# AUDIOMETER

## Design Manual

Jonathan Peer
University of Maryland
ENEE440 "Microprocessors"
Fall 2014

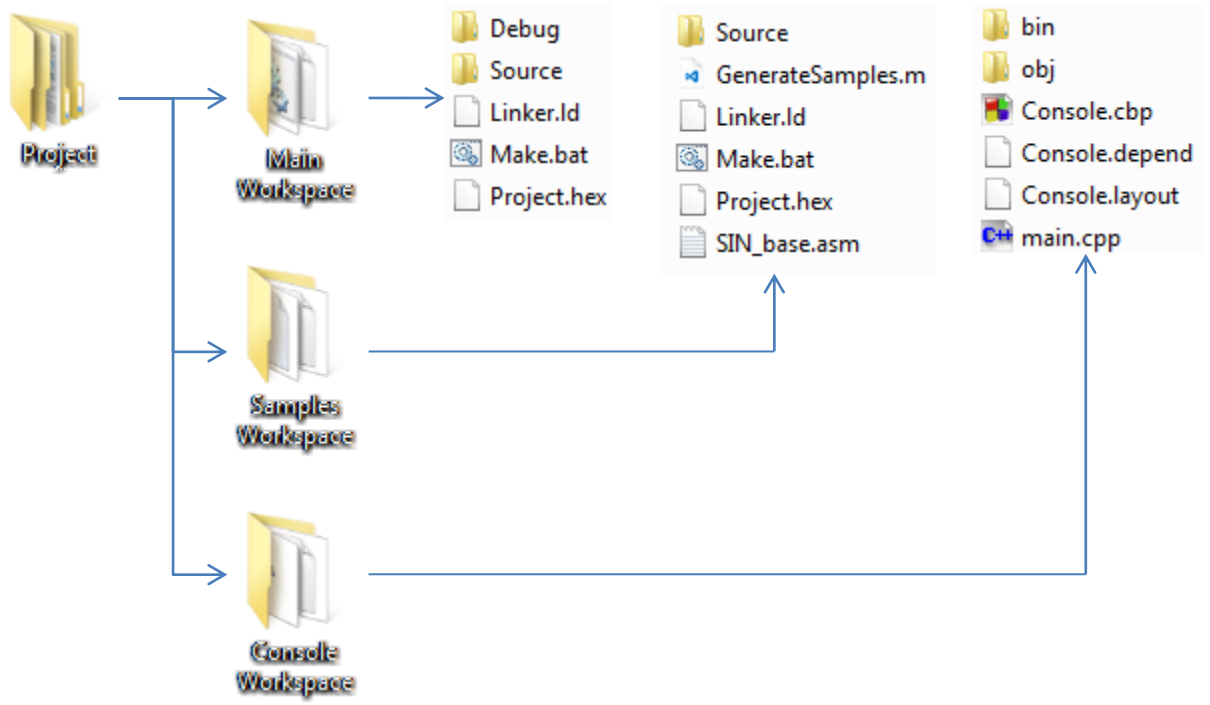
# 1.0 Project Structure

When you open the project, it'll be divided into three workspaces: main, samples, and console.

The main workspace contains the bulk of the project and produces the hex file used to program the board.

The samples workspace produces 40,000 samples of a sine wave, and a hex file used to transfer them to the board (it won't overwrite the program from Main).

The console workspace has a Code::Blocks project for a console program. It uses the Windows API and was written for the MinGW compiler (MinGW is a Windows port of GCC). Instructions on how to use the console can be found in the User Manual accompanying this.



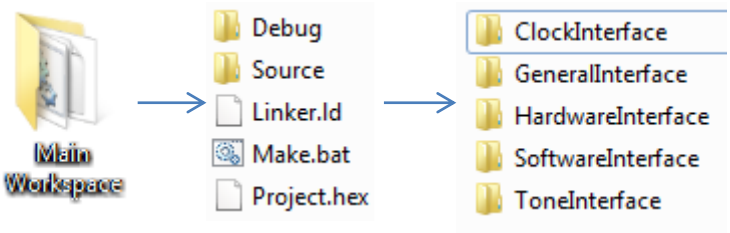

The contents of the workspace are fairly self explanatory.

"Debug" contains a Keil project, which I used for Keil's hardware debugger. The project files are updated when you run "Make." You don't need to do anything when you update the source code and recompile.

The "Make" files are batch scripts that compile the project and produce the hex files. The source files used in the compilation are located in the "Source" folder to keep them separate from the project files. The "Linker" files are linker scripts also used in the compilation. The "Project" files are the hex files you need to program the board. And "SIN_base" is an assembly file used by the Matlab script "GenreateSamples" to produce "SIN.asm" in the "Source" folder. It populates "SIN_base" with 40,000 samples of a sine wave.

# 2.0 Main Workspace

When you go into the Source folder of the Main Workspace, it'll be divided into five more folders. They represent subsystems (and General Interface, which is a catch-all).




## 2.1 The General Interface

General contains the startup file, the main function, two header files (one for Assembly, one for C), and two additional files which implement bit and pin operations because they're used by multiple subsystems.

The header files contain the base addresses of peripherals, the register offsets from the base addresses, data types used for sampling button input, and an Assembly macro, "CALL," which implements the C calling convention, described below.

```
Syntax:
CALL functionName, p0, p1, p2, p3, p4, p5, p6, p7

Description:
The parameters are optional. You don't need to specify more than are required by the
function. The parameters can be register names, symbol names (like labels), and literals
(without the preceding '#' or '=' sign).

Example:
CALL PIN_setValue, B, 6, 1 // set output pin B6 to 1
```

In Main, you see two things. One, the functions are prepended by CINT, TINT, HINT, and SINT. And two, I use functions like SINT_update and HINT_update. I used a few programming conventions to help make the code clear.

```
#include "Header.h"

void main () {

	// The order matters here.
	CINT_init(); // initialize clock interface
	TINT_init(); // initialize tone interface
	HINT_init(); // initialize hardware interface
	SINT_init(); // initialize software interface

	while (1) {
		SINT_update();
		HINT_update();
	}
```

```
}
```

I prepend function and variable names with the name of the file in which they're located ("CINT_init" can be found in "CINT.asm"). Also, the subsystems all have files that abbreviate their names ("Clock Interface" has "CINT.asm"). These files act as interfaces through which the subsystems interact with one another. It's not practical. But it helps illustrate how the peripherals / subsystems interact with one another, and helps you find the definition of function calls.

```
.global TIM7_Handler
.thumb_func
TIM7_Handler:

        @; set up stack frame
        PUSH {R7,LR}

        @; Drive P24 IO operations with TIM7
        CALL P24_update

        @; clear pending bit
        CALL BIT_setBitfield, (TIM7 + TIM_SR), 0, 0, 1

        @; restore stack frame
        POP {R7,LR}

BX LR
```

To keep peripherals as self-contained as possible and support that earlier idea of better seeing how they interact with one another, I use functions like "P24_update." If you look at the TIM7 interrupt handler, even though "TIM7.asm" is in the same folder as "P24.asm," it doesn't modify P24 directly. It calls P24_update, and the P24 peripheral then modifies itself.

That's about it as far as programming conventions. I make liberal use of comments, formatting, and other visual queues to help you digest the code, especially the assembly. Even though I write in assembly, I often translate the larger functions into into C in the comments, and show where registers are being used or are vacant. Other than that, as you saw in Main, the General Interface doesn't really do anything. It just calls calls _init and _update functions for the different subsystems.

## 2.2 The Clock Interface

Clock contains system clock settings, or files generated by the STM32F4 Clock Config Wizard (the spreadsheet you showed us in class). It also contains "STK" which initializes the SysTick peripheral to provide fractional second keeping. The peripheral triggers an interrupt at a 1kHz interval, and the interrupt handler increments a global variable, CINT_time.

## 2.3 The Hardware Interface

Hardware is responsible for the P24, its inputs and outputs, and the state machine that makes them behave as they should in the audiometer.

"P24" manages the anode and cathode latch with an array of five anode and cathode patterns assigned digits 1, 2, 3, and 4 of the seven segment display, and then all 6 LEDs together in the fifth pattern. The anode patterns are fixed. When written to the latch, they enable their assigned output, and disable the rest. The cathode patterns are changed by functions like LED_setOn and DISP_setDigit and in turn change which LEDs and LED segments are lit up or not. And because there are not enough pins to keep them all on at the same time, the 5 outputs are turned on and off in a round robin scheme.

| **ANODE_LATCH** | | | **CATHODE_LATCH** | | | |
|---|---|---|---|---|---|---|
| B4 | AN_EN | | C1 | CA_EN | | |
| C11 | AN_CLK | | D2 | CA_CLK | | |
| C2 | DIGIT_1 | | C5 | CA_A | LED_5 | SW_11-12 |
| A1 | DIGIT_2 | | B1 | CA_B | LED_6 | SW_7-8 |
| C4 | DIGIT_3 | | A1 | CA_C | LED_1 | SW_13 |
| B1 | DIGIT_4 | | B5 | CA_D | | SW_1-2 |
| B11 | AN_R | | B11 | CA_E | | SW_3-4 |
| B0 | AN_G | | C2 | CA_F | LED_4 | |
| B5 | ROT_ENC | | C4 | CA_G | LED_2 | SW_9-10 |
| C5 | COLON | | B0 | CA_DP | LED_3 | SW_5-6 |

A table showing the overlapping use of cathode pins.

The TIM7 interrupt handler is triggered at a 5kHz interval, calling P24_update which, while written in Assembly, would do the following in C. P24_index is incremented, reset to 0 when it reaches 5, and used as an index to write an anode and cathode pattern to the latches.

The fact that the 5 outputs are only on for $1/5^{th}$ of the time means that they have an effective refresh rate of 1kHz. And the fact that the rotary encoder and buttons are polled only when P24_index equals 0 means that they too have an effective refresh rate of 1kHz.

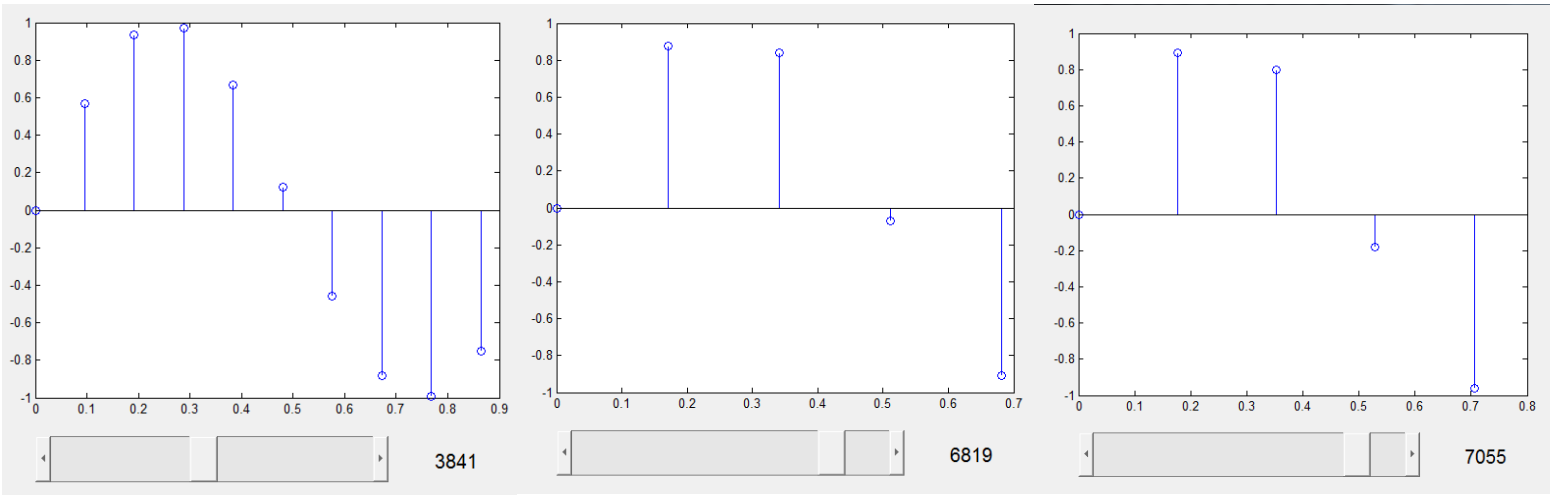
```
if (P24_index == 0) {
        ROT_poll();
        BTN_poll();
}
PIN_setValue(B, 4, ~(0)); // disable anode latch output
P24_setAnodePattern(P24_anodePattern[P24_index]);
P24_setCathodePattern(P24_cathodePattern[P24_index]);
PIN_setValue(B, 4, ~(1)); // enable anode latch output
P24_index = (P24_index + 1) % 5;
```

# 2.4 The Tone Interface

I decided that the tone would have a playback frequency of 40kHz. And instead of implementing the sine function, I would produce my samples by periodically selecting samples from a larger sample set. In order to make the selection period an integer (always picking something like every $10^{th}$ sample instead of something like every $10.75^{th}$ sample), I would need 40,000 samples to start. This would also allow for each frequency to be its own selection period (125 Hz means pick every $125^{th}$ sample, 8000 Hz means pick every $8000^{th}$ sample, and so on) which is convenient.

From the 40,000 samples, I select a total of 12,800. This means that, instead of selecting a subset that comprise just one period, as in the illustrations below, I produce a sample set with 40 to 2,560 periods,

(depending on the frequency of selection). I did this for two reasons. One, with just on period, the difference between the first and last samples results in a popping sound when played back. And two, with just one period, entire ranges of frequencies result in having the same number of samples being played back, (6819 and 7055 Hz below both only have 5 samples, and this is true up until 8000 Hz). So in addition to the popping sound, they produce the same tone.




"SIN" sets the frequency by putting a subset of the original 40,000 samples in a 12,800 sample buffer. The larger sample set is stored in ROM at 0x08010000 by the hex file in the samples workspace. The main program starts in RAM at 0x08000000. It's not large enough to overlap. It shouldn't anyway since it's RAM and ROM.

To produce the sound, the TIM6 interrupt is triggered at a 40kHz interval. It signals the DAC to initiate a DAC-DMA request. The DMA transfers a sample from the sample set to the DAC's input registers. Then, the DAC converts the sample to an analog value and it comes out on pin PA4. At this point, the tone is being generated, but it needs to be forwarded to the CS43L22 audio buffer so it can be heard by headphones.

To do this, I initialized the I2C peripheral to communicate with the CS43. The CS43 has its own DAC, and would normally accept samples via I2S. But I configured it in analog passthrough mode in which it buffers the analog signal on pin PA4 instead.

Despite having written the wrapper "I2C," preparing all the initialization settings and implementing mute and volume controls in "CS43," and integrating it with the overall design of the audiometer, the CS43 never acknowledges its address when I communicate with it via I2C. So the tone never reaches the audio port.

## 2.5 The Software Interface

Software contains a USB-VCP driver. I added some functions to read and write arrays of bytes, instead of just bytes, in "USB." And I implement a control protocol in "SINT."

The console program translates script commands from text to numbers. For each command, it generates a buffer. The first byte tells the audiometer how many more bytes will follow. The second byte tells it what command to perform. If there are more bytes, they act as parameters to the command.

| Command | Command Number |
|---|---|
| Press Button | 1 |
| Hold Button | 2 |
| Release Button | 3 |
| Turn Rotary | 4 |
| Wait | 5 |
| Get Records | 6 |

"Turn rotary" for example has two parameter, a byte for direction (0 = CCW, 1 = CW) and a byte for the number of detent stops (0 – 255 stops). You have one byte for the command, and two for the parameters. If the script command was "turn rotary 1 15," the buffer would look like this:

```
Buffer
byte[0] = 4
byte[1] = 4
byte[2] = 1
byte[3] = 15
```

Sending the user records to the computer is similar. A byte tells the computer how many records will be sent. Then the user records are sent in 3 byte buffers. The first byte is signed and represents the intensity, -10 to 110. The second two bytes are incidentally signed and represent the frequency, 125 to 8000. The first byte contains the most significant bits. The second contains the least.

# 3.0 Conclusion

This concludes my design. It's fairly straightforward because there's a near one to one correspondence between the theoretical solution and the code. As I've mentioned, I tried to make sure that the code is readable. I made the project files (Make, Linker, Startup) from scratch. And I picked up some interesting programming tips like defining the interrupt handlers with weak aliases. So, in addition to being able to add and remove source files without having to edit the Make file, I can add and remove interrupt handlers without having to edit the Startup file.