ECE216 Project: Software Based Open Loop Motor Control And More!

John K. Luebs

September 11, 2009

Abstract

Highly integrated microcontrollers such as the Microchip PIC16 series offer a relatively inexpensive way to do some powerful operations with minimal external hardware.

In this report, a demonstration of basic microcontroller operations involving analog input, LCD display output, and firmware based stepper motor control are documented. The objectives of the hardware design were simplicity and exploitation of the microcontroller's integrated features and the use of mainly very commonly available components. Parts such as the motor drive can easily be replicated with discrete components to perform a similar function.

The firmware design is entirely written in assembly language with judicious use of macros and demonstrates the efficiency and code compactness that can be achieved with a relatively primitive microprocessor. It also presents a broad coverage of embedded assembly implementation and design techniques. Both the coding and reading, if not the world's best source code were at the very least quite instructive to the author in writing multifaceted, compact embedded programs.

A review of the design process with some lessons learned is also made.

1 Description

The initial specification for this project was for a hardware design with a basic demonstration of electro-optical sensing and motor control output in response to sensor input along with some basic user feedback (blinkenlights). The final design based on this specification was a hardware design with the following features:

- A differential light sensor using CdS cells and producing a buffered analog output for connection to the microcontroller.
- A very inexpensive bipolar stepper motor with relatively coarse 7.5 degree step angle controlled by software and setup for PWM based microstepping.
- Optical limit switches for the motor drive taken from surplus parts.
- A user input interface using 4 momentary switches
- User feedback provided by a 2x16 LCD with controller and 4 LEDs driven by the microcontroller.

The project implementation is on a breadboard and a quickly prototyped control panel for display of user feedback and input.

1.1 Demonstration of Microstepping

The basic idea for software control of microstepping came from Microchip application note AN906, however the implementation provided was for a more limited device and did not take into account some non-idealness of open-loop control. The motor control source code presented with this project is pretty much an original work.

The fundamentals of an open loop microstepping control and stepper motor physics are documented in an accessible way by Douglas Jones of the University of Iowa CS department, available at `http://www.webcitation.org/5jfBSLxHa`.

The method used here is a degenerate form of sine-cosine microstepping of a low-performance bipolar stepper motor where one pole is kept in a fixed (100% on) state while the opposite is pulse width modulated proportional to the cosine of the desired angle. This provides maximum holding torque which was important for the low torque, high resistance motor used.

2 Objective

The primary objective of this project was as a learning tool and demonstration of a number of features that are common to most modern microcontrollers.

Hardware design emphasizes simplicity and use of commonly available parts, over a more finely tuned design with careful attention paid to part application and sourcing. Another lower priority (but not disregarded) in the design is careful design for low or battery powered operation.

3 Hardware Overview and Operation

This section runs through the electrical hardware design and roughly follows each page of the attached schematic.

Along the way there are comments and rationale on some hookup decisions.

3.1 Microcontroller

The high level of integration in the PIC16F88 series makes the microcontroller circuitry almost trivial.

The main external components are a 20MHz ceramic resonator which allows for the fastest rated operation of the part and a 32.768kHz watch crystal with drive capacitors. The demo program does not utilize the full capability of this speed. This is somewhat due to the fact that interrupt driven routines are used to improve efficiency. The one weak area of the PIC16 series is any kind of multi-step math operations, especially those requiring any kind of division. To allow flexibility for larger sized division operations, the fast rated operation is useful. For many applications, even battery operated ones, the extra power requirements for 20MHz operation are immaterial. This is especially the case if the MCU power-down features are used.

The watch crystal was not in the first design. It was added for two reasons: One is to give greater low-end range in the rate of the motor control interrupt. The other is that it offers a more flexible way to implement power-down mode functions with low-power oscillator based wakeup. This PIC16 series also offers an RC based ultra-low power wake-up facility that was not utilized.

3.1.1 Pin Assignment/Package Considerations

The 40-pin PIC devices give a good deal of flexibility in pin assignments. It is more than just having 12 extra pins. With the smaller devices, oftentimes the alternate functions for the various peripheral modules can lead to difficulty in designing optimal pin assignments, or prevent effective use of all the pins. By the time fixed alternate function pins are assigned, there either might not be enough GPIO lines. Even if there are enough pins available, the alternate functions being assigned to certain ports may require splitting up natural data or address busses to external components and thus complicating the software.

A 28-pin PIC 16F886 was examined and it was determined that connecting both an LCD display and the motor drive control lines would be possible but extremely cumbersome for firmware.

The operator buttons are connected to Port B of the device because these lines feature built-in in toggleable weak pullups. This means that external resistors are not required on these pins. The motor limit switches connect to this port for the same reason. `RB6/ICSPDAT` and `RB7/ICSPCLK` were left unconnected to allow an in-system programmer or debugger to be connected without interference from other circuitry. The

`RB3/PGM` line was assigned to a button, which means that low-voltage programming cannot be supported unless this is modified.

3.2 User Interface

The user interface consists of an 2x16 character mode LCD display connected with an industry standard HD44780 LCD controller. This chip is extremely easy to operate. The only drawback for some designs is the fact that its 8-bit interface requires 11 pins. Reserving the data lines on one port of even a 48-pin PIC part is impossible if certain PWM alternate functions are used, or it is just very constraining. The HD44780 has a 4-bit interface mode, that requires only a slightly more complex software interface.

The interface includes 4 LEDs for feedback of various program states. These are connected directly to the microprocessor with no interface hardware except current limiting resistors. The LEDs are connected with resistors so that the maximum current sink is well within the maximum total rating of the PIC part. This is calculated from the rated lowest forward voltage drop of the various LEDs used, so that a big enough resistor is used. High intensity LEDs can typically be used with larger resistors for better light output efficiency at the expense of some consistency over the operating temperature range.

Finally, there are 4 button inputs. In the prototype design, cheap buttons with atrocious make-break bounce characteristics are used. The software takes care of the bounce, and external circuitry is not necessary to correct for bounce. Although external pullup resistors can be connected, if necessary, the PIC16F88 series has individually controllable weak pullups on its PORTB pins, which are perfect for this application.

In the prototype, the controls were placed in a separate box and a DB-15 connector was used for neatness and easy transport. This setup is not a suggestion of an appropriate production design. With local connection to the MCU, the user interface components shouldn't need external circuitry. In a production design with a "remote" connection with some user accessible connector, appropriate buffering circuitry is usually needed.

3.3 Motor Drive

The primary consideration in the design of the motor drive was part cost and availability. There are number of interesting closed-loop step motor driver ICs, such as those from Allegro Microelectronics. However, these parts are not commonly available in a part bin, they are bit expensive, and closed-loop operation requires an appropriate current sense resistor in the high-side drive circuit (either "hard" to find, or delicate and expensive).

A TI SN75440 quad Darlington-driver (designed for motor drive applications) IC was chosen because it is relatively easy to come by, cheap, requires no external diodes, and is simple to interface. It is similar to the L298 bridge-driver. If none of these parts are available, the functionality can easily be replicated with discrete logic and a Darlington-driver, or some MOSFETs.

To allow various types of microstepping current control, the driver IC is connected so that both the driver enable lines and the base drive can be pulse-width modulated. This PIC16F88 series has a PWM output "steering" feature that allows directing the single PWM output to different pins without external circuitry.

3.4 Sensor Circuitry

The differential light sensor consists of nothing more than two CdS cells connected in a voltage divider configuration. Ideally, when each CdS cell receives an equal amount of light, the output voltage will be exactly 50% the supply voltage. CdS cells are available over a wide range and even two of the same rated value will have dissimilar response, so the equal light case will give an output somewhere in the middle, but only within 10% or so of ideal. In some instances this may not matter, and in any case it can be adjusted in hardware or software. Since this sensor is to be used with a microcontroller based system, no external adjustment component is provided.

The components used in this case are rated for the 1k to 10k decade, although higher values should work just as well.

If the impedance of the cells is low enough, they could be connected directly to an ADC input on the MCU. However, with 100k valued cells, the integration time would become quite long for the ADC in a PIC 16F MCU, especially in newer parts with higher resolution and larger hold capacitors. For this reason, it is a good idea to buffer the sensor to allow a lower impedance. This also makes interfacing more versatile.

There are now a large number of inexpensive low-voltage rail-to-rail op-amp/buffers that could be put to this purpose, but none of them seem to be a common staple of part bins like the 741, LM311, or LM321. If such a newer component is available, this should probably be used and then the entire circuit can be driven with 5V and ground.

In this case, the very common LM321 is used. This is an older and common bipolar opamp with ability for single-supply operation. In many simple opamps, hitting the supply rails can cause phase-inversion. This makes using these with a single-supply quite difficult, especially if the output will reach near ground in normal operation. The LM321 doesn't have this problem, and the output can reach the lower rail, and can saturate without any inversion. However, the output can only get to 1.5V below the upper supply. For the application of this demonstration, this actually might not be a big problem, as with a 5V supply, values around 2.5V will still be useable. However, the response is quite asymmetric and the flexibility of the sensor is greatly diminished.

In order to use the entire ADC range, the CdS cells are connected with a 5.1V zener diode and a current limiting resistor which works as a simple, albeit slightly inefficient way to run the sensors from a 12V input and only operate across 5V. The entire board is now powered with 12V. A 1k resistor is placed on the output of the opamp in series, in order to limit current and protect the ADC in case the opamp drives too high a voltage.

3.4.1 Choice of optoelectronic

There are two basic choices for an optoelectronic detector: a photodiode or a photoresistor. A typical photoresistor is a CdS (cadmium-sulfide) cell, so both are technically semiconductors although their operation and application are quite different.

A photodiode for light detection purposes is typically reverse biased which allows the response time of the diode to be relatively fast. A photodiode in this manner can be used to read information modulated in light at a very high rate. Even with relatively unsophisticated circuitry, a data rate of 100 kb/s is easily achieved. In any case, a bare photodiode requires circuitry to amplify its tiny signal current into a more useful level or a voltage signal. For simple switching applications, a phototransistor can sometimes be used directly, which automatically has a high builtin current gain.

A CdS cell acts as a light sensitive variable resistor. Compared to a reverse-biased photodiode it is quite slow. However CdS cells have a couple advantages in certain applications: For one, they have a much broader response spectrum in the visible range, so their "sensitivity" is more inline with human vision in terms of perceived light levels. Another advantage is their simplicity, since they are as easy as hooking up a resistor. Finally, the slow response works like a very low pass filter which is the desired effect in many light level detection applications.

4 Firmware Design

The only initial specification for firmware design was that it be implemented using the macro assembler. This helps demonstrate the machine level considerations of the MCU and also allows much greater flexibility on the PIC16 series.

The PIC16 CPU core has a very peculiar design compared to typical processors used as targets for higher level languages. This is probably due to its heritage as a deeply embedded peripheral I/O processor for larger computer systems. It was not originally designed as a general purpose processor. The lack of a general purpose stack for variable passing, and large addressable memory space make it very difficult to implement even a hosted C language environment without numerous limitations. Even with some sophisticated C compiler implementations, code compactness and density suffers.

The final firmware implementation demonstrates a number of techniques and priorities:

- Use of interrupt handling and considerations for preventing bugs and race conditions.
- Strong separation of an "application" and long running code and efficient time-critical interrupt handling code.
- Use of conditional compilation to allow selecting different application parameters at assembly time for different hardware implementations.
- Making judicious use of macros and using them to hopefully improve code readability and not add confusion
- Simple use of third party canned routines with minimal modifications, for performing tedious to optimized operations such as multiplication and division.
- Use of inline source documentation techniques in addition to line comments and basic function descriptions.

4.1 Basic Structure

The firmware source is targeted for the MPASM assembler suite, which is overall a decent assembler with few major warts. The macro facilities are not amazing but they are suitable for most things and quite typical of similar MCU targeting products.

This assembler can produce relocatable code that can be linked with an external linker. This can be done to modularize commonly used code and help assemble modularized programs that have decoupled modules with strict interfaces. These are all good things, but there are tradeoffs, including potentially less efficient function calling, greater usage of register space, and the need for more complex tricks for inter-procedure data passing. While for the seasoned developer this may make complex things much simpler, some of the tradeoffs mentioned can be pitfalls for the beginner.

So, the code was "modularized" by separating certain functional units into multiple files that are all included in a top level assembler input file, which also included the demo application code.

The third party math routines provided in Microchip Application Notes (including AN617), were "modularized" in a similar way.

There were plenty of registers for this application, and no shortage for further addition of features, but a method of "safely" conserving register locations across the application globally is shown in the use of the `CBLOCK` definition. A general label name is used in the `CBLOCK` with a size reservation of 0 bytes, and then additional names are listed with the 0 bytes of storage, except for the last one. The names are placed on the same line to clarify this usage.

The application, as completed, currently fits in well under 1 page (2048 words). Even with some of the partially implemented additional features including a 32-bit/16-bit division routine, the code would be around the ball park of 1 page. This easily fits in the PIC16F887 chosen, and the smaller, cheaper version of the part could be considered for this application.

4.1.1 In-Line Documentation

Because the purpose of this project, including the source, is mainly to demonstrate concepts, the comments in some places are slightly more verbose and expository on the function of individual instructions.

In addition to this, a pattern for inline documentation is also used for many entry points and at the "module" level. Entry points that are made to implement top-level "functions" have a comment block preceded by a line consisting of "`;**`". This line can easily be picked up by external text processing tools, and formatted documentation could be generated using a simple program written in a scripting language. This is similar to the design of tools like Javadoc and Doxygen.

4.2 Use of a ubiquitous utility Timer interrupt

For certain MCU applications, a free-running periodic (possibly adjustable) timer interrupt can be useful for a number of things.

For example, PORTB, where the buttons are connected can generate an interrupt on change. However, it is much simpler and costs little in overhead to simply poll the button state and do debouncing processing all in one step.

A key design consideration here was the worst case time taken in this interrupt, and ensuring that a great deal of algorithmic complexity was not included in the IRQ handler. The time taken in this interrupt has a direct effect on the latency and jitter in changing the drive state of the motor control.

4.2.1 Button debouncing

Button debouncing implements a fairly straightforward software key debouncing, inspired by a clocked D-latch and XOR, except there is an n-bit shift register (up to 8 bits with the current code).

Every Timer 0 overflow (about 19.5kHz at 20MHz `Fosc`), the port bits for the buttons are polled. Each button has a single register reserved, and the last reading for each key is set in the LSb of its state register. Using XOR and AND tests with the Z-flag, the registers are checked for having all 1-s or all 0-s. A register bitmap exists for keys that are pressed and another for keys that are released. The interrupt will only set these bits, the application must acknowledge them by clearing them. It works out fairly efficiently, and the response rate/bounce immunity can be controlled reasonably easily.

4.2.2 Simple User-Feedback

Coding LED blinking in delay loops can become quite cumbersome amongst other code. Another way to get simple blink patterns is to simply set the desired blink parameters in a variable, and the interrupt handler will do the port changes as necessary.

This project does the bare minimum, each LED can either be enabled or disabled for blinking in a variable named `led_blink_mask`. Period and timing are not controllable, but these would be trivial to add if they were deemed worthwhile for the application.

4.2.3 Arbitrary Timing of periodic events

The last major thing the TMR0 overflow interrupt is used for is to set flags (essentially shadows of the interrupt flag) and increment another 8-bit counter to allow application code to schedule delays and periodic events without busy waiting.

For example, in light following mode, the sensor and control readings are read as fast as possible, but screen updates need to happen much slower. Using a flag and additional variable, the fast TMR0 overflow interrupt (19.5 kHz) can be used for much slower update of processes. LED blinking is a special case handled directly in the IRQ handler. This is considered OK, because it is quite trivial and low overhead.

4.3 Motor Control

The motor control code is modularized across two files, one including the core functions for state changes and application calls, and the other includes the code that runs in the interrupt handler.

The compare mode of the standard (non-enhanced) compare/capture/PWM (CCP) module of the MCU is used to schedule the motor interrupt. The compare module compares a programmed count value against the counter incremented by the Timer 1 (TMR1) hardware. This timer hardware can be driven by the prescaled master oscillator (Fosc/4) or it can be driven by clocking on the `T1OSC` pin. Because the hardware is run at 20MHz and the prescaler for Timer 1 is small, it is hard to generate low frequencies even though the timer is 16-bits. For this reason, the integrated LP oscillator with a 32.768kHz watch crystal is utilized to give a greater range of frequency control.

If higher step frequencies with higher accuracy is desired it would be possible to add a switch to change the source of Timer 1.

The interrupt handler for the compare match runs through a simple procedure based that implements a continually running process and utilizes flag bits for communication to and from application (non-interrupt) code.

1. Disable (mask) the interrupt source CCP match
2. Check if limit switches reached. If so, then set notification flag and exit.
3. Check if a stop request (flag) is set. If so, then set notification flag and exit.
4. Check if user has set a position target to move to. If they have, then see if that target has been reached and if it has then signal and exit.
5. Update the step position based on the requested direction.
6. Check the new position based on the initialized values for limit switch travel distance and zero (home) position. If the new position is a few phases beyond these values, assume a hardware problem, deenergize motor windings and call an error handling routine.
7. Call the function that sets all motor control pins and PWM based on the new current step position.
8. Reschedule the interrupt based on the programmed period and set the CCP registers. Re-enable the interrupt.

The microstepping feature as described previously uses PWM to set a desired motor winding frequency. The ECCP module is used to implement up to 10-bits of PWM duty cycle resolution at a range of frequencies practical for small motor applications. The demonstration runs with a PWM frequency of 19.5kHz, which seemed to prevent most coil audio emanations and allowed for good current control. The code uses conditional compilation to allow a couple PWM frequencies and resolutions. These can be extended and the selection made with a simple one-line source change (or separate configuration file).

The ECCP enable mode (including output inversion) is exploited to allow easy control of slow-decay output with only one side of the bridge switchable. The ECCP output steering is also used to allow control of all driver PWM “modes” without any external glue logic.

4.3.1 PWM Decay

An issue that arose early in the design was the issue of accuracy in open-loop current control of an actual motor. A motor can be modeled as a combination of ideal components including an inductance, resistance, and current source. However the response of these elements is very complex and varies with static and dynamic friction and other inertia in the application, resonances, and the properties of the motor construction. This is one reason why closed-loop control is so useful and absolutely necessary in certain types of control.

However, for certain applications, especially those with relatively consistent physical characteristics and operating conditions, open loop control is fully serviceable.

With a PWM controlled motor, there are couple of ways to utilize the switching elements (which include the “flyback” diodes and transistors). The terms used in the context of a motor device are “slow-decay” and “fast-decay” based on the way the stored winding energy is discharged. Figure 1 shows the basic current path for the discharging for the two operational modes.

After “charging” the coil with the transistors as shown by the solid line, if both the transistors are “disabled” or shutoff, the current will discharge by flow through the flyback diodes in the opposite direction of charge and the decay will be fast. If, on the otherhand, the driver one side is kept “shorted” to the charging polarity, and only one side is turned off, or shorted, then current will not flow through the flyback diode, and the decay will be slower.

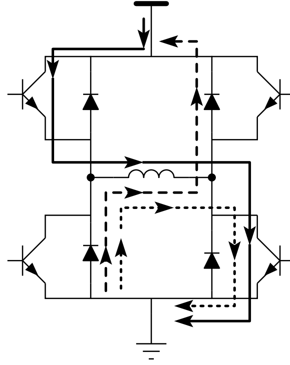

Figure 1: Diagram showing discharge flow (dotted line is slow-decay and dashed line is fast decay) for a charge current shown by solid line

The trade-off between the two is that slow-decay allows more consistent current control across duty cycles, but at higher PWM frequencies the motor response can be limited. There is a patented design used in some driver ICs that switches between the drive modes depending on the state of the pole cycle (increasing or decreasing) and the relative duty cycle. This is called “mixed-decay.”

In this design, the mode is statically switchable and the control code implements the drive logic, but there are no tuned open-loop tables and this mode was not tested. It is felt that perhaps some form of hybrid decay is the only viable improvement from “slow-decay.” Simply statically switching will not improve operation.

4.4 LCD and Text Output

The LCD source is inline documented and its operation is straightforward. The core code shows what a typical “library” of functions for driving an external peripheral would look like. Code to drive different types of I/O peripherals and memory chips would probably look quite similar.

A number of functions to do BCD (binary-coded decimal) conversion to allow decimal output without any complicated and time consuming division routines are shown. A traditional add-3 and shift method as used in programmable logic is used, as well as a very straightforward looped method sufficient for 8-bit numbers. An example of using statically generated code from a script is used to scale (multiply or divide) a number by a fixed constant.

The lack of instructions for direct read/write access to a block of read-only memory (such as program memory) makes implementing display of text strings rather interesting. A module showing a very straightforward method for string storage and display is implemented. The method is not the most storage efficient, but makes the addition of new text strings very simple for the developer. The 14-bit word length of the PIC means that two ASCII characters can be stored in one program memory address. If a great deal of text needs to be stored, then the EEPROM read registers can be used and a “packed” data storage can be used. Implementation of this requires a more complex string output routine as well as more sophisticated use of the assembler.

4.5 Frontend operation

The application that is seen by the user is implemented by mutually exclusive loops that implement each “menu” or operational mode as a state machine.

Each mode is entered by simply jumping to an initialization entry point. The mode operation is then self contained.

One such mode is the Calibration Menu Mode. The screen is shown in Figure 2.

The text along the bottom are labels for commands that correspond to the function of each button.

This mode source code demonstrated handling of a number of issues in processing user input. The text along the bottom changes depending on the mode selected by the button presses. In this mode, the user can see the currently calibrated sensor offset and sensitivity values. The user can then "unlock" the values and adjust them by using a light source on the sensor and using the control knob to adjust sensitivity. After the user has done this adjustment, the settings can be tested without committing them. If necessary, the calibration can be performed repeatedly and then finally saved to non-volatile storage. Code is implemented to prevent the user from testing settings while the controls are "unlocked." The EEPROM programming utilizes artificial delays and the LED output to give operation feedback.

The implementation of this mode in source is rather interesting and instructive even though this particular design from a user perspective was found difficult to use in practice.

Figure 2: Design of Calibration Menu including use of "softkeys" on screen

The other mode implemented is a basic status display application of the light "following." The layout is shown in Figure 3. On the display are shown user-friendly decimal displays of stepper motor positions, frequency, and sensor state.

Also implemented is selection of the light tracking/following mode: It can either use a fixed response speed based on the control knob position, or it can use a speed proportional to the sensor output. The current mode is indicated with an LED.

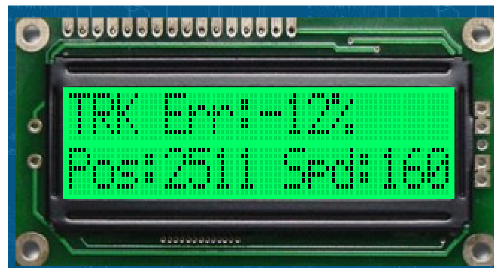

Figure 3: Tracking/Follower Mode Screen Display Elements

4.5.1 Tracking/Following Routine

The tracking application uses a single, simple routine to use sensor readings and make motor commands (`FollowerMotionUpdate_Simple`).

This routine implements a mode that simply changes direction based on sensor input with no control of speed, and a basic speed control based on the sensor difference. The speed control uses a fixed (hard-coded) linear response slope and offset.

Provisions for an adjustable slope using a fixed-point number were started. This involved the use of a 32-bit division routine to compute the period from the adjustable period in addition to a relatively fast 8-bit multiply.

4.5.2 Motor Home Routine

The initial measurement of motor travel, finding limit switches, and position counter reset are handled in a single routine. The procedure used is documented throughly in the source code.

4.5.3 Motor Fault Handler

As described previously, the motor control code optionally checks the assumed position of the motor against what it expects the limits should be. If it is too far out, it assumes something has failed in hardware and shuts down the motor and calls an arbitrarily implemented function so the application can report the error and take some action. This demo displays user feedback including LCD messages and LED flashing. Appropriate precautions are taken for the jump out of the motor interrupt handler, and the reenablement of interrupts. After user notification, the motor system can be reset.

5 Design Comments

The overall design met the basic objective of demonstrating a number of programming techniques for different areas of PIC16 applications, and a broad scope of various issues in embedded design. The source code met the objectives by including much more explanation and commentary then might be normally found in production source code. In general a good rule for commenting might be to comment "what is being done," not how it is done. However, as this source code was demonstrative of techniques peculiar to the architecture, more explanation of how things were done was included.

The source code design is a bit inconsistent in some style conventions and use of certain patterns. Some of the design issues are primarily due to the fact that a concrete idea was not considered up front. Instead the design was much more organic, and different areas of design focus in the software were tried as the software was written. A positive side-effect is that a couple different ways of doing similar things end up being shown.

The "softkey" design concept was directly inspired by its use on HP graphing calculators, although the concept is much older and in ubiquitous use. Although the idea is good in theory and changing labels in a constrained screen space is useful, the exact implementation in the project is quite cumbersome and confusing. Although entirely practical from a user perspective, the source code demonstrates the challenges associated with user feedback and text display on the PIC16.

5.1 Testing and Debugging

A lot of care was taken during code design to ensure code correctness and no improper interaction of interrupt handling and other code. Critical sections and appropriate use of primitive locking were thought about in advance. This probably contributed to an easier system to get working and keep working as more code was added.

Testing was done on both a unit level and then a complete system level. Many of the basic unit tests unrelated to hardware were performed with the simulator. The system test consisted of running the software through all user functions and running certain operations for long periods of time. Some features added near the end were not extensively tested, but the core motor control and demonstration were.

Testing uncovered a number of simple bugs but only one difficult to trace bug. However, it was not a fundamental logic bug, but a silly one due to lack of experience with the platform. The bank select bits were not saved in the interrupt handler, even though bank switching had to be performed for PWM hardware control. This manifested as an occasional "crash" in the PWM output, visible as erratic switching on the PWM output lines. A number of causes were considered, such as latchup or oscillations on the LCD lines

during tristating (which are on the same port), before the cause was found by careful reading of sequence of events.

5.2 Assembler Use

The assembler macro facilities were used for 3 things:

- Make certain very frequent operations that are generally useful to any program, such as register to register transfers possible with a single line. Basically implementing "missing" pseudo-instructions
- Codify patterns specific to the program that are repetitive or involve operations that should be done at compile time (such as calculations)
- As a workaround for the limited call stack of the part.

The use of some of the pseudo-instructions built into the assembler was begun after reading some code on the Internet, not from reading the MPASM manual. This was only done after the fact. The manual page gives a warning that these "pseudo-instructions" are legacy and should not be used. The author doesn't entirely agree with this blanket statement. A general avoidance of any type of pseudo instruction essentially prevents any use of any type of macro facility. While it is true that macros can be abused quite easily to convolute code and can hide complex or expensive operations behind a single innocent looking line, they are very helpful for certain assembly operations that become very repetitive and error prone.

This issue was drove home when the author wrote an `lcall` macro to handle code page (high bits of PC) selection for code that might be in different memory "page". The assembler gave an error, and it was found out that a pseudo-instruction of the exact same name (a long call) with the exact same functionality was already implemented.

5.3 Design Commentary Summary

If there would be one (non-exclusive) overall negative critique of the project that I might readily agree with, it is a lack of focus with the expense that some parts of the design might not be as fleshed out in enough detail as they should be. While this is quite defensible, I would counter that most of those failings in the design were reviewed carefully and they can all be chalked up to the author's own learning process and improving the author's abilities. Most of the project's failures were the author's successes in the long term.

6 Further Directions/Extensibility

Some of the code in the project, such as LCD control and key press debouncing give a decent "application framework" for easy extensibility to the features provided. Adding a new "application mode" is quite trivial. In fact, another mode was planned, but it was becoming apparent at the end that implementing, testing, and debugging a new mode and its interaction with the rest of the system would take too much time away from things like improved documentation and commenting of what was already there. But, for example, a diagnostic/debug menu would be quite nice to have. The hacks for debugging in there now got a bit cumbersome to use a short while after the code base grew at a quicker rate.

The motor control can be used as a base for experimentation in slightly more sophisticated open and even closed loop control schemes.

One simple change would be to switch the Timer 1 source depending on frequency range to allow greater accuracy across the entire range of useable step frequencies, including high frequencies for higher performance motors.

7 Conclusion

The project met basic objectives for a demonstration of microcontroller input/output and various software design features and priorities peculiar to this type of embedded design.

Hardware design was quite simple, but this illustrates the power of the integration available even in a very low-cost microcontroller.

The firmware and source code, in addition to demonstrating the rote use of the PIC16 CPU and simplest peripherals, also showed the practical use of some of the more advanced features such as the ECCP module PWM and steering features. It also showed the use of patterns for implementation of deeply embedded assembly level source, organization, and documentation methods. The firmware project in its entirety includes multiple foci, and is not a compact representation of a single concept. Although this might diminish its utility as a review tool somewhat, it was also a better learning experience for the author.

References

[1] [Jones] Jones, Douglas W. *Microstepping of Stepping Motors*. Retrieved on 1 Aug 2009. `http://www.webcitation.org/5jfBSLxHa`

[2] [AN906] *AN906: Stepper Motor Control Using the PIC16F684*. 14 Nov 2006. Microchip Inc.

[3] [AN907] *AN906: Stepping Motor Fundamentals*. 23 Feb 2004. Microchip Inc.

[4] [AN544] *AN906: Math Utility Routines*. 26 Aug 1997. Microchip Inc.