ALiCE II:
PIC Microcontroller for a Line-Crawling Robot Designed for Manitoba Hydro

Maciej Borkowski
Christopher Henry
Dan Lockery
Peter Schilling
James F. Peters, Supervisor
{*maciey,dlockery,chenry,jfpeters*}*@ee.umanitoba.ca*
University of Manitoba
Computational Intelligence Laboratory
ENGR E3-576 Engineering & Information Technology Complex
75A Chancellor's Circle
Winnipeg, Manitoba R3T 5V6
Tel.: 204.474.9603
Fax.: 204.261.4639

Submitted 3 July 2005, revised 30 March 2006

UM CI Laboratory Technical Report
Number TR-2006-010
April 8, 2006

University of Manitoba ALiCE II

Computational Intelligence Laboratory
URL: http://www.ee.umanitoba.ca/research/cilab.html

Project no. 2006-010

ALiCE II:
PIC Microcontroller for a Line-Crawling Robot Designed for Manitoba Hydro

Maciej Borkowski
Christopher Henry
Dan Lockery
Peter Schilling
James F. Peters, Supervisor
{maciey,dlockery,chenry,jfpeters}@ee.umanitoba.ca
University of Manitoba
Computational Intelligence Laboratory
ENGR E3-576 Engineering & Information Technology Complex
75A Chancellor's Circle
Winnipeg, Manitoba R3T 5V6
Tel.: 204.474.9603
Fax.: 204.261.4639
Submitted 3 July 2005, revised 30 March 2006

CI Laboratory TR-2006-010

April 8, 2006

Abstract

The is the first of a series of research reports that were begun in 2005, and are now being completed in preparation for the completion of the first phase of the ALiCE II project. This report focuses on introduction of a Peripheral Interface Controller (PIC) microcontroller in the design of the line-crawling robot affectionately named ALiCE II. PIC was originally an acronym for Programmable Intelligent Computer introduced by General Instruments in connection with its PIC1650. ALiCE II is an acronym for Autonomous Line-Crawling Equipment II, a second generation version of a new family of autonomous line-crawling robotic devices using swarm intelligence system engineering design principles introduced during the past 3 years. ALiCE II represents the combined efforts of Maciej Borkowski, Dan Lockery, Christopher Henry (alpha order) with some help from Peter Schilling during the summer of 2005. The main architect of ALiCE II has been Dan Lockery. ALiCE I was a single line-crawling robot designed by Vitaliy Degetyarov in 1999 as part of his M.Sc. project, which was also funded by Manitoba Hydro.

This research work has been funded by Manitoba Hydro.

Contents

List of Figures

1 Introduction

This research report focuses on discoveries made recently and work being done on the design of a line crawler robot. By contrast with our original controller design, we have changed our approach for serial communication, specified and implemented control of a dc motor, and set up control for the servo motors. Each of these steps has been researched, built (as required) and gone through the verification steps to ensure proper operation. In addition to the control aspects of the line crawler, we have continued on with the physical construction of the robot and encountered some interesting problems which will also be discussed in this report.

2 Serial Communication

The first area discussed in this report is to address the serial communication link with the PIC microcontroller, the problems encountered and the steps taken to resolve those problems. Several different approaches were attempted, including different compilers, software and hardware control of the serial link. The final solution is discussed in detail and it has been tested and verified to ensure proper operation and interfacing with a variety of devices.

Initially, we were having great difficulty in getting the serial communication link operational. One of the main problems encountered was that after some preliminary research, I had decided to make use of a compiler that provided 'canned' routines for serial communication. After specifying several bit values in key registers, it should have operated the onboard hardware UART of the PIC 16F871 that we are currently using. This appeared to be a nice reliable and conveniently easy to use implementation for a serial communication link. Unfortunately, when using code provided by other individuals, it is necessary to reproduce their exact situation or often the code will not work properly. This was discovered in our setup since we were unable to get the serial link working properly using the code generated by PICBasic PRO [8]. After spending time examining the link parameters and what is required for preliminary setup as well as maintaining communication, we decided to abandon PICBasic PRO. The reason why is that there was not enough flexibility provided and at that stage we were unsure of some of the individual requirements for proper communication since we had been unable to get things working up to that point.

The first major change that was employed when looking at the serial communication link was to switch to a compiler that allows greater flexibility and doesn't require a substantially large cash commitment to start with. After spending a bit of time researching some of the PIC enthusiast websites, I came across the CC5X compiler by B. Knudsen Data [2]. The two most important points that were attractive about this compiler were the facts that it is a free compiler to use and it offers a great deal more flexibility than some of the 'Basic' language compilers. In addition to these points, it also offers a feel much more like traditional 'C' programming. Since a large portion of the project's interfacing software is written in C or C++, it is also helpful in keeping things on a similar platform. The CC5X compiler provides support for most of the common 14-bit core devices, including the 16F871, which is the device we are currently using.

The next major step in getting the serial communication link to work properly was to dissect the requirements for establishing a software controlled serial communication link. Since we were working on a software communication link, we had to decide on a set of parameters for the link. When I was examining the error rates with varying baud rates, it turned out that using a 9600bps rate provided one of the best error rates based on throughput. We decided to use a standard RS232 communication protocol of 8-N-1, or eight data bits, no parity bit and one stop bit. To set up software control for this particular scheme, the first place to start is to examine the data rate and how it has to be harnessed to ensure proper communication. Using a 9600bps data rate, this implies that each bit must be approximately 104μS in length. The length of each bit is of extreme importance as without proper control over the timing, communication will never work properly

(as experienced with the previous attempts).

The RS232 serial communication link is bi-directional, the first attempt at establishing a serial link was to have the PIC send a byte pattern out to any possible device. This is easier to determine if it is working properly since we will immediately be able to receive the byte pattern. The alternative of using an incoming byte to the PIC would require some means to indicate a successful byte pattern was received. As a result, a single direction was used initially to establish a uni-directional serial communication link. The

Figure 1: RS232 8-N-1 Protocol Diagram CMOS logic level [1]

diagram in figure 1 shows RS232 communication using the 8-N-1 protocol. This diagram is shown using the CMOS/TTL logic levels. A logic '0' or 'space' corresponds to positive voltage and a logic '1' or 'mark' corresponds to negative voltage. As can be seen in figure 1, the start bit corresponds to a 'mark' or logic '1'. As a result, the serial software interface begins with generating a start bit and then the byte patterns are forwarded and finally a stop bit is provided (a logic '0' or 'space'). This pattern repeats for each byte that is pushed out on the serial connection and so long as it obeys the 104μS bit-length rule, the receiving end will see the expected results [4]. To achieve the appropriate bit lengths from this software, delays needed to be inserted and tested. The delays were added via NOP() commands as well as looping for extended delays. The means to test the length of the delays was to use MPLAB SIM, provided in the MPLAB IDE from Microchip [6]. MPLAB SIM allowed us to simulate the 4MHz oscillator controlled PIC and how it would respond to the code that was implemented. Once the appropriate 104μS delay was perfected, code provided from the Spark Fun Electronics tutorial was modified for our system to implement the serial control software. The code that was used is provided in the appendix at the end of the report.

Using the software controlled approach as discussed in the previous paragraph, we were able to get information out from the PIC. I sent out the 'Hi' pattern and after ensuring proper wired connections, we managed to see the expected values from the PIC, receiving the information using a serially connected laptop running hyper-terminal in windows XP. Once the transmit section of the PIC was verified, the next step was to attempt to have it receive information from the serial link and then indicate that it received the correct information. This was one of the most important steps in developing the interface to the micro controller since without a serial link in, we cannot control devices externally. The same method was attempted for receiving data through a standard I/O port pin. With the receive bits though, it was important to remember that once a start bit (logic '1', or 'mark) has been detected, we need to test each bit in the approximate middle of the allowed time for each bit (since each bit lasts approximately 104μS, half of that would be 52μS). The same piece of code from the Spark Fun Tutorials was modified for 9600bps with a 4MHz oscillator [4]. Testing the serial link input proved it to be somewhat unstable, the first few attempts provided no results and it wasn't until a few changes were made and the level converter IC mentioned in the previous set of notes (Linear Technologies Level Converter IC LTC1383) [7] that we were able to see some sign of input present.

Since the serial link was not particularly stable using the software control I decided at this stage that it was important to investigate the use of the PIC's onboard UART. Initially, the PICBasic PRO compiler attempted to make use of the onboard UART but something was not quite working properly in the setup and since we were only using a demonstration version of PICBasic PRO, it was deemed better to drop that attempt. Regardless, the idea of having the timing or 'bit-bashing' taken care of in hardware was quite appealing since it was a bit of a headache to make sure that everything was set properly. The UART has several registers associated with it for transmit and receive capabilities. They pertain to both the receive

(RX) and transmit (TX) operations.

First, similar to the previous software control of the serial link, we started with the transmit section where the PIC is responsible for sending a simple bit pattern. The protocol selected is still the same as discussed earlier (8-N-1 at 9600bps). The RCSTA register contains 8 bits for specifying the receive settings [3]. The most significant bit, B7 is the enable (1 = enable, 0 = disable), we need to enable the serial link, so B7 = 1 [3]. Next, B6 is referred to as RX9, or the bit to set for 9-bit communication (8-data bits and 1 parity bit), since we are not using 9-bit communication, B6 = 0 [3]. Next, B5 is referred to as SREN which is important for synchronous communication only, we are using an asynchronous serial link so B5 = 0 [3]. B4 is CREN, which means continuous receive, as a result, this is enabled, or B4 = 1 [3]. B3 is referred to as ADDEN or address detect enable enables the interrupts when bytes are received [3]. Since we are not interested in having an interrupt driven system, B3 is cleared (B3 = 0). B2 is referred to as FERR, or framing error [3]. The suggested setting for asynchronous communication was to clear bit B2 as it is not necessary (B2 = 0) [3]. B1 is referred to as OERR or overrun error, this can be a bit of a nuisance depending upon how the serial link is used [3]. Since we want to have a seamless one-way serial link from external RS232 devices to the PIC controlled devices, we chose to disable the overrun error (B1 = 0). This ensures that the device will not stop receiving bytes until the error is cleared (this is what happens if B1 = 1) [3]. Since we are not planning on issuing long streams of commands (byte after byte of data), this will never be a problem for us. Finally, B0, referred to as RX9D refers to the 9th bit of data that can be used for a parity bit, we cleared this bit since we are not using 9-bit communication (B0 = 0) [3]. As a result of the associated bit status for each of the bits discussed, the value of RCSTA corresponds to 90 hex. This is the setting required to operate the PIC 16F871's onboard UART properly.

In addition to the RCSTA register, there is a TXSTA register for the TX settings. The most significant bit, B7 is referred to as CSRC, which refers to as the clock source select [3]. Since we are using asynchronous communication, this is a don't care bit and we chose to clear this bit (B7 = 0). Next, B6 is referred to as TX9, this selects 9-bit transmission, since we are using 8-bit transmission, B6 is cleared (B6 = 0) [3]. B5, or TXEN (transmission enable) is set (B5 = 1) to enable UART transmissions [3]. Next is B4, or SYNC (selects UART mode), we chose asynchronous or clearing B4 (B4 = 0) [3]. The bit corresponding to B3 is unused in the TXSTA register so we clear it (B3 = 0) [3]. B2, or BRGH is the high baud rate select bit [3]. We set B2 so that we were using the high speed baud rate generator since it provides better error rate (less than one percent error) when the UART operates at 9600bps (B2 = 1) [3]. Next, bit B1 is referred to as TRMT which is the transmit shift register status bit, this is a read-only bit so we didn't bother setting it (left B1 = 0) [3]. Finally, B0 or TX9D allows for the 9^{th} bit of data to be transmitted, since we are not using 9-bit transmissions, B0 is cleared (B0 = 0) [3]. The resulting bit pattern for the TXSTA register corresponds to 24 hex.

There are a couple of other settings required to make the PIC's onboard UART happy and have it working at the appropriate baud rate and protocol. The TXIE bit must be either set or cleared, since we do not want to use interrupts to begin with, TXIE is cleared (TXIE = 0). Interrupts can be helpful in some cases, but for our purposes, there is no need to use interrupts with the PIC, it is better to let it work on controlling all the devices and poll for information on a regular schedule. Commands will not be issued rapidly enough to fill the 2-byte FIFO in the receive section of the onboard UART [3]. Last, but not least, the SPBRG byte must be set, this specifies the specified baud rate generator to generate the appropriate baud rate. We want to use 9600bps, and with BRGH set to high (as mentioned in the previous paragraph) this means that SPBRG will be set to 25 [3]. This corresponds to 9600bps, and with the other settings for the previous registers, that takes care of all of the different requirements for the PIC hardware.

After setting all of the respective registers and bits to the desired specification, the next step was to write a simple program to make use of the hardware UART. The operation of the UART is as shown in figure 2. To transmit out bytes on the serial port (PORTC, pin 6), simply transmit a byte to the TXREG and it will go through the TSR, or transmit shift register (shifted out serially, bit by bit) [3].

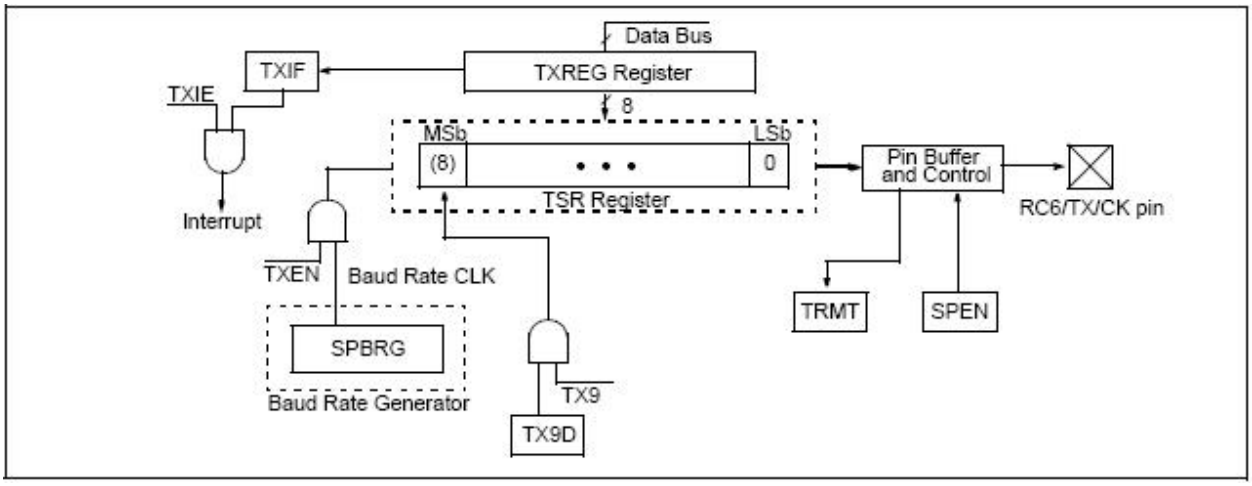


Figure 2: PIC 16F871 UART TX-Section [3]

The software control required to send out bytes on the serial link to a hyperterminal window are extremely simplified now that all the timing is taken care of in hardware. Simply passing bytes in to the TXREG will have them shifted out serially. We connected a laptop running hyperterminal in WindowsXP and were able to see the serial bytes without any difficulty. The important thing to note is that the RS232 level converter was required to obtain proper communication. This is the Linear Technologies device mentioned earlier (the LTC1383).

The next logical step in working on the serial link for the PIC was to move on to testing the RX part of the UART. Previous tests proved that this was a little bit more difficult with the software controlled version, but I suspect that it was due to the requirement to specify all of the parameters, including overrun error. When the overrun error bit is not cleared, no further information can be received and that is likely what was happening and why we were not seeing any information coming in on the serial link. The registers were already set up as well as the baud rate generator, all that remained was to understand the RX half of the UART, which is shown in figure 3. The main difference is that when data is received and the RCIF bit goes high, that indicates that a byte has been received. Once the RCREG is read, the RCIF bit clears and the next byte can be read when it is received. The RX-section has 2-byte FIFO which allows the RX section to store up to two bytes in the FIFO and have a third bit being received before an overrun error occurs. This allows for most situations that we should experience when controlling devices with the PIC for the line crawler robot. The projected protocol is going to use a single byte to issue most commands and as a result, since we will be polling the RCIF bit every 20 milli-seconds approximately, we should never experience an overrun error.

After implementing some simple code to link the serial input to the serial output using the hardware UART system discussed in the previous paragraphs (see RXTest.c in appendix), we successfully managed to communicate through the serial port. The experimental setup was to connect the PIC, PORTC, pin 6 to the RX line of the PC's hyperterminal input, PORTC, pin 7 to the TX line of the PC's hyperterminal output. The connection from the PIC to the laptop was interfaced through the LTC1383 level converter as well to ensure proper RS232 and CMOS logic levels were used. The result was a complete, serially connected PIC that receives bytes from the hyperterminal link and returns whatever it receives to the hyperterminal screen (essentially results in an echoing terminal). After experimenting with this setup a number of times, we were able to ensure that the serial link with the UART-based-timing operated correctly and with excellent stability and an almost non-existent error rate.

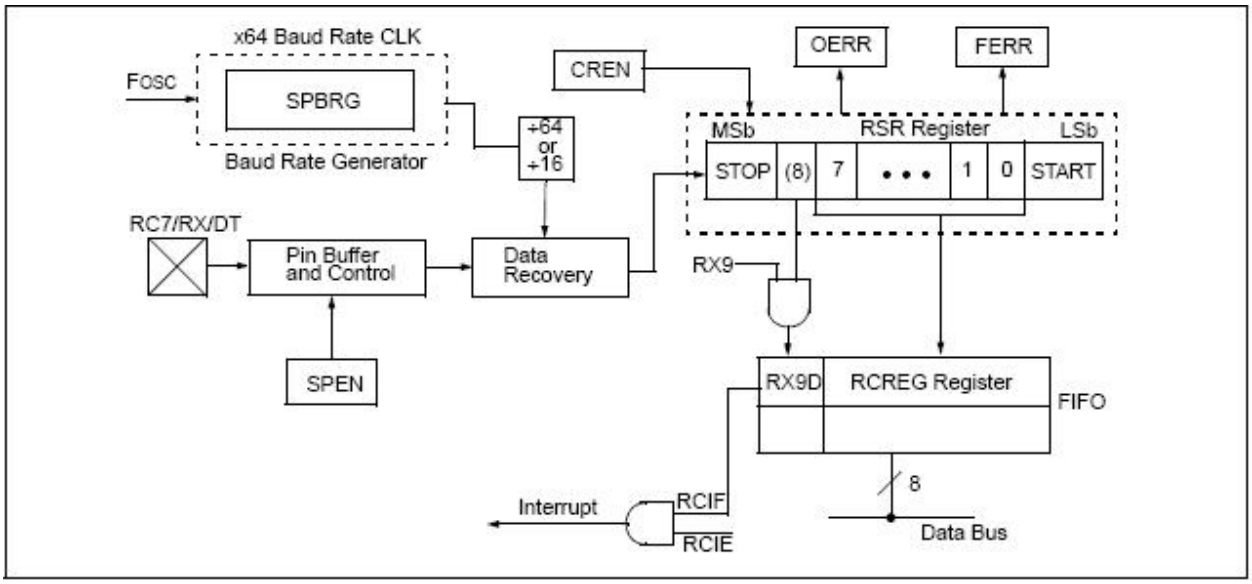


Figure 3: PIC 16F871 UART RX-Section [3]

3 PWM Setup

Further to the previous set of notes where we were specifying a dc motor based on the physical models and the estimated requirements. The next step is to look into PWM control using the PIC so that we can have forward and backward motion as well as braking and a quiescent mode. To achieve this level of control, the Texas Instruments IC was selected since it is an intelligent h-bridge, giving us these features in a nice small packaged IC. The implementation and testing of this chip required a little bit of setup and experimental work. The operation of the chip was not exactly how it was initially perceived to be and as a result we had to make some minor modifications to achieve the level of control that we were hoping for. Once we had the experimental setup in place, we proceeded to test and verify the correct operation of the PWM motor control via the PIC 16F871.

The first step was to put together a simple test rig for the PWM chip once it arrived from Digikey. Figure 4 shows the TI chip mounted on a 'surfboard' which conveniently fits right into a breadboard, allowing us to wire up the connection interface to the PIC and the DC motor. Once the board was soldered, the next step was to wire up all of the power and ground pins as well as the input and output control pins. After completing the wiring for the PWM control circuitry, the next step was to investigate how to go about getting the PIC to control the motor.

Similar to the onboard UART discussed in the previous section of this report, the PIC contains an on-board PWM module as well [3]. Port C, pin 2 is the output for the PWM module for providing a PWM signal to control a DC motor. The TI chip has two inputs, one logic level input (IN2) and one PWM input (IN1). To obtain a PWM signal out on Port C pin 2, we need to configure the PIC to provide this signal. The PWM period needs to be decided upon, and the PWM Duty Cycle is then specified, as well as the resolution that we are capable of. We have a 10-bit resolution PWM output and the following equations describe how the PWM period, duty cycle and resolution all play a part in determining what the PWM module sends out to the respective device.

$$PWMPeriod = [(PR2) + 1] * 4 * T_{OSC} * (TMR2) \tag{1}$$

$$PWMDutyCycle = (CCPR1L : CCP1CON < 5 : 4 >) * T_{OSC} * TMR2 \tag{2}$$

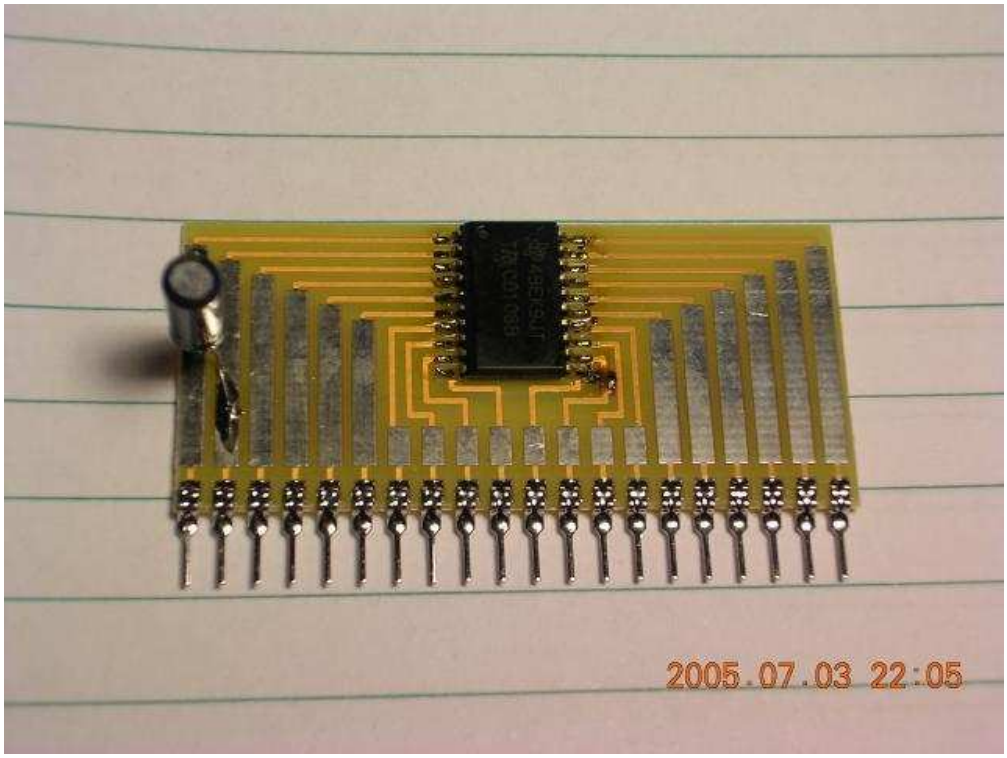


Figure 4: Texas Instruments TPIC0108b Intelligent H-Bridge [10]

$$MaxResolution = \frac{log(\frac{F_{OSC}}{F_{PWM}})}{log(2)} bits \tag{3}$$

We selected a duty cycle of 50% to begin with since it would provide a large enough change in velocity to make a difference from 100%. Keeping this in mind, generating the required bit pattern was fairly simple to calculate, as shown in (2) [3]. The necessary bit pattern is 10 0000 0000 in binary. This is equal to 512 decimal, which is exactly half of 1024, which is 2^{10}, resulting in a 50% duty cycle. For selecting the PWM period or frequency (they are the reciprocal of one another), we decided to employ the slowest possible frequency with the 4MHz oscillator. The maximum delay is achieved through increasing the value of PR2 (it is the only adjustable quantity from (1) above). This means using a value of 0xFF hex for PR2, the prescale value for TMR2 was selected as 16 (possible choices were 1, 4 or 16, with 16 giving the slowest possible frequency) and plugging in these values to equation (1), the reciprocal of the resulting period turns out to be 244Hz which is the PWM frequency. The maximum resolution available to us using these parameters and following (3) provides us with a potential of 14 bits. Keeping these equations and parameters in mind, we went about configuring the appropriate registers and memory locations in the PIC to provide the 50% duty cycle at a rate of 244Hz (see the file pwm.c in the appendix for code listing).

As mentioned in the previous paragraph, there are several registers and bits that need to be set for the PWM module to produce the desired pulse train at the output on port c pin 2. More specifically, we need to set the value for PR2, as discussed already, the value of 0xFF hex provides us with the maximum possible delay or the slowest possible frequency. We were trying to see if we could attain a slow enough frequency to drive servo motors with the PWM signal as well but unfortunately with our oscillator speed we were unable to get anything slower than 244Hz, which is much too fast for the servo motors which require refresh rates closer to 50Hz. The next value which we needed to set was the value for TMR2, which is placed in the T2CON register location. This is the prescale timer value, and we used a value of 16 for it. The next step

was to specify the duty cycle, which was done by entering the value for CCPR1L and for the two bits in CCP1CON (bits 4 and 5). This provides ten-bit duty cycle specification, and as mentioned in the previous paragraph, we used a value of 512 to specify a 50% duty cycle. As a result, CCPR1L contained the value of 128 (1000 0000) and CCP1CON, bits 4 and 5 both contained a 0. The final remaining task was to setup port c appropriately to operate as an output for the PWM module on pin 2. This meant clearing the necessary port c pin, in our case, we cleared the entire port since we weren't using any other pins for this part of the experiment. With these registers configured, the output from the PWM module is activated as soon as the PIC powers up and is operational. The final test to make sure that this worked properly was to hook up the PWM output (port c, pin 2) to IN1 on the TI chip and then manually change the direction of the motor by altering the state of IN2 (connect to ground or power to provide logic levels). The OUT1 and OUT2 pins were connected across the dc motor allowing full h-bridge control to drive the motor. When the PWM signal was connected to the motor we discovered that it did not provide any appreciable difference whether we used a PWM signal or solid logic levels. As a result, it appears that the PWM discussion in the title of the TPIC0108b might be a little bit misleading. Providing logic levels coinciding with the truth table on the data sheet for the TI device provides ample control and as a result, this eliminated the need for using the PWM module. Instead, two output pins on port c are able to control the direction and the braking of the dc motor. The truth table can be seen below in table 1.

Table 1: Intelligent H-Bridge Control Scheme [10]

$IN1$	$IN2$	$OUT1$	$OUT2$	$MODE$
0	0	Z	Z	Quiescent Supply Current Mode
0	1	LS	HS	Motor Turns Clockwise
1	0	HS	LS	Motor Turns Counter Clockwise
1	1	HS	HS	Brake Mode

This table demonstrates that through simple IO control, we can control the dc motor and not even bother with the PWM module. The downside to this type of control is that we will be running at maximum speed (PWM provides some degree of control over the speed of the motor since it gives a varying DC level to the motor), however the Texas Instruments IC that was selected appears to have no difference when the PWM signal is applied compared to when standard logic levels are applied to the inputs. As a result, the control scheme was simplified using table 1 and two output pins. This renders the PWM material ineffective for our motor control. One benefit that we still have working in our favour is that the DC motor is still relatively straight forward to control and since it is a low speed device (with an almost 300:1 gear ratio), the wheels won't turn very fast even when it is at full speed.

4 Motor Control via the PIC

The next step in device control was to include the RC servo motors. These motors are position control servos and they are responsible for positioning the line crawler grip mechanism as well as the camera. The initial approach for control was to use a PICBasic PRO, a 'basic' language compiler for the PIC provided by MicroEngineering Labs [8]. A demonstration version of PICBasic PRO was made available to us on their website (as listed in the references) and it provides nice canned routines for controlling servo motors as well as dc motors. We ended up migrating to a different compiler later on, CC5X, which provides slightly more control from the users perspective and it also uses the C language as mentioned previously. Another problem that was encountered ended up being the generation of control signals to send out to the servos. A general

servo motor has a control signal that is centered at approximately 1.5mS and usually has a pulse width of approximately +/- 0.5mS. The resolution for the step size is much smaller (in the order of microseconds) and as a result, such short delays required special attention. After that point in the work, servo control was verified to ensure proper operation.

The first step that bares mention that we addressed was the problem with using canned routines for controlling the motors. We were able to achieve 10μS resolution with the 'PULSOUT' command provided in PICBasic PRO [8]. In addition to that, a PWM routine provides a pulse width modulated signal out to control the dc motor as well. These were both convenient routines for providing us with a quick fix to control our devices, however since they are fixed routines in the basic compiler, we have no control over the specific details of how they operate. The assembly language file generated was cryptic enough that it became too much work to sort through to find out how to change the specific details. As a result, we began looking for programming alternatives, and it was at that point where we came across CC5X. This compiler makes use of the C language, and it is a complete version that is free to use (not just a demo like the PBP compiler). The flexibility in coding and using the C language is more convenient and comfortable for us since the rest of the code for our project is written in C as well.

Since we opted to use CC5X, this meant that to control servo motors we needed to develop our own routines to provide the control signals (as opposed to PBP which contained the canned control routines). The means to provide control signals is as simple as setting a logical pin high and then at some point later in time re-setting the same pin low. The most important part of the control is in the length of time that the pin is held high for. I wrote an additional delay file for generating a variable amount of delay with an 8.08μS resolution with an error of +/- 0.25%. As a result, we have slightly better resolution and a much better understanding of how the signals are generated since we had to create the operation from scratch (so to speak). The code written for the delays as well as for the rc servo motor control can be found in the appendix (minid.c, and servo.c).

Once the code was written and compiled successfully, the next step was to test and verify the new servo control scheme. We were able to achieve the desired resolution (approximately 8μS) of about 2 degrees of movement per step. The initial control scheme allowed us to move in single step increments from 0 degrees all the way to 180 degrees. We used the Hi-Tec servo that will be used for the upper grip actuator as a test subject. A simple routine (servo.c) was generated to position the servo at 0 degrees and to step through all the way to 180 degrees and then to return back to 0 degrees using the same step sizes. This demonstrated the delay routine as well as the control method and proved that both operated as expected. Future expansion to this method of control will include providing external control to operate the servo motor remotely.

The next step was to extend the control to include the dc motor as well. The approach used was described in the previous section (as seen in table 1). Since only two inputs were required, pins on port b of the PIC were assigned (PB2 & PB3). Simple logic control of these two pins provides us with a clockwise, counter-clockwise, braking, or stopping action of the dc motor. The TPIC IC provided us with the h-bridge circuit to isolate the motor control circuit eliminating problems with back-emf when voltage changes occur with each transition from state to state [10].

Two separate control programs were written to allow single or two byte control over all of the devices. We have included support for 4 servos (at present), and one dc motor. There is still room for expansion should any further devices be required. The reason that two separate control programs were written was due to the fact that the experimental test setup involved using a serial link from a laptop to the PIC. To keep things moving along quickly, simple ASCII codes were used for the preliminary protocol. For example, 'U' indicated moving the servo up, 'D' down, 'F' indicates the dc motor should rotate forward and 'B' backwards, and 'S' stop. These are just a few of the commands included in the single byte control program. The two-byte control program was written specifically for the experimental test setup that included the TS-5500. The code for both of these can be found in the appendix (PICControl.c, and PICControl2.c). After working out the last few bugs, both of these control programs allowed us to operate each of the devices

connected to the PIC, proving that both the control scheme as well as the serial link were operating correctly.

5 Construction of Alice II

Now that a number of different aspects of the control scheme and serial link have been tested and verified, the next step is to move into a more permanent setup. The areas addressed included physical construction of Alice II, moving from breadboarded circuitry to a more permanent proto-board solution, adding electronics and the camera to the structure. Each of the different areas of construction generated its own interesting set of problems to solve that are covered in this section of the report.

One of the first difficulties encountered in the physical construction stage was part of adding traction to the nylon wheels. The original trial run for testing the line grip showed that the nylon wheels have a tendency to slip when the angle of inclination increases above 15-20 degrees. As a result, some solution for the traction problem was necessary. Rather than creating a new set of wheels made out of a different material, rubber strips were added to provide much greater traction. The original design of the nylon wheels left enough room to add approximately 1mm thickness of either some sort of glue or rubber to improve the traction. Figure 5 demonstrates the strips of rubber included to provide the required traction for scaling

Figure 5: Added traction to line grip wheels

the incline of the power line. The main problem encountered with adding traction to the wheels involved using adhesives to attach the rubber to nylon. Several alternatives were attempted, including Weldbond, Krazy Glue, Loc-Tite, Contact Cement, Epoxy, and 2-sided tape. Weldbond and 2-sided tape both proved to be unsuccessful, Krazy glue on its own also failed, it wasn't until we finally switched to Loc-Tite that we stated wondering about priming the wheels first. In order to get a cyanoacrelate based glue (Krazy Glue) to bond properly, we needed to sand the wheels somewhat and make use of a primer. Once the surface was lightly sanded (emery paper, 240 rating) and the primer (Loc-Tite, Super Glue Activator) applied, the rubber

material adhered nicely to the nylon, providing us with the desired traction for scaling the gradient of the sky wire.

Next in the physical construction phase was to update the wiring harness, add the payload for the platform and the new dc motor. The wiring harness had been built already, but it was meant to interface with the handyboard (the previous controller used to operate the line grip). As a result, we needed to extend some wires and shorten others as well as removing the pin headers that were used to interface with the handyboard. The payload for the platform included the TS-5500, the PIC controller, the battery pack and the voltage regulator. The most important step in setting up the payload was to include the TS-5500, mounted on .75 inch standoffs. Space was allowed to plug in the wireless card, the USB cable attached to the camera

Figure 6: TS-5500 mounted in platform

and space around and underneath for the remaining components. Figure 6 shows an image of where the TS-5500 is mounted and that there is enough space around it to attach all of the necessary peripherals. In addition to this, as mentioned already, there is a .75 inch gap underneath where the battery packs will reside as well as the PIC controller and the voltage regulator.

The final remaining item for the platform payload was to add the camera and its control mount that consisted of two servo motors for position control. To eliminate as much weight as possible, rather than using some form of mechanical harness, two servo motors were used as mounts and fixed directly to the underside of the platform. With our adhesive experience, we selected two-sided foam tape to fix the camera/servos in place underneath the platform. Figure 7 shows how the camera and the servos are mounted, allowing us a wide range of vision to acquire images for power line inspection.

The next problem investigated during the construction phase was removing the old dc motor and including the newly specified replacement motor. The new motor was substantially smaller than its predecessor even though it is rated with much greater torque (it is a gear-head motor with a 297:1 gear ratio). Since the new motor was so much smaller, we required a mounting bracket to mate the shaft with the gear train to en-

Figure 7: Camera mounted underneath Alice II platform

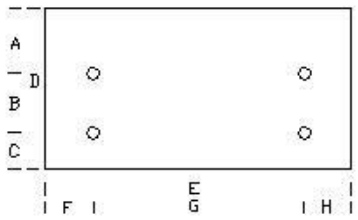
sure proper operation of the locomotion stage in the line grip. The best approach for developing a mounting bracket is to have the machine shop manufacture the bracket that we specified (see figure 8). In the interim whilst we were waiting for the machine shop to manufacture the part for us, we have been making do with a wooden shim as a replacement. This allows us to see how everything will fit together to ensure that the drive train will work properly.

The next stage in construction involved moving from a breadboarded prototype circuit to a perf-board based, more permanent solution. This involved planning a board layout and including pin headers to attach all of the peripherals as well as the communication link to the TS-5500. The main problems encountered during the fabrication of the prototype board were spacing issues based on a large soldering iron tip. Besides that, everything went quite smoothly and after verifying the correct pins were soldered, the board was tested and the end result operated exactly the same as its bread-boarded counterpart. Figure 9 shows the completed perf board version of the PIC controller. This includes all of the IC's, the PIC 16F871, LTC1383 (RS232 level converter), and the TPIC0108B (PWM chip). With the PIC control board tested and verified by running the PICControl programs and ensuring proper operation, this set us up to add all of the devices and the battery packs to the platform payload to see how everything fits together. The space in the platform was sufficient to house all of the individual devices and there still remains room for extra payload as required. Figure 10 shows all of the additional components that comprise the payload for Alice II all placed within the platform.

The last step in the current construction was to add the camera to the platform. This was accomplished by using two servo motors connected together to provide 2 degrees of freedom, allowing us to rotate into position to view a wide enough range to cover all of the different positions necessary to acquire the images of the power grid during the travels of Alice II. The camera mount consists of two rc servo motors necessary for position control as well as 2-sided foam tape. The foam tape is responsible for keeping everything fastened.

Motor Mounting Bracket

- Note: this bracket is specified as aluminum, 0.5cm in thickness (or 0.197 inches thick).



Measurements:

A - 1.5cm (0.59055") in length to centre.
B - 1cm (0.394") in length from centre of hole to centre of hole.
C - 0.5cm (0.197") in length from centre of hole to edge
D - 3cm (1.18") in length from edge to edge
E - 3.8cm (1.5") in length from edge to edge
F - 0.5cm (0.197") in length from edge to centre of hole
G - 2.8cm (1.1024") in length from centre of hole to centre of hole
H - 0.5cm (0.197") in length from centre of hole to edge

Notes:
- All holes need to be tapped for 6-32 machine screws
- Material recommended is aluminum 1100 with 0.5cm (0.197") thickness.

Figure 8: Construction drawing for simple mounting bracket for dc motor

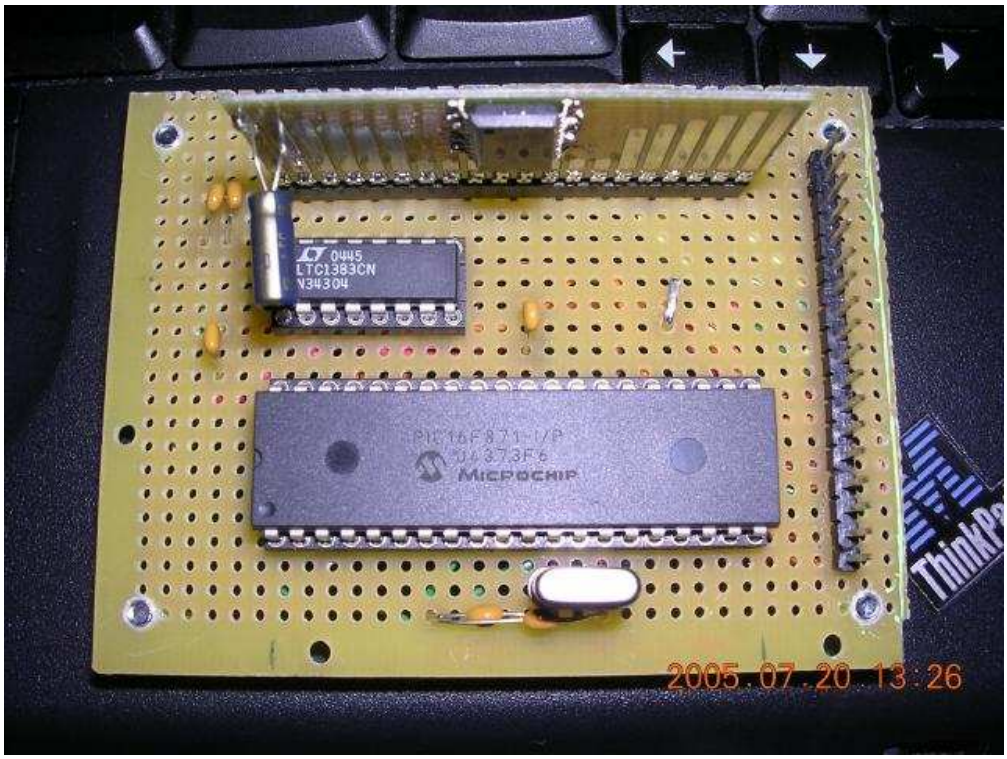


Figure 9: PIC Controller board and peripherals

Figure 10: Alice II payload

This reduced the amount of hardware brackets necessary to mount the camera, reducing the necessary parts and weight. We were still able to achieve the range of vision required even though the camera is mounted directly onto the base of the platform.

This is where we are currently at with the construction of Alice II, there are still a few items left to be done before Alice II is complete. This includes, adding an aluminum motor mount bracket for the dc motor, running the wires to the TS-5500 DIO ports once the sockets and wires arrive. With the addition of these two items, we will have the preliminary physical construction completed and ready to interface with the electronics and software.

6 What happens next

Currently, we are in the process of duplicating Alice II as well as finishing off the first one. Peter is currently fabricating an additional PIC controller board as well as working on finalizing construction of the second Alice II. Once the parts on order have arrived, we will be completing the final assembly of the first Alice II bot. Chris has been working on setting up a software interface between the TS-5500 and the PIC to control the locomotion and position of Alice II. Based on the code provided by Maciek, we have been able to verify the operation of the DIO with the contact switches, we are still in the process of adding support for the infra-red sensors to let us know when the grip closes completely. In addition to that, a proximity sensor, also of the infra-red variety is going to be added experimentally to detect obstacles at a distance since this will allow us greater flexibility in our obstacle avoidance routines. I am going to be working on completing the construction of the first Alice II and setting up the experimental work to test and verify the robot for proper operation. In addition to that, I will be extending the new control protocol to the PIC control programs. Maciek and I were involved in generating the protocol that makes use of a single byte to transmit device

selection as well as instruction for operating each device.

A Appendix: CC5X Code

A.1 minid.c

```
/*

These are some simple delay routines so that I can try to achieve variable delay
times possible with the PIC 16F871 using a 4MHz oscillator. Include this file
when delays are needed since the delay.c include file is not the greatest for
fine resolution.

Date: June 15,2005
Author: Dan Lockery
Version: 1.0

*/

//Note that this works well with a single pass (input of 1)
//The loops require less effort to run multiple times, so the time
//for subsequent delays shortens (making it difficult to have
//consistency). As a result, it has been left as is, call this w/
//input value of 1 and it provides exactly 100 microseconds.

void delay_100us(uns8 inp)
{

int i,l; // establish counter

while(inp ≥ 0)
{
for(l = 0; l ≤ 4 ; l++); // this is to add time delay (100uS)
nop();
inp–;
}

}

void delay_5us(void)
{

nop(); // try one NOP to see what sort of resolution we get

}

void delay_var(uns8 in)
{

while(in ≥ 0)
```

```
in–; // decrement until reaches zero before returning

}
```

A.2 PICControl.c

```
/*

This is an interface for serial control of all devices linked to the PIC 16F871
that we will be using for the Alice II robot. The servo motors will have to be
updated at a rate of approximately every 20ms. Whenever a serial byte or pair
of bytes is received, it is important to take action and adjust the position of
servos or direction/speed of the DC motor in question. This will be handled
after each of the servo control signals are completed and not during since this
would cause unforeseen jerk action and potentially damage the robot or cause
trouble in other ways.

Author: Dan Lockery
Date: June 27, 2005
Version: 1.0

*/

#include c:/Program Files/cc5x/16F871.h // This is the device header file
#include c:/Program Files/cc5x/minid.c // This is the delay file
#include c:/Program Files/cc5x/Delay.c // contains the millisecond delay

void main()
{

// First, set up the serial communication link.
RCSTA = 144; // assign RCSTA 90h
TXSTA = 36; // enables 8-bit asynch communication with high speed BRG
SPBRG = 25; // this sets the baud rate to 9600bps since BRG = 1
TXIE = 0; // this keeps the interrupts disabled
PORTC = 0b.0000.0000;
TRISC = 0b.1000.0000; //0 = Output, 1 = Input - PORTC.7 is the RX of UART for 16F871

// Next, set up PortB as outputs since they will run the motors!
PORTB = 0b.0000.0000;
TRISB = 0b.0000.0000; // This configures port B as all outputs

uns8 pos1; // this is the variable that holds the position of the servo motor
uns16 del; // this is the variable that holds the delay for repeating ctrl signals.
uns8 device; // this holds the device that we are going to be altering
uns8 pos; // this will hold the position of the servo that we want to change (can RX new)
uns8 instr; // this will hold the instruction for what to do based on the device!
```

```
PORTB.2 = 0; // setting pins 2 and 3 low will init DC motor in quiescent mode
PORTB.3 = 0; // quiescent mode now achieved.
PORTB.4 = 0; // start servo motor ctrl signal low, ready for info
PORTB.5 = 0; // start out servo 2 in low position (ready for when used)
PORTB.6 = 0; // start out servo 3 in low position as well.
PORTB.7 = 0; // start out servo 4 in low position as well

del = 20; // this will correspond to a 20 milli-second delay (always)
pos1 = 136; // default position, put servos at this angle to start

while(1) { // loop infinitely to run the motors and receive info from rs232

// To start with, position the servo motors (at the beginning of every loop)
pos = pos1; // assign the position bit with pos1
PORTB.7 = 1; // set servo pin high for control pulse
delay_var(pos); // call variable delay for providing the position
PORTB.7 = 0; // after the delay, put control pulse low

// Repeat for each additional servo. This could mean a potential
// extra time consumption of up to ~8mS for all 4 servos
// keep this in mind when working on the delays.

// now that the servo has been positioned, check for an RX byte?
if(RCIF) { // what to do when we get a pair of bytes serially
nop(); // just in case something was immediately received, let it settle.
device = RCREG; // first byte is the device number
while(!RCIF);
nop(); // again, just in case we need settling.
instr = RCREG;
// Now that we have the device and instruction, decide what to do with them
if(device == '0') { // this means we are controlling the servo
if(instr == '0') { // this means we need to set the motor moving clockwise
PORTB.2 = 0; // set 'In1' to 0 for CW motion
PORTB.3 = 1; // set 'In2' to 1 for CW motion
} // end of inner if for clockwise motor turning
else if(instr == '1') { // this means we need to set the motor moving CCW
PORTB.2 = 1; // set 'In1' to 1 for CCW motion
PORTB.3 = 0; // set 'In2' to 0 for CCW motion
} // end of inner if for CCW motion
else if(instr == '2') { // this means we need to turn the brakes on!
PORTB.2 = 1; // set 'In1' to 1 for braking
PORTB.3 = 1; // set 'In2' to 1 for braking
} // end of inner if for braking action
else if(instr == '3') { // this means we need to turn the DC motor off
PORTB.2 = 0; // set 'In1' to 0 for DC motor off
PORTB.3 = 0; // set 'In2' to 0 for DC motor off
}
```

```
else;
// faulty command, don't do anything
}// end of what to do when the selected device is the DC motor!
else if(device == '1') { // this means we are controlling the RC servo motor
pos1 = instr; // second byte w/ servo control is position
}// end of else if for controlling RC servo motor #1
else;
// else, do nothing

}// end of what happens when a byte is received serially

delay_ms(del); // pause for 20 milliseconds


} // end of inner while loop

}
```

A.3 PICControl2.c

```
/*

This is an interface for serial control of all devices linked to the PIC 16F871
that we will be using for the Alice II robot. The servo motors will have to be
updated at a rate of approximately every 20ms. This version of the PIC control
program is meant to be operated by a user at a terminal with single byte input
instructions. As a result, the servos can only be moved incrementally, either
positively or negatively (depending on the current position). The DC motors
will either be turned on CW, or CCW, have the brake set or stop completely.

Author: Dan Lockery
Date: June 29, 2005
Version: 1.0

*/

#include c:/Program Files/cc5x/16F871.h // This is the device header file
#include c:/Program Files/cc5x/minid.c // This is the delay file
#include c:/Program Files/cc5x/Delay.c // contains the millisecond delay

void main()
{

// First, set up the serial communication link.
RCSTA = 144; // assign RCSTA 90h
TXSTA = 36; // enables 8-bit asynch communication with high speed BRG
```

```
SPBRG = 25; // this sets the baud rate to 9600bps since BRG = 1
TXIE = 0; // this keeps the interrupts disabled
PORTC = 0b.0000.0000;
TRISC = 0b.1000.0000; //0 = Output, 1 = Input - PORTC.7 is RX of UART

// Next, set up PortB as outputs since they will run the motors!
PORTB = 0b.0000.0000;
TRISB = 0b.0000.0000; // This configures port B as all outputs

uns8 pos1; // variable that holds position of the servo motor
uns16 del; // variable that holds delay for repeating ctrl signals.
uns8 pos; // position of servo that change (can RX new position)
uns8 instr; // instruction for what to do based on selected device

PORTB.2 = 0; // pins 2 and 3 low inits DC motor in quiescent mode
PORTB.3 = 0; // quiescent mode now achieved.
PORTB.4 = 0; // start servo motor ctrl signal low, ready for info
PORTB.5 = 0; // start out servo 2 in low position
PORTB.6 = 0; // start out servo 3 in low position as well.
PORTB.7 = 0; // start out servo 4 in low position as well

del = 20; // this will correspond to a 20 milli-second delay
pos = 136; // start out servo at 0 degree mark

while(1) { // loop infinitely

// position the servo motors (at beginning of every loop)
PORTB.7 = 1; // set servo pin high for control pulse
delay_var(pos); // call variable delay for providing the position
PORTB.7 = 0; // after the delay, put control pulse low

// The above would be repeated for each servo. This could mean
// extra time consumption of up to  8mS for all four servos
// keep this in mind when working on the delays.

// now that the servo has been positioned, check for an RX byte?
if(RCIF) { // what to do when we get a byte serially
nop(); // just in case something was immediately received, let it settle.
instr = RCREG;
// Now that we have the instruction, decide what to do with them
if(instr == 'f' || instr == 'F') { // controlling the DC motor
PORTB.2 = 0; // set 'In1' to 0 for CW motion
PORTB.3 = 1; // set 'In2' to 1 for CW motion
} // end of inner if for clockwise motor turning
else if(instr == 'b' || instr == 'B') { // set motor moving CCW
PORTB.2 = 1; // set 'In1' to 1 for CCW motion
PORTB.3 = 0; // set 'In2' to 0 for CCW motion
} // end of inner if for CCW motion
```

```
else if(instr == 'h' || instr == 'H') { // turn the brakes on
PORTB.2 = 1; // set 'In1' to 1 for braking
PORTB.3 = 1; // set 'In2' to 1 for braking
} // end of inner if for braking action
else if(instr == 's' || instr == 'S') { // turn the DC motor off
PORTB.2 = 0; // set 'In1' to 0 for DC motor off
PORTB.3 = 0; // set 'In2' to 0 for DC motor off
} else if(instr == 'u' || instr == 'U') { // move servo one step up
if(pos ≤ 234)
pos++; // increment only if we haven't reached end of rotation
} else if(instr == 'd' || instr == 'D') { // move servo one step down
if(pos ≥ 137)
pos–; // lower servo motor if we are within range of motion
} // end of the else if for moving the servo.
else;
// faulty command, don't do anything
}// end of what to do when a serial byte is received


delay_ms(del); // pause for 20 milliseconds


} // end of inner while loop

}
```

A.4 PWM.c

```
/*

This is another piece of simple code to experiment with the PWM module
on the PIC 16F871. The idea is to get the PWM pin CCP1 to output the
pulse train that drives the DC motor. We will need to control a couple
of pins on the PWM chip the direction pin (either 0 for reverse, 1 for
forward), also in2 corresponds to the PWM signal in. As of the initial
run, we don't quite know what to expect so this piece of code will help
us discover what is going to happen.

Author: Dan Lockery
Date: June 22, 2005
Version: 1.0

*/

#include C:/Program Files/cc5x/16F871.h

void main()
{
```

```
PR2 = 0xFF; // Maximum Delay Possible to keep low frequency
T2CON = 16; // Timer prescale value for setting up 244Hz PWM frequency
CCPR1L = 0b.1000.0000; // Set Duty Cycle
CCP1X = 0;
CCP1Y = 0; // LSB Duty Cycle (note: DC = 50%, or 512)

PORTC = 0b.0000.0000;
TRISC = 0b.0000.0000;

while(1); // should drive PWM now that everything has been configured.
}
/* After testing this code out, it works well, the only problem is that when
using the PWM signal instead of using straight 1's and 0's for control there
is no difference in output performance. This implies that we might as well
just switch to simple two line control for direction and braking of the DC
motor. That will come once the servo motor problem has been dealt with as
it is part of the interface.
*/
```

A.5 RXTest.c

```
/*
This is a simple piece of code to test TX function of the PIC 16F871 UART.
The settings that will be used are as follows:
RCSTA = 0x90
TXSTA = 0x24 (high speed BRH and TX enable, 8bit asynch communication)
TXIE = 0 (keep the interrupts off for now, less hassle)
SPBRG = 25 (translates to 9600 bps with BRG = 1 and F_Osc = 4MHz)
TXREG = data input goes out on TX pin

Author: Dan Lockery
Date: June 21, 2005
Version: 1.0

*/


#include c:/Program Files/cc5x/16F871.h // include device defn

void main()
{
// First, initialize the UART module to prepare for serial communication

RCSTA = 144; // assign RCSTA 90h
TXSTA = 36; // enables 8-bit asynch communication with high speed BRG
SPBRG = 25; // this sets the baud rate to 9600bps since BRG = 1
TXIE = 0; // this keeps the interrupts disabled
```

```
PORTC = 0b.0000.0000;
TRISC = 0b.1000.0000; //0 = Output, 1 = Input - PORTC.7 is RX of UART for 16F871


// At this stage, the UART has been initialized, send out
// a simple bit of data (received by hyperterminal perhaps or TS-5500)
while(1) { while(!RCIF); // wait for 1 byte in from rs232
nop(); // a little bit of settling time
TXREG = RCREG; // send back whatever was given to the PIC
while(!TXIF);
}
}
```

A.6 servo.c

```
/*

This is the first attempt at developing a simple servo control routine
to allow the PIC to control servo motors. Delay routines will be included
to help in the pulse width modulation requirements. The servo that is being
used for preliminary testing is the HiTec HSR-5995TG.

The goal is to setup a simple servo routine to start by positioning the servo
at one extreme and then to increment the servo until it reaches the end of
its positioning capabilities. At which point, it will return to the start
and then begin moving again.

Author: Dan Lockery
Date: June 27, 2005
Version: 1.0

*/

#include c:/Program Files/cc5x/16F871.h // This is the device header file
#include c:/Program Files/cc5x/minid.c // This is the delay file
#include c:/Program Files/cc5x/Delay.c // contains the millisecond delay

void main()
{

// Use a single output port to control a servo motor. I'm going to use
// a pin on PORT B since there aren't any special output pins needed there.

PORTB = 0b.0000.0000;
TRISB = 0b.0000.0000; // This configures port B as all outputs

uns8 pos; // this will hold the servo position (range from 136 - 235)
```

```
uns16 del; // this is the delay for the 20ms control pulse

// Set servo pin low to begin with
PORTB.7 = 0;
del = 20;// I always want a 20 millisecond delay

// start servo at 0 degrees and then move at 8.08uS increments
while(1)
{

pos = 83; // this sets the starting position for the servo motor

while(pos ≤ 187) { // do until we reach the end of the position.

PORTB.7 = 1; // set servo pin high for control pulse
delay_var(pos); // call variable delay for providing the position
PORTB.7 = 0; // after the delay, put control pulse low
pos++; // increment the position for the next loop
delay_ms(del); // pause for 20 milliseconds

} // end of inner while loop for moving the servo to 180 degrees

while(pos ≥ 83) { // cycle back to original position

PORTB.7 = 1; // ser servo pin high for control pulse
delay_var(pos); // places servo at position 'pos'
PORTB.7 = 0; // after the delay, put the control pulse low again
pos–; // decrement the position counter
delay_ms(del); // pause for 20 milliseconds before moving again

} // end of inner while loop for moving the servo back to 0 degrees

} // end of exterior while loop

}
```

References

[1] fCoder Group International: Web Site: *fCoder Group*
$http://www.lookrs232.com/rs232/waveforms.htm$

[2] CC5X Main Page: C Compiler for PIC Micros: *B Knudsen Data*
$http://www.bknd.com/cc5x/index.shtml$

[3] Microchip Technology: PIC16F870/871 Data Sheets *Microchip Technology*
$http://ww1.microchip.com/downloads/en/DeviceDoc/30569b.pdf$

[4] Spark Fun Electronics: Website tutorial for PIC Micros:
Spark Fun Electronics $http://www.sparkfun.com$

[5] Clark D. and Owings, M., *Building Robot Drive Trains*
McGraw-Hill, New York, NY (2003) pp. 46-49

[6] Microchip Website: MPLAB IDE for PICs *Microchip Electronics*
$http://www.microchip.com$

[7] Linear Technologies: Level Converter Data Sheets for LTC1383: *Linear Technologies*
$http://www.linear.com/$

[8] Micro Engineering Labs: PICBasic PRO Compiler: *Micro Engineering Labs*
$http://www.microengineeringlabs.com$

[9] Chuck's Robotics: Website: *Chuck's Robotics*
$http://www.mcmanis.com/chuck/robotics/tutorial/h-bridge$

[10] Texas Instruments: Website: *Texas Instruments*
$http://www.ti.com$