humidity & 1 for pressure) then we are now up to 50us to poll, convert and save the sensor data. This means that even if the SensorStick was doing nothing but polling the sensors, it could at most do 20k S/s for each sensor. After adding in the time required for interrupt handling, USB data transfers and other activities, you can easily see how important it is to keep track of the timing across all activities in the device.

Now, the above estimates are a general estimate of how long it takes to poll the sensors. To truly determine how long it takes to poll the sensors, it would have to be measured much more accurately. The main point to take away from this is that it takes a lot of work (reading through data sheets, looking at timing diagrams and tables, calculating timing, etc.) to get to a point where you have a general idea of how long things are going to take in your real-time system.

Unfortunately, if your estimates are off, even by a little bit, it will be very difficult to know. There are no easy debugging tools or compiler or linker errors that will tell you that your timing is incorrect. Your system will fail in ways that are difficult to understand and even more difficult to debug. Often it is useful to decrease the polling rate to see if the problem disappears. If it does, then you can suspect that timing may be an issue at which point you break out your oscilloscope and start looking at signals on pins.



① – Software sets ADxCON. SAMP to start sampling.

② – Sampling starts after discharge period. TSAMP is described in the *"PIC32MX Family Reference Manual"* (DS61132).

③ – Software clears ADxCON. SAMP to start conversion.

④ – Sampling ends, conversion sequence starts.

⑤ – Convert bit 9.

⑥ – Convert bit 8.

⑦ – Convert bit 0.

⑧ – One TAD for end of conversion.

Figure 3 - PIC32 A/D Conversion (10-Bit Mode) Timing Characteristics

Timing is an important issue for embedded system programming. You need to schedule enough time for all of the operations that need to be performed, especially those time-consuming operations like data acquisition. Otherwise, you will find undetermined behavior such as random data loss and communication errors.

3.4. Host-to-Device Communication and Packet Format

In embedded system programming, packet design is important since your hardware needs to communicate with the host computer. In most cases, the host sends a command to the device and the device performs some action which may include sending data back to the host.

First, one needs to decide which communication protocol will be used. The PIC32MX offers USB modules, so we can use USB to communicate with the host. There are various USB device classes used to identify a device's functionality and to load a device driver based on that functionality. Among them, USB communications device class (or USB CDC) is quite popular. It is adopted in our firmware. The firmware emulates a COM port. On the host, the device will be recognized as a virtual serial port.

Next, one should decide what kind of data will be transmitted back and forth between the device and host. To ease the communication we wrap data into packets. A data packet is basally a byte series containing certain information. Typically, there are two types of packets: a command packet sent from host to the device and a response or data packet returned from the device to the host. One needs to design the packet format so that the device knows how to deal with the command and the host knows how to deal with the response packet.

A popular method which is adopted here is dividing all the packets into two parts: header and data. The idea is similar to the http data packet sent on the web. You can design each part as you like. In the given sample project, they are defined as follows (look at the file common.h in the sample for details):

```
typedef struct __attribute__((packed)) packetHeader_t {
  uint8_t         start;           // the PACKET_START character
  uint8_t         command:4;       // the command type [if applicable]
  uint8_t         type:4;          // the packet type
  uint16_t        size:12;         // the size of the packet payload
  uint16_t        unused:4;        // unused bit
  uint8_t         cs;              // the packet checksum
} packetHeader;
```

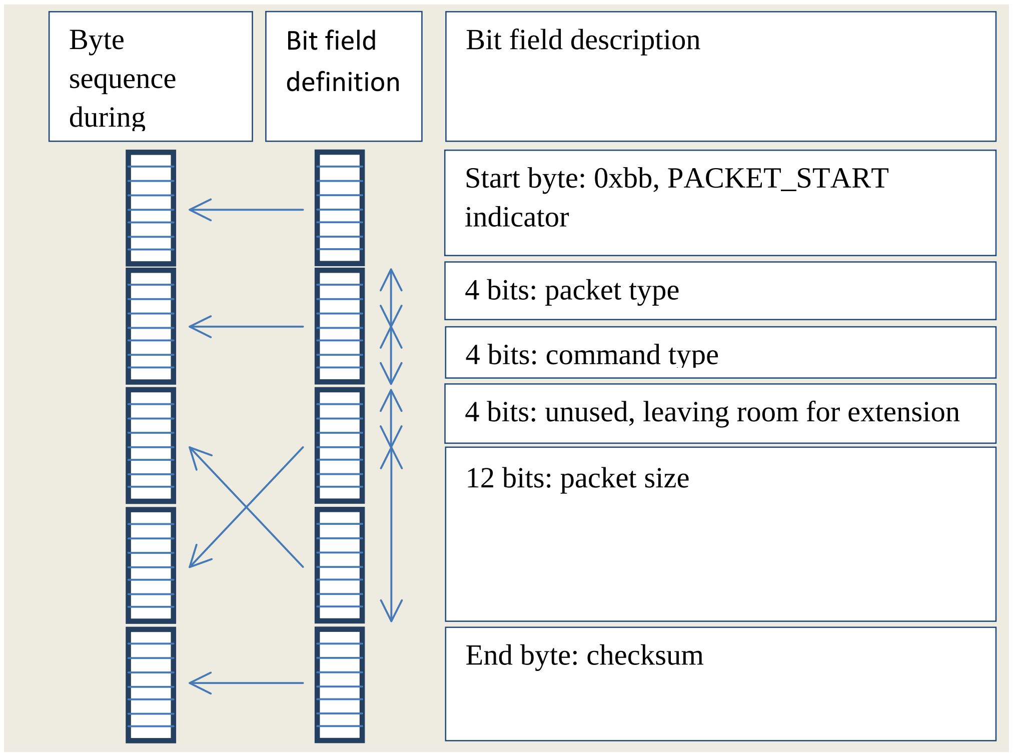


Figure 4: Packet Diagram for packet header. Note that when you define bit field, least significant bits comes first. Also note that the USB communication adopts little endian, which means the least significant byte is sent first when a data type consists of multiple bytes. But in definition, a big endian is easy to understand. That's why the fourth byte in the definition is sent before the third byte.

This definition consists of five bytes. The first byte is PACKET_START indicator and is fixed as 0xbb. It is used to identify the start of a packet from a byte series. The choice of "0xbb" is arbitrary. You can choose other values if you prefer but both the embedded firmware and host software must be changed if a different value is used. The principle is to choose something which is unlikely to appear in the packet. Note that this value might appear in the packet content as well, so one can use the following information to confirm that it is indeed a starting byte in the program of both device and host. The second byte is separated into two parts. The first four bits represent the command type (like start, stop, etc) if it is a command packet from the host. It is unused if it is a response packet from device. The second four bits represent the packet type, either command or response (called data packet in the program). The following two bytes is again divided into two parts. The first 12 bits is indicating the size of the packet after the header. This lets the developer know when to stop while reading the byte series. The next four bits are unused and are available for future extension. They will be useful for project assignments. The last byte is the checksum. It stores the last byte of the sum of each byte of the packet after the header. It is a popular technique to confirm that the packets are not corrupted during communication.

In our project, there are three types of command from the host: CONFIGURE, START & STOP.

```
typedef enum {
  CONFIGURE = 0,
  START,
  STOP
} commandType_t;
```

For the CONFIGURE packet, the payload is designed to be two bytes containing all the state information for the sensor units on the SensorStick. Therefore, the size field in the header will be 2. For the other two command packets, there is no payload so the size field in the header is 0. They simply tell the device to start and stop.

There is only one response data packet. After the device receives a CONFIGURE or START command, it starts to collect sensor measurement data every 10ms, and then construct them into a data packet that is sent back to the host.

The definition of the data packet is shown below. It contains the packet header and the payload which is the data sample time stamp and the sensor data from the SensorStick.

```
typedef struct __attribute__((packed)) dataPacket_t {
  packetHeader     header;           // packet header
  uint64_t         time_stamp;       // packet time stamp (# us since start)
  accAxes          motion;           // accelerometer data
  photoDetectors   photo;            // photodetector data
  uint16_t         temperature;      // temperature data
  uint16_t         humidity;         // humidity data
  uint16_t         pressure;         // barometric pressure
  compassPoints    compass;          // compass data
} dataPacket;
```

Byte sequence in data packet definition	Byte sequence description
	5 bytes: packet header
	8 bytes: time stamp, indicating the running time in micro second unit
	2 bytes: accelerometer output in x axis
	2 bytes: accelerometer output in y axis
	2 bytes: accelerometer output in z axis
	2 bytes: 1st photo detector output
	2 bytes: 2nd photo detector output
	2 bytes: 3rd photo detector output
	2 bytes: 4th photo detector output
	2 bytes: temperature sensor output
	2 bytes: humidity sensor output
	2 bytes: pressure sensor output
	2 bytes: magnetic compass output in x axis
	2 bytes: magnetic compass output in y axis
	2 bytes: magnetic compass output in z axis

Figure 5: Byte diagram for data packet. There are 39 bytes total. Notice that we don't address the magnetic compass data in this class. One can use these fields to output other information.

As you can see, the data packet basically attaches each sensor measurement data to the packet header. Since the collection rate is as high as 100 Hz, what the host will see is a continuous byte series. In this case, the start byte is extremely useful to identify received data.

3.5. Malab programming

We will use MATLAB to write the host program that communicates with the device. A basic MATLAB program will be provided as a starting point. The ssPCcomm.m file is responsible for the host-to-device communication. If you change the packet definition in firmware you will also need to modify the ssPCcomm.m file in MATLAB. If you wish to design a GUI interface in MATLAB it is useful to use the GUIDE command in MATLAB.

3.6. Sample Projects

The following are example projects that you may find helpful as you learn more about embedded systems firmware programming.

3.6.1. Exercise 1: Add additional LED error codes

This is an important technique to debug the firmware without using the debugging mode of the IDE. For firmware, you won't be able to use the normal printf command to debug your program. In most cases, some LEDs will be included on the device to help convey information. The information is encoded in an error code. Note that the error code can be quite versatile. Static LED on-off patterns are mostly used. Using dynamic flash patterns on the LEDs offer almost infinite possible combinations.

There is no single correct error code for this project. The following are some possible error codes. Note that LED1 is used to indicate that the device is running, so we have three other LEDs for error indication which give us 8 possible static error codes. In the project, LEDS_NO_ERRORS is defined, indicating that there is no error. For this exercise, we'll add three additional error codes. LEDS_PACKET_ERROR indicates that the packet received is not a valid packet since the header of the packet does not start with a 0xbb. LEDS_CS_ERROR indicates that the checksum of the packet is different from the value received, so the packet is corrupted during transmission. LEDS_CMD_ERROR indicates that the command type of the packet is invalid.

Add the three new LED error definitions in the file common.h as follows:

```
// define LED error codes
#define LEDS_NO_ERRORS              LED_2_OFF;LED_3_OFF;LED_4_OFF":

#define LEDS_PACKET_ERROR           LED_2_OFF;LED_3_ON;LED_4_OFF
#define LEDS_CS_ERROR               LED_2_OFF;LED_3_ON;LED_4_ON
#define LEDS_CMD_ERROR              LED_2_ON;LED_3_ON;LED_4_OFF
```

In the `parse_packet()` function in the file command.c, modify the code near the beginning of the function so that it looks like this:

```
if (header->start != PACKET_START) {
  LEDS_PACKET_ERROR;                // set bad packet error
  return;                           // no start sequence, bad packet
}
checksum = compute_checksum(&packet[sizeof(packetHeader)], header->size);
if (checksum != header->cs) {
  LEDS_CS_ERROR;                    // set bad checksum error
  return;                           // checksums don't match
}
```

Now modify the default case in `parse_packet()` function so that it looks like the code below.

```
default:
  LEDS_CMD_ERROR;
  break;
}
```

To test your new error codes, we are going to deliberately send an incorrect packet. We will do this by revising the given MATLAB program. Since the packets are constructed in the file ssPCcomm.m, all the changes are made in this file.

For LEDS_PACKET_ERROR, change the start byte in `function configpacket()`. You will see the third LED light up after you press start button.

```
   PACKET_START_BYTE=hex2dec('b1');%should be ‘bb’
```

For LEDS_CS_ERROR, uncomment the “`packet.checksum=packet.checksum+1;`” in “`function binarystring=packet_to_string(packet)`”. You will see the third and fourth LED light up after you press startbutton.

For LEDS_CMD_ERROR, uncomment the “`packet.command=5;`”, in “`function binarystring=packet_to_string(packet)`”. You will see the second and third LED light up after you press start button.

This concludes this exercise. This has demonstrated how to add a new feature to the firmware and then test it using the software running on the host.

3.6.2. Exercise 2: Add LED control command.

This exercise will show you how to design and implement command packets. The most basic function firmware should handle is to accept a command from the host and then perform a task. Allowing the host to control the LEDs on the SensorStick is an easy way to demonstration a new command type. It demonstrates using the host to direct the device to perform some action.

There are originally three types of commands defined: CONFIGURE, START and STOP. For this exercise, we'll add one more called LED_CONTROL.

Modify the commandType_t typedef in the file common.h as follows:

```
typedef enum {
  CONFIGURE = 0,
  START,
  STOP,
  LED_CONTROL
} commandType_t;
```

To be compatible with our original packet design, we follow the same packet structure: packet header + payload. Ideally, you would add a new command packet type for this purpose. To keep this exercise simple, we will take advantage of the unused bits in the packet header definition. Each of the four unused bits corresponds to the four LEDs. Notice that all the command packets received are dealt with in `parse_packet()`function in the command.c file. To handle our new command, we need to add an additional case to the `switch` statement.

```
case LED_CONTROL:
{
  if (header->unused & 1) LED_1_ON;
  else LED_1_OFF;
  if (header->unused & 2) LED_2_ON;
  else LED_2_OFF;
  if (header->unused & 4) LED_3_ON;
  else LED_3_OFF;
  if (header->unused & 8) LED_4_ON;
  else LED_4_OFF;
  break;
}
```

On the host side, one should also write a function to create the LED control command. Design a User Interface (UI) to let users to control the LED. For extra points design a GUI to do the same thing.
Again the main file you will need to revise is ssPCcomm.m, add the following code in the file (or download the revised multimeter MATLAB for the firmware course):

```
obj.ledCtrl=@public_ledCtrl;

function public_ledCtrl(ledc)
        if this.connected
            ledpacket(ledc);

fwrite(this.SensorStickSerial,this.led_decstring,'uint8','async');%send stop
command to board
        end
end
function ledpacket(ledc)
        % not prompted for
        p_type='COMMAND';
        c_type='LED_CONTROL';
        dump=true;

        % START packet assembly
        PACKET_START_BYTE=hex2dec('bb');% from settings.py
        PACKET_START_CHAR = '\xbb';% from settings.py
        PACKET_HEADER_SIZE = 5;% bytes from settings.py

        % from hard coded configuration
        ledpacket.start=PACKET_START_BYTE;
        ledpacket.packet_type=packet_type_convert(p_type);
        ledpacket.command=command_type_convert(c_type);
        ledpacket.size=command_size(ledpacket.command);
        ledpacket.dump=dump;
        ledpacket.revision=bin2dec(num2str([ledc.l1  ledc.l2  ledc.l3
ledc.l4]));

        ledpacket.checksum=0;

        ledbstr=packet_to_string(ledpacket);% start
        this.led_decstring=todecimalstring(ledbstr);% for sending to
SensorStick
end
```

The MATLAB multimeter GUI has been revised to support LED control. Download the revised multimeter MATLAB for the firmware course and run it. After you press the connect & run button, you will be able to control the LEDs by checking and unchecking the Compass/Humidity/Pressure/Temperature checkboxes. Each box represents one LED. Check and uncheck the various checkboxes and then press the **LED control** button to see the result. Note that the magnetic compass is not enabled.

3.6.3. Exercise 3: Add a new duration running mode

There is currently 1 mode in which to run the device: load and go. For this exercise, we are going to add a new mode: running for a fixed duration. To implement this new mode, you will need to send additional information in the configure packets. In this particular case, we want the device to run for a predetermined period and stop automatically. To support this new mode, we need to add one additional mode (we'll call it TIMED_RUN) to the `sensorMode_t` typedef and one additional field called duration in the `configPacket` structure in the file common.h as follows:

```
typedef enum {
  LOAD_AND_GO = 0,
  TIMED_RUN
} sensorMode_t;

typedef struct __attribute__((packed)) configPacket_t {
  packetHeader     header;          // packet header
  uint16_t         mode:2;          // the desired mode
  …
  uint32_t         duration;        // # of seconds
} configPacket;
```

Notice that `ProcessIO(void)` in the file main.c is the function that is called inside the infinite loop in the function `main()`. Just before the end of the function, add the following code to check whether the elapsed time is greater than the predetermined duration variable sent from host.

```
if ((mode == TIMED_RUN) && (elapsed_time >= duration)) {
  mINT0IntEnable(FALSE);
  DisableIntT2;
  elapsed_time = 0LL;
  WHITE_LED_OFF;
  LED_1_OFF;
}
```

Write or modify MATLAB code to test the new duration command. Calibrate timing with MATLAB and explain any systematic differences you find.

3.6.4. Exercise 4: Change timer setting and find the largest possible sample rate

The sampling rate is defined in the file common.h and used in the file timers.c.

```
#define RATE       100

void init_timers(void)
{
  OpenTimer2(TIMER_COMMON_ | T2_ON | T2_PS_1_8, (5000000 / RATE));
  ConfigIntTimer2(T2_INT_OFF | T2_INT_PRIOR_3 | T2_INT_SUB_PRIOR_2);
  reset_counters();
}
```

By the definition above, timer2 is set to run at 100Hz, which means that an interrupt will be generated every 10ms. When an interrupt occurs, the ISR (interrupt service routine) `Timer2Handler(void)` will be called. By changing RATE, you will change the sensor sampling rate and therefore effect the amount of data that is sent to the host. Try increasing the sampling rate by changing the value of `RATE`. You may have to do this several times to see the effect. At some point, you will see the host start to lose data packets. There are several possible reasons. (i) The host computer is too slow to handle all the data sent, so it simply drops the data which it can’t process. (ii) The COM model serial communication is not fast enough to handle the sampling rate. (iii) The device does not have enough time to perform the tasks (collecting sensor data in this case) for each cycle. Here, the second one is the mostly likely bottleneck for a PC. 5kS/s is probably the highest sampling rate you can reach and still obtain reliable data.

3.6.5. Exercise 5: Add sampling rate control for each sensor

In this exercise, you will enable the host to control the sampling rate of each sensor independently. You will need to add fields to set each rate to the configPacket structure in the file common.h. There are six different sensors, so we add five more fields (we're not using the compass for this class). Note that a rate over 100S/s is meaningless for temperature and humidity sensors, since the temperature and humidity won't chance very quickly and the sensors themselves won't response to rapid changes. So an 8-bit integer is enough for these two sensors.

```
typedef struct __attribute__((packed)) configPacket_t {
  packetHeader     header;          // packet header
  uint16_t         mode:2;          // the desired mode
  …
  uint16_t         motion_rate;     // accelerometer collection rate
  uint16_t         photo_rate;      // photodetector collection rate
  uint16_t         pres_rate;       // pressure collection rate
  uint8_t          temp_rate;       // temperature collection rate
  uint8_t          hum_rate;        // humidity collection rate
} configPacket;
```

In addition to adding the rates to the configuration packet, we will define a new data type called `rateGroup` and masks for determining the maximum rate value used in determining when to send out combined data packets. Add the following code to the file common.h:

```
typedef struct __attribute__((packed)) rateGroup_t {
  uint16_t         motion_rate;     // accelerometer collection rate
  uint16_t         photo_rate;      // photodetector collection rate
  uint16_t         pres_rate;       // pressure collection rate
  uint8_t          temp_rate;       // temperature collection rate
  uint8_t          hum_rate;        // humidity collection rate
} rateGroup;

// define the masks for determining the maximum rate value used in
// determining when to send out combined data packets
#define MOTION_MASK                 0x1
#define PHOTO_MASK                  0x2
#define PRESSURE_MASK               0x4
#define TEMPERATURE_MASK            0x8
#define HUMIDITY_MASK               0x10
```

From the previous exercise, we know that 5kS/s is the appropriate highest sampling rate, so set maximum rate to be 5000 for most of the sensors and 100 for the slower sensors (temperature

& humidity). To do this, add the following defines in the file common.h:

```
#define MAX_WX_RATE                  100
#define MIN_WX_RATE                  1
#define MAX_MOTION_RATE              5000
#define MIN_MOTION_RATE              1
```

In the file timers.c, modify the init_timers() function so that it looks like this:

```
void init_timers(void)
{
  OpenTimer2(TIMER_COMMON_ | T2_ON | T2_PS_1_8, (5000000 / MAX_MOTION_RATE));
  ConfigIntTimer2(T2_INT_OFF | T2_INT_PRIOR_3 | T2_INT_SUB_PRIOR_2);
  reset_counters();
}
```

The above function sets timer2 to be 5kHz, which means the ISR is called every 0.2ms (200µs). Therefore, only sample rates whose period is an integer multiple of 200µs are valid. To make the program more robust, we will add a function to check whether the rates sent from host are valid. Note that 1000000/rate gives the period with microsecond unit.

Add the valid_rates() function to the file command.c. That function is shown below.

```
BOOL valid_rates(rateGroup* packet)
{
  BOOL            valid = TRUE;    // validation error flag
  uint32_t        units;           // ISR clock units
  uint32_t        max_rate;        // max rate value

  max_rate = 0;
  units = 1000000 / MAX_MOTION_RATE; //period in us
  if ((packet->motion_rate >= MIN_MOTION_RATE) &&
      (packet->motion_rate <= MAX_MOTION_RATE)) {
    if ((1000000 % packet->motion_rate) == 0) {
      if (((1000000 / packet->motion_rate) % units) != 0) {
        valid = FALSE;
      }
    } else {
      valid = FALSE;
    }
  } else {
    valid = FALSE;
  }
```

```
if ((packet->photo_rate >= MIN_MOTION_RATE) &&
    (packet->photo_rate <= MAX_MOTION_RATE)) {
  if ((1000000 % packet->photo_rate) == 0) {
    if (((1000000 / packet->photo_rate) % units) != 0) {
      valid = FALSE;
    }
  } else {
    valid = FALSE;
  }
} else {
  valid = FALSE;
}
if ((packet->pres_rate >= MIN_MOTION_RATE) &&
    (packet->pres_rate <= MAX_MOTION_RATE)) {
  if ((1000000 % packet->pres_rate) == 0) {
    if (((1000000 / packet->pres_rate) % units) != 0) {
      valid = FALSE;
    }
  } else {
    valid = FALSE;
  }
} else {
  valid = FALSE;
}
if ((packet->temp_rate >= MIN_WX_RATE) &&
    (packet->temp_rate <= MAX_WX_RATE)) {
  if ((1000000 % packet->temp_rate) == 0) {
    if (((1000000 / packet->temp_rate) % units) != 0) {
      valid = FALSE;
    }
  } else {
    valid = FALSE;
  }
} else {
  valid = FALSE;
}
if ((packet->hum_rate >= MIN_WX_RATE) &&
    (packet->hum_rate <= MAX_WX_RATE)) {
  if ((1000000 % packet->hum_rate) == 0) {
    if (((1000000 / packet->hum_rate) % units) != 0) {
      valid = FALSE;
    }
  } else {
    valid = FALSE;
  }
} else {
  valid = FALSE;
}
```

```
  if (valid) {
    // figure out the maximum data rate so we know when to send the packet
    if (packet->motion_rate > max_rate) {
      rate_mask = MOTION_MASK;
      max_rate = packet->motion_rate;
    }
    if (packet->photo_rate > max_rate) {
      rate_mask = PHOTO_MASK;
      max_rate = packet->photo_rate;
    }
    if (packet->pres_rate > max_rate) {
      rate_mask = PRESSURE_MASK;
      max_rate = packet->pres_rate;
    }
    if (packet->temp_rate > max_rate) {
      rate_mask = TEMPERATURE_MASK;
      max_rate = packet->temp_rate;
    }
    if (packet->hum_rate > max_rate) {
      rate_mask = HUMIDITY_MASK;
      max_rate = packet->temp_rate;
    }
  }
  return valid;
}
```

In the file timers.c, modify the Timer2Handler() function so that it matches the code below:

```
void __ISR(_TIMER_2_VECTOR, ipl3) Timer2Handler(void)
{
  elapsed_time += 1000000 / MAX_MOTION_RATE;
  if (++photo_count >= photo_rate) {
    get_photos(&data_packets[create_index].photo);
    photo_count = 0;
    if (rate_mask & PHOTO_MASK) {
      send_data = TRUE;
      sample_count++;
    }
  }
  if (++acc_count >= motion_rate) {
    get_motion(&data_packets[create_index].motion);
    data_packets[create_index].motion.zero_g = zero_g_detected();
    acc_count = 0;
    if (rate_mask & MOTION_MASK) {
      send_data = TRUE;
      sample_count++;
    }
  }
  if (++presr_count >= pres_rate) {
    data_packets[create_index].pressure = get_pressure();
    presr_count = 0;
    if (rate_mask & PRESSURE_MASK) {
      send_data = TRUE;
      sample_count++;
    }
  }
  if (++tempr_count >= temp_rate) {
    data_packets[create_index].temperature = get_temperature();
    tempr_count = 0;
    if (rate_mask & TEMPERATURE_MASK) {
      send_data = TRUE;
      sample_count++;
    }
  }
  if (++humid_count >= hum_rate) {
    data_packets[create_index].humidity = get_humidity();
    humid_count = 0;
    if (rate_mask & HUMIDITY_MASK) {
      send_data = TRUE;
      sample_count++;
    }
  }
  if (++comp_count >= comp_rate) {
    get_compass(&data_packets[create_index].compass);
    comp_count = 0;
    if (rate_mask & COMPASS_MASK) {
      send_data = TRUE;
      sample_count++;
    }
  }
  mT2ClearIntFlag();
}
```

3.6.6. Exercise 6: Use push button to start and stop the LED

There is a push button on the end of the SensorStick. At present it is only used for bootloader programming. You can design your firmware to enable the button to actually control something. This project is an example in which one uses the button as a switch for the LED that is positioned among the four light sensors. There are two methods to do this: polling and using an interrupt. These methods are illustrated in the following.

Method 1: Polling

For this method, you will use timer3 to poll the status of the push button. Change the following functions in the file timers.c to match the code below.

```
void init_timers(void)
{
  OpenTimer2(TIMER_COMMON_ | T2_ON | T2_PS_1_8, (5000000 / RATE));
  ConfigIntTimer2(T2_INT_OFF | T2_INT_PRIOR_3 | T2_INT_SUB_PRIOR_2);
  OpenTimer3(TIMER_COMMON_ | T3_ON | T3_PS_1_16 , 250000);
  ConfigIntTimer3(T3_INT_OFF | T3_INT_PRIOR_3 | T3_INT_SUB_PRIOR_1);
  reset_counters();
}

void __ISR(_TIMER_3_VECTOR, ipl3) Timer3Handler(void)
{
  static BOOL     swp=1;           // previous switch info checked by timer3

  if ((sw2==0) && (swp==1)) {
    if (photo_led) {
      WHITE_LED_OFF;
      photo_led=0;
    } else {
      WHITE_LED_ON;
      photo_led=1;
    }
    swp=0;
  } else if ((sw2==1) && (swp==0)) swp=1;
  mT3ClearIntFlag();
}
```

Add following two lines in the function `UserInit(void)` in the file main.c.

```
mInitSwitch2();    // enable switch button
EnableIntT3;       // start timer3
```

Method 2: Interrupts

This is a more straightforward and preferred method. Since the push button is connected to the special GPIO pin, INT0, you can make used of the interrupt for that pin. To start, enable the INT0 interruption by modifying the init_push_button() function in the file utility.c so that it matches the following code (only the call mINT0IntEnable () to is changed):

```
void init_push_button(void)
{
  mINT0IntEnable(1);
  mINT0ClearIntFlag();
  …
}
```

Then, add the code in the `ExtInt0Handler()` function in utility.c so that it matches the following code:

```
if (photo_led) {
  WHITE_LED_OFF;
  photo_led=0;
} else {
  WHITE_LED_ON;
  photo_led=1;
}
```

3.6.7. Exercise 7: Perform a moving average calculation on real-time data

The previous exercises were all about controlling various aspects of the SensorStick. However, the computing power of the embedded system can also be exploited. It can be used to take some computing burden off the host computer and to reduce data throughput to the host. The addition of digital data filtering is one example. A very simple digital filter is a moving average.

Let’s focus on the accelerometer for the moment. You will use the compass data fields in Figure 5 as the moving average of accelerometer data.

First define the number of data points in the moving average calculation in the file common.h.

```
#define NAVE       10
```

Then in the file timers.c, we add a function to calculate the moving average of accelerometer data. Here we deliberately multiple all the values of accelerometer data by 10 so that the last digit of the integer part of the average value will be actually the first decimal value. You can redefine the data packet and use float/double to store moving average, but it will require you to redesign the host program received data processing logic as well. For the purposes of this exercise, we’re going to use a technique of converting a floating point value to an integer to save bytes when transferring the data.

```
void movingAve(int index)
{
//calculate moving ave for motion*10 so that we will have one significant
// value after decimal point when divide by 10

  float factor=10.0/NAVE;
  if (index>=NAVE){
    data_packets[index].compass.x=data_packets[index-1].compass.x-
(data_packets[index-NAVE].motion.x-data_packets[index].motion.x)*factor;
    data_packets[index].compass.y=data_packets[index-1].compass.y-
(data_packets[index-NAVE].motion.y-data_packets[index].motion.y)*factor;
    data_packets[index].compass.z=data_packets[index-1].compass.z-
(data_packets[index-NAVE].motion.z-data_packets[index].motion.z)*factor;
  } else if (index>=1) {
    data_packets[index].compass.x=data_packets[index-1].compass.x-
(data_packets[index-NAVE+NUM_DATA_PACKETS].motion.x-
data_packets[index].motion.x)*factor;
    data_packets[index].compass.y=data_packets[index-1].compass.y-
(data_packets[index-NAVE+NUM_DATA_PACKETS].motion.y-
data_packets[index].motion.y)*factor;
    data_packets[index].compass.z=data_packets[index-1].compass.z-
(data_packets[index-NAVE+NUM_DATA_PACKETS].motion.z-
data_packets[index].motion.z)*factor;
  } else {
    data_packets[index].compass.x=data_packets[NUM_DATA_PACKETS-1].compass.x-
(data_packets[index-NAVE+NUM_DATA_PACKETS].motion.x-
data_packets[index].motion.x)*factor;
    data_packets[index].compass.y=data_packets[NUM_DATA_PACKETS-1].compass.y-
(data_packets[index-NAVE+NUM_DATA_PACKETS].motion.y-
data_packets[index].motion.y)*factor;
    data_packets[index].compass.z=data_packets[NUM_DATA_PACKETS-1].compass.z-
(data_packets[index-NAVE+NUM_DATA_PACKETS].motion.z-
data_packets[index].motion.z)*factor;
  }
}
```

Then use this function after the sensor data retrieval in `Timer2Handler()`.

```
void __ISR(_TIMER_2_VECTOR, ipl3) Timer2Handler(void)
{
  …
  data_packets[create_index].humidity = get_humidity();
  movingAve(create_index);
  send_data = TRUE;
  sample_count++;
  mT2ClearIntFlag();
}
```

Extra work: allow users to input the number of points in calculating moving average.

3.6.8. Exercise 8: Power spectrum

The last exercise is more sophisticated. You will perform a Fourier transform on the light sensor data and output the real-time light intensity spectrum directly to the host. This is a simple form of a spectrum analyzer. You can find the FFT program `void four1(float data[], unsigned` `long nn, int isign)` from “Numerical recipes in C” [1]. To make things simple, we will only perform a FFT on light sensor 1 and keep the RATE fixed to be 500 samples per second. To get the spectrum, you will always need a window of continuous data points (preferably 2^n due to the efficiency of the FFT algorithm). One way to achieve this is to continuously move the window by one whenever a new light sensor sample is taken. However, this approach will be problematic. It means we will have to perform the FFT at the same rate as we are sampling the sensor. Since the FFT is time consuming, and we only have very small amount of time between sampling intervals, we can only perform FFT on small contiguous portions (or windows) of data. Our strategy is performing the FFT not at every sample, but at a fixed amount of relatively large time intervals T_{fft} (say 0.5s). It means we are updating the spectrum every T_{fft}. To keep programs simple, we will perform FFT every time the packets array is full. Since NUM_DATA_PACKETS is set to be 200 in the program, it means T_{fft}=1/500*200=0.4s. As with the previous exercise, we are going to use the x axis data field for compass to save the intensity spectrum of light sensor output. Notice that there is only 65 data points in intensity spectrum and they are put in the last 65 positions in compass x field of the data packets array.

Add the following two lines in “common.h”

```
#define RATE       500
#define FFTWINDOW  128
```

Add the following two functions in the file main.c:

```
//The following function is from “Numerical recipes in C”.
void four1(float data[], unsigned long nn, int isign)
/******************************************************************************
Replaces data[1..2*nn] by its discrete Fourier transform, if isign is input
as 1; or replaces data[1..2*nn] by nn times its inverse discrete Fourier,
transform if isign is input as -1.  data is a complex array of length nn or,
equivalently, a real array of length 2*nn.  nn MUST be an integer power of 2
(this is not checked for!).
******************************************************************************/
{
  unsigned         long n,mmax,m,j,istep,i;
  float            wtemp,wr,wpr,wpi,wi,theta;
  float            tempr,tempi;

  n=nn << 1;
  j=1;
  for (i=1;i<n;i+=2) { /* This is the bit-reversal section of the routine. */
    if (j > i) {
      SWAP(data[j],data[i]); /* Exchange the two complex numbers. */
      SWAP(data[j+1],data[i+1]);
    }
    m=nn;
    while (m >= 2 && j > m) {
      j -= m;
      m >>= 1;
    }
    j += m;
  }
  mmax=2;
  while (n > mmax) { /* Outer loop executed log2 nn times. */
    istep=mmax << 1;
    theta=isign*(6.28318530717959/mmax); /* Initialize the trigonometric
recurrence. */
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    wr=1.0;
    wi=0.0;
    for (m=1;m<mmax;m+=2) { /* Here are the two nested inner loops. */
      for (i=m;i<=n;i+=istep) {
        j=i+mmax; /* This is the Danielson-Lanczos formula. */
        tempr=wr*data[j]-wi*data[j+1];
        tempi=wr*data[j+1]+wi*data[j];
        data[j]=data[i]-tempr;
        data[j+1]=data[i+1]-tempi;
        data[i] += tempr;
        data[i+1] += tempi;
      }
      wr=(wtemp=wr)*wpr-wi*wpi+wr; /* Trigonometric recurrence. */
      wi=wi*wpr+wtemp*wpi+wi;
    }
    mmax=istep;
  }
}

void spectrum(){
  int i;
  float lightsensor[FFTWINDOW*2+1]={0};
  for (i=0;i<FFTWINDOW;i++){
    lightsensor[i*2+1]=data_packets[i+NUM_DATA_PACKETS-FFTWINDOW].photo.one;
    lightsensor[i*2+2]=0;
  }
  four1(lightsensor,FFTWINDOW,1);
  data_packets[NUM_DATA_PACKETS-FFTWINDOW/2-1].compass.x=lightsensor[1];
  data_packets[NUM_DATA_PACKETS-1].compass.x=lightsensor[FFTWINDOW+1];
  for (i=1;i<FFTWINDOW/2;i++)       {
    data_packets[i+NUM_DATA_PACKETS-FFTWINDOW/2-
1].compass.x=sqrt(lightsensor[i*2+1]*lightsensor[i*2+1]+lightsensor[i*2+2]*li
ghtsensor[i*2+2])*2.0;
  }
}
```

Then in function `ProcessIO(void)`, call the function spectrum() as shown in the code snippet below:

```
if (send_data) {
  …
  if (create_index >= NUM_DATA_PACKETS) {
    create_index = 0;
    spectrum();
  }
  …
  }
```

On the host side, the “lightsensor-firmware” folder contains the customized MATLAB program. It shows the most recent received spectrum and real time light intensity data. You will find that the florescent light spectrum peak is not exactly at 120 Hz. This is because our FFTWINDOW is too small and we haven’t applied any window functions. You can increase the value of NUM_DATA_PACKETS and FFTWINDOW to determine the maximum FFTWINDOW window that can be used. You may now try different window functions to understand their effects. Armed with the FFT technique, you can write time-domain digital filters such as simple low/high/band pass filters and also much more sophisticated filters.

Reference:
[1] http://www.fizyka.umk.pl/nrbook/bookcpdf.html