December 2009

Educational Signal Processing Platform

Daniel James Cullen
Worcester Polytechnic Institute

Jonathan Michael DeFeo
Worcester Polytechnic Institute

Educational Signal Processing Platform

A Major Qualifying Project Report

submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

By:

Daniel J. Cullen

Jonathan M. DeFeo

Project Advisor:

Professor D. Richard Brown III

22 December 2009

Abstract

Digital Signal Processing (DSP) education is often limited by the high cost to entry for the platforms commonly used in college laboratories. Since most students cannot reasonably afford a $400 board (in addition to a textbook, tuition, etc.), this project created a new DSP platform at a much more reasonable price ($50), while maintaining the same level of educational content and value available through more expensive hardware. The new platform consists of hardware designed for audio processing using Microchip's dsPIC33F microcontroller, as well as a set of software libraries to facilitate its use. New laboratory assignments were also formulated to achieve the same learning outcomes as the assignments that use the more expensive hardware, but with additional value due to the resource-constrained nature of the dsPIC, which is more representative of the non-ideal scenarios that students will face in their careers. The DSP platform developed in this project is more educational and cost-effective than other existing platforms, and therefore makes DSP instruction more easily accessible to students.

Contents

1 Introduction

In order to effectively teach any subject in engineering, a practical component where the students implement what they have learned in the classroom in a laboratory setting is critical. Digital signal processing is no different and this necessity has lead to the major DSP manufacturers creating educational kits based on their own processors. WPI's own DSP course, ECE 4703, uses Texas Instrument's kit based on the C6000 series DSP. This board provides a huge number of features, both in terms of input and output (such as switches, audio codec, and a microphone connection) as well as a very powerful, super-scalar, floating-point enabled DSP. This would seem to be an excellent solution, since it provides everything a student would need in a single package.

There is, however, a major problem with this solution: the cost. At $400, it is not practical to own a large number of these boards. The WPI lab currently has 12, which means that enrollment in the course is limited to 24 students, thus ensuring the waitlist to gain entry to the course is always full. The goal of this project was to create a board that only cost $50, thus eliminating this issue. This would allow students to purchase their own boards so that they could take them home and use them in any setting. This major upside of this is that course enrollment is now no longer artificially limited by number of available DSP boards - now, as many students that are interested in taking it can because the main barrier to entry (cost) has been removed. Furthermore, students could take their boards home with them which allows for more open-ended experimentation.

This issue has not been addressed by any of the major DSP manufacturers. The DSP chips alone cost nearly $50, without any of the supporting circuitry, PCB, connections, or audio interface. In addition, their chips are not viable for a hobbyist solution due to their package (ball grid array) and massive complexity (over 100 pins). On the lower end, there are many hobbyist platforms for doing control-type work such as in a robotics application. However, these systems are not nearly powerful enough for real-time, audio DSP. This leaves a major gap in the market for an affordable, complete package that is capable of real time

DSP on audio frequency signals.

This project's approach is to keep the entire cost of the board, including the processor, audio interface, and all supporting circuitry under $50. This is a cost low enough that would not be a burden to students while at the same time giving enough flexibility and power to teach the same concepts as a much more expensive board. It is also roughly the cost of the textbook currently used in ECE4703; the board could be purchased instead of the text to keep the financial requirements as low as possible. This is accomplished using Microchip's dsPIC processor; a 16-bit, DIP-package, single-issue, fixed-point processor which is capable of executing 40 MIPS. The price for the processor is under $4. It has DSP-specific instructions which enable it to execute commonly used functionality such as a multiply-accumulate in a single clock cycle. This affords huge flexibility and makes it a very powerful platform for audio DSP. Since it is in a DIP package, students could assemble the boards themselves as well as utilize it for other projects, adding further value to the platform.

The dsPIC platform is capable of teaching students the same concepts as the $400 TI DSK. Real time filters, microprocessor architecture, audio processing, and algorithmic optimization all can be taught just as effectively. A further advantage is that this system is tailed specifically for the ECE 4703 course with its educational needs in mind. An issue with the TI board is that it provides a sense of limitless power to the student, in that nothing in the ECE 4703 course comes close to stressing it. The dsPIC platform's capabilities are much closer to the requirements for the course's assignments. This will require students to write better, more organized code since they won't have so much power to effectively "waste." Another issue is that the TI platform's fixed and floating point performance is nearly identical. This inevitably begs the question, "Why would anyone use fixed point?" The dsPIC platform answers this question through demonstration - floating point code is extremely slow due to the hardware limitations. This gives students a real-world example of why fixed-point is preferable and much more widely used. The dsPIC platform provides the same educational functionality as many of the boards released by other manufacturers at a

much lower price point.

2 Background

2.1 Survey of Microprocessors

Originally, the idea was to use an older CPU architecture which could be purchased extremely inexpensively. The Zilog Z80 and 68000 were briefly considered. However, it was determined that the 8-bit registers were too small to do accurate signal processing. The Z80 also did not have a RISC-like instruction set and used archaic x86 addressing modes which would be pretty non-valuable information for a signal processing course. The 32-bit 68000 is inexpensive and has a familiar ISA but seemed slightly impractical compared to the dsPIC, which was discovered later.

TI's and AD's line of DSP chips were investigated. However, it was determined that they would be far too expensive to use in this application. The chips themselves were not extremely expensive, but due to their 192-256 pin BGA configurations would require a very expensive PCB design. Since this project is unlikely to produce thousands of these boards, construction costs would be prohibitive.

The focus of the project instead shifted to Microchip's dsPIC. A 16-bit microcontroller, the dsPIC has a specialized instruction set which enables high-performance in signal-processing applications. The chips have a lot of hardware integrated, such as flash memory and DAC/ADC. They are also extremely inexpensive and come in a variety of packages ranging from 18-pin SOIC, 40-pin DIP, and 100 pin SOIC, depending on how many inputs/outputs are needed. This significantly reduces PCB complexity and cost since a single-layer PCB could be used. The PIC instruction set is RISC-like and dyadic, similar to most every modern DSP. PICs are also widely used in industry, providing students with experience and knowledge that they could potentially use or apply in the workplace (which is the goal of most engineering courses at WPI). Since the dsPIC is not nearly as powerful as the TI DSPs currently used in the lab (40 MIPS vs. 400+ MIPS), this would force students into a "real-world" mindset where they are faced with non-ideal hardware and restrictions. This will

help to emphasize the value of fixed point vs. floating point (and emphasize why fixed-point is so much more prevalent in real-world DSP operations), require more strenuous optimization techniques (circular buffering, assembly programming), and demonstrate the effects of having 12-bit analog/digital converters compared to doing everything in full 32 bit (and how to make the best of it). Furthermore, the actual hardware will be much more accessible so the actual operation of the components should be more transparent than the somewhat complex TI hardware.

An issue with the dsPIC is that is does not have an FPU of any kind. However, there are open-source floating point libraries that enable floating data types in the code. The effectiveness of these libraries is unknown and a major part of the project might be optimizing them in assembly so that their performance is adequate for signal-processing. An idea being considered (to reduce cost and make the class a little bit more interesting) would be to produce PCBs and have the students assemble the system as their first lab assignment. The downsides being that this would require open soldering stations and someone to maintain parts kits.

The other issue with students doing assignments at home would be that most students do not have oscilloscopes and function generators. There are open-source software oscilloscopes and function generators that use the input/output on the sound card which could be an adequate solution when students are not in a WPI lab. Switches and LEDs on-board are nearly mandatory in this case since they will allow students to do some basic debugging without any other external tools.

Market research was conducted to determine if such a low-cost, educational signal processing platform existed. The findings were that no such product currently existed and there was ample room in the market for this type of device.

The most inexpensive floating point equipped DSP chip TI currently manufactures is the C6720. Running at 200MHz, it features 64 general purpose registers (a huge number for our purposes), DMA and extensive memory access features, hardware PLL (for variable

clock rates), several serial interconnects, and probably most useful for our purpose, circular addressing which greatly speeds up memory access times when using circular buffers. The cost of the chip by itself is $10.83 per unit; $8.125 when ordering 500 or more.

Although powerful, its cost is prohibitive when considering that the entire DSK should have a final cost of less than $50. The cost of PCB manufacture and other supporting components is significant. Spectrum Digital sells their kit based on this chip for $395.01. The other downside is that it can only be programmed with Code Composer Studio. Also, since the chip only comes in a 256 contact PBGA package, a complex PCB design will be required, further adding to the cost.

TI also sells their line of OMAP processors, based on an ARM core. These are very powerful, general purpose processors and can only be purchased in quantities over 100, at $55.25 per unit. Obviously this chip is far outside of the scope of this project.

Analog Devices' floating point DSP, called SHARC, is closer to the type of processor needed for this project. At $6.64 per unit at 1000 ordered units ($7.97 per unit), the price is more reasonable and the feature set is not quite as extensive. It does still come with a lot of features that are completely unnecessary for this project, such as Dolby Digital decoding and 24-bit, 96kHz sampling rates. The performance is also much higher than necessary. It has the same unfortunate issue as the TI chip such that it only is available in a 136-pin BGA package, making breadboarding impossible and leading to complex, possibly multi-layered PCBs. AD sells a development kit based on this chip for $500.

Arduino is a very inexpensive, fully open-source development platform featuring an ATmega168V 8-bit microcontroller. Although very inexpensive and easy to build, it is hugely limited by the processor. It also uses a simplistic programming language which eliminates a lot of the optimization techniques that could be employed in a DSP system. Arduino seems designed more for simple electronics prototyping and projects, rather than a real-time computational platform.

The dsPIC family of DSPs are, more or less, higher performance microcontrollers. They

are all only 16-bit, and do not have hardware floating point support. However, there are freely available, open-source libraries that give the chips software support for 16-bit floating point. The dsPIC will push 40MIPS, which compared to the TI and AD chips (which do 600-900), seems meager. However, it gives around 900 instructions per sample at 44.1kHz to work with. Using efficient coding techniques, this should be doable, especially considering that the ISA has been tweaked to allow for single-cycle multiply accumulates and single-cycle division. A 10-bit DAC and ADC is included, but it also has an SPI bus so an external codec could be used easily. While the performance is somewhat limiting, the major advantage is that the unit cost is just $2.76. Also, the package is an 18-pin SOIC which is much easier to handle than hundreds of BGA pins. A development kit with an LCD screen, multiple serial and audio inputs, buttons, and switches is sold for $300.

Towards the end of the project, we discovered a competing product from TI [6]. It is a development board that includes a TMS320C5505 (a low power, 16-bit, fixed-point DSP) and the AIC 32 codec, similar to the AIC23 found on the currently used DSK board. It also includes a full copy of Code Composer Studio, TI's development environment. This entire package costs only $49, making it very competitive to our platform. We did not have time to fully investigate this product, although we believe it warrants further research.

2.2 FPGA Soft-Core Embedded Microprocessors

One option that the project team explored was using a soft-core microprocessor inside an FPGA. This approach has several advantages. The first advantage is that it most students already own a Xilinx Spartan 3 FPGA evaluation board from taking other courses offered in the department. An embedded microprocessor in the FPGA would allow them to save money by reusing hardware for both their digital logic and real-time DSP classes. Since the school already has licenses for the Xilinx development tools (used for other courses), there would be no additional cost to the department for selecting this embedded microprocessor approach. Another advantage of this FPGA approach is that it allows the possibility of labs

involving hardware filters (i.e., in VHDL code). Such labs would be beneficial to students because it would illustrate the tradeoffs between hardware and software implementations, as well as give the students valuable exposure to FPGA-based signal processing, which is prevalent in industry.

The project team performed a few initial tests of the feasibility of this approach. The team built a simple Xilinx Embedded Development Kit (EDK) project, consisting of a Xilinx MicroBlaze soft processor running on the Xilinx Spartan 3 FPGA evaluation board. In terms of size and speed, the Spartan 3 appeared to be sufficient. The MicroBlaze processor easily fit into the available logic resources of the FPGA chip. The Spartan 3 also runs on a 50MHz clock, which should be sufficiently fast for the audio signal processing projects in the ECE 4703 labs.

Even though these initial findings were promising, the project team soon determined that the FPGA embedded microprocessor approach was not feasible for several reasons. One problem was that the Spartan 3 board does not have audio jacks or even an ADC or a DAC. Adding external hardware to support DSP would seem like a hack. However, the major problem was that the overall system is impractical for the purposes of the ECE 4703 course. The process of re-building and re-downloading every time the software code is changed is time consuming and difficult to make streamlined. The student needs a good software development suite with a debugger, code profiling tools, optimizing compiler, etc., all of which are unavailable in the Xilinx EDK tools. Xilinx ChipScope is too low-level to be used as a substitute for a debugger. The MicroBlaze is also a general microprocessor, not a DSP, so it cannot be used to teach about DSP architectural concepts such as pipelining, parallelization, and DSP-optimized instructions (e.g., MAC). In all, the FPGA approach adds unnecessary complexity and confusion, which results in more potential for things to go wrong and increased difficulty of debugging. When learning how to implement digital filters and other DSP algorithms for the first time, it is much easier to implement everything purely in software than in this mixture of hardware and software in the FPGA. The bottom line is

that the FPGA approach is not practical for use in the ECE 4703 class.

2.3 Audio Codec Selection

Although the dsPIC has a 12-bit DAC on chip, it was discovered that it could only operate at 10 bits per channel when the channels were being simultaneously sampled. This is an issue if any kind of adaptive algorithm were implemented; these algorithms such as system identification rely on simultaneously sampled channels. It was determined in that case that it was advantageous to use an external codec. The AIC23 was decided on quickly due to the fact that it is the same chip used on the current DSK platform and will allow for an easier transition to the new hardware. It also integrates all of the analog circuitry necessary for the platform. It is reasonably inexpensive at less than $5 per chip when ordered in quantity.

2.4 Development Tools

The development tools are a critical component of this project. There are three parts to the set of development tools: the device programmer, the integrated development environment (IDE), and the compiler suite. Before we can begin designing any hardware or writing any software, we must have these tools; programs cannot be compiled without a compiler and they cannot be downloaded to the dsPIC hardware without a device programmer!

More importantly, a good set of development tools is critical for the overall feasibility of restructuring the ECE 4703 course. If the students do not have a set of tools that are powerful yet easy to use, then this entire dsPIC project would be impractical to use in the ECE 4703 curriculum, even if it were the best microcontroller in the world. It is also important to consider the cost of the development tools. One of the goals of this project is to minimize cost, and the development tools add an additional cost to the overall system. Fortunately, Microchip offers development tools that are powerful, easy to use, and relatively inexpensive. The availability of inexpensive, high-quality development tools is key.

2.4.1 Device Programmers

The programmer is used to get the code from MPLAB onto the dsPIC. It contains a USB connection to the programming interface, which connects to the dsPIC board. It also functions are a real-time debugger. Debugging is an important feature since it gives much needed feedback about variable contents during execution. The PICKit2 has limited support for debugging, but for the purposes of ECE 4703 should be sufficient. There are more powerful debuggers available, but are cost-prohibitive. The ICD2 has similar functionality but costs significantly more than the PICKit2; all it adds is support for RS 232, which many students will not have available.

The PICKit2 also supports UART, which has been made an important part of the platform and course. The PICKit2 is by far the cheapest programmer, which is a major advantage since cost saving is the primary goal of this project.

Most interestingly, though, is that it is possible to build the PICKit2 for much less than it would cost to buy it, since Microchip supplies the firmware necessary to create such a device. Using an older PIC and a USB interface chip, the programmer could be integrated onto the PCB which makes the platform even more portable. It seems then, that the PICKit2 offers so many advantages over other programmers that there is no reason to pursue them.

2.4.2 Integrated Development Environment

Microchip provides a free integrated development environment (IDE) called MPLAB for developing software for Microchip products. MPLAB is the de facto IDE for PIC programming. It has all of the major features common to all integrated development environments, such as a text editor, a notion of projects and workspaces, compiler and linker integration, build scripts, and integrated hardware debugger support. The debugging capabilities include the standard mechanisms for setting breakpoints and stepping through code, as well as watch windows for monitoring variables and registers. The disassembly listing window is also quite useful because it lists the lines of C code and the associated lines of assembly code, allow-

ing the user to see exactly how the C code is implemented. Unlike the Texas Instruments Code Composer Studio IDE, MPLAB does not have a built-in grapher tool for plotting data from microprocessor memory, but this is a non-essential feature. Overall, we have found the MPLAB IDE to be powerful and simple to use, and it is therefore suitable for use by students in ECE 4703.

2.4.3 Compilers and Assemblers

A reasonably powerful compiler is also essential when developing code for any microprocessor. For the ECE 4703 course, it is not enough simply to have an assembler; a high-level programming language such as C is required because it is significantly easier for students to learn the DSP algorithms in C before attempting assembly language implementations, hence the need for a suitable C compiler. Microchip offers several different C compilers that integrate seamlessly with the MPLAB IDE. For high optimization and performance, Microchip[1] offers the HI-TECH line of compilers [23], but unfortunately, the commercial versions are too expensive (for ECE 4703 students), and the capabilities of the free ("Lite") version are extremely limited. The MPLAB C30 Compiler is another option, offering high levels of optimization that take advantage of many of the dsPIC's DSP-specific features, but the full commercial version costs $8,950 per site license [18]. Fortunately, there is also a student version of MPLAB C30 Compiler that is free for academic use [17]. This academic version includes many of the features of the commercial version, including full ANSI-C compliance, standard C libraries, optimized DSP libraries, and modest optimizing compilation support. Moreover, the academic version of MPLAB C30 does not impose any code size restrictions, unlike the free versions of some other compilers. After using the MPLAB C30 student version extensively over the course of this project, we have determined that easily meets all of the needs of students in the ECE 4703 course. We have therefore selected the academic version of the Microchip MPLAB C30 C Compiler for use in the ECE 4703 course.

[1] Microchip owns the HI-TECH Software company.

There also exist a few open-source C compiler alternatives. However, the MPLAB C30 student version met all of our needs, and we had neither time nor need to research the open-source compilers further. The MPLAB C30 compiler also makes more sense because it is much more widely used and is supported by Microchip. An open-source compiler may still warrant future research, especially for those who wish to develop dsPIC code under Linux.

3 Reference Design and Implementation

Our background research led us to select the dsPIC33FJ128GP802 for our low-cost digital signal processor. Our next task was to design and implement a reference platform for signal processing using this dsPIC33F chip. There are two main parts of our reference design: the hardware, which consists of a circuit board containing the dsPIC33F and all supporting electronics; and the software libraries, which contains functions and drivers used to configure the dsPIC33F's registers. Since the ultimate goal is to have students in ECE 4703 use this board for their laboratory assignments, two of our major design considerations were simplicity and cost.

This chapter of the report explains the key design aspects of both the hardware and the software, as well as the process we used to test and debug the complete system. For even further details about any particular aspect of the system, feel free to refer to the *Detailed Guide* document in Appendix D.

3.1 Hardware Design

This section explains how the project team designed the hardware platform.

3.1.1 Overview

The hardware platform is intended to be a learning tool for students and a building block for hobbyists to use in signal processing and control systems projects. We therefore designed the hardware with simplicity in mind: an efficient, straightforward design makes a great learning tool because it allows the users to quickly grasp how all the pieces work. Moreover, the open-source nature of the hardware and software allows gives advanced users the ability to modify the hardware for use in a variety of projects ranging from audio processing to robotics. A simple, efficient design using minimal number of components also allows us to minimize the cost of the board (e.g., fewer parts, less PCB complexity, etc.), which is another

reason why our board would be appealing to students and hobbyists.

The hardware consists of the dsPIC33FJ128GP802 chip, a header for connecting the PICkit2 programmer, status LEDs, an AIC23 stereo audio codec chip, 1/8-inch stereo audio connectors, oscillator crystals, user-configurable LEDs and DIPs, and DC-to-DC converter for battery power. This section discusses each of these pieces, one at a time.

The final versions of the hardware schematics can be found in Appendix A. The schematics were drawn with gschem, which is part of the GNU Engineering Design Automation (gEDA) toolsuite. These gEDA tools were selected because they are free and easy to use, making them an excellent choice for hobbyists and students. Although the gEDA tools are only available under Linux, the tools are well supported; have been under active development for over ten years. The toolsuite also includes utilities for creating PCB layouts from the schematics, schematics, which is important for the eventual production of this hardware platform.

3.1.2 dsPIC33FJ128GP802 Connections

The supporting electronics for the dsPIC33FJ128GP802 are designed directly from the recommendations in the datasheet [13] and the dsPIC33F family reference manual [12]. The minimum recommended connection setup consists of the following:

- Decoupling capacitors across the power supply rails to reduce switching noise.
- The dsPIC33F requires an additional capacitor connected to its VCAP/VDDCORE pin. The datasheet recommends a 16V capacitor with value from 4.7μF to 10μF.
- The MCLR pin needs a 10kΩ pull-up resistor.
- The PIC programmer VPP line should be connected to the MCLR pin of the dsPIC33F. The PIC programmer's ICSP DATA and CLK lines can be connected to any of the following pairs of programming pins on the dsPIC33F: PGEC1/PGED1, PGEC2/PGED2, or PGEC3/PGED3.

The dsPIC33FJ128GP802 contains many more peripherals than it has available pins. In order to give the designer flexibility, it therefore supports a system of remappable peripherals, in which the user can configure the software registers to connect certain peripherals to certain pins.

In choosing which pins to use for which peripherals, we considered several factors. First, certain peripherals are only compatible with certain pins. For example, we do no have a choice about which pins we use for the oscillator crystal or the power supply rails. The dsPIC33F's built-in audio DAC also is compatible with only certain pins. After assigning all of these pins which we had little-to-no choice about, we started to plan out which pins to use for each of the remappable peripherals. Considerations for the physical design and board layout drove most of these decisions. For example, we tried to assign the DCI/I2S bus signals to adjacent pins, which helps to keep the trace lengths the same (for better clock/data signal synchronization). We also tried to consider layouts that would result in minimal amounts of crossing wires, which simplifies PCB design (fewer layers) and gives better signal integrity. Our choice for which set of programming pins to use (PGEC1/PGED1) was driven by all of these considerations; PGEC1/PGED1 are closest to the MCLR pin, which makes these lines easy to route over to the ICSP header connector; moreover, the pins for PGEC2/PGED2 and PGEC3/PGED3 could instead be remapped and reused for space-efficient SPI and DCI/I2S bus layouts. The last pins we assigned were the DIP and LED pins because we had the most flexibility in selecting them; almost all of the dsPIC33F pins support general purpose I/O; moreover, the DIP and LED signals are relatively low-frequency and therefore do not have the associated switching noise or trace length design constraints. We went through several iterations of pin-reassignment over the course of the project as we finalized the board's set of features before we finally arrived at an optimal configuration. The final pin assignments are given in Table 1.

Table 1: Pin Assignments for the dsPIC33FJ128GP802 (organized according to location on DIP package)

dsPIC33FJ128GP802			
1	MCLR	28	AVDD
2	AN0 = ADC_L	27	AVSS
3	AN1 = ADC_R	26	DAC1LN
4	PGED1; RP0 = UART TX	25	DAC1LP
5	PGEC1; RP1 = UART RX	24	DAC1RN
6	RB2 = DIP_0	23	DAC1RP
7	RB3 = DIP_1	22	RB11 = LED_1
8	VSS	21	RB10 = LED_0
9	OSC1	20	VCAP/VDDCORE
10	OSC2	19	VSS
11	RP4 = SPI_CLK	18	RP9 = DCI_CSCK
12	RA4 = SPI_CS	17	RP8 = DCI_CSDO
13	VDD	16	RP7 = DCI_CSDI
14	RP5 = SPI_SDO	15	RP6 = DCI_COFS

3.1.3 Interface for built-in ADC and DAC

The dsPIC33F contains internal analog-to-digital converter (ADC) and digital-to-analog converter (DAC) peripherals. For various reasons explained below, we decided to use an external audio codec chip (the AIC23), rather than the internal ADC And DAC, for audio input and output. However, we have still allocated the pins for the ADC and DAC (rather than use them for other remappable peripherals or more LEDs/DIPs) so that these peripherals could be used in future projects. Our plan was to create large vias on the PCB design so that students and hobbyists could use the ADC and the DAC in their own projects. Our design does not place AC coupling capacitors in front of the ADC and DAC connections to allow for use in low-frequency control systems applications. Another potential future use of the ADC and DAC is demonstrations of audio aliasing in the ECE 4703, something that is not possible with the AIC23 chip due to the AIC23's built-in anti-aliasing filter. The dsPIC33F's many peripherals within a single package are one of its major features.

3.1.4 ICSP Header

The dsPIC33F uses Microchip's In-Circuit Serial Programming (ICSP) interface for device programming. A summary of the ICSP pins and their descriptions is given in Table 2. For flexibility, the dsPIC33FJ128GP802 supports three pairs of ICSP clock and data pins, any pair of which can be used for device programming. As mentioned previously, we selected the PGEC1/PGED1 pins after considering physical layout issues, accommodating other peripherals, flexibility, etc. Thus, we have connected the MCLR, PGEC1, and PGED1 pins of the dsPIC33F to the ICSP header.

After the device is programmed, the ICSP programming pins can be mapped to peripherals and used for other things. In our design, the ICSP pins are used for a UART and for the debugger. Our software libraries configure the remappable UART peripheral, which interfaces to the PICkit2 programmer's "UART Tool" on the PC. The PICkit2 debugger also uses the ICSP pins for communication. Since the UART and the debugger use this same set

Table 2: ICSP Pin Descriptions

ICSP Interface			
Pin	**Name**	**Primary Function**	**Secondary Function**
1	VPP	Programming Power; Provides Target MCLR	(None)
2	VDD	Provides Target VDD	(None)
3	GND	Provides Target GND	(None)
4	DAT	Programming Data	PICkit2 UART RX (dsPIC TX)
5	CLK	Programming Clock	PICkit2 UART TX (dsPIC RX)
6	AUX	(Not used)	(None)

of pins (PGEC1/PGED1), the UART and debugger cannot be used simultaneously. However, this limitation is minor because the user generally would never have any need to use both of these simultaneously for debugging purposes. Since the dsPIC33FJ128GP802 only has a limited number of pins, carefully-planned pin reuse such as this is essential and leads to an efficient design.

While still on the topic of the UART, there are a few more points worth noting. The UART interface is an invaluable tool for debugging, data capture, and control of the dsPIC while the system is running, and we therefore decided that some sort of UART would be an indispensable part of our design. Typically, UART communication between an embedded system and a PC is handled using an RS-232 interface. However, such an interface would require additional hardware (e.g., a MAX3232 chip to convert to the correct +/-15V RS-232 voltage levels, a DB-9 connector, etc.), which would add additional cost and complexity to our design. Moreover, most students probably do not have DB-9 connectors on their

laptops, so another interface (such as USB) would be preferred, but that would add even more complexity to the design. We then discovered that the PICkit2's PC software contains a feature called the "UART Tool" for communicating with the target PIC's UART peripheral via the ICSP pins (using the 0V and +3.3V signaling levels). Thus, we were able to attain all of the functionality of the UART without requiring any additional hardware complexity.

Unlike other PIC families, the dsPIC33F uses low-voltage programming and a dedicated MCLR' pin, so no circuit protection diodes are necessary. More information about the dsPIC33F's ICSP programming interface can be found in the dsPIC33F datasheet [13] and the "dsPIC33F/PIC24H Flash Programming Specification" [14].

3.1.5 AIC23 Stereo Audio Codec

Although the dsPIC33FJ128GP802 has a built-in analog-to-digital converter (ADC) and a built-in audio digital-to-analog converter (DAC), we decided to use an external audio codec chip. Although the external audio codec adds additional cost and complexity to the board, we felt that it would give much greater flexibility. For certain laboratory projects, such as the adaptive filter, simultaneous sampling is a critical feature. Although the dsPIC33F's ADC supports simultaneous sampling, the resolution is only 10 bits, which is somewhat limited (i.e., the signal-to-noise ratio (SNR) increases with the number of bits). Another reason why it makes sense to use an external audio codec chip is because it contains all of the amplifiers and drivers circuitry needed for line-level and headphone-level inputs and outputs, all within one convenient package. If we were to use the dsPIC33F's built-in ADC and DAC, we would still need additional op-amp chips to buffer the inputs and to drive the outputs. Therefore, if we need additional circuits anyway, it makes more sense to get a full audio codec chip with more features in a single package.

We also considered practical issues concerning the adoption of our board for use in the ECE 4703 laboratories. The process of migrating the course from the current hardware (the TI TMS320C6713 DSK boards) to our new dsPIC33F board would probably take some time,

due to the fact that the new laboratories need testing and the entire curriculum must be modified to focus on the dsPIC33F hardware. During this migration period, it is likely that both the DSK and the dsPIC platforms would be used simultaneously. It would be convenient for the user if our dsPIC board behaved identically to the DSK board from an analog signaling standpoint. For example, if the DSK board were disconnected from the function generator and the dsPIC board were connected instead, the input and output voltage amplitudes should be identical. This would facilitate switching between the two boards when testing each system, since the user would not have to worry about changing function generator levels. Ideally, the dsPIC board would eventually replace the DSK completely, but since it will take time and effort to achieve this, it is important to consider the practical interoperability issues for the transition period.

We conducted some background research into available audio codec chips from the major integrated circuit manufacturers (including Texas Instruments, National Semiconductor, Analog Devices, and Microchip), and we found that the Texas Instruments AIC23 featured the necessary compromise between functionality and complexity. The competing options either had many more unneeded features (at the expense of many more pins and increased PCB design difficulty), or were too limited (for example, either contained only an ADC or only a DAC, or did not support simultaneous sampling). We wanted to choose a device which had both an ADC and a DAC in the same chip, since reducing the number of chips generally simplifies the complexity of the hardware (the software may need to be more elaborate, but software is easier to deal with than hardware). Another major feature of the AIC23 was that we already knew that it works all of the current ECE 4703 labs. A known working reference design is invaluable when designing and debugging something, especially when time is limited. One of the main drawbacks to the AIC23 is that it it is a surface mount component, which makes it more difficult for students and hobbyists hobbyists to work with when assembling their own boards, but it is difficult to find chips these days that are not surface mount. Another drawback of the AIC23 is that it is relatively expensive (about $6

each when not purchased in bulk).

Nevertheless, we concluded that the advantages of the AIC23 outweighed the disadvantages, so we added the AIC23 to our design. In order to design the supporting electronics for the AIC23, we referred to the Texas Instruments TMS320C6713 DSK board reference design [8], as well as another reference design from Texas Instruments [9]. We also followed the recommendations in the AIC23 datsheet [7]. The design is relatively straightforward, requiring only a handful of passive components (such as resistors, AC coupling capacitors, bypass capacitors, and so on). No external amplifiers or anti-aliasing filters are required, since amplification and anti-aliasing filtering are performed internally. Our design also includes a separate oscillator crystal to provide a clock for the AIC23, as discussed in greater detail below.

3.1.6 Audio Connectors

The board contains one stereo line input connector, one stereo line output connector, and one stereo headphone out connector. Each of these connectors is a stereo 1/8" female TRS connector. Although the AIC23 also supports a microphone input, we did not feel that including a microphone connector would be worth the increased cost, space, and layout complexity.

3.1.7 Oscillator Crystals

Our board contains two oscillator crystals: one for the dsPIC33F, and one for the AIC23. We could get away with using the same crystal for both devices if we want to, but crystals are relatively inexpensive and using separate crystals simplifies the design. (Sharing a crystal between both devices would introduce a dependency relationship between the dsPIC33F and the AIC23, which makes the design less flexible (i.e., if a hobbyist wants to remove the AIC23 later) and less intuitive to understand. In addition, sharing a crystal would result in a more complicated PCB layout, due to increased trace length and layout complexity.) Besides, the

HC-49US crystal package does not take up much space on the board.

We selected a 12MHz crystal for the AIC23 because it is one of few supported crystal frequencies listed in the AIC23 datasheet [7]. On the other hand, our choice of crystal frequency for the dsPIC33F was somewhat arbitrary, since the dsPIC33F's clock PLL is so flexible. We initially selected 8MHz since it is a common, moderately-fast crystal frequency. However, in future revisions of this this board, it may make sense to change this to a 12MHz crystal so that it is the same frequency as the AIC23's crystal. (From a board manufacturing standpoint, it makes more sense to reduce variety of parts). We simply chose the dsPIC33F crystal frequency before we chose the AIC23 crystal frequency, and never had any reason to change it.

3.1.8 LEDs and DIPs

Our reference design contains a few user-controlled LEDs and DIP switches. These components are inexpensive, space-efficient, and can be useful for basic debugging and for adding interactivity to simple labs. We selected low current (1.0mA) LEDs that do not need drivers (e.g., BJTs), thus preserving the simplicity of design.

The design also contains a power LED and a reset LED, which indicate when the power is on and when the dsPIC33F is being programmed, respectively. As we designed the board, we found that these two indicators of the board status were indispensable; without them, it is not always obvious when the power is on or if the programmer is holding the device in reset.

3.1.9 Power Supply

The design features two different options for power: it can be powered either from the PICkit2 programmer (USB power) or from two AA batteries. Although the students will have the board connected to the PICkit2 programmer for the majority of the time, the battery power option is still useful for giving demonstrations or for performing experiments that require

mobility.

All of the electronics on the board run on a 3.3V supply. The PICkit2 programmer automatically checks for the presence of this supply voltage. If the batteries are installed, then the switching step-up DC-to-DC converter maintains the regulated 3.3V supply, and the PICkit2 programmer determines that it does not need to drive the supply. When batteries are not used, the PIkit2 programmer will automatically provide the power for the board, since it can detect the absence of an external supply.

The first step in designing the DC-to-DC converter circuit was to determine the system's power requirements. When operating at 40MIPs, the dsPIC33F has an average current draw of about 60mA and a maximum of 90mA, according to its datasheet. The AIC23 is specified to draw no more than 26mA, even when all of its features are enabled. Each of the board's four low-current LEDs draws only about 1 to 2 mA of current when run continuously. Ignoring all other negligible sources power dissipation, we can thus conservatively estimate that the entire board under heavy use will draw no more than 120mA.

In order to verify these rough estimates with real data, we used a multimeter to measure the board's current draw under typical laboratory operating conditions. We programmed the dsPIC to run the program from Laboratory #3 (a fixed-point FIR filter at 44.1kHz, while blinking an LED with a 50% duty cycle). We connected the meter in series with the power supply in order to measure the current drawn by the entire board. Under these conditions, we measured an average current of 88mA. We therefore concluded that we would need to design the DC-to-DC converter to handle a typical load of about 100mA at 3.3V.

We decided to use the MAX756 step-up DC-to-DC from Maxim Semiconductor. The MAX756 is designed to operate at the 100mA, 3.3V specifications with over 80% efficiency. The MAX756 is also simple to use, requiring only a couple external capacitors, a Schottky diode, and an inductor. Moreover, it is one of the few DC-to-DC converter chips currently on the market that is still available in a DIP package, which makes it easy to use for prototyping. Furthermore, we already had experience using the MAX756 in a project for another course,

and the circuit design could be easily reused with only minor modifications. Considering that we had only a limited amount of time to design and test the DC-to-DC converter circuit, the MAX756 was a logical choice.

The next step was to select the type and number of batteries. We chose AA batteries because they are a standard form factor and are available in either alkaline or rechargeable. We next decided to minimize the number of batteries used in order to reduce the size and cost of the design. This is one of the reasons why we selected a step-up, rather than step-down, DC-to-DC converter. The choice for the number of batteries also depended on the desired battery life. Since each laboratory session for the ECE4703 course takes typically an entire afternoon, our design requirement was 4 hours of continuous use at minimum. Rechargeable NiMH batteries typically have about 2300 to 2500 mAh of capacity (for example, the standard Energizer NiMH rechargeable batteries are rated for 2500mAh) and an average voltage of about 1.2V. Alkaline batteries are typically somewhat higher in capacity and voltage. The following calculations give the expected battery life when using one NiMH AA battery, assuming 75% efficiency and output of 100mA at 3.3V:

$$PowerIn * Time * Efficiency = PowerOut * Time$$

$$(1.2V) * (2500mAH) * (0.75) = (3.3V) * (100mA) * (N\ hours)$$

$$N \approx 6.81\ hours$$

These calculations show that we would easily be able to achieve 4 hours of battery life using a single battery. However, in our final design, we decided to use two batteries for two reasons. First, experimentation with the DC-to-DC converter circuit showed that the MAX756 did a much better job maintaining regulation (i.e., less ripple and closer to 3.3V DC) for loads of 100mA when using two batteries rather than one. This is because there is less of a voltage difference between the input and the output, and the DC-to-DC converter does not have "work as hard" to maintain the regulation (for example, it does not have to draw as much input current). The second reason for selecting two batteries is that the

MAX756 performs significantly more efficiently for 100mA output with 2.4V input than with 1.2V input. According to the efficiency curves in the datasheet, we can expect close to 85% efficiency for two battery operation, as compared with 75% efficiency for one battery. The following calculations give the expected battery life for the two-battery configuration:

$$(2.4V) * (2500mAh) * (0.85) = (3.3V) * (100mA) * (N\ hours)$$

$$N \approx 15.45\ hours$$

Our expected battery life for our final design is therefore just over 15 hours of continuous, heavy operation. This easily meets our 4-hour minimum requirement.

3.2 Software Design

As we wrote the C and assembly code required to initialize the dsPIC33F's configuration registers and peripherals, it soon became clear that we would to organize all of this code into a set of software libraries. There is a significant amount of code required to configure the dsPIC33F, most of which is quite esoteric and certainly outside of the scope of the ECE 4703 course. The students in ECE 4703 must spend their time implementing DSP algorithms, not dabbling in the details of the SPI initialization registers. This is indeed already the case for ECE 4703; the students currently use the Texas Instruments DSK board support libraries. It therefore makes sense that we provide the students with the basic software libraries they need for the labs. Since there is a significant amount of code that must be performed in every single project that uses our dsPIC platform, it makes sense to create a set of libraries and then reference these libraries from the main program. This is common practice in industry. It is much cleaner and more maintainable to update a single set of libraries, rather than copy-paste many lines of source code into every single program.

The software libraries we eventually created provide the infrastructure required to initialize all of the dsPIC33F's configuration registers and peripherals. Our libraries are also easy for students to use, since they are designed specifically for the ECE 4703 laboratory

assignments; the student only needs to import a few header files and call a few simple functions. We have also used Doxygen to prepare reference documentation for the libraries. The source code for all of the libraries is available for curious students to learn more about the low-level configuration of the dsPIC33F.

The main features of the software libraries are as follows:

- compiler directives to initialize the dsPIC33F's configuration registers
- functions to set up the dsPIC33F's PLL and clock system
- functions to initialize the SPI and DCI/I2S peripherals to communicate with the AIC23
- functions to the AIC23's configuration registers (enable AIC23, specify sample rate, etc.)
- functions to configure the dsPIC33F's UART and to create a software console interface
- preprocessor macros to facilitate benchmarking using the dsPIC33F's timers
- macros to facilitate manipulation of general purpose I/O (e.g., LEDs and DIPs)
- assembly commands to allocate stack and heap space required for certain C functions

The best way to learn how to use these software libraries is to follow the laboratory procedure documents (see Appendix B) and to refer to the fully-working sample code for each laboratory (see Appendix C). For even more information about how the configuration registers are initialized, see actual library source code and its accompanying Doxygen documentation. Finally, for an extremely detailed description of the configuration for each component, please refer to the *Detailed Guide* in Appendix D.

To compile the C code for our software libraries and for the sample laboratory projects, we used the academic version of Microchip's MPLAB C30 compiler. Although the academic version has limited support for code optimization (you must purchase the commercial versions of the compiler for this), the compiler works fine for the purposes of the ECE 4703

laboratories. The MPLAB C30 compiler most of the standard C library functions, such as printf (our board libraries initialize the UART for use with the printf function). The C30 compiler also provides software support for floating point math, which is used in some of the ECE 4703 laboratory assignments. Specific instructions for MPLAB project setup and code compilation are available in the laboratory procedure documents and the *Detailed Guide*.

3.3 Prototype Testing and Debugging

In any large project with many complicated pieces, the problem of debugging the entire system all at once is intractable. However, if the system is decomposed into smaller pieces, and each piece is carefully tested individually, the problem becomes manageable. The case of our reference design was no different. We started with a simple system, then gradually added layer after layer of complexity, until our platform was fully functional. This section explains how we carried out the prototype development and testing.

Our first task was to successfully program the dsPIC33F. After setting up the dsPIC33F on the protoboard with the minimum required electronics (bypass capacitors, pullup resistors, etc.), we connected the PICkit2 programmer and attempted to get the dsPIC33F to communicate with the PC. We did not need to connect any power supplies, since the PICkit2 can supply power to the dsPIC33F. Even this simple task required us to learn many things about the dsPIC33F, such as how its ICSP programming interface works, how the dsPIC33F is powered, and how its configuration registers should be set up to specify which pins are used for programming. (For more information about any of these topics, please refer to the *Detailed Guide* in Appendix D.

3.3.1 Hello, World!

Our next goal was to create our equivalent of the ever-popular "Hello, World!" program: a simple blinking LED. In order to get the blinking LED working, we first analyzed all of the configuration registers to ensure that everything was configured properly. For example,

we configured the clock and PLL for the desired instruction clock rate (40MIPs using the internal RC oscillator). We also made sure to disable the watchdog timer. We connected an LED, using a simple 2N3902 NPN BJT to drive it (since the LED we had required at least 5mA but the dsPIC33F's I/O pins are only capable of driving 4mA). After writing some code to manipulate the I/O pins and more code for a software delay loop, we successfully got the LED blinking. Moreover, the rate at which the LED was blinking convinced us that the dsPIC33F was successfully operating at 40MIPs (which we also verified with an oscilloscope on the clock output pin). We wrote versions of the blinking LED program in both C and assembly, in order to practice MPLAB and test both the MPLAB C30 compiler and MPLAB ASM30 assembler.

3.3.2 PLL Configuration

Now that we had a basic system running, we carefully began to expanded functionality, adding one feature at a time. We first added a clock crystal for a more accurate clock and changed the PLL configuration accordingly. We also configured the hardware timer peripherals, which involved learning how the interrupts worked. These timers are important because we later use them for benchmarking and system performance measurement (not to mention create a more elegant blinking LED). More importantly, we also configured the dsPIC33F's UART. The PICkit2 programmer software on the PC has a feature call the "UART Tool", which is a small terminal program that can be used to communicate via the PICkit2 programmer and the dsPIC33F's UART peripheral. The UART proved to be an invaluable tool in debugging and data collection throughout this project.

3.3.3 AIC23 Debugging

The next stage in development was by far the most involved and time consuming: connecting and debugging the AIC23 audio codec. The first step was building the hardware. Since the AIC23 is only available in surface mount packages, we purchased a Schmartboard (see [22]),

which is basically a break-out board for prototyping with surface mount components. We carefully soldered the AIC23 chip to the Schmartboard and soldered some headers to the Schmartboard so that we could connect the AIC23 to the dsPIC33F. Although we carefully checked for shorts, as well as for continuity between the AIC23 and the rest of the circuit, we later found that some of the hardware connections were intermittent. (Indeed, the Schmartboard is not quite as easy to use as the Schmartboard company claims.) Figure 1 shows our breadboard of the complete prototype of our reference design.

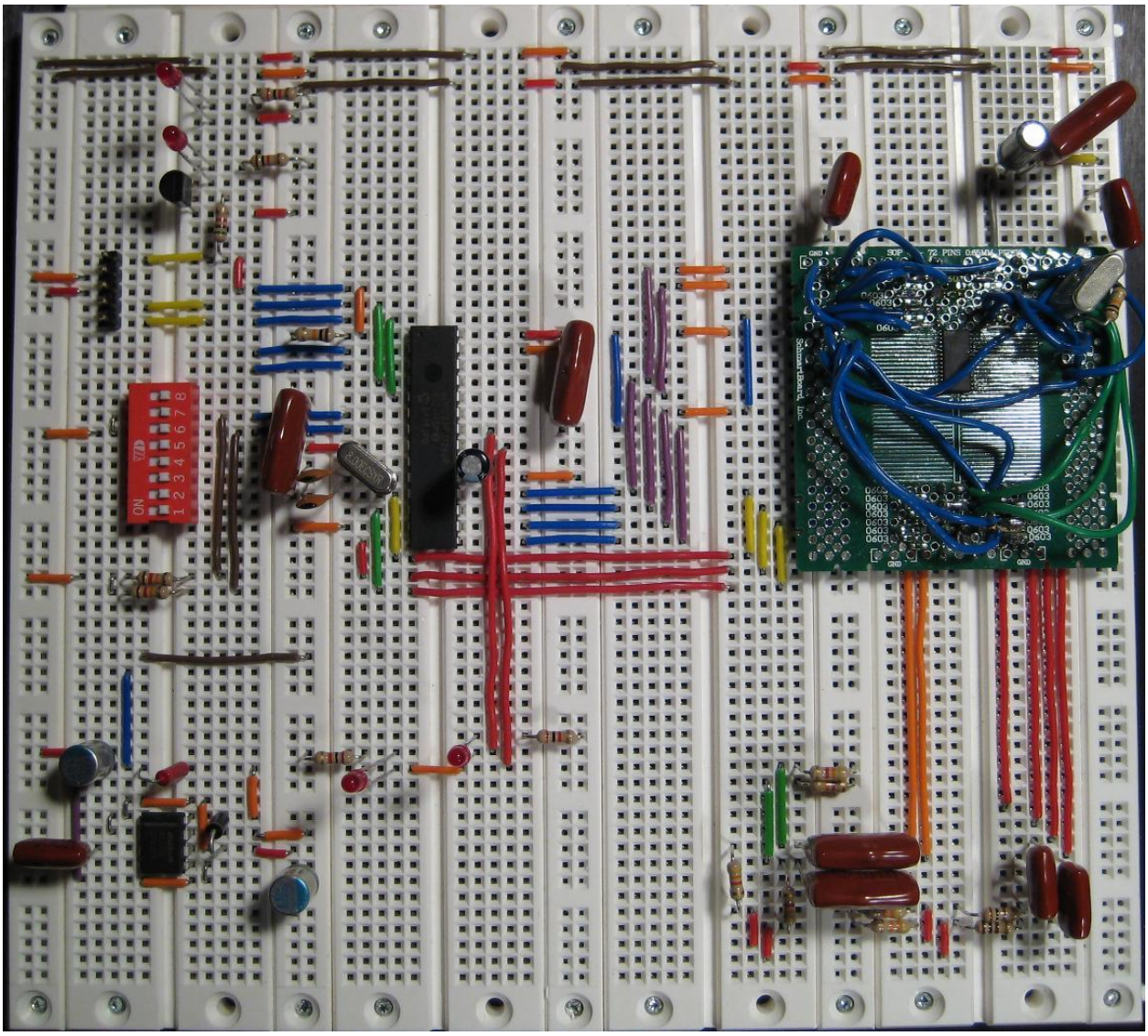


Figure 1: Breadboard of prototype reference design

3.3.4 SPI Interface Debugging

Now that that AIC23 hardware was built, it was time to configure the dsPIC to interface with it. Recall that the AIC23 uses an SPI interface for configuration settings, and an I2S-compatible data interface for the audio data. Since the AIC23 must be configured before it can send or receive audio data, we started with the SPI interface. We set up dsPIC33F's SPI control registers and tried sending configuration words to the AIC23 via the SPI interface. We used a multi-channel oscilloscope in order to verify that the SPI output from the dsPIC33F was working properly. We had hoped that by writing the correct SPI configuration registers, we could enable analog bypass mode on the AIC23 audio codec, which would give us a simple indication of whether or not the SPI and AIC23 was working. Unfortunately, the AIC23 audio codec was unresponsive. At this point, we had no idea whether the problem was hardware-related or software-related; for all we knew, the AIC23 chip itself could have been defective. We needed more information about the correct behavior of a known-working AIC23 configuration, so we carefully used the oscilloscope to analyze the behavior of the AIC23 on the TMS320C6713 DSK board. (Our complete data can be found in the **"Detailed Guide"** document.) Using this data, we eventually found that our problems were caused by poor solder connections between the AIC23 and the Schmartboard. Once we fixed the solder connections, our AIC23 chip began to behave exactly like the uninitialized AIC23 chip on the TMS32C6713 DSK. After making a few changes in the dsPIC33F regarding the behavior of the SPI framing signal, we were able to successfully put the AIC23 audio codec into bypass mode, and we concluded that SPI communication between the dsPIC33F and the AIC23 was successfully established.

3.3.5 I2S Debugging

We next worked towards configuring the dsPIC33F's DCI peripheral for I2S-format audio data communication with the AIC23. This was a tedious process because the configuration register configurations in both the dsPIC33F and the AIC23 for I2S communication are

rather involved. After much testing and analysis of the I2S signaling using the oscilloscope, we convinced ourselves that we were properly configuring the dsPIC33F and the AIC23. However, for some reason, the dsPIC33F refused to respond to the incoming data, even though all of the bus signals looked correct. The problem finally proved to be an internal pin failure on one of the dsPIC33F's input pins. After replacing the defective chip, the communication between the dsPIC33F and the AIC23 immediately began working correctly. We verified the correct operation of the dsPIC33F by sending various test signals in and out of the AIC23 codec.

3.3.6 The Importance of Good Documentation

In summary, most of the problems we had with getting the hardware working were related to poor documentation and to intermittent hardware failures. The AIC23 datasheet (see [7]) contains a great deal of information, but it does leave gaping holes in certain areas. We hope that our *Detailed Guide* document (see Appendix D) will help to explain certain things that this datasheet leaves out. Intermittent hardware failures are extremely difficult to detect, but diligence and methodical testing minimizes their occurrence.

At this point, most of our hardware was built and functional, so our efforts became primarily software-related. We spent some time gathering all of our working code into a set of software libraries, as discussed earlier. Once this software library infrastructure was in place, we began the process of rewriting the ECE 4703 laboratories and code to run on our new hardware platform. We also began the process of benchmarking and measuring performance of our system, which is the topic of the next section.

4 Performance Evaluation

The dsPIC33F must have sufficient computational abilities in order to be feasible for adoption into the ECE 4703 curriculum. If its DSP performance is too limited, the dsPIC33F platform will be impractical for use in an introductory real-time DSP course. For example, a certain amount of floating-point capability is required because floating-point arithmetic is significantly easier for students to program than fixed-point. Students start with simple floating-point systems and gradually add layers of complexity, working up towards fixed-point implementations. If the students were forced to start with fixed-point math because the hardware could not accommodate floating-point, students would have a tremendous amount of difficulty, since they would be unable to gradually build up complexity over time. The floating-point laboratories are intended to prepare the students for the transition to fixed-point. The hardware's overall performance must be sufficient to provide a suitable basis for practical, thought-provoking, and educational assignments.

After building the hardware prototype, we performed a considerable amount of computational performance analysis to ensure that our dsPIC33F platform would be practical for laboratory assignments. Figure 2 summarizes the numbers of FIR filter coefficients possible for different datatypes as a function of the sampling frequency. (Note that the number of coefficients scales linearly with respect to the sampling period but is inversely proportional to the sampling frequency, since $T_s = \frac{1}{2}/f_s$.) The following section explains the results of our performance evaluation and research in further detail.

4.1 Floating-Point Performance

Although the dsPIC33F lacks a hardware floating-point unit, the MPLAB C30 C compiler comes with a set of software floating-point libraries. The drawback of doing floating-point processing in software is that it is much slower (by at least two orders of magnitude) than using fixed-point arithmetic. However, even though the software floating-point performance

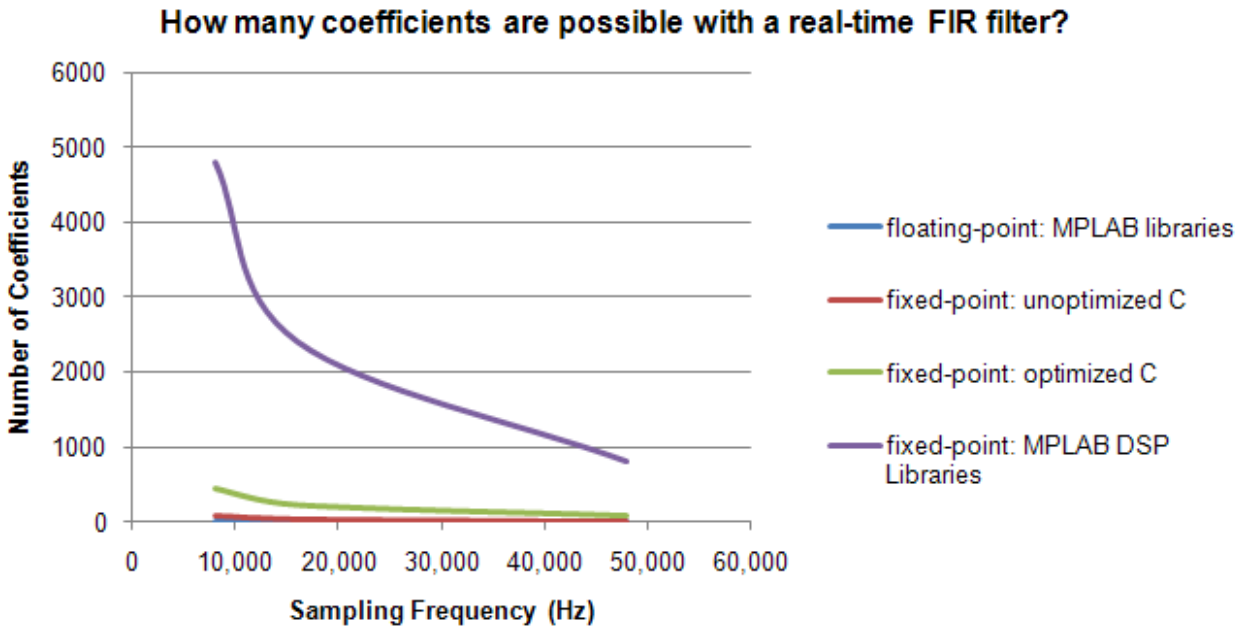


Figure 2: Summary of dsPIC33F Computational Performance

is poor, we found that the floating-point performance should still be sufficient for the first filtering lab. Once the students get their filters working in floating-point, they will quickly be able to switch to fast fixed-point processing for the subsequent labs.

We performed a few simple tests to measure the performance of the floating-point instructions. We set up one of the dsPIC33F's hardware timer peripherals and used it like a stopwatch to measure the amount of time various floating-point operations took to execute. For example, we found that single-precision addition instruction required 120 cycles and that a single-precision multiplication instruction required 117 cycles. More details about the process of configuring the dsPIC33F's timer and using it for benchmarking can be found in the Laboratory #4 Procedure document provided in Appendix B.

It turns out that Microchip's website [20] already provides a summary of the performance of their software floating-point libraries. A few of the more interesting numbers from the website are summarized in Table 3. The benchmarks provided by Microchip are quite close (within a few clock cycles) to the numbers we measured ourselves; the slight discrepancy is undoubtedly due to the fact that our measurements also include the clock cycles required to

move the 32-bit operands in and out of working registers.

Table 3: Floating-Point benchmarks (from Microchip's website [20])

Floating-Point Operation	**Clock Cycles**
addition	122
subtraction	124
multiplication	109
division	361
remainder	385
cosine	3249
sine	2238
exp	530
log	2889
sqrt	493

The audio samples sent back and forth between the AIC23 audio codec and the dsPIC are generally 16-bit signed integers. If we want to perform our signal processing in floating-point, we first need to cast the received integer samples as floating-point values. After our processing is done, we need to cast the floating-point values back into integers. We were curious how many clock cycles these casting operations required, so we benchmarked them. The results are summarized in Table 4. Notice that even the casting operations require a significant number of clock cycles.

When using a 40MHz instruction clock rate and an 8kHz audio sampling rate, there are (40E+6)/(8E+3) = 5000 clock cycles available to process the data in real-time. We measured that a single floating-point multiply-and-accumulate (MAC) operation requires about 272 cycles. If we perform the division, we see that we can only perform about 18 MAC operations and maintain real-time operation. If we take into account the cycles required to cast to and from floating-point, this number becomes about 16 MAC operations. This roughly translates

Table 4: Benchmarks for casting between floating-point and integer datatypes

Type of cast	**Clock Cycles**
float *rightarrow* 16-bit signed int	138
float *rightarrow* 16-bit unsigned int	136
16-bit signed int *rightarrow* float	185
16-bit unsigned int *rightarrow* float	193

to an FIR filter that is approximately 16 coefficients in length, or a DFII-SOS IIR filter that is about 8 coefficients in length.

To summarize, the software floating-point performance is abysmal. Even at the low, 8kHz sampling rate, the maximum possible filter order is quite small. Moreover, this performance cannot be improved using the optimizing C compiler because the floating-point software libraries are already optimized; floating-point emulation simply takes many clock cycles to perform. However, even though the performance is not great, it should still be sufficient for the purposes of the ECE 4703 laboratories. The students will be able to learn how to successfully implement their filters using all of the basic techniques, such as circular buffering and efficient convolution. After mastering the basics, the students will be equipped to move on to fixed-point arithmetic. The floating-point exercises are purely academic it is with the fixed-point processing that much higher filter orders are possible, allowing for more exciting signal processing applications.

4.2 Fixed-Point Performance

We also used the hardware timer to measure the performance of basic fixed-point arithmetic operations. Since the students would be implementing the filters in straight C code for most of the laboratories (except for Laboratory #4, the assembly code optimization lab), we performed these benchmarks using C code (not assembly). The results of these simple tests

are summarized in Table 5. It is not surprising that the long integer (32-bit) operations take longer than the regular 16-bit operations because the dsPIC33F is a 16-bit microprocessor and is hence designed for 16-bit math. Also notice that many of these operations, such as 16-bit additions and 16-bit multiplications, take four cycles to execute. However, from the dsPIC33F documentation, we know that these 16-bit operations should require only a single cycle. The discrepancy is due to the fact that our measurements also reflect the clock cycles required to move the operads in and out of the working registers (we can confirm this by taking a look at the disassembly listing window in MPLAB).

Table 5: Benchmarks for basic fixed-point operations

Fixed-Point Operation	**Clock Cycles**
16-bit integer addition	4
16-bit integer multiply	4
32-bit integer addition	8
32-bit integer multiply	14

If we follow a similar analysis to that above in the floating-point performance section, we see that if there are 5000 clock cycles available for real-time processing at an 8kHz sampling rate, we can perform over 700 multiply-and-accumulate (MAC) operations. This is significantly better than the roughly 16 MAC operations that we could achieve with floating-point arithmetic.

However, there is still room for improvement. We know from Microchip's documentation that the dsPIC33F should be able to perform a MAC operation in a single clock cycle. The C code in our basic test above obviously did not perform the MAC operations this efficiently, since they required substantial overhead to move data in and out of working registers. We could write hand-optimized assembly code to use the dsPIC's addressing modes and data paths much more efficiently; such is the focus of Laboratory Assignment #4 (see Appendix B). Alternatively, if we want to ignore the gory details of hand-coded assembly,

we can get significant performance increases by turning on the optimizing C compiler.

Although the free student version of the MPLAB C30 compiler does not perform all of the DSP-specific instruction optimizations, it still provides tremendous performance improvements over unoptimized compiled C code. For example, without optimization, the 11-coefficient FIR filter from Laboratory #3 required 618 clock cycles to execute. With full optimization enabled, the same filter required only 122 cycles to execute. Thus, **the optimizing C compiler improved performance by approximately a factor of 5.**

Microchip also provides an excellent set of free DSP libraries with its MPLAB C30 compiler. These libraries consist of C-callable assembly functions for various types of filters, transforms, and matrix arithmetic. These libraries are efficient, highly-optimized, and easy to use. For example, the example source code for Laboratory #6 (see Appendix C) uses the FIR filter from libraries, and as a result, all of the vocoder processing (three 100-coefficient filters and two modulations) requires fewer than 580 clock cycles to complete.

Microchip's website [11] provides a table which summarizes the capabilities of the DSP libraries and the required numbers of clock cycles. Table 6 summarizes a few of the more relevant DSP functions. The term "block" indicates that each type of filter can be used to calculate filter results for a range of 'N' time values. In the real-time systems of the ECE 4703 laboratories, we generally only compute the output value for one instant in time each time that the DCI interrupt service routine gets called. Therefore, we set N=1, and the equation for the number of clocks required for the FIR filtering becomes 57+M. We benchmarked this FIR filter code and obtained results that were within a few clock cycles of the predicted value (57+M). Thus, **Microchp's DSP library FIR filter code is extremely efficient, scaling linearly with the number of coefficients (M) and requiring only a modest amount of initialization overhead (57 clock cycles)**.

We can use the 57+M equation to extrapolate to find the maximum FIR filter order that can be used for real-time operation. Assuming a 40MHz instruction clock, we can achieve up to about **2400 coefficients at a 16kHz sampling rate or 800 coefficients at 48kHz.**

Table 6: MPLAB C30 DSP Libray Performance (from Microchip's website [11])

DSP Function	**Clock Cycles Required**
Block FIR	53 + N*(4+M)
Block IIR Canonic	36 + N*(8+7*S)
Block IIR Lattice	46 + N*(16+7*M)
Complex FFT	Test case: N=64 requires 3739 cycles

Key: N = # of samples, M = # of taps, S = # of sections

In conclusion, there are several ways to write efficient fixed-point code for the dsPIC33F. Our fixed-point performance analysis has showed that the dsPIC33F excels in fixed-point arithmetic, and it is more than powerful enough for the purposes of the ECE 4703 curriculum. The real-time vocoder algorithm of Laboratory #6 successfully illustrates that the dsPIC33F is computationally powerful enough to accommodate interesting and educational laboratory assignments.

5 Laboratory Redesign

The goal of this project was to develop a platform which could duplicate the pedagogical functionality (this is, to teach students about digital signal processing) of the expensive TI DSK in a cheaper, easily constructed package. The success of this hinged on re-examining the current curriculum for ECE 4703 and through testing the dsPIC hardware, it was determined that many of the laboratory assignments would have to be changed in order to create the same educational value while at the same time taking into account the performance limitations of the dsPIC. The most significant limitation being that the dsPIC has no dedicated hardware for performing floating point operations.

A summary of the major differences beween the original DSK labs and the new dsPIC labs is given in Table 7. Appendix B contains the new laboratory procedure documents. The next several paragraphs explain the major changes in greater detail.

5.1 Lab 1: Development Environment

Lab 1 was mostly focused on familiarizing oneself with the DSK and Code Composer Studio. It involved using switches to make LEDs turn on and understanding the basic functionality of the board. The revised Lab 1 is mostly the same - it provides a functional tutorial of the MPLAB development environment, introduces students to the dsPIC hardware, and has them run simple C code to make an LED turn on and off using a switch. In addition, it now has a emphasis on the UART protocol which the dsPIC supports. The assignment involves the students setting up and testing the UART interface, which becomes an invaluable debugging tool that they may use in all of the subsequent labs. This is a major deviation from the TI-based assignment, since the CCS automatically diverted printf() commands to the console in the development environment. MPLAB has no such functionality, so the UART interface is used in its place. This is educationally valuable since UART is a very common, standardized communication protocol and familiarity with it will give students an-

Table 7: Comparison between original labs and new labs

Lab	**Original Version**	**New Version**
Lab #1	Intro to CCS IDE	Intro to MPLAB IDE
	LED and DIPs	LEDs and DIPs
	Hello, World! (console)	UART
	CCS Grapher	Plot data with MATLAB
Lab #2	Floating-point FIR	Floating-point FIR
	Floating-point IIR	Floating-point IIR
Lab #3	Fixed-point FIR	Fixed-point FIR
	Fixed-point IIR	Fixed-point IIR
Lab #4	Assembly optimization	Assembly optimization
	Floating-point	Fixed-point
		Benchmarking techniques
Lab #5	Floating-point FFT	Fixed-point FFT
Lab #6	Adaptive filter	Vocoder

other tool to use in their future classes, projects, or careers. The original Lab 1 also explored Code Composer Studio's capabilities for graphing data from microprocessor memory. Since MPLAB does not have built-in graphing tools, the revised Lab 1 suggests that the students extract the data using either the debugger (Watch Window) or the UART, and then plot it using MATLAB. This practice using MATLAB will be a helpful refresher for the students in preparation for subsequent labs.

5.2 Lab 2: Floating Point FIR/IIR Filters

Lab 2 involved using floating point coefficients to create real time FIR and IIR filters. The core of this assignment remains unchanged - students will still utilize all floating point math to compute real-time filters. The major change here is that floating point operations must now be emulated with software libraries and performance is significantly degraded when compared to the TI platform which has dedicated floating point ALUs. The assignment was changed to have the students create a 10-tap filter at 8kHz sampling rate - a major change from the 41-tap, 44.1kHz filter they would have created using the TI hardware. Although this change reduces the complexity of the filter, it retains the educational goal from the original lab. Students will still be able to observe the mathematical concepts of filtering being applied in real-time. The decrease in performance has an additional education benefit; the idea that fixed-point operations are in most cases preferable to floating point is reinforced and should help students more completely understand the "why" behind fixed-point filters. This point was obscured by the fact that floating point and fixed point performance is more or less identical on the TI hardware. Since there is now a major disparity on the dsPIC hardware, students will quickly realize how powerful fixed-point operations are and will have more context to understand their implementation and use.

5.3 Lab 3: Fixed Point FIR/IIR Filters

Lab 3 moves the students entirely into fixed point. This lab now has the students using a much larger, 44-tap filter at 44.1kHz sampling rate. The students will fully be able to appreciate the benefits of fixed point math as they will easily compare its performance to the floating point filters in the previous assignment. This is more reinforcement to a common theme: fixed point operations are, in the majority of cases, preferable.

5.4 Lab 4: Code Optimization

Lab 4 has been significantly changed. The original assignment had the students use hand-optimized assembly code to create a real time filter and compared their computational efficiency with compiler-optimized C code. Now, students are tasked with using assembly code to create the largest filter they possibly can - in fixed point only. This change is due to the fact that the free, student version of MPLAB has only limited support for compiler optimization, as well as the fact that the dsPIC's hardware is not as parallel as the TI processor, thus eliminating a possible pathway for optimization. The dsPIC does however support DSP-specific instructions (e.g., a single-cycle multiply-accumulate) which the C compiler does not take advantage of. Students will use these instructions, combined with the advantage of fixed-point math, to create very complicated filters. This demonstrates the power of fixed point math, hand-coded assembly, and the dsPIC's architecture. Another major change is due to the fact that MPLAB does not support the same profiling options that CCS does. Students must now use the hardware timers provided on the dsPIC to measure how many clock cycles their filters are using. This is a very accurate way of benchmarking performance and additionally gives students experience working with hardware timers, which are common and extremely useful.

5.5 Lab 5: Fixed-Point FFT

Lab 5 remains mostly the same. The focus of the lab remains frame based processing and algorithmic efficiency. The major changes are that now everything is done in fixed point, with the complex twiddle factors calculated ahead of time in Matlab and imported into the code, and there is no longer a comparison to TI's optimized FFT algorithm since we are no longer using TI's processor.

5.6 Lab 6: Real-Time Vocoder

Lab 6 has been completely changed. The original assignment involved adaptive least-mean-squares (LMS) FIR filters for system identification and for adaptive noise removal. The new assignment requires the students to design and build a real-time vocoder. The main reason for the change was that adaptive filters are notoriously difficult to implement with fixed-point arithmetic. (The topic of fixed-point adaptive filters could easily fill an entire graduate-level DSP course.) Adaptive filters require a large amount of dynamic range; for example, adaptive step-sizes on the order of 10^{-6} to 10^{-12} are common in systems with adaptive filters of several hundred coefficients and input samples values in the range of ± 32768. Fixed-point numbers have a limited amount of dynamic range, which makes them difficult to use for adaptive filters. On the other hand, floating-point numbers have much greater dynamic range, but a floating-point adaptive filter would not be practical to implement on the dsPIC because the dsPIC lacks a hardware floating-point unit. Similar to the adaptive filter laboratory, the new vocoder laboratory introduces the student to an interesting signal processing application and teaches useful DSP techniques. Unlike the adaptive filter, the vocoder is much easier to implement in fixed-point arithmetic. In the new laboratory, the students learn new signal processing techniques, such as modulation (which has many applications in communication systems), and how to implement these techniques efficiently in software (for example, using look-up tables to obtain values of the cosine function used for modulation). The end result is a working vocoder; students can speak into a microphone and hear a pitch-shifted version

of their voices on the output of the speakers, completely processed in real time.

5.7 Remarks

In general, we tried to keep the basic learning content the same between the original DSK labs and the new dsPIC labs wherever possible. Many of the real-time DSP techniques, such as efficient FIR and IIR filtering, circular buffering, and compiler code optimization easily carried over from the original labs to the new labs. In certain situations, differences between the two hardware platforms necessitated changes in the labs. For example, the DSK supports instruction parallelization and pipelining, whereas the dsPIC contains a highly-optimized single path of execution. However, the learning outcomes in this example have not changed; the student still gains knowledge of efficient DSP microprocessor architectures. In other situations, limitations of the dsPIC necessitated changes; for example, the dsPIC's lack of floating point support resulted in a shift in focus from writing algorithms in floating point to heavily optimizing fixed-point code. The final type of changes between the original and new labs includes changes made in order to take advantage of the capabilities of the new hardware. For example, the vocoder algorithms in the new versions of Labs #5 and #6 are well-suited for fixed-point implementation, and the dsPIC33F excels in fixed-point processing. It is our hope that these re-designed laboratories will form a solid basis for a dsPIC33F curriculum for a re-imagined version of the ECE 4703 course.

6 Conclusions

Engineering is the art of balancing project requirements with cost. Running fixed-point code, the platform is more than capable of running any filter utilized in the ECE 4703 course. The architecture is simple enough for students without extensive computer engineering experience to understand and appreciate. The cost of the platform is comparatively low, which allows many more students to access the course. Most importantly though, it has a higher educational value than the current platform: by reinforcing real-world conditions, the effectiveness of fixed-point versus floating-point, and by giving students a platform that they can use in future work. In this way it is better than currently existing solutions; it is portable and much more flexible than any other system. Because the dsPIC is also an extremely capable microcontroller, it is not pigeon-holed into only being used for DSP applications. Controls, communications, and general-purpose processing are all within its grasp. The ISA is very similar to ARM and many other architectures. The computer architecture knowledge gained from this platform is easily applied to many other systems.

To fully complete the board and get it ready for a classroom environment, two things remain. First, a PCB design that students could use to assemble their own boards on would further streamline the process. While students could complete the project on a breadboard, they would need to purchase one much larger than the one provided in their ECE kits. This defeats the cost advantage of going with a breadboard. Second, the PICkit2 programmer needs to be integrated onto the board. There are significant cost savings in providing the programming hardware on the platform, rather than forcing the students to purchase it separately. This further increases the platform's portability.

This project provided a thorough examination of the current DSP course at WPI, spawned many new ideas, and provided a new look at the course material. This could be an ongoing process to continually improve the course. This project could be continued in several ways; perhaps even with other MQPs. First, cost effective integration of the programming interface into the reference board would make the hardware package totally complete; all students

would need is their board and a USB cable. Another direction might be a revision of Lab 6: what about investigating and implementing other, more advanced vocoder algorithms using the dsPIC hardware? Phase vocoders and time domain harmonic scaling would be interesting and non-divergent paths to consider.

References

[1] Rabiner, Lawrence R., and Schafer, Ronald W., . *Digital processing of speech signals / Lawrence R. Rabiner, Ronald W. Schafer* . Prentice-Hall, Englewood Cliffs, N.J. : , 1978 .

[2] Audacity. `http://audacity.sourceforge.net/`.

[3] R.V. Cox, R.E. Crochiere, and J.D. Johnston. Real-time implementation of time domain harmonic scaling of speech for rate modification and coding. *Solid-State Circuits, IEEE Journal of*, 18(1):10–24, Feb 1983.

[4] D. Cullen and J. DeFeo. dsPIC33F Audio Signal Processing Platform. MQP Final Report. `http://spinlab.wpi.edu/`.

[5] D. Cullen and J. DeFeo. The dsPIC33F Detailed Guide. `http://spinlab.wpi.edu/`.

[6] Texas Instruments. . `http://focus.ti.com/dsp/docs/dspplatformscontento.tsp?sectionId=2&familyId=114&tabId=2429`.

[7] Texas Instruments. AIC23 Datasheet. `http://www.ti.com/`.

[8] Texas Instruments. DSK Datasheet. `http://www.ti.com/`.

[9] Texas Instruments. Interfacing the TMS320F2833x to the AIC23B Stereo Audio Codec - spraaj2.pdf. `http://www.ti.com/`.

[10] Nasser Kehtarnavaz. *Real-Time Digital Signal Processing Based on the TMS320C600*. Elsevier, 2005.

[11] Microchip. dsPIC DSC DSP Algorithm Library. `http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en023598`.

[12] Microchip. dsPIC33F Family Reference Manual. `http://www.microchip.com/`.

[13] Microchip. dsPIC33FJ128GP802 Datasheet. http://www.microchip.com/.

[14] Microchip. dsPIC33F/PIC24H Flash Programming Specification. http://www.microchip.com/.

[15] Microchip. Getting Started with dsPIC30F Digital Signal Controllers User's Guide. http://www.microchip.com/.

[16] Microchip. Microchip Forums - PICkit2 FAQ. http://www.microchip.com/forums/printable.aspx?m=270347.

[17] Microchip. Mplab c compiler for academic use. http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en536656.

[18] Microchip. Mplab c compiler for pic24 mcus and dspic dscs. http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en010065.

[19] Microchip. MPLAB User's Guide - MPLAB_USER_GUIDE_51519c.pdf. http://www.microchip.com/.

[20] Microchip. PIC24 MCU / dsPIC DSC Math Library. http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2680&dDocName=en022432.

[21] Microchip. PICkit2 Programmer/Debugger User's Guide (51553E).

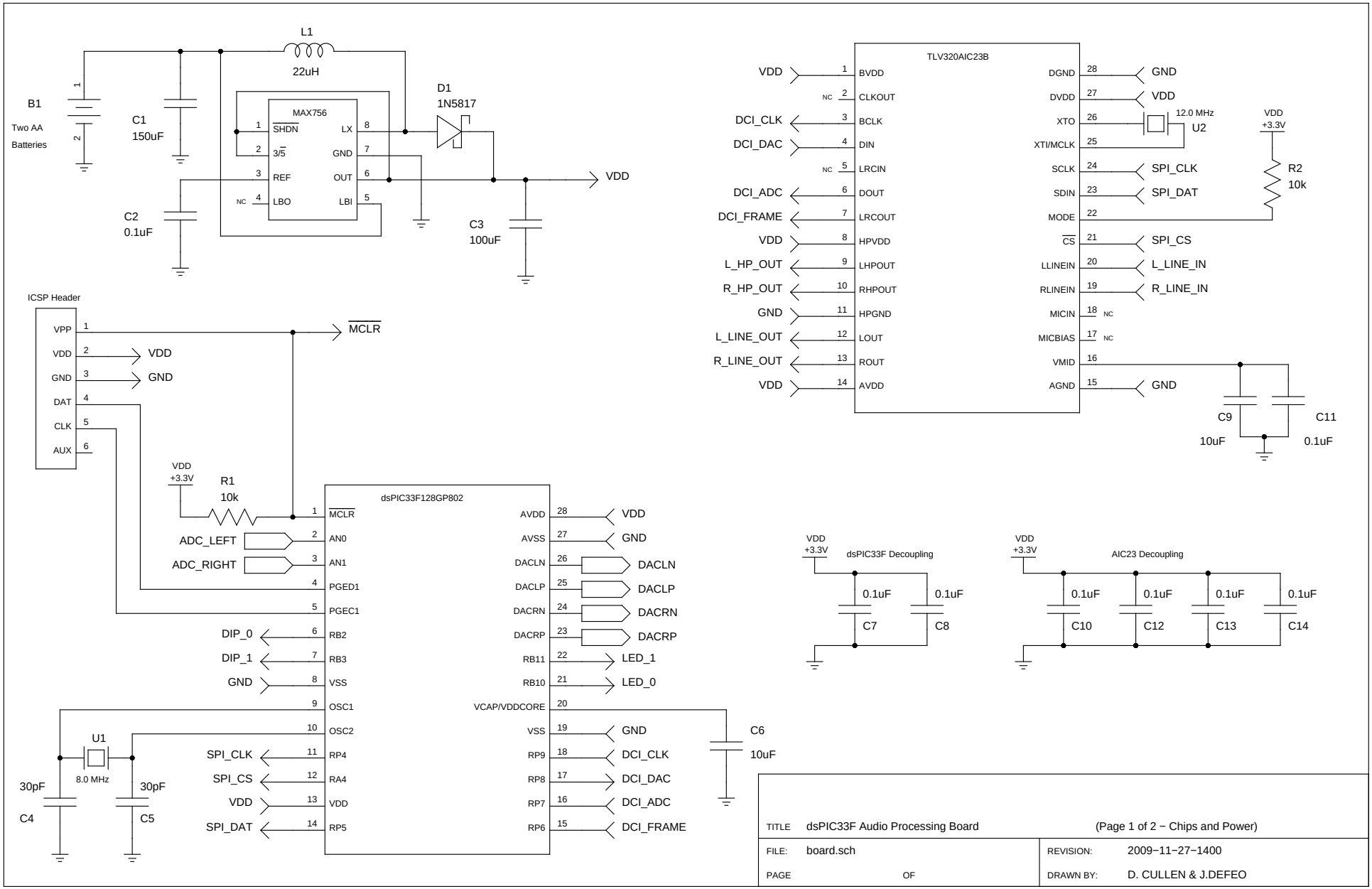
[22] Schmartboard. Schmartboard. http://www.schmartboard.com/.

[23] HI-TECH Software. http://www.htsoft.com/.

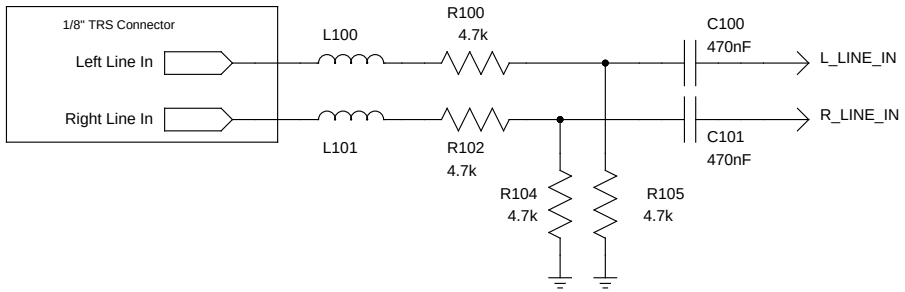
[24] Wikipedia. Vocoder. http://en.wikipedia.org/wiki/Vocoder.

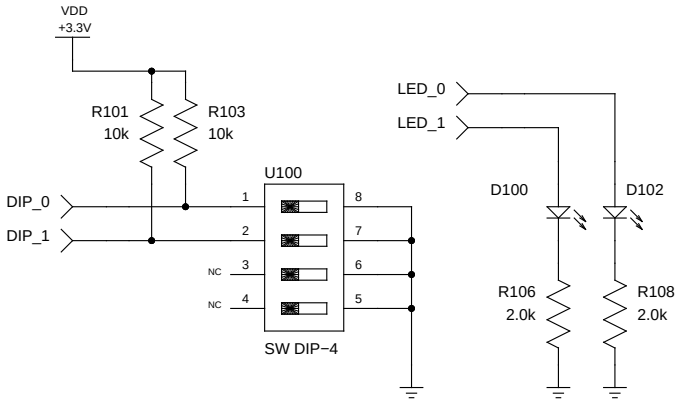
Appendices

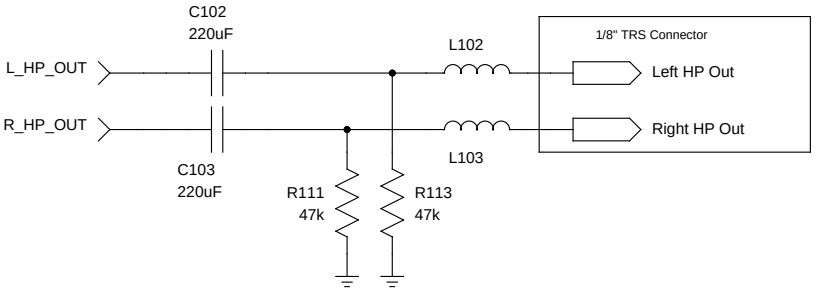
A Hardware Schematics

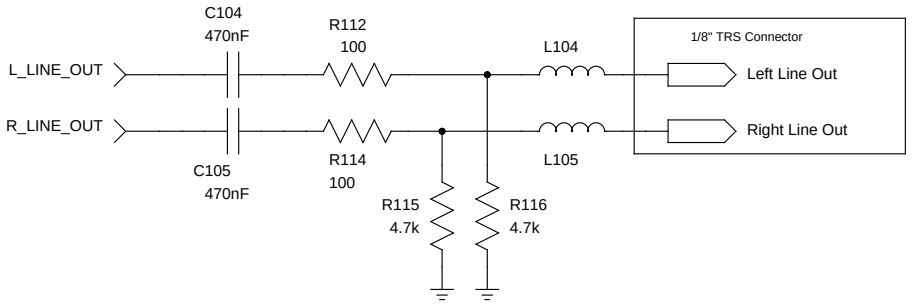


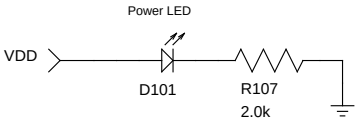
TITLE	dsPIC33F Audio Processing Board	(Page 1 of 2 − Chips and Power)	
FILE:	board.sch	REVISION:	2009−11−27−1400
PAGE	OF	DRAWN BY:	D. CULLEN & J.DEFEO

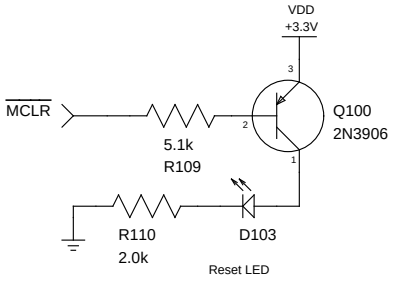












TITLE	dsPIC33F Audio Processing Board	(Page 2 of 2 − Input and Output)	
FILE:	board.sch	REVISION:	2009−11−27−1400
PAGE	OF	DRAWN BY:	D. CULLEN & J.DEFEO

B Laboratory Procedures

B.1 Laboratory #1

ECE4703 Laboratory Assignment 1 for the dsPIC33F

Last updated: 2009-12-04

The goals of this laboratory assignment are as follows:

- to familiarize the student with dsPIC33F and board hardware
- to familiarize the student with the MPLAB integrated development environment (IDE) and to guide the student through the process of building and testing simple projects
- to explore the various capabilities of the dsPIC33F platform

1 Introduction

This lab assignment is intended to familiarize you with most of the tools used to develop interesting real-time DSP systems in ECE 4703. A portion of this assignment is self-study and is necessary to give you a better understanding of the capabilities and structure of the lab tools. The remainder of this assignment is a hands-on exploration of the lab tools where you will develop three small C programs that will run on the dsPIC33FJ128GP802, interface with the basic I/O on the board, and perform some simple functions.

2 Prerequisites

This lab assumes the following:

- You have already properly set up the hardware and verified that it works properly. The hardware may be built on a breadboard or on an actual PCB; both are acceptable. For more details about hardware setup, please refer to [1].
- You will be using using the PICkit2 programmer.
- The latest versions of MPLAB, the MPLAB C30 Compiler, and the PICkit2 software have already been installed. Please see [1] for detailed installation instructions.

By D. Cullen and J. DeFeo. Based on Dr. Brown's ECE 4703 Lab #1 B Term 2008 document.

3 Self-Study

Begin by familiarizing yourself with the dsPIC33F board technical reference (available as a PDF file on the course web page). You do not need to memorize everything in this document but you should become familiar with the key features and basic operation of the board. In Chapter 2, focus on the AIC23 codec (Section 2.2) and the LEDs and DIP switches (Section 2.5). Also be sure to check out the board layout and connector information in Chapter 3. You should know the function of all connectors on the board and where the main components are located.

Next, download the dsPIC33F board library code from the course website. These libraries contain drivers that facilitate configuring and interfacing with all of the peripherals and features of the board. Be sure to browse through the html documentation (generated with Doxygen) provided with the libraries, in order to get a sense of which function calls are available and how to use them.

You may also want to download the **"Detailed Guide"** [1] document from the course webpage. This document provides detailed descriptions of how all of the hardware and software board work. If you ever have a question about any particular aspect of the system, you might find this document valuable.

At this point, you should be familiar with the board's hardware and the function calls that you can use to interface to the hardware. The final self-study task is to familiarize yourself with the MPLAB integrated development environment.

4 Programming Assignments

Please create separate MPLAB projects for each of the following programs. Your code should work correctly and must be liberally commented to receive full credit. Your code should compile without any errors or warnings.

4.1 Program 1: Hello World

In this part of the assignment, you will learn how to create projects with Microchip's MPLAB integrated development environment (IDE). Typically, the first program that developers write is a program that prints the text "Hello, World!" to a console. Since we won't be setting up the console until the second part of this assignment, we will create an even simpler version of the 'Hello World' program: a blinking LED.

4.1.1 Getting Acquainted with MPLAB

This section guides you through the process of creating a new MPLAB project, importing the required libraries and header files, writing a simple main program, and then compiling, downloading, and running the program.

1. Start the MPLAB IDE v8.36 software.
2. Create a new project using the Project Wizard, as explained in the following steps:
   (a) Select Project→Project Wizard... from the menu.
   (b) The "Project Wizard" dialog window should open. Click "Next."
   (c) On the next screen, select "dsPIC33FJ128GP802" for the device and click "Next."

(d) Select "Microchip C30 Toolsuite" in the "Active Toolsuite" drop-down box. An example screenshot of this step is provided in Figure 1. When you are finished, click "Next."

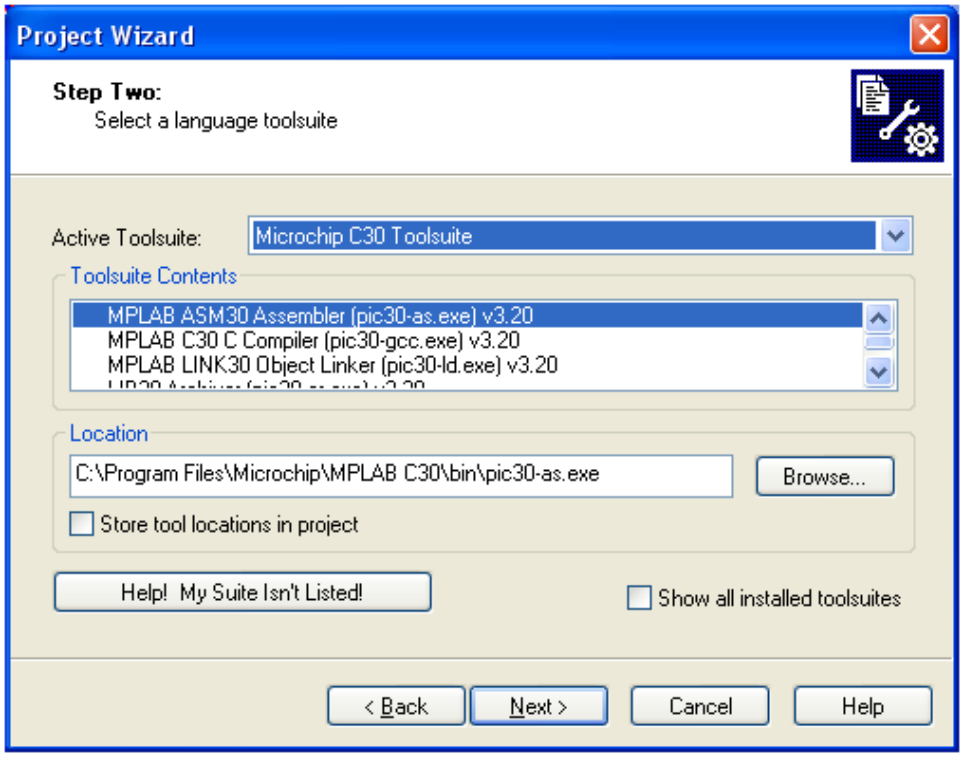


Figure 1: Select the Microchip C30 Toolsuite

(e) The next screen will ask you to create a new project file. Choose a suitable place to save your project files, then click "Next."

(f) The next screen asks you to "Add existing files to your project." We will do this later, so just click "Next" to skip ahead to the next screen.

(g) Click "Finish" on the final screen.

3. Next, you will need to set up the paths to the header (.h) files for the board libraries. If you have not already done so, you can download the board support software libraries from the course webpage (`http://spinlab.wpi.edu/`). To add the libraries to your project, right-click on the project in the Project Window and select "Build Options..." as shown in Figure 2. This should open the "Build Options" dialog box.

4. In the "Build Options" dialog window, click the "Directories" tab. Select "Include Search Path" from the drop-down box. Add a new entry for the path to the library "include" directory. An example screenshot is shown in Figure 3. Click "OK" when you're done to save your settings and close the "Build Options" dialog.

5. Now you must select the library archive (.a) file that contains the pre-compiled library code (corresponding to the header files you just included). When setting up the "Include Search Path" (see above), you may have noticed that there was a "Library Search Path" option in

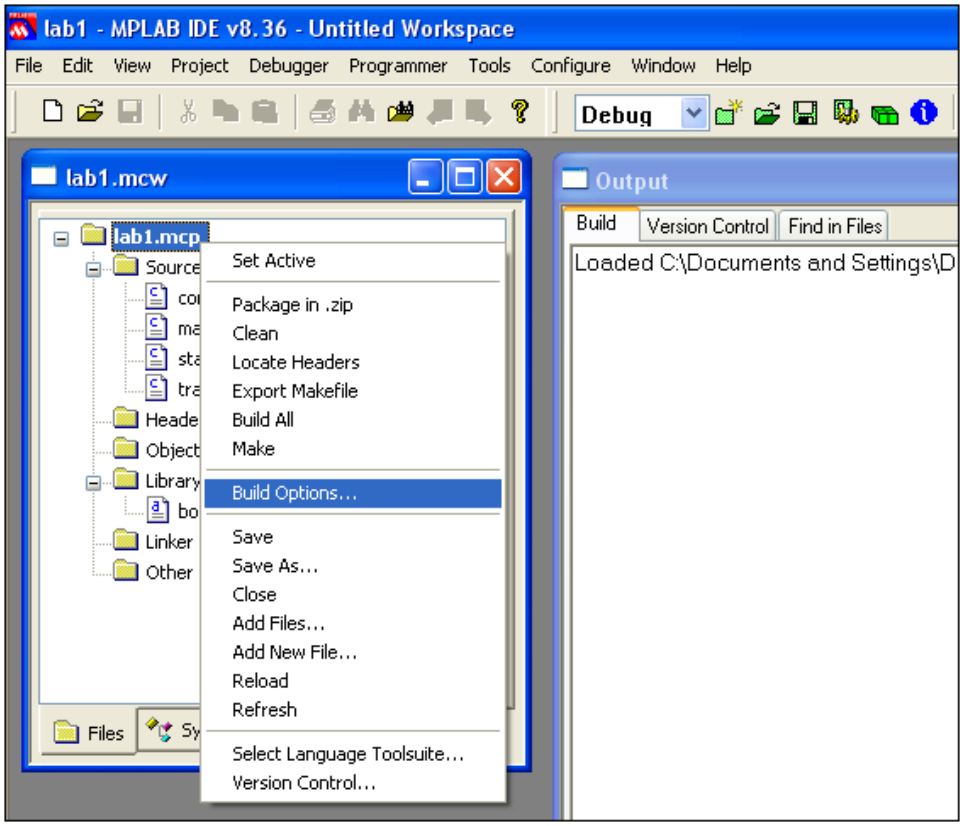


Figure 2: Open the Build Options

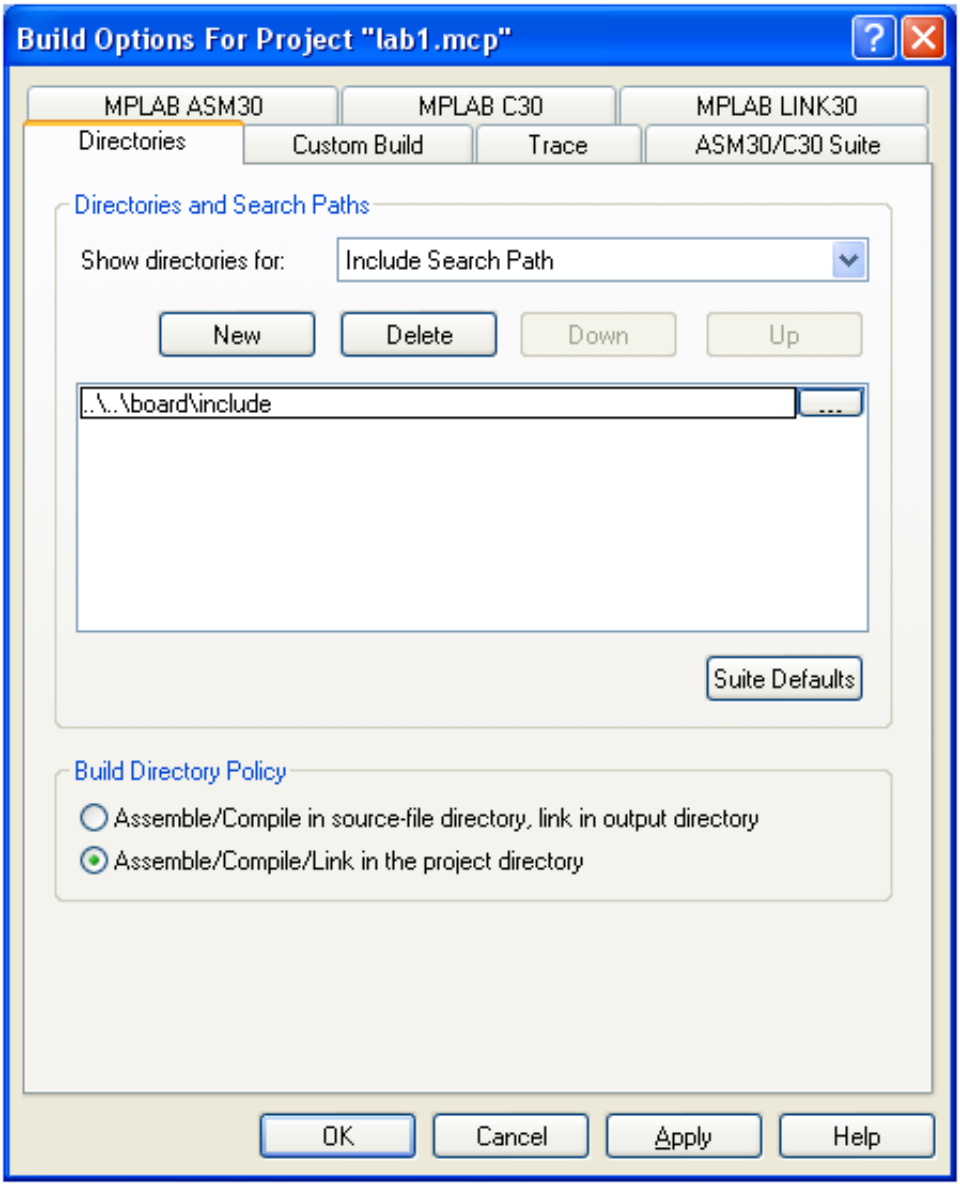


Figure 3: Add library search path

the drop-down box. However, the MPLAB linker seems to have trouble finding our library if we specify the search path there. Instead, we will add our library directly to the project. Right-click on the project in the Project Window, click "Add Fiels...," then browse to the "lib" directory and add "board.a" to the project.

6. Next, create a new C source file for your project using File→New, and save this file in your project directory. Name it something sensible, such as 'main.c'. Be sure to save it as a C file with the .c extension. After creating the file, be sure to add this file to the project using the using the "Add Files..." mechanism.

7. The code that you should use for this program is shown in Listing 1.

Listing 1 Hello, World!

```
#include <p33FJ128GP802.h>
#include "dspic.h"
#include "gpio.h"
#include "swdelay.h"

int main (void)
{
  dspic_init();
  gpio_init();

  while(1)
  {
    LED_0 = 1;
    swdelay(32);
    LED_0 = 0;
    swdelay(32);
  }
  return 0;
}
```

8. You should also add the following files to your project: "traps.c," "config_regs.c," and "stack_and_heap.c." These files should be located in the "src" directory of the board libraries. You will not need to modify these three files; simply add them to your project. The purposes of each of these files are as follows:

   - The file "traps.c" contains interrupt service routines which execute in the event of severe hardware failures. It is perfectly fine if you choose not to include "traps.c"; MPLAB automatically creates default ISRs for each ISR that you do not explicitly define. Besides, hardware failures should not happen under normal operation, and if they do, we do not need fail-safe operation for the purposes of this lab. We simply include "traps.c" for completeness and for compliance with Microchip's PIC coding conventions. You do not need to name the file "traps.c" for that matter, but we call named it "traps.c" to follow Microchip's coding conventions.

- The file "config_regs.c" contains code that properly initializes the dsPIC33F's configuration registers. If you forget to include this file, your code will compile without errors, but you will get a big yellow warning message in the PICkit2 programmer tool ("Warning: No configuration words in hex file") when you attempt to program the dsPIC. It is very important that you include "config_regs.c" in your project because the configuration registers are responsible for critical settings such as clock frequency, PLL, code memory write protection, programming pin selection, etc.
- The file "stack_and_heap.c" allocates some heap memory, which is required for certain stdio functions such as printf(). If you forget to include "stack_and_heap.c", you will get an error message when you attempt to build the project, reminding you to allocate some heap space.

9. To build the project, go to Project→Make or press the "F10" key. You should see "BUILD SUCCEEDED" in the build output window.

10. Now we will download the program to the dsPIC and test it.

    (a) At this point, you should check your wiring to make sure that the dsPIC board is connected to the PICkit2 programmer, and that the PICkit2 programmer is connected to your PC.

    (b) Now, select the programmer in MPLAB: Programmer→Select Programmer→PICkit 2. You should see some lines appear in the MPLAB output window, and the last line should say something like "PICkit2 Ready." If you see any errors, recheck your connections.

    (c) Programmer→Program. You should see some activity on the dsPIC board LEDs and some status messages in the MPLAB PICkit2 output window.

    (d) After programming, MPLAB holds the device in reset state by default. To start the execution of your program, click Programmer→ Release from Reset. V_{DD} should already be turned on by default. If for some reason it is not, simply run Programmer→Set Vdd on.

    (e) You should see one of the LEDs (LED_0) blinking.

    (f) Congratulations! You have just successfully programmed the dsPIC.

4.1.2 Explanation of the blinking LED source code

We will now explain each of the parts of the program, step by step.

- The "p33FJ128GP802.h" header file defines all of the register names, constants, preprocessor macros, etc. for the dsPIC33FJ128GP802. We will include it in every program. It is extremely useful for programming the dsPIC, and we suggest that you take a look at it. It can be found in "C:/Program Files/Microchip/MPLAB C30 Suite/Support/dsPIC33F/h/p33FJ128GP802.h."

- Be sure to call the "dspic_init()" function at the beginning of your main program. This calls routines to initialize the dsPIC33F's configuration registers with many important settings (such as disabling the watchdog timer, setting up the PLL and clock settings, etc.). You must include the "dspic.h" header file at the top of your file in order to access this "dspic_init()" library function.

- The "gpio.h" header file defines a set of preprocessor #define constants for accessing the LEDs and DIP switches. This header file also includes the declaration of the "gpio_init()" function, which configures the LEDs and DIPs in the dsPIC. You must call this function at the beginning of your main program before you can use the LEDs or DIPs.

- The "swdelay()" function is defined in the board libraries for your convenience. It implements a simple software delay loop (busy waiting). You must include the "swdelay.h" header file in order to be able to call this function. You can create your own software delay routine, if you wish, using counters. For reference, the dsPIC33F instruction clock is currently set up to operate at roughly a 40MHz clock rate. Most of the dsPIC instructions execute in a single clock cycle, so you can compute roughly the amount of times you need the software to sit in a loop (or nested loops) to count the desired time interval. In later lab assignments, we will introduce timers, which are much more efficient than software delay loops (since the timers run in hardware and use interrupts).

- For more details about the software libraries, please refer to the Doxygen documentation html pages provided with the libraries (available on the course website. Additional explanations can be found in the **"Detailed Guide"** document.

4.1.3 Further Experimentation

Now that you have the LED blinking, extend your program to include the DIP switches. Set up the DIP switches so that turning on either DIP switch turns on the other LED (LED_1). Note that the DIP switches are active low (i.e., 0=ON and 1=OFF).

4.2 Program 2: UART

The UART is an invaluable tool for debugging and data logging. In this section of the laboratory, we introduce the UART. The UART essentially allows us to call printf() commands within the C code in order to send the data out to a text terminal. We can also use the terminal to send data to the dsPIC. We can even write a tiny console program that will allow us to interact with the program while it is running on the hardware, allowing us to perform tasks such as change settings or check status.

1. In your C code, you must add a few things to get the UART working. You wil need to add #include statements for "stdio.h" and "uart.h." You will also need to add a call to the "uart_init()" function, just after the call to the "dspic_init()" function. After adding these things, compile your program as usual.

2. The "uart_init()" function sets up the dsPIC to use the programming pins (PGED1 and PGEC1) as UART1's RX and TX pins. We can use the PICkit2 UART Tool to interact with the dsPIC's UART. Unfortunately, the UART Tool is not built into MPLAB, which is why we also have the "PICkit2 Programmer Tool" installed. Start the PICkit2 Programmer tool. You'll notice that the PICkit2 Programmer tool can be used to program the dsPIC with the .hex programming file that the compiler generates when you compile your program in MPLAB. Try using the PICkit2 Programmer software to program the dsPIC. Although you must use MPLAB to compile your programs, you might find the PICkit2 Programmer to be easier to use than MPLAB for programming the dsPIC and running your programs.

(Try using MPLAB only for compiles and using PICkit2 to download the hex file and for its UART Tool.)

3. In the PICkit2 Programmer tool, select Tools→UART Tool... from the menu. Set the baud rate to 9600, make sure the VDD box is checked (to turn power to the dsPIC), then click the "Connect" button. You should see "Hello, World!" being printed to the console!

4. Notice that we used the carriage return/line feed (CRLF) character (\r\n) rather than just a \n in the printf. This is so that the newlines are interpreted correctly by the UART Tool terminal program.

5. Now that we have learned how to send data from the dsPIC to the PC, it is time to learn how to send data from the PC to the dsPIC. The board software libraries include functions for a terminal console system which facilitates handling input commands from the UART. The code shown in Listing 2 demonstrates how to use the console code to process simple user commands. If you are interested, you can look at the library source code to learn how the console works.

6. Try using the PICkit2 UART Tool to send commands to the dsPIC. Verify that the code works as you expect.

4.3 Program 3: AIC23 Stereo Codec

In this part of the assignment, you will have the opportunity to test the AIC23 stereo audio codec. The AIC23 codec contains both an analog-to-digital converter and a digital-to-analog converter. It is optimized for audio applications and is highly configurable. You will create a program to read 16-bit data samples for each channel (left and right channels) from the AIC23, perform some trivial signal processing, and send the samples back out through the AIC23.

1. The AIC23 supports a bi-directional serial data protocol called I^2S for communicating with the dsPIC. The dsPIC's peripheral used to handle the I^2S protocol is called the Data Converter Interface (DCI). The dsPIC also uses an SPI (Serial Peripheral Interface) bus to configure the AIC23's settings (such as volume, muting, sample rates, etc.).

2. The "AIC23_init()" function from the board support library initializes the dsPIC's DCI and SPI registers with the correct settings. This function also takes care of configuring the AIC23's own configuration registers via the SPI interface. In order to use this function, you must include its header file, "aic23.h."

3. In addition to calling the AIC23_init() function, you must also define the dsPIC's DCI interrupt service routine. This interrupt service routine gets executed every time the AIC23 sends a new data sample to the dsPIC. In this ISR, you can use the "AIC23_get_samples() function to access the new data samples that have just arrived from the AIC23. After performing your signal processing algorithms, you can call the "AIC23_set_samples()" function to put the output samples into the transmit registers to be sent to the AIC23 the next time a DCI interrupt occurs.

4. In this lab, we will configure the ACI23's sampling rate to 44.1kHz. This means that every (1/44.1) ms, the AIC23 sends a new data sample to the dsPIC, each time triggering the DCI interrupt. Thus, in order for our system to run in real time, all of the filtering or signal

Listing 2 UART Console

```
#include <p33FJ128GP802.h>
#include "dspic.h"
#include "uart.h"

void update_console(void);

int main (void)
{
  dspic_init();
  uart_init();

  while(1)
  {
    update_console()
  }
  return 0;
}

void update_console(void)
{
  // Checks if any new commands have arrived and processes them.

  unsigned char* cmd_buffer_ptr = cmd_get_buffer();

  if (cmd_get_flag())
  {
    switch(cmd_buffer_ptr[0])
    {
      case 'h':
printf("Hello, world!\r\n");
break;
      default:
printf("Unrecognized command.\r\n");
break;
    }
  }
  cmd_clear();
}
```

processing operations that we apply to the input data must complete within (1/44.1) ms so that the processed data can be sent out before the next input samples arrive.

5. Example code is provided in Listing 3

Listing 3 Audio Sampling Code

```
#include <p33FJ128GP802.h>
#include "dspic.h"
#include "aic23.h"

void __attribute__ ((interrupt, no_auto_psv)) _DCIInterrupt(void);

int main (void)
{
  dspic_init();
  AIC23_init(AIC23_44);

  while(1)
  {
    // (Just wait here for DCI interrupt to get called.)
  }
  return 0;
}

void __attribute__ ((interrupt, no_auto_psv)) _DCIInterrupt(void)
{
  signed int left, right;

  // Get new samples:
  AIC23_get_samples(&left, &right);

  // Send out the samples:
  AIC23_set_samples(&left, &right);

  // Clear the interrupt flag:
  IFS3bits.DCIIF = 0;
}
```

6. Try compiling and running this code. Use the function generator to provide a 1kHz sine wave as input to the line input audio connectors on the AIC23. Use the oscilloscope to observe the input and output waveforms. You should see that the output is the same as the input, since we are not performing any processing on the data in the dsPIC.

7. Try printing the left and right channel sampel data to the UART.

8. Try inverting each sample (multiplying it by -1) before sending it to the output. Try scaling each input sample by a constant before sending it to the output.

9. Feel free to experiment with a variety of different input signals (such as sinusoids, square waves, and even music).

10. Finally, implement a clip detector. Let LED_0 be the clip indicator for the left channel, and let LED_1 be the clip detector for the right channel. Since the samples are signed 16-bit integers, each has a range of -32768 to +32768. One way to implement a clip detector is to turn on an LED when magnitude of the sample exceeds a certain threshold (for example, `abs(sample)` $\geq$ `32000`). Unfortunately, if clipping occurs very infrequently, the LED may not stay lit long enough for the human eye to detect. Try experimenting with different values of the clipping threshold and the duration for which the LED remains lit. Ideally, the clip indicator will provide a clear indication of clipping but still turn off relatively quickly once the voltage drops below the threshold.

5 Additional Exploration

After completing the above programs, you should continue to experiment and familiarize yourself with MPLAB and the dsPIC board. Here are a few things to try:

1. **Try out the debugger.** In addition to functioning as a programmer, the PICkit2 also can function as debug interface. In MPLAB, you'll notice that there is a "Debugger" menu and a "Programmer" menu. The two modes are mutually exclusive; either you configure the dsPIC and PICkit2 to be in debugger mode or in programmer mode. This is because the PICkit2 loads some some special code in the memory of the dsPIC that is used for debugging. (The dsPIC debugging is implemented in software, not hardware; in other words, the compiler inserts instructions throughout the program for debugging, so that execution can be stopped when the dsPIC arrives at breakpoints or is single-stepped through code.) You can still program the dsPIC when in debugger mode; being in debugger mode just implies that you have extra functionality enabled for debugging.

    - Debugger→Select Tool→PICkit 2 to enable the debugger. You might get a warning telling you that programmer and debugger modes cannot be loaded at the same time; just click 'OK' and proceed.
    - Try setting some breakpoints in the code and running to the breakpoints. You will find that the dsPIC executes instructions up to, *and including*, the line at which the breakpoint is set. This is different from other debuggers that you may have used, which generally stop before executing the instruction of the line at which the breakpoint is set. Don't worry, this is not an error; this is how it is supposed to work. Microchip refers to this as "skid-stop debugging."
    - You can only set a maximum of two breakpoints at a time when using the PICkit2 debugger.
    - If you have too many variables open in the watch window or the special function register (SFR) watch window, and you let the program free run, MPLAB might stop responding for a few seconds. This is because it is busy transferring lots of data. Simply be patient and wait 10 or 15 seconds or so; MPLAB generally will recover. The solution to this problem is to reduce the number of variables in the watch window, or close the SFR watch window while the program is executing. This reduces the amount of data that

needs to be transferred in real-time, and consequently MPLAB will be less prone to hanging.

- As always, refer to the extensive documentation provided by Microchip for more information.

2. **Capturing and plotting data**. It is useful to be able to capture data from the system and plot it. There are two main ways to accomplish this: using MPLAB's debugger/watch windows or using the UART.

   - In your C code, create two arrays to store the most recent 1024 samples from the ADC for each channel.
   - Try using the MPLAB debugger watch windows to view the contents of these arrays. Try copying the data out of the watch window and pasting it a text file, and then use try using MATLAB to plot the data.
   - Use printfs and the "Log to File" feature of the PICkit2 UART Tool to dump the data to a text file. You can use the printf to output the data in any desired format (for example, any of the following might work: ascii hex, ascii decimal, comma-separated values, formatted already in vector syntax to be pasted directly into a MATLAB .m file, etc). Then plot the data using MATLAB. (Hint: Depending on how you formatted the data with your printf, you might find MATLAB's fopen/fscanf useful. If you formatted your data to look like a .m file, you can load it with the 'load' command.)

6 In Lab

Teams of two are permitted. In the case of an odd number of students in the course, one team of one/three will be formed with permission of the instructor. You will keep these lab partner(s) for all of the laboratory assignments in this course. You and your lab partner(s) will submit joint project code and lab reports that receive a single grade.

7 Specific Items to Discuss in Your Report

Please refer to the general report guidelines provided on the course web page for an overview of the ECE4703 report format. Since the first two programs in this assignment were quite simple, you do not need to discuss them in the report. Your report should focus on your methods, solutions, and results obtained with Program 3, especially the design of the clip detector. In addition to discussing the design of your clip detector, you should also perform a few simple experiments with a function generator and an oscilloscope and report on the following characteristics of the AIC23 codec:

1. What is the maximum peak-to-peak input voltage (at the "line in" input) that the AIC23 codec can accept before clipping occurs?

2. What is the maximum peak-to-peak output voltage (at the "line out" output) the AIC23 codec can generate?

3. What happens when you try to sample a sinusoid with frequency higher than the Nyquist rate? To test this, set the sampling frequency of the codec to 44.1kHz and use the function generator to provide a 25kHz sinusoidal input to the codec.

4. What happens when you sample a signal with DC offset? To test this, use the function generator and apply a one volt offset to a 1kHz sinusoidal input with a one volt peak-to-peak voltage. Plot the input buffer using Matlab. What does the DC offset look like after sampling? Can you explain what is going on here?

5. Try generating an output with a DC offset by adding a constant to each sample before writing the sample to the AIC23 codec. Look at the result on the scope. What happens?

6. Finally, determine where the codec configuration is written in your code and take a look at the AIC23 codec datasheet (a link is provided on the course web site). Notice that, among other configuration options, the codec has registers that allow for left/right line input channel volume control in 1.5dB steps. In Sections 3.2.1 and 3.2.3 of the datasheet, it is stated that the ADC and DAC each have a full-scale range of 1.0 VRMS. Using what you know about the codec configuration, can you explain how these specifications agree or disagree with the first and second results that you obtained regarding the maximum peak-to-peak input and output voltage?

Where appropriate, include plots generated in Matlab and/or screenshots from the oscilloscope.

8 Final Remarks

Please be aware that each of the laboratory assignments in ECE4703 will require a significant investment in time and preparation if you expect to have a working system by the signoff period on the assignment's due date. This course is run in "open lab" mode where it is not expected that you will be able to complete the laboratory in the scheduled official lab time. It is in your best interest to plan ahead so that you can use the TA and instructor's office hours most efficiently.

References

[1] D. Cullen and J. DeFeo. The dsPIC33F Detailed Guide. `http://spinlab.wpi.edu/`.

B.2 Laboratory #2

ECE4703 Laboratory Assignment 2 for the dsPIC33F

Last updated: 2009-12-04

The goals of this laboratory assignment are as follows:

- to familiarize you with the digital filter design tools in Matlab,
- to familiarize you with floating point real-time finite impulse response (FIR) filtering on the DSPIC33F,
- to familiarize you with floating point real-time infinite impulse response (IIR) filtering and two different realization structures on the DSPIC33F, and
- to demonstrate some of the differences between real-time FIR and IIR filtering.

1 Filter Specifications

While the code you write for this assignment will allow you to realize almost any type of filter, you will be evaluated on your ability to realize a band-stop filter that satisfies the following requirements:

- First passband: 1000 Hz
- First stopband: 1500 Hz
- Second stopband: 2000 Hz
- Second passband: 2500 Hz

Use the Matlab filter design tools, e.g. fdatool, to design filters for each part below that satisfy all of the requirements. Be sure to note your design choices in your report. Use fdatool to plot the impulse response, magnitude response, and phase response of your filter. As shown in class, you can export your filter coefficients to C header files from fdatool. To do this correctly, please make sure you have selected the correct filter realization structure first, and that you are exporting the filter coefficients as the correct data type. All filter coefficients in this assignment should be generated as single precision floats.

Please use a least-squares FIR type and 8000 Hz sampling rate. Specify the order of the filter to be 10. These numbers are necesary to get floating point code running in real time on the dsPIC.

Why such low numbers? The dsPIC does not have hardware support for floating point arithmetic. These procedures must be emulated in software! This is, as you will soon discover, is quite slow. On average, a floating point add takes 328 clock cycles. On the dsPIC, an integer add takes

By D.Cullen and J.DeFeo. Based on Dr. Brown's ECE 4703 B-Term 2008 lab documents.

only a single clock cycle. Big difference! In the next lab, you will be doing all of your filtering using integers because of this huge performance gap. The goal of this lab is to demonstrate that gap effectively as well as demonstrate some real-time filtering.

2 Prerequisites

This lab assumes the following:

- The hardware has already been properly set up and it has been verified that it works properly. Hardware setup on a breadboard or on an actual PCB board are both acceptable.
- That you are using the PICkit2 programmer.
- The latest versions of MPLAB, the MPLAB C30 Compiler, and the PICkit2 software have already been installed.

3 Programming Assignments

Please create separate MPLAB projects for each of the following programs. Your code should work correctly and must be liberally commented to receive full credit. Your code should compile without any errors or warnings.

3.1 Part 1: Floating Point DF-I FIR filtering

Create a new MPLAB project and write source code that satisfies the filter constraints described in the Filter Specifications section using a direct form I FIR filter with single-precision floating point coefficients. Your filter should run in real-time on the DSK at a sampling frequency of 8kHz. A suggested outline for your program is given below (you are welcome to any approach that makes sense): 1. Declare variables including a single-precision floating point buffer for the input samples. Al- ternatively, you could declare the input sample buffer as short, and then cast these shorts to single-precision floats each time you multiply the input samples with the filter coefficients. Feel free to see which approach runs faster. 2. Initialize the dsPIC, codec, set the sampling rate, set up interrupts, etc. 3. Initialize the input buffer array to zero. 4. In the ISR: (a) Read in the input sample from the right channel of the AIC23 codec. Because we are reading the channels in stereo, you will also get the left channel too but you dont need to use it. (b) Put the input sample into the input buffer. (c) Compute the filter output using the direct form I calculations with your floating-point filter coefficients and the input buffer (either stored as floating point or dynamically cast as floating point during the calculations). See the Kehtarnavaz textbook for examples if you are unsure how to do this. (d) Cast the floating-point final result to a short for output to the DAC. You may want to check for overflow here and perhaps even light up an LED if overflow is detected. (e) Write the short filter output to the codec (right channel).

Convert the final result to a 16-bit signed integer (short datatype) only prior to output by the AIC23 DAC.

3.2 Part 2: Floating-point DF-II Single-Section IIR filtering

Create a new MPLAB project and write source that satisfies the filter constraints described in the Filter Specifications section using a direct form II single-section IIR filter with single-precision

floating point coefficients and single precision floating point intermediate results. You can use fdatool to generate appropriate filter coefficients in the appropriate filter structure and export these coefficients to a header file for use in your MPLAB project. Your filter should run in real-time on the dsPIC. Use an elliptic type IIR filter.

3.3 Part 3: Floating-point DF-II Second-Order-Sections IIR filtering

Create a new MPLAB project and write source that satisfies the filter constraints described in the Fil- ter Specifications section using a direct form II second-order-sections IIR filter with single-precision floating point coefficients and single precision floating point intermediate results. You can use fdatool to generate appropriate filter coefficients in the appropriate filter structure and export these coefficients to a header file for use in your MPLAB project. Your filter should run in real-time on the dsPIC. Please write one DF-II SOS function that you can call multiple times from your interrupt service routine. It is up to you how to do this, but the idea here is that your overall filter can be realized by calling the DF-II SOS function several times, each time passing the output from the last DF-II SOS to the input of the next DF-II SOS. All intermediate results should be kept as single precision floats. The usual difficulty with this part of the assignment is in interpreting the header files that Matlab generates for DF-II SOS filters. Please see the course web site links and files page for some information on how to correctly interpret these header files.

4 In Lab

Teams of two are permitted. In the case of an odd number of students in the course, one team of one/three will be formed with permission of the instructor. You will keep these lab partner(s) for all of the laboratory assignments in this course. You and your lab partner(s) will submit joint project code and lab reports that receive a single grade.

5 Specific Items to Discuss in Your Report

In your report, provide theoretical and experimental magnitude response results in the same plot (use different colors or line styles) for each part of the assignment. Your plots should look professional with axis labels, grid lines, and legends as necessary. Your plots should provide immediately convincing evidence to the grader that your filter running on the dsPIC satisfies the requirements and that that it agrees with the magnitude response predictions from Matlab. Be sure to explain any discrepancies (there may not be any). Discuss any scaling considerations that you used to get your filters working correctly and to avoid overflow in the intermediate results and the output.

6 Final Remarks

Please be aware that each of the laboratory assignments in ECE4703 will require a significant investment in time and preparation if you expect to have a working system by the signoff period on the assignment's due date. This course is run in "open lab" mode where it is not expected that you will be able to complete the laboratory in the scheduled official lab time. It is in your best interest to plan ahead so that you can use the TA and instructor's office hours most efficiently.

B.3 Laboratory #3

ECE4703 Laboratory Assignment 3 for the dsPIC33F

Last updated: 2009-12-04

The goals of this laboratory assignment are as follows:

- to familiarize you with the fixed-point digital filter design tools in Matlab and
- to familiarize you with fixed-point real-time FIR and IIR filtering on the DSPIC33F,

This assignment extends the work that you did in Lab 2 to redesign your filters to use only fixed-point calculations. Fixed-point processing is used in many applications requiring high speed, low power, and/or low cost. The dsPIC33F processor has no hardware floating point unit and as you saw in Lab 2, has extremely poor floating point performance. By switching to fixed point, you will be able to create much higher order filters than in floating point. The goal is to realize exactly why fixed-point is preferable and so prevalent in most DSP applications: you can use very inexpensive parts to create high order, effective filters.

1 Filter Specifications

While the code you write for this assignment will allow you to realize almost any type of filter, you will be evaluated on your ability to realize a band-stop filter that satisfies the following requirements:

- First passband: 7200 Hz
- First stopband: 9600 Hz
- Second stopband: 12000 Hz
- Second passband: 14400 Hz

Use the Matlab filter design tools, e.g. fdatool, to design filters for each part below that satisfy all of the requirements. Be sure to note your design choices in your report. Use fdatool to plot the impulse response, magnitude response, and phase response of your filter. As shown in class, you can export your filter coefficients to C header files from fdatool. To do this correctly, please make sure you have selected the correct filter realization structure first, and that you are exporting the filter coefficients as the correct data type. All filter coefficients in this assignment should be generated as single precision floats.

You may use any FIR type this time and 44100 Hz sampling rate. This is a major change from the previous lab, and further demonstrates the performance advantage of fixed-point calculations!

By D.Cullen and J.DeFeo. Based on Dr. Brown's ECE 4703 B-Term 2008 lab documents.

Prior to the generation of the header files, your coefficients should be quantized to a coefficient word length of 16 bits (including the sign bit), i.e. the data type for all filter coefficients should be signed integer. Double check the header files that Matlab generates to be sure that all of the integer filter coefficients fall in the range -128 to +127. You should also confirm that Matlab is giving the best Q-representation that will allow you to represent your filter coefficients with minimum quantization error while also avoiding overflow. Note that the optimum number of fractional bits might be a negative number. This is ok it just means that you are shifting the decimal point to the right rather than to the left. Be sure to note the number of fractional bits implicit in your fixed-point filter coefficients and comment your code with this information. Use fdatool to plot the impulse response, magnitude response, and phase response of your filter with quantized coefficients. Compare these results to the unquantized coefficients and discuss any differences in your report.

2 Prerequisites

This lab assumes the following:

- The hardware has already been properly set up and it has been verified that it works properly. Hardware setup on a breadboard or on an actual PCB board are both acceptable.
- That you are using the PICkit2 programmer.
- The latest versions of MPLAB, the MPLAB C30 Compiler, and the PICkit2 software have already been installed.

3 Programming Assignments

Please create separate MPLAB projects for each of the following programs. Your code should work correctly and must be liberally commented to receive full credit. Your code should compile without any errors or warnings.

3.1 Part 1: Floating Point DF-I FIR filtering

Create a new MPLAB project and write source code that satisfies the filter constraints described in the Filter Specifications section using a direct form I FIR filter with16 big signed integer coefficients. coefficients. Your filter should run in real-time on the DSK at a sampling frequency of 44100kHz. You may want to do this in two steps:

1. Use the quantized filter coefficients. Like Lab 2, include some code to keep track of the largest positive and largest negative valued intermediate results (including the final output) in the filter. Test your code with full-scale sinusoids in the passband and stopband, as well as with white noise. You shouldnt have any overflow when using floating point intermediate results (except perhaps at the output), but this step will help to determine if you will need to perform any scaling to avoid overflow when the intermediate results are fixed-point.

2. Make the code 100results. How many fractional bits should your intermediate results have? Use your results from the previous part to perform any scaling as needed to avoid overflow in all intermediate results and at the output.

3.2 Part2: Floating-point DF-II Second-Order-Sections IIR filtering

Create a new MPLAB project and write source that satisfies the filter constraints described in the Fil- ter Specifications section using a direct form II second-order-sections IIR filter with single-precision floating point coefficients and single precision floating point intermediate results. You can use fdatool to generate appropriate filter coefficients in the appropriate filter structure and export these coefficients to a header file for use in your MPLAB project. Your filter should run in real-time on the dsPIC. Please write one DF-II SOS function that you can call multiple times from your interrupt service routine. It is up to you how to do this, but the idea here is that your overall filter can be realized by calling the DF-II SOS function several times, each time passing the output from the last DF-II SOS to the input of the next DF-II SOS. All intermediate results should be kept as single precision floats. The usual difficulty with this part of the assignment is in interpreting the header files that Matlab generates for DF-II SOS filters. Please see the course web site links and files page for some information on how to correctly interpret these header files.

4 In Lab

Teams of two are permitted. In the case of an odd number of students in the course, one team of one/three will be formed with permission of the instructor. You will keep these lab partner(s) for all of the laboratory assignments in this course. You and your lab partner(s) will submit joint project code and lab reports that receive a single grade.

5 Specific Items to Discuss in Your Report

In your report, provide theoretical and experimental magnitude response results in the same plot (use different colors or line styles) for each part of the assignment. Your plots should look professional with axis labels, grid lines, and legends as necessary. Your plots should provide immediately convincing evidence to the grader that your filter running on the dsPIC satisfies the requirements and that that it agrees with the magnitude response predictions from Matlab. Be sure to explain any discrepancies (there may not be any). Discuss any scaling considerations that you used to get your filters working correctly and to avoid overflow in the intermediate results and the output.

Please comment about the performance differences between floating and fixed point. Why use fixed point when you have to worry about overflow, fractional bits, and quantization? What did moving to fixed point allow you to do that you could not accomplish with floating point?

6 Final Remarks

Please be aware that each of the laboratory assignments in ECE4703 will require a significant investment in time and preparation if you expect to have a working system by the signoff period on the assignment's due date. This course is run in "open lab" mode where it is not expected that you will be able to complete the laboratory in the scheduled official lab time. It is in your best interest to plan ahead so that you can use the TA and instructor's office hours most efficiently.

B.4 Laboratory #4

ECE4703 Laboratory Assignment 4 for the dsPIC33F

Last updated: 2009-12-04

By D. Cullen and J. DeFeo. Based on Dr. Brown's ECE 4703 B-Term 2008 Laboratory documents.

1 Introduction

Many applications require us to fully exercise the capabilities of the microprocessor in order to achieve optimum signal processing performance. For example, as you saw in previous labs, filters of higher order generally exhibited sharper transitions between passband and stopband, as well as greater attentuation in the stopband. However, we are often unable to achieve maximum performance from C code alone, even if we use optimizing C compilers. In order to push the capabilities of the dsPIC33F to the limit, we must resort to assembly language to hand-optimize critical sections of code.

In this laboratory, you will re-implement your FIR and IIR filter algorithms from the previous laboratory assignment in assembly code. You will then optimize your assembly code to use as many coefficients as possible while still maintaining successful real-time operation.

The goals of this laboratory are as follows:

- to introduce the student to techniques for profiling code, such as using the dsPIC33F's hardware timers
- to familiarize the student with assembly programming on the dsPIC33F
- to introduce the student to various techniques to optimize dsPIC33F assembly code

2 Benchmarking

Before you begin to the process of optimization, it is important to have a few tools at your disposal for measuring the performance of your code. Microchip sells several commercial tools (both hardware and software) for profiling your code[1] but these tools are expensive. Fortunately, you can still do a significant amount of performance analysis for free, using only MPLAB and your dsPIC board. In this part of the lab assignment, you will learn ways to measure the amount of time (in clock cycles) required for the dsPIC33F to perform various tasks.

[1] For example, the Microchip Real In-Circuit Emulator (ICE)

2.1 Hardware Timers

The dsPIC33F contains five 16-bit hardware timer peripherals, four of which can be paired up with each other to form two 32-bit timers. These hardware timers are useful because they run in the background while the dsPIC performs other tasks. If we configure a hardware timer to increment every clock cycle and then we use the timer like a stopwatch, we can measure the exact number of clock cycles required to execute a certain section of code. Since the dsPIC33F's instruction clock frequency is 40MHz, a 32-bit timer should give us plenty of time ($40\text{MHz}/(2^{32}) \approx 107$ seconds) to profile any of the DSP algorithms used in this course.

The dsPIC board libraries (i.e., the ones you downloaded from the course website in previous labs) provide some useful functions and macros for initializing and controlling the dsPIC33F's timers. It is recommended that you use these functions and macros because it librates you from having to know all of the esoteric details of timer initialization and control.[2] To use these libraries, first make sure you include the appropriate header file:

```
#include "benchmark.h"
```

Then, call the benchmarking timer initialization function at the start of your main program:

```
int main(void)
{
   ...
   dspic_init();
   benchmark_timer_init();
   ...
   while(1) { ... }
   ...
}
```

The benchmarking timer is modeled after a stopwatch, with the commands "start," "stop," "read," and "clear." To start the timer, use the following macro:

```
benchmark_timer_start();
```

To stop the timer, use the following macro:

```
benchmark_timer_stop();
```

To read the 32-bit value from the timer, you can use the following macro to obtain the most-significant and least-significant 16-bit words from the timer count registers:

```
unsigned int msw, lsw;
benchmark_timer_read(msw,lsw);
```

To clear the stopwatch timer in preparation for timing something else, use the following macro:

```
benchmark_timer_clear();
```

[2] For example, reading the value of a 32-bit timer requires that the operations be performed in a certain order; otherwise incorrect values will be read. These details can be found in Section 11 of the dsPIC Family Reference Manual [2].

You can use the UART and printf() statements (introduced in Laboratory #1) to print out display the results from performing the benchmarking. Note that the printf() function that comes with MPLAB C30 C compiler does not support 32-bit integers[3], which is why the `benchmark_timer_read()` function breaks the 32-bit timer count into most-siginficant and least-significant words. Therefore, in your printf() calls, you will have to print out these two words separately. (For clarity, you might consider printing them in hexadecimal format using the printf()'s "%X" format specifier.)

Another important thing to note is that the number of clock cycles required to perform the instructions-under-test is equal to one less than the count of the benchmarking timer. This is because it takes one instruction cycle to stop the counter (using the `benchmark_timer_stop()` macro), and during this time the timer increments an additional tick. In other words, simply remember to subtract one from the value of the stopwatch in order to get the true profiling time for your section of code.

The reason why the board support libraries use preprocessor macros instead of function calls is that preprocessor macros get textually substituted into the code, whereas function calls generally involve branch instructions. The macros avoid the latency of the branch instructions and therefore guarantee perfectly accurate benchmark count values.

Exercises: Try using the benchmark timer libraries to measure the number of clock cycles required to perform simple arithmetic. How many clock cycles are required to add two 16-bit integers? If the ADD instruction only takes one clock cycle, why do you measure about 3 or 4? (In the next section, you'll discover why). How many clock cycles are required to multiply two 16-bit integers? How about addition and multiplication for long (32-bit) integers? (Why do 32-bit integer operations take significantly longer?) Perform the same experiments for floating-point numbers and explain what you find. Finally, use the benchmarking timer to measure the number of clock cycles required to perform each type of filtering operation from Laboratory #2 and Laboratory #3.

2.2 Disassembly View

Another way to determine the number of clock cycles required to execute a piece of code is to simply analyze the disassembly listing. Although this is not practical for larger, more complicated pieces of code involving loops and function calls, it can be quite useful for simpler pieces of code because it shows exactly what instructions the C compiler generates for each line of C code.

To access the disassembly listing in MPLAB, simply compile your project then select "View → Disassembly Listing" from the main menu. The Disassembly Listing window shows each line of your C code and the corresponding assembly instructions generated by the compiler.

Exercises: Take a look at the disassembly code listing for the 16-bit integer and 32-bit integer additions and multiplications that you profiled earlier. You should now be able to explain the values you measured in the previous section of the lab (for example, why it requires more than a single clock cycle to add two 16-bit integers with C code).

3 C Code Optimization

As you worked through the previous sections of this laboratory, you probably noticed that the C code generated by the compiler is quite unoptimized; it generally takes several assembly instructions

[3]Probably because the dsPIC33F is a 16-bit microprocessor and therefore Microchip deemed 32-bit printfs unnecessary.

to perform even the most basic arithmetic operations. To increase the performance of your code, you can try enabling the C code optimization features of the compiler.

To enable code optimization, open the "Build Options" for your project, and click the "MPLAB C30" tab. Select "Optimization" from the drop-down box. On the little "Optimizatoin Level" diagram, click on the circle next to the "3" to select optimization level 3, which optimizes execution speed (at the expense of code size). You may also try clicking on the checkboxes (for example, the "unroll loops[4]" checkbox) or playing with the other drop-down boxess to select additional options. When you are done, click "OK" to save your settings and close the dialog box. Then recompile your project.

Exercises: Try enabling C compiler optimization in your FIR and IIR filter projects from Laboratory #2 and Laboratory #3. Use the benchmarking timer to measure the performance of your code. How much faster does it run?

You may notice only modest performance increases by enabling the optimizing compiler. This is because you are using the free (student) version of the C compiler, which only has limited support for code optimization, and does not even take advantage of the dsPIC's special DSP instructions! Microchip sells commercial versions of compilers with far more optimization support that do take advantage of the DSP instruction set, but unfortunately these tools can be quite pricy.

4 MPLAB C30 DSP Libraries

Microchip provides an excellent set of DSP libraries with the free student version of the MPLAB C30 C compiler. These libraries are written in assembly langauge for performance but they are designed to be invoked as C function calls, which makes them fairly straightforward to use. These libraries support a variety of DSP-related operations, including functions for performing block[5] FIR processing, block IIR processing, Fast Fourier Transforms (FFTs), and even matrix inversion.

In addition to your typical project configuration, you must do the following things in order to use these libraries in your project.

1. Add a line `#include "dsp.h"` to your *main.c* program. Note that you do not have import this file into your project or set up the path to this file in the build options because this library header file is already in the compiler's path by default.

2. Copy the *fir.s*, *firdelay.s*, and *firinit.s*, files from `C:\ProgramFiles\Microchip\MPLABC30\src\dsp\asm` into your local project source directory. After doing so, be sure to add these files to your project. (The reason for first copying them into your project is simply so that you do not accidentally modify the good copies of them in the "Program Files" directory.)

3. Copy the file *dspcommon.inc* from `C:\ProgramFiles\Microchip\MPLABC30\src\dsp\inc` into the same directory as your MPLAB project (.mcp) file. Note that you only need to copy this file to the MPLAB project directory; you do not need to actually import it into your project.

[4] "Unroll loops" performs "inline expansion," which causes the compiler to use implement each iteration of the loop with separate instructions. This eliminates the overhead required for branching, at the expense of using more program memory.

[5] The term "block" in this context means that the functions can compute the system response for multiple time instants. All of the filtering you have done in labs up until now has been computing the output of the filter for a single time instant (i.e., every time the dsPIC receives a new audio sample). Do not worry; although the DSP libraries can handle convolution for multiple time instants, they still support efficient processing of single time instants.

4. **Extremely Important:** Do not call these libraries directly from within your DCI interrupt service routine. Instead, set a flag in the ISR that indicates that a new sample arrived, and perform the filter processing in your main loop. The system will hang if you call the library filtering function from within your ISR. The reason for this is that the libraries put the dsPIC into a special DSP mode of operation, and the system loses track of what it is doing if you call these functions from within the ISR. Refer to the MPLAB C30 DSP library documentation for more details. This is a minor inconvenience but it is necessary for correct operation.

It is highly recommended that you read through the small amount of documentation that comes with these DSP libraries. This HTML-based documentation can be found in `C:\ProgramFiles\Microchip\MPLABC30\docs\dsp_lib`. For more details, you can take a look at the actual library source code, located in `C:\ProgramFiles\Microchip\MPLABC30\src\dsp`. Microchip also provides example code which illustrates how to use the MPLAB C30 DSP libraries; check the MPLAB C30 compiler directories and the Microchip website. If you follow the provided documentation, you should be able to get these libraries working without too much trouble.

These DSP libraries are quite efficient. For example, processing a single time instant using the block FIR filter requires only about 57+M clock cycles [1], where 57 is the small amount of initialization overhead and M is the number of filter coefficients. See Microchip's website [1] for more data about the performance of these libraries.

Try using the MPLAB C30 libraries in your C programs from Laboratory #3 to perform the filtering. Measure the performance and compare it to your results from Laboratory #3.

5 Assembly Programming

C is a great high-level language for writing embedded programs, but there are limits to even the best optimizing C compilers. Sometimes, in order to squeeze the maximum amount of performance out of a processor with limited resources, it is necessary to write certain sections of the code in assembly language. In this section of the lab, you will be re-implementing your FIR and IIR filtering functions as C-callable assembly instructions. This mixed C and assembly approach allows you to write your high-level logic in C code (which otherwise would be very tedious to code in assembly) and optimize certain critical sections by hand in assembly (for maximum performance).

5.1 The Basics

You are already famililar with ".c" and ".h" files for C programs. The equivalents for assembly code in MPLAB are ".s" and ".inc" files. The ".s" files are assembly source code files (for some reason, MPLAB uses the extension ".s" instead of ".asm") and the ".inc" files are the standard assembly include files (for example, the file `C:/ProgramFiles/Microchip/MPLABASM30Suite/Support/dsPIC33F/inc/p33FJ128GP802.inc` contains all the register definitions for your dsPIC).

The are two ways to call assembly instructions from C code. One way is to use what are called "inline assembly" commands directly in your C code. For example, the compiler directive

```
__asm__("instruction");
```

in your C code will cause the compiler to insert the specified assembly instruction into your program. This is great for adding a few quick lines of assembly code to your program, without needed to create a separate assembly source code file. However, for longer, more involve assembly programs, specifying the `__asm__()` directive in your C code for each assembly instruction becomes tedious.

Therefore, for the purposes of this lab, you will be using the second way to call assembly instructions from C code, which is to create c-callable assembly instructions. This involves first creating a separate ".s" assembly source code file containing your desired assembly function. Then, you can call this assembly function from the C source code. You can even pass arguments back and forth between the C and assembly, provided that you follow the correct mechanism for doing this, as discussed in lecture.

Here are some additional resources that you might find helpful when learning assembly language:

- Section 7.1 of [3].
- Assembly code template files located in `C:/ProgramFiles/Microchip/MPLABASM30Suite/Support/templates/assembly`. Technically, these templates are designed for the dsPIC30F, but much still applies also dsPIC33F.

5.2 The Challenge

Your task is to write two C-callable assembly functions. The first function should implement the FIR filter convolution, which includes all of the required loops, multiplications, and accumulations. The second function should implement a DFII-SOS filter. You may perform all of your high-level logic in C (such as initialization, circular buffering, etc.), but your filtering should be done in assembly code.

Your goal is to find the maximum number of coefficients for each type of filter that can be successfully implemented on the dsPIC33F in real-time. To do this, you must optimize your assembly code as much as possible in order to create filters of the maximum number of coefficients.

How efficient can you make your code? Can you beat the performance of Microchips MPLAB C30 DSP libraries?

5.3 Suggestions

Here are a few suggestions for how to proceed:

1. At the beginning, do not try to optimize your code. Simply try to get your filter working. After it is working, you can start to optimize it, step by step.
2. Some ways you can try to increase code performance include:
    - Use the dsPIC33F's special DSP-optimized instruction set. For example, make use of the DSP MAC instruction.
    - Use the increment syntax to automatically increment array indices on the same instruction that you perform other tasks; this saves you from needing a separate instruction to increment your indices.
    - Rather than compute array indices separately, use array addressing modes to increase performance.
    - To implement your convolution, do not use if-statements within your loops. Instead, split the convolution into two separate loops to eliminate unnecessary test instructions.
3. It highly recommended that you create some flowcharts or diagrams to plan out your assembly code before you begin coding.

4. You may find the benchmarking timer and Disassembly Listing window that you explored earlier in this lab useful when analyzing code.

5. You may also find it useful to set breakpoints with the debugger to analyze your code. You might also decide to try using the MPLAB Simulator as an alternative to the debugger.

6 Specific Items to Discuss in Your Report

Your report should focus on the design of your efficient assembly language code. What was your process for developing the code? What techniques did you employ to increase performance?

You should also provide an analysis of the performance of the FIR and IIR filters before and after assembly optimization. What are the maximum numbers of coefficients were you able to obtain using your optimized FIR and IIR assembly code? What are the maximum numbers of coefficients possible using the unoptimized C code from the previous laboratory? How does this performance compare to the floating-point filter implementations from Laboratory #2?

7 Possibilities for Further Independent Study

There are many more ways to squeeze extra performance out of the dsPIC with clever assembly language programming and full utilization of the dsPIC's features. If you have time and curiousity, you may consider exploring some of these techniques, such as DMA and circular buffer addressing.

References

[1] Microchip. dsPIC DSC DSP Algorithm Library. `http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en023598`.

[2] Microchip. dsPIC33F Family Reference Manual. `http://www.microchip.com/`.

[3] Microchip. Getting Started with dsPIC30F Digital Signal Controllers User's Guide. `http://www.microchip.com/`.

B.5 Laboratory #5

ECE4703 Laboratory Assignment 5 for the dsPIC33F

By D. Cullen and J. DeFeo.

Last updated: 2009-12-22

The goals of this laboratory assignment are:

- to develop an understanding of frame-based digital signal processing,
- to familiarize you with computationally efficient techniques that achieve performance gains not through compiler optimization but, rather, through clever algorithm optimization, and
- to allow you to experimentally verify the theoretically predicted computational requirements of the FFT and DFT.

1 Problem Statement

The FFT is an efficient algorithm for calculating the DFT and is used in a variety of signal processing applications. The most common implementation method for the FFT is the radix-2 decimation-in-time Cooley-Tukey algorithm where a large DFT is broken down into smaller DFTs by splitting the input samples into odd and even sets. The outputs of the smaller DFTs are reassembled in a special way to form the final result and the overall amount of computation required is much less than a direct calculation of the DFT. This assignment has two parts. First, you will implement the basic direct DFT, run it in real-time for various values of N, and profile the execution time to observe the computational trends. The second part of the assignment is to implement the radix-2 decimation-in-time Cooley- Tukey FFT, run it in real-time for various values of N, and profile the execution time to observe the computational trends. All math in this assignment should be performed in fixed point and all of your code will be written in C.

2 Part I: Implementation of the DFT

The first part of this assignment is to implement the direct DFT as a function written in C. Remember that the dsPIC does not have any kind of floating point unit, so you must remember to scale your sample values appropriately to avoid overflow! Another consequence of this is that the trigonometric functions in the math.h library in C must be emulated in software since they use floating point values. Do not use them! Calculate your twiddle factors in Matlab first; scale them to Q15 format (multiply by 32768) and then create an array in your program with these values. You will have to calculate these twiddle factors for each N value of the DFT. Your function must be written generally to allow for any value of N that is an integer power of 2, i.e. N = 2, 4, 8, 16, 32...

A suggested flow diagram for the assignment is shown in Figure 1. It is recommended that you use a double buffering technique here since all processing in this assignment is frame-based (your

Kehtarnavaz textbook [1] discusses a triple-buffering technique in Lab 6, but the third buffer can be eliminated since the DFT/FFT output overwrites the DFT/FFT input). Let the left channel represent the real part of each input sample and let the right channel represent the imaginary part of each input sample. The (complex-valued) incoming samples are placed in one buffer (with 2N float elements) and the last complete frame of (complex-valued) samples is stored in another buffer (with 2N float elements). Your ISR will simply accept new input samples and fill up the incoming buffer; no processing other than incrementing a global index and checking for a full buffer is performed in the ISR. When the incoming buffer is full, you should swap buffers (the incoming buffer becomes the complete buffer, and vice versa) and begin computing the DFT on the complete buffer in your main code. Your DFT code will compute 2N outputs and will overwrite the current complete buffer. Your ISR will also be running while you compute the DFT and will be filling up the new incoming buffer. To run in real-time, you need to complete your computation of the DFT before the next incoming buffer is full. You do not need to send any output signals to the AIC23 codec. Upon completion of the DFT, you should clear any flags that would initiate another DFT and check the status of DIP switch 0. If it has been pressed, you should compute the magnitude of the DFT and store the result in a third buffer. The DFT magnitude buffer will have N float elements. You can then plot the magnitude of the DFT output using Matlab by exporting the magnitude values using the UART tool. After you are certain that your DFT is working correctly, profile the execution of your DFT function with and without compiler optimization for various values of N. Increase N until the DFT can no longer execute in real-time and note the maximum value of N that your DFT can handle in real time. Note that, when DIP switch 0 is not pressed, your code should be continuously filling input buffers and computing a DFT each time the buffer is filled. Your code should not be computing the magnitude of the DFT unless DIP switch 0 is pressed since computing the magnitude of the DFT will take a lot of extra cycles and will prevent you from increasing N to interesting values. The magnitude function in your code is only to allow for verification that the DFT is working correctly; it should not be running during the normal operation of your code. To test your DFT, configure the AIC23 codec for 8kHz sampling rate and connect an interesting test signal like a 500Hz square wave to the line-in jack.

3 Part II: Implementation of the FFT

In this part, you will replace your DFT function from Part I with a Cooley-Tukey radix-2 decimation-in-time FFT function. To receive full credit for this part of the assignment, you are required to support at least N = 2, 4, 8, 16, 32. It may be tricky to write one general function that works for any value of N, hence it is acceptable to write separate functions, e.g. fft2, fft4, fft8, ... If you choose to take this approach, it is also acceptable to have your higher-numbered FFT func- tions call the lower number functions, reassembling the outputs of the lower numbered functions appropriately. Make sure your FFT function is called identically, i.e. has the same function prototype, as the DFT function in Part I. Test and profile your FFT as described in Part I. The output of the FFT should be identical to that of the DFT but, for large enough N, you should see that your FFT executes faster than the DFT.

4 In lab

You will work with the same lab partner as in the prior laboratory assignments. Please contact the instructor if your lab partner has dropped the course or if you have concerns about your lab

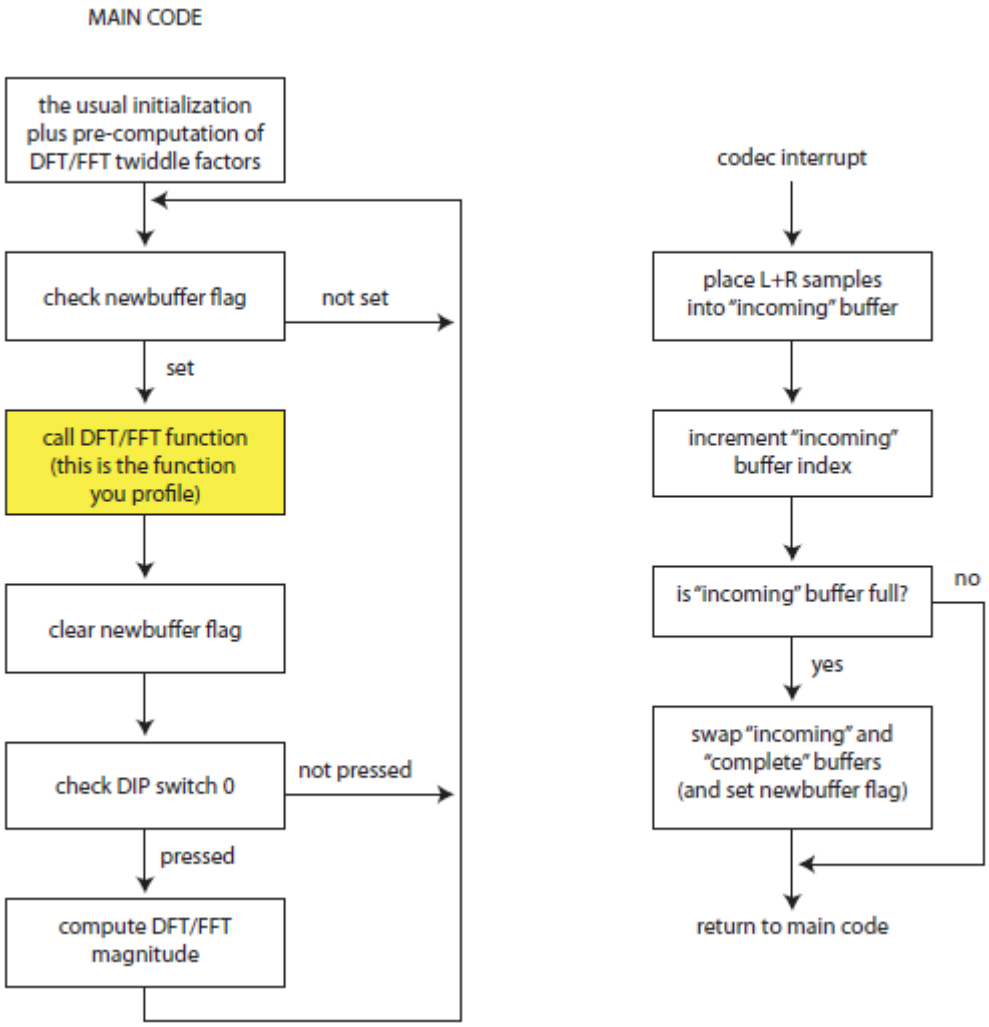


Figure 1: Suggested logical flow of your DFT/FFT code.

partner's performance on the prior assignment.

5 Suggested Procedure for Software Design

1. Begin by at least skimming Chapter 9 and Lab 6 in the Kehtarnavaz text [1]. There are several good examples in here that may give you ideas on how to start the assignment.

2. Make sure your DFT code works before progressing to the FFT. You will need the DFT to check the results of the FFT, so it important that you fully test your DFT code and are confident that it is working correctly. It is recommended that you test your DFT on a buffer with known values and compare the results to Matlab's FFT function. Your DFT function should give exactly the same results as Matlab's FFT function, at least to the precision of the Q15 format.

3. Write and test your FFT code for smaller values of N first. Recall that, at N = 2, the FFT and the DFT perform the same calculations. So you should already have a working fft2 function. When writing your FFT code for higher values of N, you can leverage the code you've already written. For example, fft4 could split the input into odd/even parts and then call fft2 twice, making sure to correctly reassemble the outputs. Similarly, fft8 could be written by calling fft4 twice and correctly reassembling the outputs. You can keep doing this until the FFT no longer runs in real time. Note: It is possible to implement this sort of functionality with recursive function calls (and you are welcome to do so just be careful about your stack and heap sizes), but recursive function calls are an advanced concept and are not required in this assignment.

4. Make sure your FFT function gives exactly the same output as your DFT function. If it doesn't then one (or both) functions are wrong. There are many ways to check your answers: Matlab, Chassaing's or Kehtarnavaz's example code.

6 Code Submission and Specific Items to Discuss in Your Report

You can submit one project for the entire assignment where the DFT or FFT is simply selected by changing a #define at the top of your main C code. Make sure that there are enough comments to make this obvious to the grader. Your code will be tested for correct functionality and profiled by the grader for select values of N to determine if the profiling results in your report are accurate.

At a minimum, you should include the following results and discussion in your report:

1. A single plot showing:

   - The exclusive average cycles of your DFT function with and without compiler optimization for N = 2, 4, 8, 16, 32, ...
   - The exclusive average cycles of your FFT function with and without compiler optimization for N = 2, 4, 8, 16, 32, ...
   - Predicted trends of O(N2) and O(N log2(N)).

   This plot should include distinct line types and a clearly labeled legend. You may want to try generating a semilog or loglog plot to see if that makes your results easier to interpret.

2. Discussion of how the DFT/FFT functions follow (or don't follow) the predicted trends.

3. Analysis and discussion on the largest value of N that can run in real-time in each of the tested cases.

Your report should also discuss any special tricks that you used to implement your FFT code.

References

[1] Nasser Kehtarnavaz. *Real-Time Digital Signal Processing Based on the TMS320C600*. Elsevier, 2005.

B.6 Laboratory #6

ECE4703 Laboratory Assignment 6 for the dsPIC33F

By D. Cullen and J. DeFeo.

Last updated: 2009-12-19

1 Introduction

Vocoders are devices used to shift the pitches of sounds. They are most commonly used with human speech; the word "vocoder" itself is a contraction of the words "voice" and "encoder." Homer Dudley began began the first work with vocoders in 1928 and received a patent for it in 1938 [2]. Over the years, vocoders have been used in music and in films. They have even been used for research in the fields of encryption and speech synthesis. Today, vocoders can be implemented entirely using digital signal processing, which gives considerably more power and flexibility than Homer's first all-analog implementations. For example, "Auto-Tune" is controversial piece of software for adjusting the pitch of a singer's voice, and it relies on some of the pitch-shifting techniques of vocoders. In this lab, you will design and build a simple vocoder to be used in real-time audio processing.

The goals of this laboratory are as follows:

- to show the student an interesting application of real-time DSP
- to demonstrate communications-related techniques such as modulation
- to teach the student how to implement the vocoder DSP techniques efficiently in software

2 Algorithm Development

How does a vocoder change the pitch of an audio signal? In this section, we will develop a simple algorithm for shifting a set of pitches.

One way to shift a set of frequencies is to use modulation, a simple technique that is widely used in communications systems. Modulation is nothing more than a point-by-point time-multiplication of the input signal and a sinusoid. The following Fourier Transform pair shows the cosine modulation frequency shift property:

$$\mathcal{F}\{w(t)cos(2\pi f_c t)\} \Longleftrightarrow \frac{1}{2}(W(f-f_0)+W(f+f_o))$$

Notice that modulation produces a magnitude spectrum consisting of two scaled copies of the original signal, shifted up and down by an amount equal to the frequency of the cosine.

Figure 1(a) shows the magnitude spectrum of a short clip of human speech, sampled at f_s = 16kHz. Let us suppose that we want to shift these pitches up or down by a small amount, say 200Hz. Modulation with a 200Hz cosine wave can give us a 200Hz frequency shift. However, if

you look back at Figure 1(a) and imagine replacing it with two shifted, scaled copies, you will find that these copies overlap, since 200Hz is much less than the bandwidth of either signal. This resulting overlap would result in distortions! Therefore, we cannot modulate directly to the desired frequency.

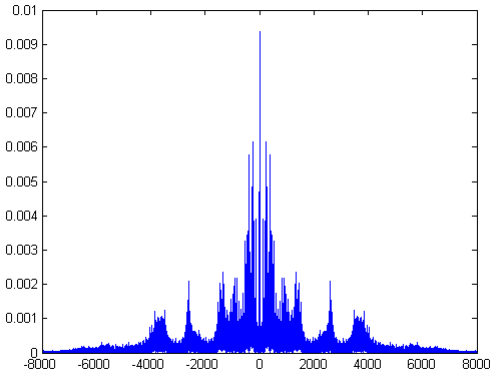
Suppose that we instead perform two modulations: the first modulation is to a frequency greater than the width of the bandwidth, hence no overlap, and the second modulation is used to shift back down to the desired target frequency. (In communications theory, this "temporary" frequency that we modulate to and from is known as the *intermediate frequency (IF)*.) Remember that each modulation produces two copies of the magnitude spectrum: one shifted up by f_0 Hz and one shifted down by f_0 Hz. Lowpass filters (LPF), highpass filters (HPF), and bandpass filters (BPF) may be required to cut out the unwanted frequency bands. This algorithm using modulations and simple filters works well and is the recommended approach for completing this laboratory assignment.

The following algorithm requires only three filters and two modulations; no FFTs are necessary. An example of a signal processed using these steps is shown in Figure 1 parts (a) through (f).

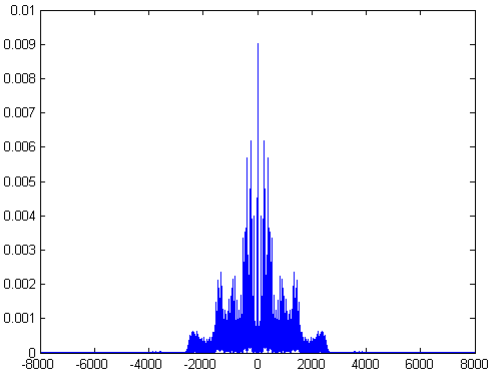
1. LPF to remove high frequency components (and noise) so that we do not get aliasing when we modulate up.
2. Modulate up to an intermediate frequency.
3. HPF to remove the symmetric half so that it does not interfere when we modulate back down.
4. Demodulate back down to baseband.
5. LPF to remove high-frequency components.
6. Perform amplitude scaling (factor of 4) to compensate for attenuation from modulations.

There are a few things to bear in mind when implementing this algorithm:

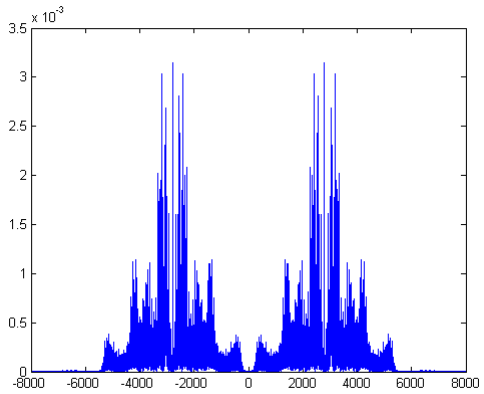
- **Remember the Nyquist Sampling Theorem.** Frequencies greater than half the sampling frequency alias and wrap back around. Be careful when performing your modulations to make sure you do not accidentally cause unwanted frequency wrapping! Selecting a relatively large gives you more frequency bandwidth to play with when you perform your modulation to intermediate frequency. It is recommended that you use a sampling frequency of 44.1kHz for this laboratory assignment. The example plots in this document (i.e., Figure 1(a)) were created using a 16kHz sampling rate, but that choice was completely arbitrary. The dsPIC has plenty of performance to allow real-time vocoder operation at 44.1kHz. Moreover, 44.1kHz instead of 16kHz means greater bandwidth and thus better-sounding audio output.
- **Different pitch shifts require different filter cutoff frequencies.** If you want to shift the pitch of your voice up by 400Hz, you will obviously need to use different modulation frequencies than if you wanted to shift your voice down by 50Hz. However, it is easy to overlook the fact that the cutoffs of the filters must change as well! For example, if you want to shift the pitch down, the highpass filter will need a higher cutoff frequency; otherwise, you will get some negative frequencies (and thus, wraparound) when you demodulate. Working out the frequency shifting on paper and testing in MATLAB will help you to avoid overlooking anything.



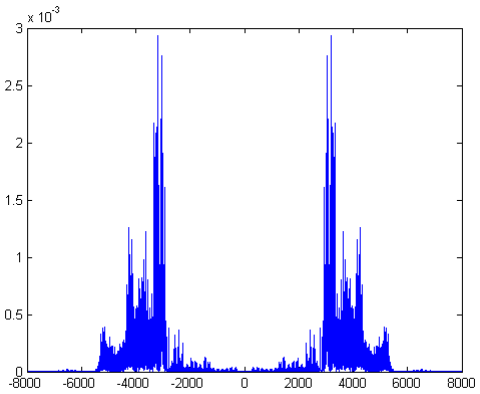
(a) Magnitude spectrum of original audio



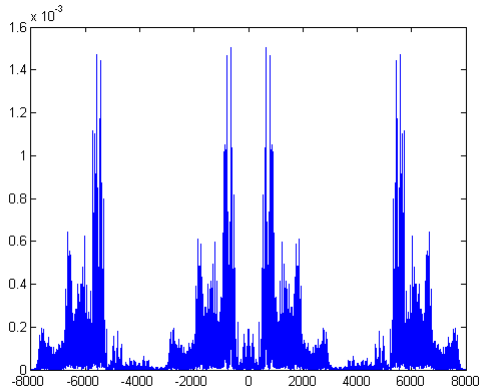
(b) Magnitude spectrum after first LPF



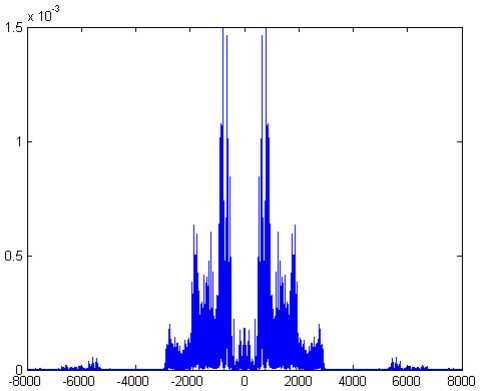
(c) Magnitude spectrum after modulation



(d) Magnitude spectrum after HPF



(e) Magnitude spectrum after demodulation



(f) Magnitude spectrum after second LPF

Figure 1: Example to illustrate vocoder algorithm

3 Vocoder Implementation in MATLAB

Your first task is to implement the suggested vocoder algorithm in MATLAB. It will not take you very long to get it working in MATLAB and doing so will teach you all the subtle nuances of the algorithm, greatly reducing the amount of time you would spend debugging on the dsPIC. You can use the *wavread*, *wavwrite*, and *soundsc* for manipulating and listening to ".wav" files. Try recording your voice in a audio tool such as Audacity [1] and processing it with your MATLAB script. Use the Fast Fourier Transform (FFT) in MATLAB to observe the magnitude spectrum.

Design your filters using the *firpm* MATLAB command, rather than FDATool, so that you can rapidly experiment with different filter cutoff frequencies and numbers of coefficients. What cutoff thresholds sound best for human speech? How much stopband attenuation is needed to prevent aliasing and unwanted distortion? Once you have selected your filter design parameters, you can enter them in FDATool in order to generate the 16-bit signed integer coefficients C header file.

You will soon discover that there are tradeoffs between available bandwidth, modulating to the intermediate frequency, and the cutoff frequencies of your filters. In order to avoid aliasing when you modulate, must perform filtering to limit the frequency ranges of your signals. However, this is at the expense of reduced audio quality, since frequency information is essentially discarded. Experimentation will allow you to select acceptable cutoff frequencies for your filters.

4 Efficient Trigonometric Function Implementation

In order to perform the modulations, you will need to compute the values of cosine functions. You will need to implement the vocoder using fixed-point integer arithmetic; the software floating-point support is too slow. Although the MPLAB C30 C Compiler has floating-point trigonometric libraries for the dsPIC33F, it does not include any fixed-point trigonometric libraries. Therefore, you will need to use another approach in order to compute the cosine values.

One of the most straightforward approaches is to create a look-up table. Since your modulation frequencies will be predetermined and unchanging, you can implement the modulations using look-up tables. You can generate the values of the cosine and store them in a static array in your C code, using the exact same technique that you use to store filter coefficients. You can use MATLAB to generate the floating-point cosine values (from -1.0 to 1.0), scale them to their appropriate fixed-point range (i.e., multiply by 32767), then print them to a text file. (You may find the *fprintf* MATLAB command handy.) If you select the 2*pi*f0*Ts term of the cosine correctly, the modulation step in your vocoder will simply involve selecting the next element from the look-up table and multiplying this value by the value of the input signal.

Be sure to test your look-up table in MATLAB before trying it on the dsPIC. Try performing the modulation using your look-up table and make sure the results are what you would expect.

5 Vocoder Implementation on dsPIC

Now that you have properly designed the filter and modulation components, you are now ready to implement the vocoder on the dsPIC. The FIR filter code will be very similar to your FIR filter code from previous labs. If you properly designed your filters and modulation look-up tables in MATLAB, the implementation on the dsPIC should be quite straightforward.

Here are a few things to bear in mind:

- **Each filter needs its own history buffer.** In previous labs, you probably only used a single filter, or perhaps two (one per each audio channel). In this lab, you will be cascading several modulation and filtering blocks. It is important to remember that each filter needs its own history buffer. You will implement each of these history buffers using the same circular buffering technique that you use to buffer the input samples from the AIC23, except now you will be storing the inputs of each of the filters. For simplicity, we recommend that you use FIR filters. You can achieve decent results with filter orders as low as 20 coefficients each. The dsPIC can easily handle three, 100-coefficient FIR filters.

- **Modulation introduces an attenuation by a factor of 2**. Each copy has half the magnitude amplitude as the orignal copy. Thus, you may have to perform scaling (amplification) so that your output is the same volume as your input.

6 Experimentation

Here are a few things to try once you get your vocoder working:

- Try attaching a microphone to the input of the dsPIC board and speakers to the output. Talk into the microphone and listen to the pitch of your voice shifting in real-time!

- Try playing some music into the vocoder and listening to the output. Does it sound out-of-key? Why?

7 Further Study

7.1 Vocoder Improvements

The vocoder you designed in this laboratory was quite simple. There are many ways to improve the performance and flexiblity of the vocoder. Traditionally, vocoders have used *filter banks*, which consists of an set of bandpass filters, each with a different passband. Using a *filter bank* has the following advantages:

- **Dynamic pitch shifting.** If you want to change the amount of pitch shift in your simple vocoder, you will need to redesign the cutoff frequencies of your filters. However, if you had a bank of bandpass filters, you could change the pitch shift simply by changing the gains of each filter and the frequencies of the oscillators; no filter redesign would be necessary.

- **Reduced sampling frequency requirements.** Recall that we needed to modulate to an intermediate frequency in order to avoid overlapping frequency bands. A bank of bandpass filters would allow us to break the spectrum into narrow frequency bands. If we individually modulated and filtered each narrow frequency band, we could shift the frequencies directly to the target frequency, without worrying about overlap. This would eliminate the need for an intermediate frequency, thereby reducing bandwidth requirements and hence reducing the sampling frequency requirements.

- **Frequency-dependent pitch shifting.** If you tried playing music through your vocoder, you probably noticed that the output sounds out-of-key. The reason for this is that musical pitches do not scare linearly, but your vocoder shifts all of the pitches by the same amount. In

a vocoder consisting of a bank of bandpass filters, different pitches can be shifted by amounts in order to maintain musical pitch relationships.

There are many other more advanced algorithms for implementing vocoders. For example, some algorithms use the "short-time Fourier Transform (STFT)" in order to adjust changing frequencies over time. Another algorithm known as "Time-Domain Harmonic Scaling (TDHS)" is widely used for time-domain processing of frequencies and pitches. These algorithms are outside of the scope of this course, but you might be interested in pursuing them on your own.

There are also many algorithms to efficiently calculate trigonometric functions. In this lab, you were able to implement the cosine modulation using look-up tables because the frequency and sampling frequency were fixed. However, if you instead wanted to dynamically change the modulation frequency, you would need to compute the cosine function in real time. Power series expansion is one way to numerically compute the cosone values. Another widespread method is CORDIC, which stands for "COrdinate Rotation DIgital Computer." If you have time, you can try implementing some of these algorithms.

References

[1] Audacity. `http://audacity.sourceforge.net/`.

[2] Wikipedia. Vocoder. `http://en.wikipedia.org/wiki/Vocoder`.

C Laboratory Code

This appendix provides the example source code for the redesigned laboratory assignments. Note that the code for Labs #2 and #5 is trivial (i.e., it is mostly generic, floating-point C code, very similar to the code already being used in the ECE 4703 lab assignments) and is therefore omitted.

C.1 Lab 1 - main.c

```
1  // Lab1 - "Hello, World!", Blinking LED, UART, AIC23 sample-in/sample-out
2  // Last updated 2009-11-04
3
4  #include <p33FJ128GP802.h>
5  #include <stdio.h>
6  #include <math.h>
7  #include "aic23.h"
8  #include "dspic.h"
9  #include "dspic_adc.h"
10 #include "gpio.h"
11 #include "status_timer.h"
12 #include "swdelay.h"
13 #include "uart.h"
14
15 void __attribute__ ((interrupt, no_auto_psv)) _DCIInterrupt(void);
16 void __attribute__ ((interrupt, no_auto_psv)) _T1Interrupt(void);
17 void update_console(void);
18
19
20 int main (void)
21 {
22   dspic_init();
23   dspic_adc_init();
24   gpio_init();
25   status_timer_init();
26   AIC23_init(AIC23_44);
27   uart_init();
28
29   while(1)
30   {
```

```
31      update_console();
32    }
33
34    return 0;
35  }
36
37  void __attribute__ ((interrupt, no_auto_psv)) _DCIInterrupt(void)
38  {
39    signed int left, right;
40
41    // Get new samples:
42    AIC23_get_samples(&left, &right);
43
44    // Multiply by 2 to compensate for divide-by-two input voltage dividers:
45    // (Note that output has 100 ohm and 47kOhm resistors, so output voltage divider is
          basically just a divide by 1.)
46    left  = left*2;
47    right = right*2;
48
49    // Send out new samples:
50    AIC23_set_samples(&left, &right);
51
52    // Clear the interrupt flag:
53    IFS3bits.DCIIF = 0;
54  }
55
56  void __attribute__ ((interrupt, no_auto_psv)) _T1Interrupt(void)
57  {
58    LED_0 = ~(LED_0);  // Toggle LED_0
59    IFS0bits.T1IF = 0; // clear interrupt flag
60  }
61
62
63  void update_console(void)
64  {
65    // Checks if any new commands have arrived and processes them.
66
67    unsigned char* cmd_buffer_ptr = cmd_get_buffer();
68
69    if (cmd_get_flag())
70    {
```

```
71      switch(cmd_buffer_ptr[0])
72      {
73        case 'h':
74    printf("Hello, world!\r\n");
75    break;
76        case '0':
77    printf("Disabling bypass mode.\r\n");
78    AIC23_bypass_mode_off();
79    break;
80        case '1':
81    printf("Enabling bypass mode.\r\n");
82    AIC23_bypass_mode_on();
83    break;
84        default:
85    printf("Unrecognized command.\r\n");
86    break;
87      }
88    }
89    cmd_clear();
90  }
```

C.2 Lab 3 - main_fir.c

```
1  // Lab 3 - "Fixed-Point FIR and IIR Filters"
2  // main_fir.c - This file is the code FIR part of the lab.
3  // Last updated 2009-12-12
4
5  #include <p33FJ128GP802.h>
6  #include <stdio.h>
7  #include <math.h>
8  #include "aic23.h"
9  #include "dspic.h"
10 #include "dspic_adc.h"
11 #include "gpio.h"
12 #include "status_timer.h"
13 #include "swdelay.h"
14 #include "uart.h"
15 #include "benchmark.h"
16
17 void __attribute__ ((interrupt, no_auto_psv)) _DCIInterrupt(void);
18 void __attribute__ ((interrupt, no_auto_psv)) _T1Interrupt(void);
```

```
19 void update_console(void);
20 void buffer_init(void);
21 long signed int perform_filtering(void);
22
23 #define BUFFER_LENGTH (11)
24 static const signed int b_coefs[BUFFER_LENGTH] = {-8192, 0, 0, 0, 0, 32767, 0, 0, 0, 0,
       -8192}; //{7789, 3420, 3420, 7789};
25 static signed int buffer[BUFFER_LENGTH];
26 volatile unsigned int buffer_index = 0;
27 unsigned int msw, lsw;  // These store benchmarking timer output.
28
29
30 int main (void)
31 {
32   dspic_init();
33   dspic_adc_init();
34   gpio_init();
35   status_timer_init();
36   benchmark_timer_init();
37   AIC23_init(AIC23_44);
38   uart_init();
39
40   while(1)
41   {
42     update_console();
43   }
44
45   return 0;
46 }
47
48 void __attribute__ ((interrupt, no_auto_psv)) _DCIInterrupt(void)
49 {
50   signed int left, right;
51   long signed int output;
52
53   // Get new samples:
54   AIC23_get_samples(&left, &right);
55
56   // Multiply by 2 to compensate for divide-by-two input voltage dividers:
57   // (Note that output has 100 ohm and 47kOhm resistors, so output voltage divider is
         basically just a divide by 1.)
```

```
58    left  = left*2;
59    right = right*2;
60
61    // Perform filtering:
62    buffer[buffer_index] = left;
63    benchmark_timer_clear();
64    benchmark_timer_start();
65    output = perform_filtering();
66    benchmark_timer_stop();
67    benchmark_timer_read(msw, lsw);
68    output = output >> 15;
69    left   = (signed int)(output);
70    buffer_index++;
71    if (buffer_index >= BUFFER_LENGTH)
72      buffer_index = 0;
73
74    // Send out new samples:
75    AIC23_set_samples(&left, &right);
76
77    // Clear the interrupt flag:
78    IFS3bits.DCIIF = 0;
79  }
80
81  void __attribute__ ((interrupt, no_auto_psv)) _T1Interrupt(void)
82  {
83    LED_0 = ~(LED_0);  // Toggle LED_0
84    IFS0bits.T1IF = 0; // clear interrupt flag
85  }
86
87  void update_console(void)
88  {
89    // Checks if any new commands have arrived and processes them.
90
91    unsigned char* cmd_buffer_ptr = cmd_get_buffer();
92
93    if (cmd_get_flag())
94    {
95      switch(cmd_buffer_ptr[0])
96      {
97        case 'h':
98    printf("Hello, world!\r\n");
```

```
99    break;
100       case 'b':
101   printf("BENCHMARK␣RESULTS:\t%04X\t%04X\r\n", msw, lsw);
102   break;
103       case '0':
104   printf("Disabling␣bypass␣mode.\r\n");
105   AIC23_bypass_mode_off();
106   break;
107       case '1':
108   printf("Enabling␣bypass␣mode.\r\n");
109   AIC23_bypass_mode_on();
110   break;
111       default:
112   printf("Unrecognized␣command.\r\n");
113   break;
114     }
115   }
116   cmd_clear();
117 }
118
119 void buffer_init(void)
120 {
121   // Initializes sample buffer with zeros.
122   // Also initializes buffer_index to zero.
123   unsigned int i;
124   for (i=0; i<BUFFER_LENGTH; i++)
125     buffer[i] = 0;
126   buffer_index = 0;
127 }
128
129 long signed int perform_filtering(void)
130 {
131   // Filters the data in the buffer and returns the filtered result.
132   // Takes no arguments because it uses global variables for everything.
133   // Use two for-loops for the convolution, rather than one for-loop
134   // with an if-statement inside, because the two for-loop convolution
135   // is MUCH faster.
136
137   signed int k = 0;            // k must be signed integer since k-- (when k==0) yields -1.
138   signed int j = buffer_index; // grab a local copy for faster access
139   signed int i = 0;
```

```
140   long signed int accumulator = 0; // must initialize accumulator to zero!
141   long signed int product = 0;
142   long signed int tmp1, tmp2;
143
144   // figured it out. it was performing 16bit-by-16bit multiplication, getting a 16bit
          result,
145   // then transferring into 32bit product. Rather than performing 16bit*16bit=32bits.
146   // So I had to use tmp1 and tmp2 to get it working.
147
148   for(k=j; k >= 0; k--)
149   {
150     //accumulator += b_coefs[i++]*buffer[k];
151     //product = b_coefs[i++]*buffer[k];
152     tmp1 = b_coefs[i++]; tmp2 = buffer[k];
153     product = tmp1*tmp2;
154     //product = product >> 1;  // Scale the product to prevent overflow. (more like reduce
          the chance of it)
155     accumulator += product;
156   }
157   for (k=BUFFER_LENGTH-1; k>j; k--)
158   {
159     //accumulator += b_coefs[i++]*buffer[k];
160     //product = b_coefs[i++]*buffer[k];
161     tmp1 = b_coefs[i++]; tmp2 = buffer[k];
162     product = tmp1*tmp2;
163     //product = product >> 1;  // Scale the product to prevent overflow. (more like reduce
          the chance of it)
164     accumulator += product;
165   }
166
167   return accumulator;
168 }
```

C.3 Lab 4 - main.c

```
1  // Lab 4 - Assembly code optimization
2  // main.c - This is the top-level C code.
3  //          This code makes calls to Microchip's free (included with
4  //          academic version of MPLAB C30 C compiler) assembly-optimized
5  //          DSP libraries for efficient filtering.
6  // Last updated 2009-12-12
```

```
7
8  #include <p33FJ128GP802.h>
9  #include <stdio.h>
10 #include <math.h>
11 #include "aic23.h"
12 #include "dspic.h"
13 #include "dspic_adc.h"
14 #include "gpio.h"
15 #include "status_timer.h"
16 #include "swdelay.h"
17 #include "uart.h"
18 #include "benchmark.h"
19 #include "dsp.h"  // Include DSP library functions. C:\Program Files\Microchip\MPLAB
       C30\src\dsp\include\dsp.h (though you don't have to manually import this because it's in
        the search path)
20
21 void __attribute__ ((interrupt, no_auto_psv)) _DCIInterrupt(void);
22 void __attribute__ ((interrupt, no_auto_psv)) _T1Interrupt(void);
23 void perform_filtering(void);
24 void update_console(void);
25 void xbuf_init(void);
26
27 // Store values from benchmarking timer:
28 unsigned int msw, lsw;
29
30 // Length of sample input buffer. Length must be mod2 for circular buffering.
31 #define XBUF_LEN (16)
32
33 // define input sample circular buffer:
34 fractional xbuf[XBUF_LEN]; // __attribute__((space(dma)));
35
36 // used for handshaking between _DCIInterrupt() and main()
37 volatile unsigned int gotsample = 0;
38 fractional newsample = 0;
39
40 // buffer index
41 volatile unsigned int xbuf_index = 0;
42
43 // Number of FIR coefficients
44 #define FIR_M (11)
45
```

```
46 // FIR Coeffecient Buffer.
47 // You may allocate filter coefficients in either program space or X-Data memory.
48 // (We have chosen to use X-data memory.)
49 fractional fir_b[FIR_M] __attribute__((space(xmemory),far))
50   = {-8192, 0, 0, 0, 0, 32767, 0, 0, 0, 0, -8192};
51
52 // FIR Delay Buffer.
53 // Basically, Microchip's DSP libraries implement an FIR filter
54 // as a tapped delay line. It copies each element from your input
55 // buffer into the tapped delay line, then performs the MACs between
56 // the delay buffer and the coefficients. I'm not sure why they chose
57 // to implement it this way, but they did, so we must create this buffer.
58 // This delay buffer must be allocated in the Y-Data memory.
59 fractional fir_z[FIR_M] __attribute__((space(ymemory),far));
60
61 // Filter struture:
62 FIRStruct FIRfilter;
63
64 // Output:
65 fractional output;
66
67 int main (void)
68 {
69   dspic_init();
70   dspic_adc_init();
71   gpio_init();
72   status_timer_init();
73   benchmark_timer_init();
74   AIC23_init(AIC23_44);
75   uart_init();
76
77   // Initialize sample buffer:
78   xbuf_init();
79
80   // Initialize the filter:
81   FIRStructInit(&FIRfilter, FIR_M, fir_b, 0xFF00, fir_z);  // Initializes filter structure
         (copies in pointers and automatically computes end addresses). Note that 0xFF00
         indicates that coefficients are defined in data space.
82   FIRDelayInit(&FIRfilter);                                 // They recommend calling this
         function to initialize tapped delay line with zeros.
83
```

```
84    while(1)
85    {
86      update_console(); // temporarily comment out for extra filter speed performance
87
88      //WORD OF CAUTION. MUST SET A "GOT NEW SAMPLE" FLAG IN INTERRUPT, THEN CALL THE
            FILTERING COMMAND FROM YOUR MAIN PROGRAM. REASON FOR THIS IS THAT THE MICROCHIP DSP
            LIBRARIES PUT THE DSPIC INTO A SPECIAL MODE AND DO SOME FIDDLING WITH THE REGISTERS.
             IT GETS SCREWED UP IF YOU CALL FIR FROM WITHIN THE INTERRUPT. SEE DOCUMENTATION
            THAT COMES WITH THE LIBRARIES FOR MORE DETAILS ABOUT REGISTERS AFFECTED BY THE
            FIR() FUNCTION.
89      // I think maybe it modifies some registers so you can't do it in DCI directly...
90      if (gotsample != 0)
91      {
92        xbuf[xbuf_index] = newsample;
93
94        benchmark_timer_clear();
95        benchmark_timer_start();
96        // ----- START SIGNAL PROCESSING -----
97        FIR(1, &output, &xbuf[xbuf_index], &FIRfilter); // Perform filtering. Get 1 sample's
              worth, not an entire convolution (which is why output buffer is a single element)
98        // ----- END SIGNAL PROCESSING -----
99        benchmark_timer_stop();
100       benchmark_timer_read(msw, lsw);
101
102       xbuf_index++;
103       if (xbuf_index >= XBUF_LEN)
104   xbuf_index = 0;
105       gotsample = 0;
106     }
107   }
108
109   return 0;
110 }
111
112 void __attribute__ ((interrupt, no_auto_psv)) _DCIInterrupt(void)
113 {
114   signed int left, right;
115   //fractional output;
116
117   // Get new samples:
118   AIC23_get_samples(&left, &right);
```

```
119
120   // Multiply by 2 to compensate for divide-by-two input voltage dividers:
121   // (Note that output has 100 ohm and 47kOhm resistors, so output voltage divider is
          basically just a divide by 1.)
122   left  = left*2;
123   right = right*2;
124
125   // Store new sample for filtering:
126   newsample = left;
127
128   // Get newest output to send out:
129   left = (signed int)(output);
130   gotsample = 1; // set flag
131
132   // Send out new samples:
133   AIC23_set_samples(&left, &right);
134
135   // Clear the interrupt flag:
136   IFS3bits.DCIIF = 0;
137 }
138
139 void __attribute__ ((interrupt, no_auto_psv)) _T1Interrupt(void)
140 {
141   LED_0 = ~(LED_0);  // Toggle LED_0
142   IFS0bits.T1IF = 0; // clear interrupt flag
143 }
144
145 void update_console(void)
146 {
147   // Checks if any new commands have arrived and processes them.
148
149   unsigned char* cmd_buffer_ptr = cmd_get_buffer();
150
151   if (cmd_get_flag())
152   {
153     switch(cmd_buffer_ptr[0])
154     {
155       case 'h':
156   printf("Hello, world!\r\n");
157   break;
158       case 'b':
```

```
159   printf("BENCHMARK␣RESULTS:\t%04X\t%04X\r\n", msw, lsw);
160   break;
161       case '0':
162   printf("Disabling␣bypass␣mode.\r\n");
163   AIC23_bypass_mode_off();
164   break;
165       case '1':
166   printf("Enabling␣bypass␣mode.\r\n");
167   AIC23_bypass_mode_on();
168   break;
169       default:
170   printf("Unrecognized␣command.\r\n");
171   break;
172     }
173   }
174   cmd_clear();
175 }
176
177 void xbuf_init(void)
178 {
179   // Initializes sample buffer with zeros.
180   // Also initializes buffer index to zero.
181   unsigned int i;
182   for (i=0; i<XBUF_LEN; i++)
183     xbuf[i] = 0;
184   xbuf_index = 0;
185 }
```

C.4 Lab 6 - main.c

```
1  // Lab 4 - Assembly code optimization
2  // main.c - This is the top-level C code.
3  //          This code makes calls to Microchip's free (included with
4  //          academic version of MPLAB C30 C compiler) assembly-optimized
5  //          DSP libraries for efficient filtering.
6  // Last updated 2009-12-12
7
8  #include <p33FJ128GP802.h>
9  #include <stdio.h>
10 #include <math.h>
11 #include "aic23.h"
```

```
12 #include "dspic.h"
13 #include "dspic_adc.h"
14 #include "gpio.h"
15 #include "status_timer.h"
16 #include "swdelay.h"
17 #include "uart.h"
18 #include "benchmark.h"
19 #include "dsp.h"  // Include DSP library functions. C:\Program Files\Microchip\MPLAB
       C30\src\dsp\include\dsp.h (though you don't have to manually import this because it's in
        the search path)
20
21 void __attribute__ ((interrupt, no_auto_psv)) _DCIInterrupt(void);
22 void __attribute__ ((interrupt, no_auto_psv)) _T1Interrupt(void);
23 void perform_filtering(void);
24 void update_console(void);
25 void xbuf_init(void);
26
27 unsigned int msw, lsw;  // these store values from benchmarking timer
28
29 volatile unsigned int gotsample = 0;  // used for handshaking between _DCIInterrupt() and
       main()
30 fractional newsample = 0;             // used for passing data from _DCIInterrupt() to
       main()
31 fractional output;                    // used for passing data from main() to
       _DCIInterrupt()
32
33 #define XBUF_LEN (256)
34 fractional xbuf[XBUF_LEN];
35 volatile unsigned int xbuf_index = 0;
36
37 // Need to store additional history buffers, since each filter has its own input sample
       hsitory.
38 fractional hpf_inbuf[XBUF_LEN];
39 fractional lpf2_inbuf[XBUF_LEN];
40
41 #define MODLUTLEN (280)
42 volatile unsigned int modlutindex = 0;
43 static const signed int modlut[MODLUTLEN] = { // modulate at 2.8kHz
44   32767,14876,-19261,-32365,-10126,23170,31164,5126,-26510,-29197,-1,29196,26509,-5127,-31165,
45   -23171,10125,32364,19260,-14877,-32768,-14877,19260,32364,10125,-23171,-31165,-5127,26509,29196,
46   0,-29197,-26510,5126,31164,23170,-10126,-32365,-19261,14876,32767,14876,-19261,-32365,-10126,23170,
```

```
  31164,5126,-26510,-29197,-1,29196,26509,-5127,-31165,-23171,10125,32364,19260,-14877,-32768,-14877,
  19260,32364,10125,-23171,-31165,-5127,26509,29196,0,-29197,-26510,5126,31164,23170,-10126,-32365,
  -19261,14876,32767,14876,-19261,-32365,-10126,23170,31164,5126,-26510,-29197,0,29196,26509,-5127,
  -31165,-23171,10125,32364,19260,-14877,-32768,-14877,19260,32364,10125,-23171,-31165,-5127,26509,
  29196,0,-29197,-26510,5126,31164,23170,-10126,-32365,-19261,14876,32767,14876,-19261,-32365,-10126,
  23170,31164,5126,-26510,-29197,-1,29196,26509,-5127,-31165,-23171,10125,32364,19260,-14877,-32768,
  -14877,19260,32364,10125,-23171,-31165,-5127,26509,29196,0,-29197,-26510,5126,31164,23170,-10126,
  -32365,-19261,14876,32767,14876,-19261,-32365,-10126,23170,31164,5126,-26510,-29197,-1,29196,26509,
  -5127,-31165,-23171,10125,32364,19260,-14877,-32768,-14877,19260,32364,10125,-23171,-31165,-5127,
  26509,29196,0,-29197,-26510,5126,31164,23170,-10126,-32365,-19261,14876,32767,14876,-19261,-32365,
  -10126,23170,31164,5126,-26510,-29197,0,29196,26509,-5127,-31165,-23171,10125,32364,19260,-14877,
  -32768,-14877,19260,32364,10125,-23171,-31165,-5127,26509,29196,-1,-29197,-26510,5126,31164,23170,
  -10126,-32365,-19261,14876,32767,14876,-19261,-32365,-10126,23170,31164,5126,-26510,-29197,-1,29196,
  26509,-5127,-31165,-23171,10125,32364,19260,-14877,-32768,-14877,19260,32364,10125,-23171,-31165,
  -5127,26509,29196,0,-29197,-26510,5126,31164,23170,-10126,-32365,-19261,14876};

#define DEMODLUTLEN (60)
volatile unsigned int demodlutindex = 0;
static const signed int demodlut[DEMODLUTLEN] = { // demodulate at 2.4kHz
  32767,19260,-10126,-31165,-26510,-1,26509,31164,10125,-19261,-32768,-19261,10125,31164,
  26509,0,-26510,-31165,-10126,19260,32767,19260,-10126,-31165,-26510,0,26509,31164,10125,
  -19261,-32768,-19261,10125,31164,26509,0,-26510,-31165,-10126,19260,32767,19260,-10126,
  -31165,-26510,-1,26509,31164,10125,-19261,-32768,-19261,10125,31164,26509,0,-26510,-31165,
  -10126,19260};

#define LPF1_M (101)
fractional lpf1_b[LPF1_M] __attribute__((space(xmemory),far)) = {
  -313,   -517,    207,      3,    172,      3,    -91,   -155,    -58,
  91,    176,    101,    -73,   -197,   -149,     43,    213,    201,
  0,   -222,   -258,    -57,    218,    318,    130,   -199,   -379,
  -222,    162,    439,    335,    -98,   -496,   -476,      0,    548,
  657,    150,   -594,   -904,   -390,    632,   1284,    819,   -659,
  -2034,  -1837,    677,   4814,   8671,  10240,   8671,   4814,    677,
  -1837,  -2034,   -659,    819,   1284,    632,   -390,   -904,   -594,
  150,    657,    548,      0,   -476,   -496,    -98,    335,    439,
  162,   -222,   -379,   -199,    130,    318,    218,    -57,   -258,
  -222,      0,    201,    213,     43,   -149,   -197,    -73,    101,
  176,     91,    -58,   -155,    -91,      3,    172,      3,    207,
  -517,   -313};
fractional lpf1_z[LPF1_M] __attribute__((space(ymemory),far));
FIRStruct lpf1filter;
```

```
88
89  #define HPF_M (101)
90  fractional hpf_b[HPF_M] __attribute__((space(xmemory),far)) = {
91    179,   2075,   -103,   -128,   -191,    -20,    182,    185,    -27,
92    -219,   -169,     81,    251,    141,   -140,   -277,    -99,    205,
93    292,     44,   -274,   -295,     28,    344,    283,   -117,   -414,
94    -252,    225,    482,    194,   -357,   -547,   -106,    519,    605,
95    -29,   -727,   -655,    238,   1017,    696,   -589,  -1485,   -726,
96    1321,   2533,    745,  -4095,  -9384,  21095,  -9384,  -4095,    745,
97    2533,   1321,   -726,  -1485,   -589,    696,   1017,    238,   -655,
98    -727,    -29,    605,    519,   -106,   -547,   -357,    194,    482,
99    225,   -252,   -414,   -117,    283,    344,     28,   -295,   -274,
100   44,    292,    205,    -99,   -277,   -140,    141,    251,     81,
101   -169,   -219,    -27,    185,    182,    -20,   -191,   -128,   -103,
102   2075,    179};
103 fractional hpf_z[HPF_M] __attribute__((space(ymemory),far));
104 FIRStruct hpffilter;
105
106 #define LPF2_M (101)
107 fractional lpf2_b[LPF2_M] __attribute__((space(xmemory),far)) = {
108       256,   -535,   -293,    -85,     85,    104,    -27,   -148,   -104,
109        70,    174,     76,   -124,   -193,    -33,    183,    197,    -28,
110      -241,   -181,    107,    293,    140,   -200,   -331,    -70,    306,
111       349,    -33,   -419,   -335,    174,    535,    280,   -360,   -646,
112      -166,    605,    748,    -36,   -941,   -835,    397,   1463,    901,
113     -1152,  -2566,   -942,   3962,   9470,  11878,   9470,   3962,   -942,
114     -2566,  -1152,    901,   1463,    397,   -835,   -941,    -36,    748,
115       605,   -166,   -646,   -360,    280,    535,    174,   -335,   -419,
116       -33,    349,    306,    -70,   -331,   -200,    140,    293,    107,
117      -181,   -241,    -28,    197,    183,    -33,   -193,   -124,     76,
118       174,     70,   -104,   -148,    -27,    104,     85,    -85,   -293,
119      -535,    256};
120 fractional lpf2_z[LPF2_M] __attribute__((space(ymemory),far));
121 FIRStruct lpf2filter;
122
123 int main (void)
124 {
125   dspic_init();
126   dspic_adc_init();
127   gpio_init();
128   status_timer_init();
```

```
129   benchmark_timer_init();
130   AIC23_init(AIC23_16);
131   uart_init();
132
133   // Initialize sample buffer:
134   xbuf_init();
135
136   // Initialize the filters:
137   FIRStructInit(&lpf1filter, LPF1_M, lpf1_b, 0xFF00, lpf1_z);
138   FIRDelayInit(&lpf1filter);
139   FIRStructInit(&hpffilter, HPF_M, hpf_b, 0xFF00, hpf_z);
140   FIRDelayInit(&hpffilter);
141   FIRStructInit(&lpf2filter, LPF2_M, lpf2_b, 0xFF00, lpf2_z);
142   FIRDelayInit(&lpf2filter);
143
144   // Some variables to use for multiplications:
145   long signed int tmp1, tmp2, product;
146
147   while(1)
148   {
149     update_console();
150
151     if (gotsample != 0)
152     {
153       xbuf[xbuf_index] = newsample;
154
155       benchmark_timer_clear();
156       benchmark_timer_start();
157       // ----- START SIGNAL PROCESSING -----
158       // LPF
159       FIR(1, &output, &xbuf[xbuf_index], &lpf1filter);
160       // Modulate. (Use long ints b/c to avoid problems because I haven't figured out better
              way yet.)
161       tmp1 = output; tmp2 = modlut[modlutindex];
162       product = tmp1*tmp2; product = product >> 14; // only shift by 14, not 15, to multiply
              by 2 to compensate for modulation attenuation.
163       output = (signed int)(product);
164       // HPF
165       hpf_inbuf[xbuf_index] = output;
166       FIR(1, &output, &hpf_inbuf[xbuf_index], &hpffilter);
```

```
167         // Demodulate. (Use long ints b/c to avoid problems because I haven't figured out
                better way yet.)
168         tmp1 = output; tmp2 = demodlut[demodlutindex];
169         product = tmp1*tmp2; product = product >> 14; // only shift by 14, not 15, to multiply
                 by 2 to compensate for modulation attenuation.
170         output = (signed int)(product);
171         // LPF
172         lpf2_inbuf[xbuf_index] = output;
173         FIR(1, &output, &lpf2_inbuf[xbuf_index], &lpf2filter);
174         // ----- END SIGNAL PROCESSING -----
175         benchmark_timer_stop();
176         benchmark_timer_read(msw, lsw);
177
178         xbuf_index++;
179         if (xbuf_index >= XBUF_LEN)
180     xbuf_index = 0;
181         gotsample = 0;
182
183         modlutindex++;
184         if (modlutindex >= MODLUTLEN)
185     modlutindex = 0;
186         demodlutindex++;
187         if (demodlutindex >= DEMODLUTLEN)
188     demodlutindex = 0;
189       }
190     }
191
192     return 0;
193 }
194
195 void __attribute__ ((interrupt, no_auto_psv)) _DCIInterrupt(void)
196 {
197     signed int left, right;
198     //fractional output;
199
200     // Get new samples:
201     AIC23_get_samples(&left, &right);
202
203     // Multiply by 2 to compensate for divide-by-two input voltage dividers:
204     // (Note that output has 100 ohm and 47kOhm resistors, so output voltage divider is
            basically just a divide by 1.)
```

```
205    //left  = left*2;
206    //right = right*2;
207
208    // Store new sample for filtering:
209    newsample = left;
210
211    // Get newest output to send out:
212    left = (signed int)(output);
213    gotsample = 1; // set flag
214
215    // Send out new samples:
216    AIC23_set_samples(&left, &right);
217
218    // Clear the interrupt flag:
219    IFS3bits.DCIIF = 0;
220  }
221
222  void __attribute__ ((interrupt, no_auto_psv)) _T1Interrupt(void)
223  {
224    LED_0 = ~(LED_0);  // Toggle LED_0
225    IFS0bits.T1IF = 0; // clear interrupt flag
226  }
227
228  void update_console(void)
229  {
230    // Checks if any new commands have arrived and processes them.
231
232    unsigned char* cmd_buffer_ptr = cmd_get_buffer();
233
234    if (cmd_get_flag())
235    {
236      switch(cmd_buffer_ptr[0])
237      {
238        case 'h':
239    printf("Hello, world!\r\n");
240    break;
241        case 'b':
242    printf("BENCHMARK RESULTS:\t%04X\t%04X\r\n", msw, lsw);
243    break;
244        case '0':
245    printf("Disabling bypass mode.\r\n");
```

```
246     AIC23_bypass_mode_off();
247     break;
248         case '1':
249     printf("Enabling␣bypass␣mode.\r\n");
250     AIC23_bypass_mode_on();
251     break;
252         default:
253     printf("Unrecognized␣command.\r\n");
254     break;
255       }
256     }
257     cmd_clear();
258 }
259 
260 void xbuf_init(void)
261 {
262   // Initializes sample buffer with zeros.
263   // Also initializes buffer index to zero.
264   unsigned int i;
265   for (i=0; i<XBUF_LEN; i++)
266     xbuf[i] = 0;
267   xbuf_index = 0;
268 }
```

D Detailed Guide

Detailed Guide

By D.Cullen & J.DeFeo

Last updated: 2009-12-21

1 Introduction

This guide is intended to be used as a reference manual. Students may find this guide useful if they wish to learn more about particular aspects of the dsPIC33F, the audio processing board, or the board's software support libraries. This "Detailed Guide" contains a great deal of knowledge that is useful but too esoteric to put in the lab procedure documents or the final project report. Other names for this guide include, "The dsPIC Lab Companion Guide" and the "More-than-you-ever-wanted-to-know-about-the-dsPIC guide."

This guide is organized into sections according to the major systems of the dsPIC33F, such as the UART, the clock/PLL/oscillator setup, SPI, DCI/I2S, timer peripherals, etc. There are also a few other sections that give miscellaneous tips about the MPLAB IDE and embedded programming in general.

2 Overview of the dsPIC33F

This section highlights some of the important features of the dsPIC33F mentioned in the datasheet.

2.1 CPU Features

The dsPIC33F is designed with a Harvard architecture.

- Datasheet p. 25 "All instructions execute in a single cycle, with the exception of instructions that change the program flow, the double-word move (MOV.D) instruction, and the table instructions. Overhead-free program loop constructs are supported using the DO and REPEAT instructions, both of which are interruptible at any time."
- Datasheet p.25: There are sixteen 16-bit working registers, W0 thru W15. W15 acts as the stack pointer for interrupts and calls.

- Datasheet p.25: Two types of instructions: MCU and DSP.
- Datasheet p.25: Instruction set designed for DSP efficiency. For most instructions, you can do reads from program memory, data memory, working register, and write to a data register all in the same clock cycle, so you can essentially do "C = A + B" types of data operations.
- Datasheet p.25: "Overhead-free circular buffers (Modulo Addressing mode) are supported in both X and Y address spaces." There is also a bit-reversed addressing mode for efficient FFT computations!

2.2 Addressing

- Datasheet p.38: "The program memory space is organized in word-addressable blocks. Although it is treated as 24 bits wide, it is more appropriate to think of each address of the program memory as a lower and upper word, with the upper byte of the upper word being unimplemented. The lower word always has an even address, while the upper word has an odd address (Figure 4-2)."

3 Clock Configuration and PLL

This section explains how configure the clock and PLL for the dsPIC33F. It also describes the clock source the AIC23 audio codec.

3.1 dsPIC33F Clock Sources

The dsPIC33F supports several options for clocking. It has an internal RC oscillator which can be configured to clock the dsPIC at speeds up to 40MIPs. The dsPIC33F also supports an external clock source. Another common option is to connect an external oscillator crystal across the dsPIC's OSC1 and OSC2 pins. The dsPIC33F supports a wide range of crystal frequency values and uses an internal PLL to adjust the instruction clock frequency up to 40MHz. In addition, the dsPIC33F supports a secondary oscillator crystal, generally intended for use with watch crystal for a real-time clock applications.

- We have a separate clock crystal for each the AIC23 and the dsPIC.
- Crystal accuracy is given in terms of ppm (parts per million) of the nominal frequency they are guaranteed to run at when under their rated capacitor load (C_L).

- You'll notice that the dsPIC33FJ128GP802 datasheet p.141 has Equations 9-2 and 9-3 for determining how to configure the PLL. However, we recommend looking at Equations 7-3 and 7-4 from Section 7.7 of the dsPIC33F Family Reference Manual, since the latter set of equations is more complete (i.e. shows how to get M, N1, and N2 from the bits stored in the registers). This expression is given in Equation (1).

$$FOSC = FIN*(M/[N1*N2]) = FIN*([PLLDIV+2]/[(PLLPRE+1)*2*(PLLPOST+1)]) \quad (1)$$

where

$$FCY = FOSC/2$$

To get the system to run at 40MIPs, see the code in our board software support libraries.

- The block diagrams in the Family Reference Manual and in the datasheet are useful, but it can take some time to understand them because they break the whole PLL system into several pieces. We have therefore redrawn the entire dsPIC33F PLL and clock system as a single block diagram, as shown in Figure 1.

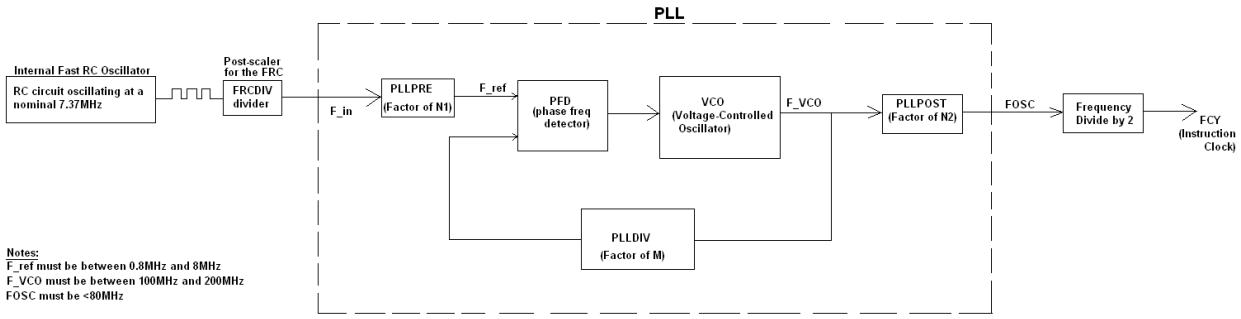


Figure 1: Block Diagram of dsPIC33F Clocking System

3.2 Fast Internal RC Oscillator

The fast internal RC oscillator (FRC) is easy to configure and allows operation up to 40MIPs. However, the drawback is that it varies significantly over temperature. According to the dsPIC33FJ128GP802 datasheet (p.330, Table 30-19 "AC Characteristics - Internal RC accuracy"), the FRC has ±2% accuracy a 7.3728MHz at 3.3V for -40°C ≤ TA ≤ +85°C. According to Section 39.6, p.18 of the Family Reference Manual, the internal fast RC oscillator oscillates at a nominal 7.37MHz. Applications can tune this oscillator from -12% to +11.625% of the nominal frequency, in 30kHz steps, using the FRC oscillator tuning register. Oscillator value is expected to remain within ±2% of the tuned value over temperature and voltage variations of the device.

$$2\% \; of \; 7.37MHz = 147.4kHz$$

So the range is 7.2226MHz to 7.5174MHz Thus, the frequency can vary over a range of about 300kHz over standard temperature (-40°C to +85 °C) at standard supply voltage (+3.3V)! (To put this in perspective, the range of human hearing is typically 20Hz to 20kHz.)

4 Watchdog Timer

Many embedded microcontrollers support watchdog timers, which are used to automatically reset the device if there is a hardware failure or a severe software error. The dsPIC33F is no exception; it supports a watchdog timer (WDT) which can be configured to operate in a variety of modes and with various user-selectable timeout intervals.

However, for the purposes of our laboratory projects, we have no need for the watchdog timer. In order to avoid the trouble of "kicking the watchdog" every once in a while, we simply disable the watchdog timer. This is handled in our board's software libraries and requires two steps: first, the configuration registers must be set to allow for software configuration of the WDT; and second, the watchdog timer must be disabled in the code by writing to the correct software registers. The former is handled in "config_regs.c" of the board libraries and the latter is handled in the dspic_init() function in "dspic.c."

For more information, refer to the datahseet [3] and and Section 9 of the Family Reference manual [2].

5 UART

5.1 Overview

The dsPIC33FJ128GP802 supports two hardware UART peripherals. The PICkit2 programmer can be used for convenient communication between a PC and the UART interface of the dsPIC. The user simply initializes the UART peripheral in software on the dsPIC to use reuse the programming pins as UART pins. The PICkit2 programmer comes with a piece of PC software called the "UART Tool," which provides a simple UART termainal interface. In the dsPIC code, the user can use standard C library functions such as *printf()* to send ASCII text to the PC. The user can also set up interrupts on the dsPIC to receive ASCII text sent from the "UART Tool" on the PC.

This chapter gives additional background behind how this all works and how to correctly configure the UART on the dsPIC33FJGP802.

5.2 How to configure the UART

This section explains how to set up the UART peripheral by writing to software registers in the dsPIC33F code. Since the initialization process is somewhat involved, it is recommended that you use the board infrastructure software libraries, which provide convenient functions to perform all of these tasks for you. However, if you need to configure the UART differently, or if you simply wish to learn more about the configuration process, keep reading!

For the specifics concerning the UART peripheral software configuration, please refer to the the board software library source code and its accompanying documentation. In addition, be sure to refer to the the dsPIC33FJ128GP802 datasheet [3] and Section 17 of the dsPIC33F Family Reference Manual [2]. The final MQP project report [1] also contains more information concerning the UART and the PICkit2.

The basic steps for setting up the UART are as follows:

1. **First decide which UART to use.** The dsPIC33F supports two UART peripherals. We recommend that you use UART1, since the MPLAB C30 C compiler automatically associates UART1 with the standard C library functions (i.e., *printf()*) by default. If you use UART2, you may have to adjust the compiler options in order to direct the *printf()* output to UART2.

2. **Write to the Peripheral Pin Select registers.** This maps the UART to the desired pins of the dsPIC. If using the PICkit2 programmer's "UART Tool" to communicate with the dsPIC, be sure to choose the dsPIC's programming pins for the UART transmit and receive. The reason for the Peripheral Pin Select mappings is that the dsPIC33FJ128GP802 contains more peripherals than it has I/O pins. See [3, Section 11.4, p.159] for more information about the Peripheral Pin Select registers.

3. **Set up the UART configuration registers.**

4. **Configure UART interrupts.** The dsPIC33F supports both transmit and receive interrupts for the UART. If you are only transmitting data, you do not need to bother to implement the receive interrupt. However, if you do wish to receive data, you will need to set up the receive interrupt to handle each character that arrives via the UART. Generally, you will not need to implement the UART transmit interrupt because you don't care about having the dsPIC alert itself every time it transmits a character.

5.3 How to use the UART

This section explains how to use the UART to transmit and receive data. There are two ways to transmit data: 1) You can write your own code to manually transmit one character at a time by

writing to the UART transmit registers, or 2), you can use the standard C library I/O functions such as *putc()* and *printf()*. In order to receive data, you generally will use the UART receive interrupts. Polling the UART receive status registers is also an option, but we recommend using interrupts. The standard C libraries also include the *scanf()* function, but we have not had much luck with getting it to work. If you need scanf-like behavior, we suggest using interrupts to capture characters into a buffer, and then using *sscanf()* to parse the acquired string. This approach requires little coding effort and is probably more robust.

5.3.1 Using the UxTXREG and UxRXREG Registers

You can transmit a single character by writing to the transmit register (UxTXREG), where 'x' is either '1' or '2,' depending on which UART you are using. According to [2, Section 17.5.1], the transmit buffer is essentially a 5-level deep FIFO. If this transmit FIFO is not full, the UART will automatically shift in the next character. (Internally, the dsPIC33F hardware must have some mechansim that alerts (e.g., through assertion of a flag) the UART periperhal when a write instruction modifies the UxTXREG register.)

This leads to the following questions: "What happens if you try to write too many bytes to the transmit register? Does it block? Does it discard bytes?" The dsPIC33F documentation [2, Section 17.5.1] answers these questions, stating: "*Once the contents of the previous location are shifted out, a new item can be added to the FIFO. The UTXBUF (UxSTA¡9¿) status bit gets set whenever the buffer is full. If a user application attempts to write to a full buffer, the new data will not be accepted into the FIFO.*"

Thus, your application code must first check the UART status register to make sure the FIFO is empty before you write to the UxTXREG register. The following snippet of code illustrates one way to do this:

```
while(U1STAbits.UTXBF); // wait here until the transmit buffer empties
U1TXREG = 'A';          // you may now transmit a character.
```

The approach shown above is actually the same approach used in the MPLAB C30 C Compiler libraries (see `C:\ProgramFiles\Microchip\MPLABC30\src\peripheral_30F_24H_33F\src\pmc\uart\putsUART1.c`.

Receiving characters using the UxRXREG is fairly straightforward. If you are using the UART receive interrupt, simply read from the UxRXREG in the interrupt code, since you know a character must have arrived in order to trigger the interrupt to get called. If you are instead using the polling approach to receive characters, simply check the receive bit in the UART status bits register in order to determine if a new character has arrived, then read from the UxRXREG register to retrieve the received character.

5.3.2 Using Standard C Library I/O Functions

Setting up a heap. In order to use the functions from "stdio.h," you must first set up a heap. If you do not create a heap and you include "stdio.h," you will get compiler errors that notify you that you need a heap.

1. **Use the MPLAB GUI.** To create a heap, go to Project→Build Options. . . →Project, then click on the "MPLAB LINK30" tab, and enter a heap size of 256 bytes into the "Heap size" box. Figure 2 shows how to set up the heapsize in the MPLAB project build options window.

2. **Specify the heap in the source code.** We recommend that you use this approach instead because it is more convenient; it's easier to define the stack and heap in the source code, rather than having to specify them in the MPLAB build options every time you create a new project. The code in Listing 1 demonstrates how to configure the stack and heap in C code using inline assembly langauge marcos. This snippet of code was taken from "stack_and_heap.c" in the board software libraries. Note that although only the heap needs to be defined for "stdio.h" to work, we have demonstrated how to allocate the stack as well, just in case you need to know how to do this for something else. For more information about setting up the stack and heap using assembly language, refer to the README document provided with the MPLAB C30 Compiler (i.e., see *"New section attributes for creating heap or stack allocations in source code"* in in `C:\Program Files\Microchip\MPLAB C30\README.html`). For more information on inline assembly language, refer to Section 9.4 "Using Inline Assembly Language" p.127 of the Microchip "MPLAB C Compiler for PIC24 MCUs and dsPIC DSCs User's Guide (UG51284H)." Section 4.11 of this same document also gives more information about the C heap.

Listing 1 Specifying Stack and Heap in Source Code

```
__asm__(".section mystack,stack");
__asm__(".space 512");
__asm__(".section myheap,heap");
__asm__(".space 256");
```

5.3.3 How does it know to use the UART for printf?

One of the purposes of this "Detailed Guide" document is to demystify the "black box" pieces of the system. For example, you may be curious about how the dsPIC knows where to send the output of the stdio functions. In other words, how does it know to use UART1 (as opposed to UART2 or

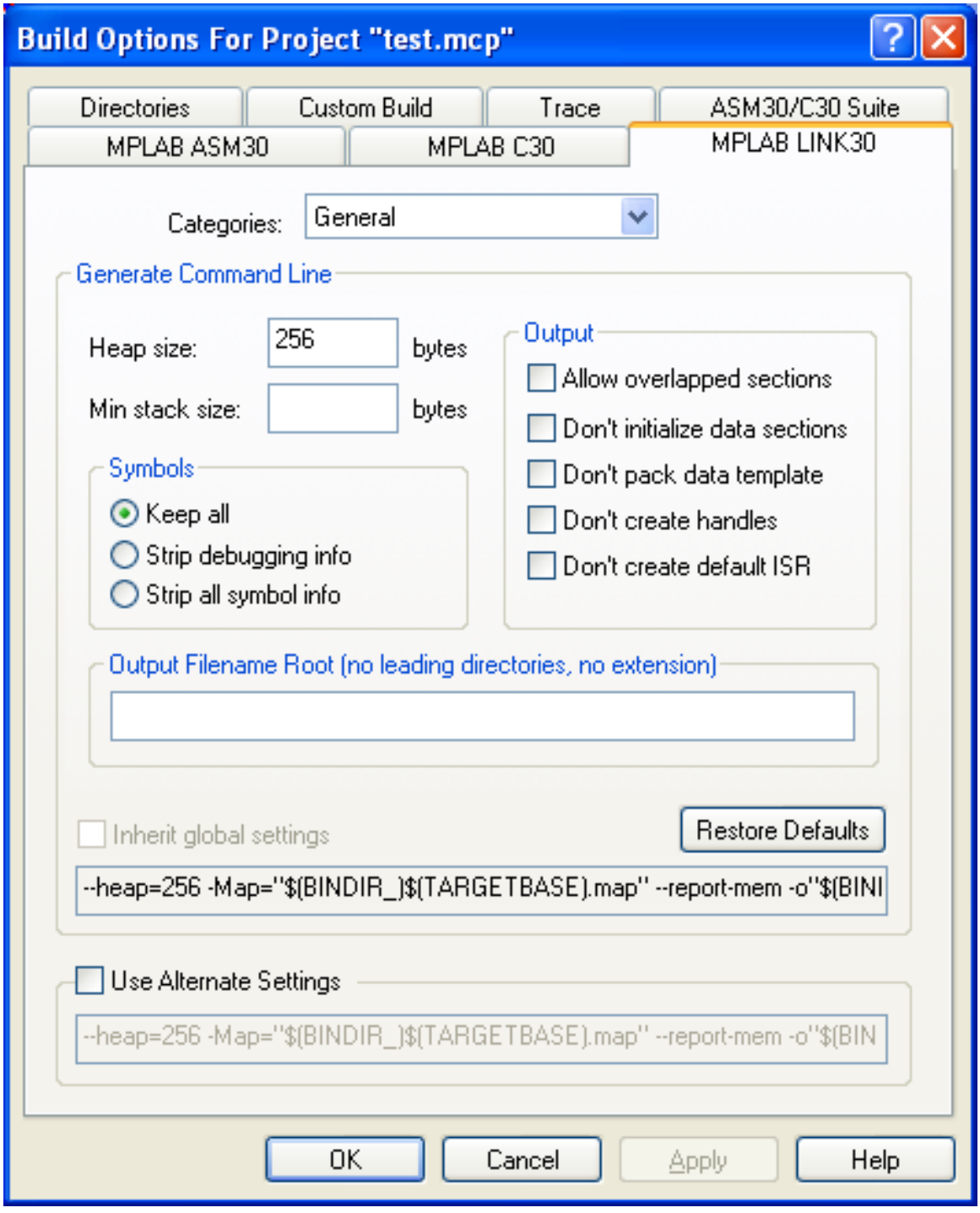


Figure 2: Setting up the heapsize in MPLAB

some other peripheral) for the output of the *printf()* function? It turns out that there is a variable used by the MPLAB C30 C compiler that tells it how to compile the *printf()* function into your code. By default, this variable is set to use UART1 as the destination. Every time the compiler sees a printf() command in your code, it inserts the appropriate instructions required for sending the data out through UART1.

The name of the variable that handles this is `__C30_UART`. If you refer to the source code file "write.c" that is part of the libraries that come with the compiler (see `C:\ProgramFiles\Microchip\MPLABC30\src\pic30\libpic30.zip`), you will see that the code checks this variable in order to determine which UART to use for writing characters. You can easily check the value of this variable in your own code, using the following:

```
extern int __C30_UART;
printf("%d\r\n", __C30_UART);
```

Running this code yields a printed value of 1, which indicates that UART1 is selected by default, as expected.

The *printf()* function relies on the *putc*function in order to actually send the printed characters to the target peripheral. If you want the *printf()* function to work with other types of peripherals, you can override the standard library definition of *putc()*. (Overriding basically means writing your own versions of these functions to send characters to your peripheral and then changing the linker options when you recompile in order include use your versions of these functions rather than the standard library's versions.) The *printf()* function will then work with your own peripheral since it calls this function in its implementation. The following are a few links that you might find useful if you ever decide to do this:

- `http://support2.microchip.com/KBSearch/KB_StdProb.aspx?ID=SQ6UJ9A009MTU`
- `http://ww1.microchip.com/downloads/en/DeviceDoc/51456E.pdf`
- `C:\ProgramFiles\Microchip\MPLABC30\docs\periph_lib\dsPIC30F_dsPIC33F_PIC24H_UART_Help.htm`
- `http://e2e.ti.com/forums/t/4701.aspx`

6 SPI Interface

The AIC23 requires an SPI interface for configuration. The SPI interface on the AIC23 consists of three wires: SCLK, SDIN, and CSL. There is no SPI data out line from the AIC23 because the

AIC23's configuration registers are write-only; there is no way to read back the settings written to the AIC23.

The CSL signal is an active low (hence the 'L' in 'CSL') chip select signal, which is used to roughly mark the start and stop of a data frame. The AIC23 looks for a high-to-low transition of the CSL signal to mark the start of the data transfer. After this transition occurs, the SPI clock must be clocked for 16 clock cycles in order to clock the 16 bits of SPI data into the AIC23. The SPI clock is not a free-running clock; it remains in its idle state (logic high) until the 16 bits of data need to be clocked, after which it returns to its idle state. The 16-bit data word is transmitted with the most significant bit (MSBit) first. After the data is clocked in, the AIC23 looks for the rising edge of the CSL signal to latch the new data into its configuration registers. The AIC23 expects the data to change on the falling edge of the clock, and it samples the data on the rising edge of the clock. Furthermore, it expects the clock polarity to be configured such that the idles state of the clock is logic low (0.0V), and the active state of the clock is logic high (+3.3V).

The timing requirements (setup and hold time) are given in the AIC23 datasheet.

Figure 3 presents some data that we collected that illustrate the SPI bus behavior.

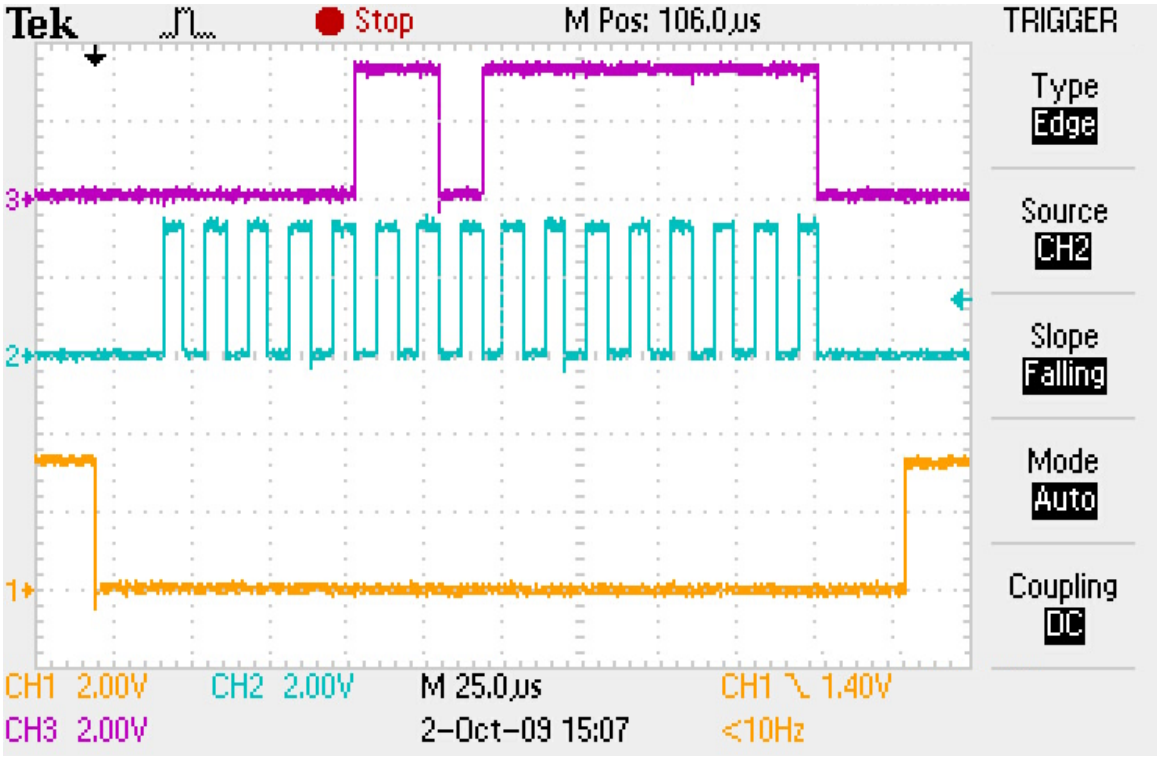


Figure 3: SPI Timing

The dsPIC33F supports two SPI peripherals. We need one of these SPI peripherals for interfacign with the AIC23. Each SPI peripheral on the dsPIC has four wires associated with it: SDI,

SDO, SCK, SS. However, we only use the clock (SCK) and data out (DOUT) lines from the SPI peripheral; we leave the SPI's SCK and SS disconnected. We must manually control the chip select (CSL) signal line (also known as slave select or SS) using a general purpose I/O pin; we cannot use the SS pin that is part of the SPI peripheral. The reason for this is that the dsPIC33F's SPI peripheral does not support the same type of type of chip select framing signal that the AIC23 expects. The dsPIC33F's SPI peripheral supports framing signals that are high for a single clock pulse to mark the start of an SPI data transfer, but it does not support a chip select signals that must be held low for the entire duration of the data transfer. However, it is not a big deal to manually assert and deassert the CSL signal with a GPIO pin; in fact, this is the way that Microchip recommends implementing SPI communications that rely on this type of chip select framing behavior (see Microchip's dsPIC33F Family Reference Manual and Microchip's dsPIC33F example SPI code). Using a GPIO pin for the SPI CSL is easy to implement, and there are generally no timing issues because the data only gets transferred when the SPI_CLK is clocked, and there is plenty of margin between asserting/deasserting the SPI_CSL signal and the start/end of the data transmission.

6.1 USB Mode versus normal mode

Since we are using a 12.000MHz crystal, we have a USB-specification compliant clock. This means that we will specify "USB Mode" in the AIC23 configuration registers, rather than "Normal Mode." (The "Normal Mode" is designed for other common oscillator crystal frequencies, such as 12.288MHz, 11.2896MHz, and 18.432MHz.) The "USB Mode" refers only to compliance with the 12.000MHz USB clock specification; it doesn't mean that the AIC23 needs to be connected to a USB port or anything like that.

Obviously note that if we set the AIC23 to "Normal Mode" and set the sampling rate control settings to 48kHz for a 12.288MHz crystal, then our sampling rates would be off because our board contains a 12.000MHz crystal.

- In "USB Mode," the default "Base Oversampling Rate (BOSR)" is 250*fs.
- In "Normal Mode," the default BOSR is 256*fs.
- Notice that for a sampling rate of fs=48kHz, we get:
- 250*48kHz = 12.000MHz ("USB Mode")
- 256*48kHz = 12.288MHz ("Normal Mode")

The AIC23 allows you to independently specify different sampling frequencies for the ADC and DAC. However, we don't need to do this; we will always use the sampling frequency for both the

ADC and DAC. The software libraries we wrote for the board always give you the same rates for both the ADC and DAC.

7 I2S Interface

- The I2S clock is a freerunning clock (unlike SPI, which is idle when not sending data)
- The dsPIC starts capturing data on either transition so that you have to manually poll the pin's state to determine which channels (L/R) you received, etc.)
- We only use LRCOUT. Data is transferred bidirectionally each time this is triggered so we don't need to use both framing signal lines from AIC23. And it works just fine. We interrupt when this occurs, do processing, then put data in the TXBUF0/TXBUF1 registers, so that it will be sent out on next interrupt. Thus, the way you use the dsPIC is very similar to using the DSK.
- You must manually poll the framing signal pin in order to determine if you are looking at the left or right channel. We have written code to do this; see our board support libraries.

8 Interrupts on the dsPIC33F

The following are a few helpful notes concerning dsPIC33F interrupts.

- Refer to Section 6 of the dsPIC33F Family Reference Manual [2].
- Interrupts priority levels can be 1 through 7, where 7 indicates the highest priority, and 1 indicates the lowest priority. The interrupt priority is set using the interrupt priority control (IPC) registers. According to [2, Section 6.1.5], *"If the IPC bits associated with an interrupt source are all cleared, the interrupt source is effectively disabled."*

9 MPLAB Tips

- Use relative paths. According to the MPLAB help file, "Specifying Project File Paths: It is recommended that you use relative paths whenever possible for project portability."

- On a similar note, the MPLAB help document (hlpMPLABIDE.chm) explains in an FAQ that "Project Files (.mcp) are portable. Workspace files (.mcw) may or may not be portable. If you try ot move your projects and workspaces between PCs, [...] you will need to update your workspace or create a new one."

- Note that you need to use backslashes for the path separators in MPLAB since it is running on Microsoft Windows and MPLAB is not smart enough to handle forward slashes.

- In MPLAB, there is a notion of "workspaces" and "projects." After you complete this project setup wizard, MPLAB will create a project (.mcp) file and a workspace (.mcw) file in the directory that you selected. What is the difference between a "workspace" and a "project"? According to p.87 of the MPLAB User's Guide [5]:

  - "A **project** contains the files needed to build an application (source code, linker script files, etc.) along with their associations to various build tools and build options."
  - "A **workspace** contains information on the selected device, debug tool and/or programmer, open windows and their location, and other IDE configuration settings."

  For the purposes of this lab, we only need to use a simple setup. For each part of each lab, we will create a new project, and MPLAB will automatically generate a workspace associated with that project. If you save the project and workspace and close MPLAB, and then later you go back and start MPLAB, be sure to open the *workspace*, rather than the *project*. That way, all of your workspace preferences get loaded. It doesn't make a huge difference, but loading the whole workspace is slightly more convenient.

- You might find it handy to have multiple projects open within a single workspace in MPLAB. For example, I found this handy when I wanted to work on the board support libraries at the same time that I was working on another project that used these libraries. (Closing one project and opening another simply to recompile would have been too tedious!)
  To enable opening two projects in the same workspace, simply go to "Configure*rightarrow*Settings..." and uncheck the "Use one-to-one-workspace model" checkbox. This is shown in Figure 4. After changing this setting, you can create a new workspace and open multiple projects simultaneously.

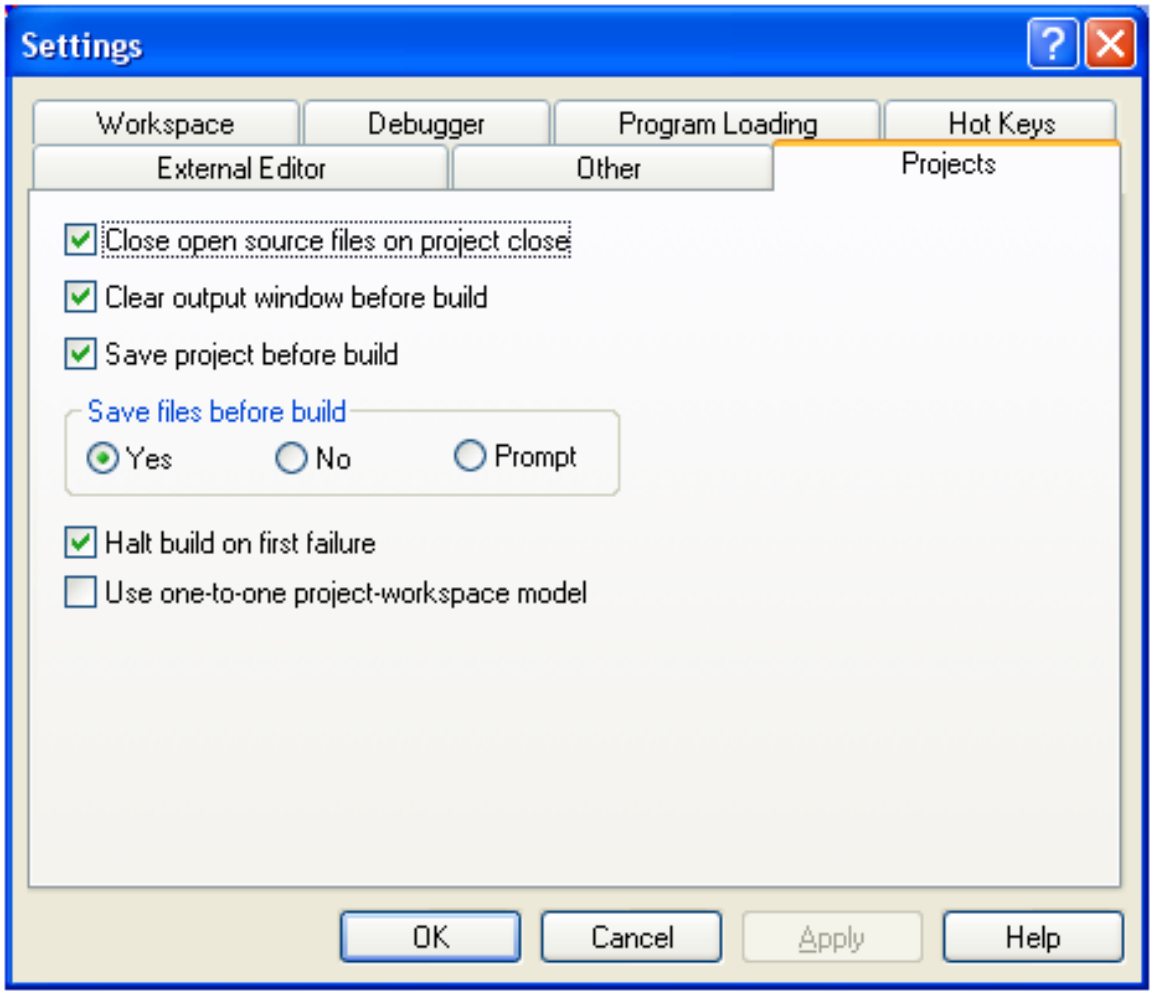


Figure 4: How to allow multiple projects per workspace

- The difference between “make” and “build all” in MPLAB is that “build all” rebuilds everything, whereas “make” only rebuilds files that changed (GNU “Make”-style behavior).
- The difference between “Add Files...” and “Add New File...” is that “Add Files...” allows you to import existing files, whereas “Add New File...” allows you to create a new blank file.

10 PICkit2 Debugger Tips

The PICkit2 debugger allows you to perform in-circuit debugging using the MPLAB development environment. Although the PICkit2's debugger capabilities are limited (compared to other Microchip debuggers such as the Microchip ICD2 or the Microchip Real ICE), the PICkit2 should still be suitable for all of the projects used in the ECE 4703 course. The following is a list of tips and helpful information concerning the PICkit2 debugger.

- **The debugger supports all of the common debugger operations:**
  "Run," "halt," "animate," "step in," "step over," "step out," etc.

- **The debugger supports the typical modes of viewing data:**
  "Call stack," "Disassembly listing," "File registers," "Program memory," "Locals," "Special Function registers," "Watch window," and "Memory Usage Gauge."

- **Execution halts after the line of the breakpoint.** This might seem like non-intuitive behavior, since many other debuggers that you may have used tend to stop before executing the line of the breakpoint, not after. Microchip calls this behavior *"breakpoint skidding."* It may time to get used to this behavior, but it stopping after executing the line of the breakpoint works just as well for debugging as stopping before executing the line of the breakpoint.

- **The PICkit2 debugger supports setting up to two breakpoints.** This should be enough for simple debugging, but you do use these up faster than you think. For example, after setting the second of two breakpoints, MPLAB gives you a courteous message informing you that the "step out" and "step over" buttons are disabled until you free up a breakpoint, since these operations require the use of a breakpoint.

- **It is important to be patient with the debugger.** Sometimes when debugging, the MPLAB IDE becomes unresponsive and you might think that you crashed the program. However, if you wait several seconds, MPLAB generally starts to respond again. The reason why the GUI appears to hang is that it is in the middle of transferring large amounts of data between the dsPIC and the PC; if you give it some time to catch up, it will recover. If you unplug the USB cable, MPLAB recovers immediately, but we do not recommend unplugging the USB cable in the middle of debugging because that confuses the debugger and results in errors until you exit and restart MPLAB. **The Microchip Forum [4] offers the following suggestions for increasing the speed of debugging:**

  - "*File Registers Window* is open. With this window open, all file registers must be read from and written to the PIC MCU during each debugging operation. This can really

slow things down! Close this window and watch only needed registers and variables using a *Watch Window instead*."

- "*Special Function Registers Window* is open. With this window open, all SFRs must be read from and written to the PIC MCU during each debugging operation. This can really slow things down! Close this window and watch only the needed SFRs using a *Watch Window* instead."
- "*Slow MCU Oscillator* results in slow debugging as the oscillator speed directly affects the debugger communications speed. Using a 32kHz clock can really slow things down!"

11 How to Compile Board Support Libraries in MPLAB

This section guides you through the process of creating a new MPLAB project, importing the board support library code and header files, and compiling them. The compiled libraries can then be used in all your software projects.

First, start MPLAB and go through the project wizard (Project*rightarrow*Project Wizard...) to create a new project. Be sure to specify the architecture as "dsPIC33FJ128GP802." Be sure to select the "Microchip C30 Toolsuite." Select a sensible location and filename (e.g., [...]/mplab/board.mcp) for the new MPLAB project file. Don't bother using the wizard to add source code to the project; we'll do this later.

Next, open up the "Build Options" dialog window for your new project. Click on the "Directories" tab. Select "Include Search Path" from the drop-down box. Add a new entry for the path to the header files directory. It is recommended that you use relative paths for this step in order to make your project portable. Remember to use backslashes for the path separators because MPLAB on Microsoft Windows does not understand forward slashes. An example screenshot is shown in Figure 5. Be sure to select the "Assemble/Compile/Link in the project directory" radio button; this will keep all the temporary compiler files from getting mixed up with your source files. Click "Apply" to save your settings.

At this point, you should still have the "Build Options" open to the "Directories" tab. Select "Output Directory" from the drop-down box. Select the destination directory for the output library archive (.a) file. For example, I use the directory "../lib/" to as the output. The sole purpose of selecting this output directory is to keep our files organized; it would still successfully compile the ".a" file if we didn't specify an output directory, but it would place this file in the cluttered MPLAB project directory by default. Be sure to select the "Assemble/Compile/Link in the project directory" radio button; it is much cleaner to keep the ".o" files from getting mixed up with your source files. Click "Apply" to save your settings.

Next, you should select the "ASM30/C30 Suite" tab in the "Build Options" dialog box. Select the "Build Library Target" radio button instead of the "Build normal target" radio button. This will allow you to produce library archive (.a) files, rather than generating a binary programming image (.hex) file. The library file produced will have the same name as your project, but with ".a" file extension, and it will be created in the output directory you specified earlier. Leave the "Build generic library" box unchecked, since our library code is designed specifically for the dsPIC33FJ128GP802. Click "OK" when you are done to save your settings and close the "Build Options" dialog.

Now we need to add the source code files to our project! We already specified the path to the header (.h) files, so all we need to import now are the source (.c) files. Right-click on the project in

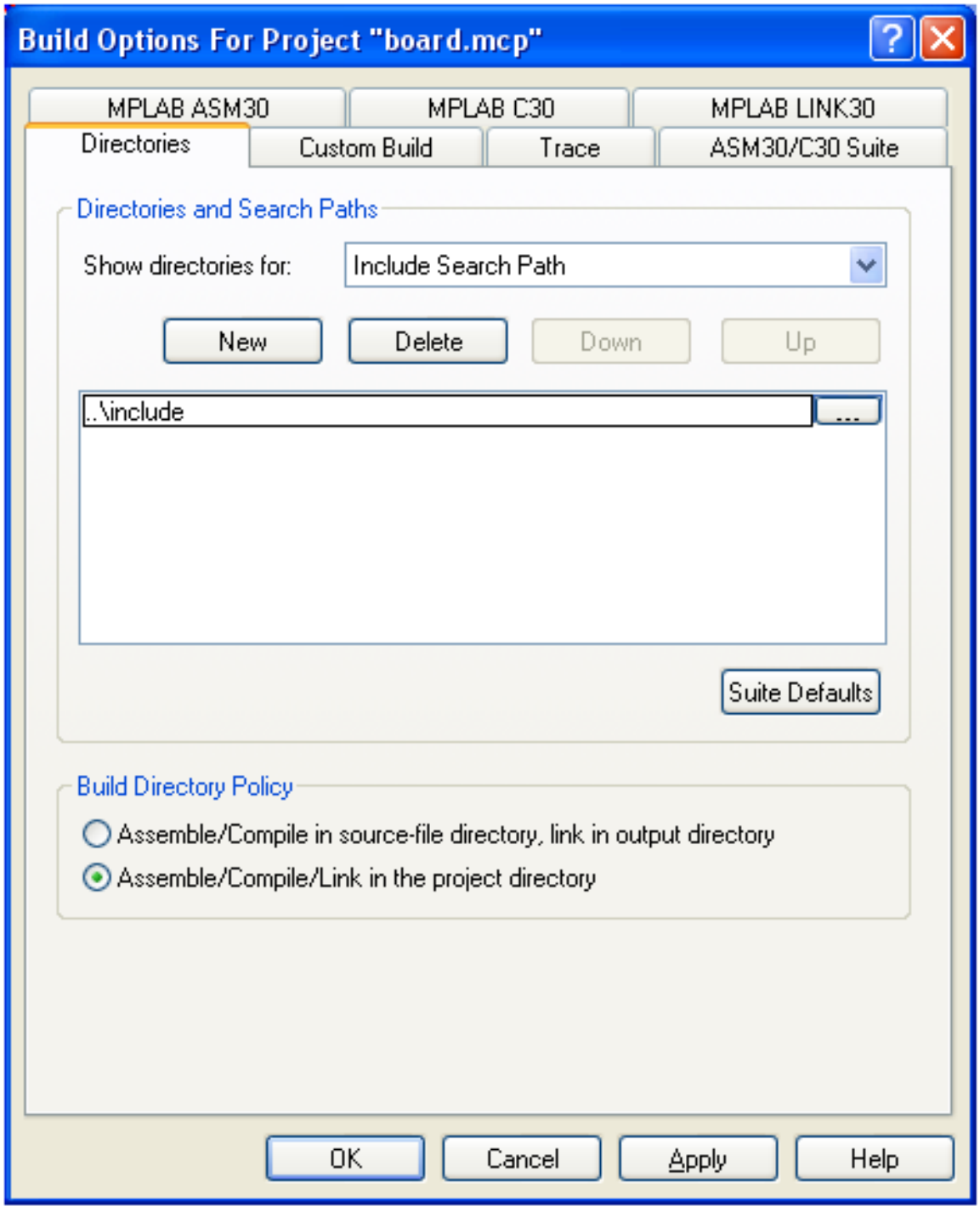


Figure 5: Build options

the Project Window and click "Add files..." Use the file selection dialog box to select all your ".c" files. At the bottom of the dialog box, you will see some radio buttons; it is recommended that you select the "User: File(s) were created especially for this project; use relative path" radio button so that the imported files will use relative paths (for better project portability). Click "Open" to add the files to the project.

Finally, press "F10" to compile all of the files!

12 Embedded Programming Tips

- The reason why we clear the interrupt flag before we enable interrupts (rather than enabling interrupts before clearing the flag) is so that just in case the interrupt flag is already set, you don't instantly get sent into the interrupt.

- The preprocessor macros for setting up the configuration registers are considered definitions (i.e., actual code), not declarations. Therefore, you cannot stick them in a header (.h) file, because importing the file in two different libraries will cause the linker to complain about multiple definitions (even if you use "#include guards" in the header files). You need to put the code for setting up the configuration registers in a '.c' file. You will need to include the this .c file in each project, which is not as slick as creating one precompiled library file, but as long as you reference the .c file from the library src directory (rather than copying it, so that we can maintain a single copy), we can achieve our goal of code reuse.

References

[1] D. Cullen and J. DeFeo. dsPIC33F Audio Signal Processing Platform. MQP Final Report. http://spinlab.wpi.edu/.

[2] Microchip. dsPIC33F Family Reference Manual. http://www.microchip.com/.

[3] Microchip. dsPIC33FJ128GP802 Datasheet. http://www.microchip.com/.

[4] Microchip. Microchip Forums - PICkit2 FAQ. http://www.microchip.com/forums/printable.aspx?m=270347.

[5] Microchip. MPLAB User's Guide - MPLAB_USER_GUIDE_51519c.pdf. http://www.microchip.com/.

E Board Software Libraries

This appendix provides the source code and documentation for the board infrastructure libraries.

E.1 aic23.h

```
#ifndef AIC23_H
#define AIC23_H

#define AIC23_8  (1)  // fs = 8 kHz
#define AIC23_16 (2)  // fs = 16 kHz
#define AIC23_24 (3)  // fs = 24 kHz
#define AIC23_32 (4)  // fs = 32 kHz
#define AIC23_44 (5)  // fs = 44.1 kHz
#define AIC23_48 (6)  // fs = 48 kHz
#define AIC23_96 (7)  // fs = 96 kHz

void AIC23_init(unsigned int);
void AIC23_get_samples(signed int *, signed int *);
void AIC23_set_samples(signed int *, signed int *);
void AIC23_bypass_mode_on(void);
void AIC23_bypass_mode_off(void);
void AIC23_SPI_write(unsigned int, unsigned int);
void AIC23_SPI_init(void);
void AIC23_DCI_init(void);

#endif
```

E.2 benchmark.h

```
#ifndef BENCHMARK_H
#define BENCHMARK_H

#include <p33FJ128GP802.h>

// We define the timer controls as macros so that there is no function call latency.
#define benchmark_timer_start(void) {T2CONbits.TON = 1;}
#define benchmark_timer_stop(void)  {T2CONbits.TON = 0;}
```

```
9  #define benchmark_timer_read(msw,lsw) {lsw=TMR2; msw=TMR3HLD;}     // Must read LSW first.
       Afterwards, MSW must be read from TMRHLD.
10 #define benchmark_timer_clear(void) {TMR3HLD=0x0000; TMR2=0x0000;} // Must write MSW to
       TMR3HLD first. Afterwords, write to LSW.
11
12 void benchmark_timer_init(void);
13
14 void __attribute__ ((interrupt, no_auto_psv)) _T3Interrupt(void);  // Declare for good
       measure, even though interrupts aren't used for benchmarking.
15
16 #endif
```

E.3 dspic.h

```
1  #ifndef DSPIC_H
2  #define DSPIC_H
3
4  void dspic_init(void);
5
6  #endif
```

E.4 dspic_adc.h

```
1  #ifndef DSPIC_ADC_H
2  #define DSPIC_ADC_H
3
4  void dspic_adc_init(void);
5
6  #endif
```

E.5 gpio.h

```
1  #ifndef GPIO_H
2  #define GPIO_H
3
4  #include <p33FJ128GP802.h>
5
6  // DEFINED CONSTANTS
7  #define LED_0       (LATBbits.LATB10)   // RB10 = LED_0 = pin 21
8  #define LED_1       (LATBbits.LATB11)   // RB11 = LED_1 = pin 22
9  #define DIP_0       (PORTBbits.RB2)     // RB2  = DIP_0 = pin 6
```

```
10 #define DIP_1      (PORTBbits.RB3)     // RB3  = DIP_1 = pin 7
11 #define TRIS_LED_0 (TRISBbits.TRISB10) // Tristate register for LED_0
12 #define TRIS_LED_1 (TRISBbits.TRISB11) // Tristate register for LED_1
13 #define TRIS_DIP_0 (TRISBbits.TRISB2)  // Tristate register for DIP_0
14 #define TRIS_DIP_1 (TRISBbits.TRISB3)  // Tristate register for DIP_1
15
16 // FUNCTION DECLARATIONS
17 void gpio_init(void);
18
19 #endif
```

E.6 status_timer.h

```
1 #ifndef STATUS_TIMER_H
2 #define STATUS_TIMER_H
3
4 void status_timer_init(void);
5
6 #endif
```

E.7 swdelay.h

```
1 #ifndef SWDELAY_H
2 #define SWDELAY_H
3
4 void swdelay(unsigned int);
5
6 #endif
```

E.8 traps.h

```
1 #ifndef TRAPS_H
2 #define TRAPS_H
3
4 #include <p33FJ128GP802.h>
5
6 void __attribute__((__interrupt__)) _OscillatorFail(void);
7 void __attribute__((__interrupt__)) _AddressError(void);
8 void __attribute__((__interrupt__)) _StackError(void);
9 void __attribute__((__interrupt__)) _MathError(void);
10 void __attribute__((__interrupt__)) _DMACError(void);
```

```
11
12 void __attribute__((__interrupt__)) _AltOscillatorFail(void);
13 void __attribute__((__interrupt__)) _AltAddressError(void);
14 void __attribute__((__interrupt__)) _AltStackError(void);
15 void __attribute__((__interrupt__)) _AltMathError(void);
16 void __attribute__((__interrupt__)) _AltDMACError(void);
17
18 #endif
```

E.9 uart.h

```
1  #ifndef UART_H
2  #define UART_H
3
4  #include <p33FJ128GP802.h>
5
6  #define CMD_BUFFER_LENGTH (32)  // 32 characters maximum
7
8  void __attribute__ ((interrupt, no_auto_psv)) _U1RXInterrupt(void);
9  void __attribute__ ((interrupt, no_auto_psv)) _U1TXInterrupt(void);
10
11 void uart_init(void);
12 void cmd_handle_char(unsigned char);
13 unsigned char cmd_get_flag(void);
14 unsigned char* cmd_get_buffer(void);
15 void cmd_clear(void);
16
17 #endif
```

E.10 aic23.c

```
1  #include <p33FJ128GP802.h>
2  #include "aic23.h"
3
4  void AIC23_init(unsigned int fs)
5  {
6    // Use this function to initialize the AIC23.
7    // fs = desired sampling frequency (used for both the ADC and DAC).
8    // If you need to change the sampling frequency at a later time,
9    // simply call this function again, specifying the new sampling frequency.
10
```

```
11    // ---------------------------------------------------------------
12    // Initialize SPI:
13    // ---------------------------------------------------------------
14    AIC23_SPI_init();
15
16    // ---------------------------------------------------------------
17    // Initialize DCI:
18    // ---------------------------------------------------------------
19    AIC23_DCI_init();
20
21    // ---------------------------------------------------------------
22    // Now write the configuration registers:
23    // ---------------------------------------------------------------
24
25    // Must configure power down control register first:
26    AIC23_SPI_write(0x0006, 0x0002);  // Power down control. Must configure power down first.
          Mic input off; everything else is on.
27
28    AIC23_SPI_write(0x0000, 0x0017);  // Left line input channel volume control
29    AIC23_SPI_write(0x0001, 0x0017);  // Right line input channel volume control
30
31    AIC23_SPI_write(0x0002, 0x00D8);  // Left channel headphone volume control
32    AIC23_SPI_write(0x0003, 0x00D8);  // Right channel headphone volume control
33
34    AIC23_SPI_write(0x0004, 0x0012);  // Analog audio path control: DAC on. Bypass mode OFF.
          line input selected. microphone muted. mic boost off.
35
36    AIC23_SPI_write(0x0005, 0x0000);  // Digital audio path control
37
38    AIC23_SPI_write(0x0007, 0x0042);  // Digital audio interface format
39
40    // Sample rate control. See Detailed Guide for explanation of why we set the AIC23 to "USB
          Mode", rather than "Normal Mode."
41    switch (fs)
42    {
43      case AIC23_8:   AIC23_SPI_write(0x0008, 0x000D); break;
44      case AIC23_16:  AIC23_SPI_write(0x0008, 0x0059); break;
45      case AIC23_24:  AIC23_SPI_write(0x0008, 0x0041); break;
46      case AIC23_32:  AIC23_SPI_write(0x0008, 0x0019); break;
47      case AIC23_44:  AIC23_SPI_write(0x0008, 0x0023); break;
48      case AIC23_48:  AIC23_SPI_write(0x0008, 0x0001); break;
```

```
49      case AIC23_96:   AIC23_SPI_write(0x0008, 0x001D); break;
50      default:         AIC23_SPI_write(0x0008, 0x0023); break; // Default: Use AIC23_44.
51    }
52
53    AIC23_SPI_write(0x0009, 0x0001);  // Digital interface activitation
54  }
55
56  static unsigned int AIC23_swapped = 0; // Private variable keeps track of I2S left/right
        alignment
57
58  void AIC23_get_samples(signed int *left, signed int *right)
59  {
60    // See Detailed Guide for further explanation of how/why we need to check alignment.
61    if (PORTBbits.RB6 == 0) AIC23_swapped = 1; else AIC23_swapped = 0;
62
63    // Receive the new data.
64    if (AIC23_swapped) { *right = RXBUF0; *left = RXBUF1; } else { *left = RXBUF0; *right =
          RXBUF1; }
65  }
66
67  void AIC23_set_samples(signed int *left, signed int *right)
68  {
69    // Fill transmit buffer with data to be sent out on the next DCI frame.
70    // See Detailed Guide for further explanation of how/why we need to check alignment.
71    if (AIC23_swapped) { TXBUF0 = *right; TXBUF1 = *left; } else { TXBUF0 = *left; TXBUF1 =
          *right; }
72  }
73
74  void AIC23_bypass_mode_on(void)
75  {
76    AIC23_SPI_write(0x0004, 0x000A);  // Analog audio path control: DAC is OFF. Bypass mode
          ON. line input selected. microphone muted. mic boost off.
77  }
78
79  void AIC23_bypass_mode_off(void)
80  {
81    AIC23_SPI_write(0x0004, 0x0012);  // Analog audio path control: DAC is ON. Bypass mode
          OFF. line input selected. microphone muted. mic boost off.
82  }
83
84  void AIC23_SPI_write(unsigned int addr, unsigned int data)
```

```
85  {
86    // ---------------------------------------------------------------
87    // Writes configuration data to AIC23 via SPI.
88    // ---------------------------------------------------------------
89    // Notes:
90    //     The technique for manually controlling the chipselect line (rather than using
          built-in SPI stuff)
91    //     comes from CE 136 microchip sample code. The same technique is used in CE 141. If
          you look at
92    //     the README file that comes with CE 141, you will see the timing diagram (in ascii
          art) that
93    //     describes this CS behavior that we need for controlling the AIC23.
94    //     For the AIC23 configuration words: The 7 MSBits are the address, and the 9 LSBits
          are the address.
95    // ---------------------------------------------------------------
96    unsigned int spi_word = 0;
97    unsigned int temp;
98
99    spi_word = ((0x007F&addr)<<9) | (0x01FF&data);
100
101   while (SPI1STATbits.SPITBF == 1) { }  // Wait for the transmit output buffer to empty
          before writing to the SPI1BUF register.
102   PORTAbits.RA4 = 0;                    // lower the slave select line
103   temp = SPI1BUF;                       // dummy read of SPI1BUF register to clear the
          SPIRBF flag
104   SPI1BUF = spi_word;                   // write the data out to the SPI peripheral
105   while (SPI1STATbits.SPIRBF == 0) { }  // wait for the data to be sent out
106   PORTAbits.RA4 = 1;                    // raise the slave select line
107 }
108
109 void AIC23_SPI_init(void)
110 {
111   // ---------------------------------------------------------------
112   // Initialize SPI
113   // ---------------------------------------------------------------
114   // Notes:
115   //     Since we're only transmitting for the SPI, not receiving, we don't need to write an
           ISR for the SPI.
116   //     Also, we have to manually assert the CS line, since the frame signals that are
          supported by the dsPIC
117   //     are not the same as what the AIC23 expects.
```

```
118     //     We are running the SPI clock at 2.5MHz (which we have configured below with the
            PPRE and SPRE bits).
119     // ------------------------------------------------------------------
120     // SPI Remappable peripheral pin setup:
121     RPOR2bits.RP4R       = 8;  // Set pin 11 (RP4) to SPI Clock (SCK)
122     RPOR2bits.RP5R       = 7;  // Set pin 14 (RP5) to SPI Data Output (SDO)
123     TRISBbits.TRISB4     = 0;
124     TRISBbits.TRISB5     = 0;
125     TRISAbits.TRISA4     = 0;  // Chip select (pin 12)
126     PORTAbits.RA4        = 1;
127     // SPI control register setup:
128     SPI1STATbits.SPIEN   = 0;  // First, we must disable SPI1 while we configure it.
129     SPI1STATbits.SPISIDL = 0;
130     SPI1CON1bits.DISSCK  = 0;
131     SPI1CON1bits.DISSDO  = 0;
132     SPI1CON1bits.MODE16  = 1;
133     SPI1CON1bits.SMP     = 0;
134     SPI1CON1bits.CKE     = 1;
135     SPI1CON1bits.SSEN    = 0;
136     SPI1CON1bits.CKP     = 0;
137     SPI1CON1bits.MSTEN   = 1;
138     SPI1CON1bits.SPRE    = 0x07;
139     SPI1CON1bits.PPRE    = 0x01;
140     SPI1CON2bits.FRMEN   = 0;
141     SPI1CON2bits.SPIFSD  = 0;
142     SPI1CON2bits.FRMPOL  = 0;
143     SPI1CON2bits.FRMDLY  = 0;
144     SPI1STATbits.SPIEN   = 1;  // Finally, enable the SPI.
145 }
146
147 void AIC23_DCI_init(void)
148 {
149     // ------------------------------------------------------------------
150     // Initialize DCI/I2S.
151     // ------------------------------------------------------------------
152     // Remappable peripheral setup:
153     RPINR24bits.CSCKR    = 9;      // DCI Clock (CSCK)
154     RPOR4bits.RP8R       = 0x0D;   // DCI Data Output (CSDO)
155     RPINR24bits.CSDIR    = 7;      // DCI Data Input (CSDI)
156     RPINR25bits.COFSR    = 6;      // DCI Frame Sync (COFS)
157     TRISBbits.TRISB9     = 1;
```

```
158     TRISBbits.TRISB8    = 0;
159     TRISBbits.TRISB7    = 1;
160     TRISBbits.TRISB6    = 1;
161     // Setup register values:
162     DCICON1bits.DCIEN   = 0;      // Disable DCI.
163     DCICON1bits.DCISIDL = 0;
164     DCICON1bits.DLOOP   = 0;
165     DCICON1bits.CSCKD   = 1;
166     DCICON1bits.CSCKE   = 1;
167     DCICON1bits.COFSD   = 1;
168     DCICON1bits.UNFM    = 0;
169     DCICON1bits.CSDOM   = 0;
170     DCICON1bits.DJST    = 0;
171     DCICON1bits.COFSM   = 0x01;
172     DCICON2bits.BLEN    = 0x01;
173     DCICON2bits.COFSG   = 1;
174     DCICON2bits.WS      = 15;
175     DCICON3             = 0x0000;
176     RSCON               = 0x0001;
177     TSCON               = 0x0001;
178     // FRM says that it's good practice to initialize transmit buffers with default values:
179     TXBUF0 = 0x0000;
180     TXBUF1 = 0x0000;
181     TXBUF2 = 0x0000;
182     TXBUF3 = 0x0000;
183     // DCI interrupt setup:
184     IPC15bits.DCIIP     = 7;      // interrupt priority level
185     IFS3bits.DCIIF      = 0;      // clear the interrupt flag
186     IEC3bits.DCIIE      = 1;      // enable DCI interrupt
187     DCICON1bits.DCIEN   = 1;      // Enable DCI.
188 }
```

E.11 benchmark.c

```
1  #include <p33FJ128GP802.h>
2  #include "benchmark.h"
3
4
5  // Here's a Timer3 ISR. It never gets executed because we never enable
6  // interrupts for our benchmarking timer. We include it here solely
7  // for the sake of completeness.
```

```
8  void __attribute__ ((interrupt, no_auto_psv)) _T3Interrupt(void)
9  {
10   IFS0bits.T3IF = 0; // clear interrupt flag
11 }
12
13
14 void benchmark_timer_init(void)
15 {
16   // Initializes timers 2 and 3 are used together as a 32-bit timer for benchmarking
17   //
18   // Initialize Timers 2 and 3 as a 32-bit timer:
19   //     Timer2 = least significant word
20   //     Timer3 = most significant word
21   //     When operating in 32-bit mode, only Timer2's T2CON control register is used;
22   //     the T3CON control register for Timer3 is completely ignored.
23   //     If you want to read the count value of Timer3 while it is running, and you want
24   //     the LSW and MSW words that you read to be synchronized with respect to each other,
25   //     then you must access the MSW through a special control register, the TMR3HLD
26   //     "holding" register. If you don't care about reading the exact count value of Timer3
27   //     while it is free-running, then you can simply just read from the TMR2 and TMR3
       registers
28   //     directly.
29   //       --> UPDATE: It seems that even if the 32-bit timer is not running, you cannot
       write to the TMR3
30   //           register directly. You must write to TMR3HLD. Otherwise, the value never
       seems to get applied!
31   //           I know for ceterain that this is the case because I tried doing it both ways.
32   //           See p.16 of FRM 11.6 for an example of how to correctly read and write from
       the timer registers.
33   //           I imagine that you also must read from the TMR3HLD register for the correct
       value.
34   //           Aside: Note that clearing the interrupt flag in the ISR automatically causes
       the hardware
35   //           to clear the count. This works correctly for both 16-bit and 32-bit counters.
36   //     When operating in 32-bit mode, only Timer3's ISR and interrupt flag/enable bits get
        used;
37   //     the ISR and interrupt flag/enable bits from Timer2 are completely ignored.
38   //
39   // With a 1:1 prescale value and 32 bits, (2^32)/(FCY) = (2^32)/40MHz = 107 seconds
       maximum before timer reaches maximum count.
40
```

```
41    T3CONbits.TON    = 0;        // disable 16-bit Timer3
42    T2CONbits.TON    = 0;        // disable 16/32-bit timer
43    T2CONbits.T32    = 1;        // Sets up Timers 2 and 3 for a 32-bit timer
44    T2CONbits.TSIDL = 0;
45    T2CONbits.TGATE = 0;
46    T2CONbits.TCKPS = 0x00;     // use 1:1 prescale value
47    T2CONbits.TCS    = 0;
48    TMR3HLD          = 0x0000;  // clear MSW of the 32-bit timer value
49    TMR2             = 0x0000;  // clear LSW of the 32-bit timer value
50    PR3              = 0xFFFF;  // load MSW of the 32-bit period value
51    PR2              = 0xFFFF;  // load LSW of the 32-bit period value
52    IPC2bits.T3IP    = 0;       // interrupt priority level
53    IFS0bits.T3IF    = 0;       // clear interrupt flag
54    IEC0bits.T3IE    = 0;       // leave interrupt disabled, since the user will be polling the
          timer; we don't want to use interrupts.
55    T2CONbits.TON    = 1;       // leave the 32-bit timer stopped, since user will start it
          when he/she wishes to start benchmarking.
56  }
57
58
59
60
61
62
63  // ------------------------------------------------------------------
64  // Some example benchmarking code:
65  // ------------------------------------------------------------------
66
67  /* // MAIN PROGRAM */
68  /* int main (void) */
69  /* { */
70  /*   sys_init(); */
71  /*   adc_init(); */
72  /*   timer_init(); */
73  /*   uart_init(); */
74  /*   gpio_init(); */
75
76  /*   while(1) */
77  /*   { */
78  /*     swdelay(128); */
79  /*     run_benchmarks(); */
```

```
80  /*   } */
81  /*   return 0; */
82  /* } */
83
84  /* void run_benchmarks(void) */
85  /* { */
86  /*   // Runs a set of benchmark tests and prints results to output. */
87  /*   // Notes: */
88  /*   //   * The resulting numbers are actually 1 instruction clock longer */
89  /*   //     than the amount of time it took to execute the instructions under test, */
90  /*   //     because it takes a clock to stop the timer at the end of the test. */
91  /*   //   * Each test is not the time it takes just to run the instruction under test, */
92  /*   //     but rather includes the time it takes to load the values into registers */
93  /*   //     before and after executing the instruction under test. */
94  /*   //   * Note that these tests are not using the special DSP instructions. */
95  /*   //     Nor are they using various efficient addressing modes; every */
96  /*   //     instruction loads values into intermediate registers before processing, */
97  /*   //     which is probably slower than it would be if the memory was used with care. */
98
99  /*   unsigned int msw, lsw; */
100 /*   float f1=7.0, f2=5.0, f3=0.0; */
101 /*   signed int si1=-12, si2=5, si3=-9;; */
102 /*   unsigned int ui1=15, ui2=18, ui3=22; */
103 /*   long signed int ls1=-12, ls2=5, ls3=-9; */
104
105 /*   benchmark_timer_clear(); */
106 /*   benchmark_timer_start(); */
107 /*   benchmark_timer_stop(); */
108 /*   benchmark_timer_read(msw, lsw); */
109 /*   printf("No test\t%04X\t%04X\r\n", msw, lsw); */
110
111 /*   benchmark_timer_clear(); */
112 /*   benchmark_timer_start(); */
113 /*   Nop(); */
114 /*   benchmark_timer_stop(); */
115 /*   benchmark_timer_read(msw, lsw); */
116 /*   printf("NOP\t%04X\t%04X\r\n", msw, lsw); */
117
118 /*   benchmark_timer_clear(); */
119 /*   benchmark_timer_start(); */
120 /*   swdelay(1); */
```

```
121 /*   benchmark_timer_stop(); */
122 /*   benchmark_timer_read(msw, lsw); */
123 /*   printf("swdelay(1)\t%04X\t%04X\r\n", msw, lsw); */
124
125 /*   benchmark_timer_clear(); */
126 /*   benchmark_timer_start(); */
127 /*   f3 = f1*f2; */
128 /*   benchmark_timer_stop(); */
129 /*   benchmark_timer_read(msw, lsw); */
130 /*   printf("MPY(FP,FP)(A=B*C)\t%04X\t%04X\r\n", msw, lsw); */
131
132 /*   benchmark_timer_clear(); */
133 /*   benchmark_timer_start(); */
134 /*   f3 *= f1; */
135 /*   benchmark_timer_stop(); */
136 /*   benchmark_timer_read(msw, lsw); */
137 /*   printf("MPY(FP,FP)(A*=B)\t%04X\t%04X\r\n", msw, lsw); */
138
139 /*   benchmark_timer_clear(); */
140 /*   benchmark_timer_start(); */
141 /*   f3 = f1+f2; */
142 /*   benchmark_timer_stop(); */
143 /*   benchmark_timer_read(msw, lsw); */
144 /*   printf("ADD(FP,FP)(A=B+C)\t%04X\t%04X\r\n", msw, lsw); */
145
146 /*   benchmark_timer_clear(); */
147 /*   benchmark_timer_start(); */
148 /*   f3 += f1; */
149 /*   benchmark_timer_stop(); */
150 /*   benchmark_timer_read(msw, lsw); */
151 /*   printf("ADD(FP,FP)(A+=B)\t%04X\t%04X\r\n", msw, lsw); */
152
153 /*   benchmark_timer_clear(); */
154 /*   benchmark_timer_start(); */
155 /*   si1 = (signed int)f1; */
156 /*   benchmark_timer_stop(); */
157 /*   benchmark_timer_read(msw, lsw); */
158 /*   printf("Cast float to int16\t%04X\t%04X\r\n", msw, lsw); */
159
160 /*   benchmark_timer_clear(); */
161 /*   benchmark_timer_start(); */
```

```
162 /*   f1 = (float)si1; */
163 /*   benchmark_timer_stop(); */
164 /*   benchmark_timer_read(msw, lsw); */
165 /*   printf("Cast int16 to float\t%04X\t%04X\r\n", msw, lsw); */
166
167 /*   benchmark_timer_clear(); */
168 /*   benchmark_timer_start(); */
169 /*   ui1 = (unsigned int)f1; */
170 /*   benchmark_timer_stop(); */
171 /*   benchmark_timer_read(msw, lsw); */
172 /*   printf("Cast float to uint16\t%04X\t%04X\r\n", msw, lsw); */
173
174 /*   benchmark_timer_clear(); */
175 /*   benchmark_timer_start(); */
176 /*   f1 = (float)ui1; */
177 /*   benchmark_timer_stop(); */
178 /*   benchmark_timer_read(msw, lsw); */
179 /*   printf("Cast uint16 to float\t%04X\t%04X\r\n", msw, lsw); */
180
181 /*   benchmark_timer_clear(); */
182 /*   benchmark_timer_start(); */
183 /*   si1 = si2*si3; */
184 /*   benchmark_timer_stop(); */
185 /*   benchmark_timer_read(msw, lsw); */
186 /*   printf("MPY(int16,int16)(A=B*C)\t%04X\t%04X\r\n", msw, lsw); */
187
188 /*   benchmark_timer_clear(); */
189 /*   benchmark_timer_start(); */
190 /*   ui1 = ui2+ui3; */
191 /*   benchmark_timer_stop(); */
192 /*   benchmark_timer_read(msw, lsw); */
193 /*   printf("ADD(uint16,uint16)(A=B*C)\t%04X\t%04X\r\n", msw, lsw); */
194
195 /*   benchmark_timer_clear(); */
196 /*   benchmark_timer_start(); */
197 /*   ls1 = ls2*ls3; */
198 /*   benchmark_timer_stop(); */
199 /*   benchmark_timer_read(msw, lsw); */
200 /*   printf("MPY(int32,int32)(A=B*C)\t%04X\t%04X\r\n", msw, lsw); */
201
202 /*   benchmark_timer_clear(); */
```

```
/*   benchmark_timer_start(); */
/*   ls1 = ls2+ls3; */
/*   benchmark_timer_stop(); */
/*   benchmark_timer_read(msw, lsw); */
/*   printf("ADD(uint32,uint32)(A=B*C)\t%04X\t%04X\r\n", msw, lsw); */

/*   benchmark_timer_clear(); */
/*   benchmark_timer_start(); */
/*   si1 += si2*si3; */
/*   benchmark_timer_stop(); */
/*   benchmark_timer_read(msw, lsw); */
/*   printf("MAC(all int16's)(A+=B*C)\t%04X\t%04X\r\n", msw, lsw); */

/*   benchmark_timer_clear(); */
/*   benchmark_timer_start(); */
/*   ls1 += ls2*ls3; */
/*   benchmark_timer_stop(); */
/*   benchmark_timer_read(msw, lsw); */
/*   printf("MAC(all int32's)(A+=B*C)\t%04X\t%04X\r\n", msw, lsw); */

/*   benchmark_timer_clear(); */
/*   benchmark_timer_start(); */
/*   f1 += f2*f3; */
/*   benchmark_timer_stop(); */
/*   benchmark_timer_read(msw, lsw); */
/*   printf("MAC(all floats)(A+=B*C)\t%04X\t%04X\r\n", msw, lsw); */
/* } */
```

E.12 config_regs.c

```
#include <p33FJ128GP802.h>

// CONFIGURATION REGISTER INITIALIZATIONS
_FBS(RBS_NO_RAM & BSS_NO_FLASH & BWRP_WRPROTECT_OFF);
_FSS(RSS_NO_RAM & SSS_NO_FLASH & SWRP_WRPROTECT_OFF);
_FGS(GSS_OFF & GWRP_OFF);
_FOSCSEL(IESO_OFF & FNOSC_PRIPLL );
_FOSC(FCKSM_CSDCMD & IOL1WAY_OFF & OSCIOFNC_OFF & POSCMD_XT);
_FWDT(FWDTEN_OFF);
_FPOR(ALTI2C_OFF & FPWRT_PWR128);
_FICD(JTAGEN_OFF & ICS_PGD1);
```

E.13 dspic.c

```
#include <p33FJ128GP802.h>

void dspic_init(void)
{
  // Disable Watch Dog Timer
  RCONbits.SWDTEN    = 0;

  // Setup PLL for 40MIPS with an 8MHz crystal:
  CLKDIVbits.FRCDIV  = 0;   // Yields divide by 1
  PLLFBD             = 36;  // M=38 --> PLLDIV=PLLFBD=M-2=36. This gives
      (8MHz*38)/(2*2)=76MHz --> FCY=38MIPS.
  CLKDIVbits.PLLPOST = 0;   // N1=2
  CLKDIVbits.PLLPRE  = 0;   // N2=2

  // According to FRM 7.7.2, it is good practice to wait for the PLL to lock after changing
      PLL configuration before executing other code:
  while (OSCCONbits.LOCK != 1) {}  // Wait for PLL to lock
}
```

E.14 dspic_adc.c

```
#include <p33FJ128GP802.h>
#include "dspic_adc.h"

void dspic_adc_init(void)
{
  // Since we don't have any ADC's configured yet, just set all pins as digital pins, rather
       than analog pins!
  // We must do this explicitly, since everything is configured as analog by default.
  AD1PCFGL = 0xFFFF;
}
```

E.15 gpio.c

```
#include <p33FJ128GP802.h>
#include "gpio.h"

void gpio_init(void)
{
```

```
  // Initialize general-purpose i/o pins.

  // Set up LEDs:
  TRIS_LED_0 = 0;  // Set to output.
  TRIS_LED_1 = 0;  // Set to output.
  LED_0 = 0;       // Leave LED off initially.
  LED_1 = 0;       // Leave LED off initially.
  // Set up DIPs:
  TRIS_DIP_0 = 1;  // Set to input.
  TRIS_DIP_1 = 1;  // Set to input.
}
```

E.16 stack_and_heap.c

```
#include <p33FJ128GP802.h>

// These assembly instructions allocate space for a stack and a heap.
// It's more convenient to define the stack and the heap in the source
// code, rather than having to specify them in the MPLAB GUI under
// project "Build Options" (the "MPLAB LINK30" tab).
// The heap is required for stdio.h (which is used for UART printf).
__asm__(".section␣mystack,stack");
__asm__(".space␣512");
__asm__(".section␣myheap,heap");
__asm__(".space␣256");
```

E.17 status_timer.c

```
#include <p33FJ128GP802.h>
#include "status_timer.h"

void status_timer_init(void)
{
  // Initialize Timer1:
  T1CONbits.TON   = 0;       // disable Timer1
  T1CONbits.TSIDL = 0;
  T1CONbits.TGATE = 0;
  T1CONbits.TCKPS = 0x03;    // prescale: FCY/256. At 40MIPs, this gives us 6.4us per tick.
  T1CONbits.TSYNC = 0;
  T1CONbits.TCS   = 0;       // Timer1 clock source is instruction clock (FCY)
  TMR1            = 0x0000;  // Clear Timer1's current count
```

```
14    PR1             = 0xFFFF;  // Timer1 period register. Triggers interrupt when this count
          is reached.
15    IPC0bits.T1IP   = 1;       // interrupt priority level
16    IFS0bits.T1IF   = 0;       // clear the interrupt flag
17    IEC0bits.T1IE   = 1;       // enable Timer1 interrupt
18    T1CONbits.TON   = 1;       // enable Timer1
19  }
```

E.18 swdelay.c

```
1
2   void swdelay(unsigned int max)
3   {
4     // You'll notice that I can't use the same loop count max values as in the assembly code
          version
5     // because this C code seems to be poorly optimized and runs with more overhead.
6     unsigned int i, j;
7     for (i=0; i < max; i++)
8       for (j=0; j < 0xFFFF; j++)
9       {} // Do nothing.
10  }
```

E.19 traps.c

```
1   // traps.c - Provided by Microchip as a code example/template.
2
3   #include <p33FJ128GP802.h>
4   #include "traps.h"
5
6   // ------------------------------------------------------------------------
7   // Primary Exception Vector handlers:
8   // ------------------------------------------------------------------------
9   // These routines are used if INTCON2bits.ALTIVT = 0.
10  // All trap service routines in this file simply ensure that device
11  // continuously executes code within the trap service routine. Users
12  // may modify the basic framework provided here to suit to the needs
13  // of their application.
14
15  void __attribute__((interrupt, no_auto_psv)) _OscillatorFail(void)
16  {
17          INTCON1bits.OSCFAIL = 0;        //Clear the trap flag
```

```
18          while (1);
19  }
20
21  void __attribute__((interrupt, no_auto_psv)) _AddressError(void)
22  {
23          INTCON1bits.ADDRERR = 0;        //Clear the trap flag
24          while (1);
25  }
26  void __attribute__((interrupt, no_auto_psv)) _StackError(void)
27  {
28          INTCON1bits.STKERR = 0;         //Clear the trap flag
29          while (1);
30  }
31
32  void __attribute__((interrupt, no_auto_psv)) _MathError(void)
33  {
34          INTCON1bits.MATHERR = 0;        //Clear the trap flag
35          while (1);
36  }
37
38  void __attribute__((interrupt, no_auto_psv)) _DMACError(void)
39  {
40          INTCON1bits.DMACERR = 0;        //Clear the trap flag
41          while (1);
42  }
43
44
45  // ------------------------------------------------------------------------
46  // Alternate Exception Vector handlers:
47  // ------------------------------------------------------------------------
48  // These routines are used if INTCON2bits.ALTIVT = 1.
49  // All trap service routines in this file simply ensure that device
50  // continuously executes code within the trap service routine. Users
51  // may modify the basic framework provided here to suit to the needs
52  // of their application.
53
54  void __attribute__((interrupt, no_auto_psv)) _AltOscillatorFail(void)
55  {
56          INTCON1bits.OSCFAIL = 0;
57          while (1);
58  }
```

```
59
60 void __attribute__((interrupt, no_auto_psv)) _AltAddressError(void)
61 {
62         INTCON1bits.ADDRERR = 0;
63         while (1);
64 }
65
66 void __attribute__((interrupt, no_auto_psv)) _AltStackError(void)
67 {
68         INTCON1bits.STKERR = 0;
69         while (1);
70 }
71
72 void __attribute__((interrupt, no_auto_psv)) _AltMathError(void)
73 {
74         INTCON1bits.MATHERR = 0;
75         while (1);
76 }
77
78 void __attribute__((interrupt, no_auto_psv)) _AltDMACError(void)
79 {
80         INTCON1bits.DMACERR = 0;
81         while (1);
82 }
```

E.20 uart.c

```
1  #include <p33FJ128GP802.h>
2  #include <stdio.h>
3  #include "uart.h"
4
5  void __attribute__ ((interrupt, no_auto_psv)) _U1RXInterrupt(void)
6  {
7    cmd_handle_char(U1RXREG); // Handle the received byte.
8    IFS0bits.U1RXIF = 0; // Clear interrupt flag
9  }
10
11 void __attribute__ ((interrupt, no_auto_psv)) _U1TXInterrupt(void)
12 {
13   // We define this TX interrupt function for completeness,
14   // but we don't actually enable TX interrupts in uart_init(),
```

```
15    // so this interrupt never gets executed.
16    IFS0bits.U1TXIF = 0; // Clear interrupt flag
17  }
18
19  void uart_init(void)
20  {
21    // Sets up UART1 for 9600-8-N-1.
22    // ----------------------------------------------------------------
23    // Notes:
24    //   REMAPPABLE PERIPHERAL SETUP:
25    //     PICkit2's UART receive is pin 4 of the ICSP connector (which is also used for PGED)
26    //     PICkit2's UART transmit is pin 5 of the ICSP connector (which is also used for
          PGEC)
27    //     dsPIC's UART receive is pin 5 (RP1) (which is also the dsPIC's PGEC1)
28    //     dsPIC's UART transmit is pin 4 (RP0) (which is also the dsPIC's PGED1)
29    // ----------------------------------------------------------------
30    // Set up remappable peripherals:
31    RPINR18            = 1;      // Set up the dsPIC's remappable pin for the dsPIC's UART1
          RX.
32    TRISBbits.TRISB1   = 1;      // According to datasheet, must manually set TRIS to input
          for the UART1 RX. (However, do not set TRIS for TX; chip takes care of that
          automatically.
33    RPOR0bits.RP0R     = 3;      // Set up the dsPIC's remappable pin for the dsPIC's UART1 TX
34    // Turn off UART while we configure it:
35    U1MODEbits.UARTEN  = 0;
36    U1STAbits.UTXEN    = 0;
37    // Configure U1MODE bits:
38    U1MODEbits.USIDL   = 0;      // want continue in idle mode
39    U1MODEbits.IREN    = 0;      // want to disable the IrDA (infrared encoder/decoder)
40    U1MODEbits.RTSMD   = 1;      // want simplex mode (not flow control mode)
41    U1MODEbits.UEN     = 0;      // want RX,TX enabled; want CTS,RTS disabled
42    U1MODEbits.WAKE    = 0;      // want to disable wakeup (since we never sleep)
43    U1MODEbits.LPBACK  = 0;      // want to disable loopback
44    U1MODEbits.ABAUD   = 0;      // want autobaud disabled (since we control baud ourselves
          and we don't want to have to always send 0x55 characters)
45    U1MODEbits.URXINV  = 0;      // want receive polarity such that idle state is '1'.
46    U1MODEbits.BRGH    = 0;      // want 16 clocks per bit period
47    U1MODEbits.PDSEL   = 0;      // want 8 bits, no parity
48    U1MODEbits.STSEL   = 0;      // want 1 stop bit
49    // Configure U1STA bits:
```

```
50    U1STAbits.UTXISEL0 = 0;       // want interrupt when char is transferred to the tx buffer
          (implies there is at least one char in tx buffer)
51    U1STAbits.UTXISEL1 = 0;       // (continued from above)
52    U1STAbits.UTXINV   = 0;       // want transmit polarity such that idle state is '1'.
53    U1STAbits.UTXBRK   = 0;       // want to disable sync break
54    U1STAbits.URXISEL  = 0;       // want interrupt when char is received into the receive
          buffer (there is one or more characters in receive buffer)
55    U1STAbits.ADDEN    = 0;       // want to disable address detect mode
56    // Transmit interrupt
57    IPC3bits.U1TXIP    = 0;       // setting transmit interrupt priority to 0 disables
          interrupt (FRM 6.1.5)
58    IFS0bits.U1TXIF    = 0;       // want to clear the transmit interrupt flag
59    IEC0bits.U1TXIE    = 0;       // want to disable transmit interrupts
60    // Receive interrupt:
61    IPC2bits.U1RXIP    = 4;       // want to set receive interrupt priority level to medium
          priority level (1=lowest; 7=highest)
62    IFS0bits.U1RXIF    = 0;       // want to clear the receive interrupt flag
63    IEC0bits.U1RXIE    = 1;       // want to enable receive interrupts
64    // Configure the baud rate generator:
65    //      When BRGH is set to 0 (which is what we have), use the formula on p.10 of FRM 17.3:
66    //      U1BRG = FCY/(16*baud_rate) - 1 = 38MIPS/(16*9600) - 1 = 246
67    U1BRG = 246;
68    // Finally turn the UART back on:
69    U1MODEbits.UARTEN  = 1;       // want to enable the UART
70    U1STAbits.UTXEN    = 1;       // want to enable TX pin for control by UART (rather than for
          control by PORT). Must enable UTXEN only after UARTEN is enabled.
71  }
72
73  static unsigned char cmd_buffer[CMD_BUFFER_LENGTH];
74  static unsigned char cmd_length = 0x00;
75  static unsigned char cmd_flag   = 0x00;
76
77  void cmd_handle_char(unsigned char ch)
78  {
79    if (cmd_flag) // ignore new input if there's a pending command
80      return;
81
82    if (cmd_length >= CMD_BUFFER_LENGTH) // if buffer already is full
83      cmd_length = 0x00;  // flush buffer and start over.
84
85    switch(ch)
```

```
86    {
87      case 0x1B: // escape character flushes buffer and starts over.
88        cmd_length = 0x00;
89        break;
90      case '\r': // carriage return latches command
91        cmd_flag = 0x01;
92        break;
93      case '\n': // newline latches command
94        cmd_flag = 0x01;
95        break;
96      default:   // for all other characters, store the values
97        cmd_buffer[cmd_length] = ch;
98        cmd_length++;
99        break;
100   }
101 }
102
103
104 unsigned char* cmd_get_buffer(void)
105 {
106   return cmd_buffer;
107 }
108
109 unsigned char cmd_get_flag(void)
110 {
111   return cmd_flag;
112 }
113
114 void cmd_clear(void)
115 {
116   cmd_length = 0x00;  // flush the command buffer
117   cmd_flag   = 0x00;  // clear flag to indicate command has been processed
118 }
```