2005 American Control Conference
June 8-10, 2005. Portland, OR, USA

A Platform for Building PIC Applications for Control and Instrumentation

Brandon Kuczenski, Philip R. LeDuc, and William C. Messner
Department of Mechanical Engineering
Carnegie Mellon University

***Abstract*— The authors have developed a minimal software environment for PIC 18F-series microcontrollers that can be used to rapidly develop mechatronic applications. The environment includes initialization of commonly utilized hardware features such as analog-to-digital conversion, pulse-width modulation control, serial communication and a programmable interrupt-driven timing loop that can be used to provide a sampling interval for a discrete-time control system. The environment also supports simple mathematical operations, interfacing with an LCD display, capturing user input and developing a flexible menu-style user interface. The software will be publicly available to facilitate rapid development of PIC-based mechatronic systems.**

I. INTRODUCTION

PICmicro® microprocessors made by Microchip [1], once called Peripheral Interface Controllers, are increasingly popular for developing inexpensive and reliable mechatronic systems. Many microcontroller projects make use of a common set of hardware features, such as serial communication, A/D conversion and PWM output, which are provided by PIC products. However, the authors have found in their research that tools to easily develop high-level applications to make use of these features are lacking.

In the process of designing a closed-loop control system for a microfluidics application [2], the authors developed a software package that can be used as a 'platform' for designing complex mechatronics applications. The platform makes use of some of Microchip's most advanced microprocessors and makes available the primary features that are most valuable for mechatronic applications and servo control applications. The platform packages these utilities within a software environment that provides the engineer with a variety of tools and utilities, such as input through a set of push-buttons, output to an LCD screen, a flexible and extensible menu-driven user interface, and a robust, interrupt-driven servo timing loop. Finally, devices built using the platform and related hardware are designed to 'stand alone,' operating without requiring a connection to a computer.

This paper details the functionality of the platform in different stages. First, hardware features and low-level software features are discussed. Then, the paper details more advanced functionality provided by the software. Finally, an example program written with the platform is described.

The platform is provided in the form of a 'template' program which can be compiled and run on PIC hardware without any modifications, providing an entry point for a device designer who is beginning a new project. The desired functionality can be written within the template's framework. The designer can thus make use of the features provided by the platform while maintaining the flexibility of writing his or her own code. The code for a simple example, an implementation of a PID compensator which is built upon the platform, is provided along with the code for the platform. The example illustrates how to make use of the template's layout to design a custom application, and is detailed in Section IV below. See Section V for information on where to acquire the code.

It is the authors' intention to make the platform and the template program freely available to the engineering community under the GNU General Public License [3]. The program may then be freely redistributed and modified, so that it can be useful to other engineers developing applications on PIC microcontrollers. Contributions from the embedded programming community are likely to make the software package more flexible and valuable. Availability of the source code is of crucial importance to this effort, because design choices that are appropriate for one purpose may not be appropriate in general.

II. FEATURES OF THE PLATFORM

There are two main functional categories of the software platform's features: (1) utilization of peripheral hardware devices available on PIC microcontrollers, and (2) a software environment within which an embedded application can operate. The result of this combination is a program that initially provides functional hardware features to the programmer, but can also be easily expanded and modified. It is written in a modular programming style that facilitates the development of custom applications. Below is an outline of first the on-chip peripherals used in the platform, and second the features provided in software.

A. Hardware Features

The software platform is designed for use on the PIC18Fxx8 family of devices [4], which are distinguished from other of Microchip's devices by the presence of on-board flash memory and a particular hardware architecture. All of these chips feature at least 16 kilobytes of program memory, 768 bytes of RAM, a maximum clock frequency of 40 MHz and a host of on-board peripherals, making

0-7803-9098-9/05/$25.00 ©2005 AACC

them more than adequate for even complex applications. The platform is written in PIC native assembly language. See the section below, entitled "Using the Platform with Higher-Level Languages" for more information.

The features included in this section are built-in to the hardware of the PIC; the platform simply activates and configures them for a typical application. Enabled features include

- polling the on-chip 10 bit Analog-to-Digital converter;
- writing to the 10 bit PWM channel;
- detection and counting of quadrature input using edge-detecting interrupts;
- transmitting and receiving serial data over the serial UART;
- initializing RAM from program memory on bootup, where desired.

The PWM output routine can be configured to use a second pin as a 'direction' flag, in effect producing a signed 10-bit output. The quadrature counter can detect inputs as fast as 30 kHz, and possibly faster, though processing time for other functions would be severely curtailed at this rate.

B. Software Features

In addition to the native hardware features of the PIC, the platform offers a number of software extensions which are of general usefulness to the embedded systems programmer. The structural software features include

- a library of functions for interfacing with the popular Hitachi HD44780 LCD screen driver chip [5];
- a way of processing user input;
- a math library;
- a programmer's interface to the serial UART;
- a programmable servo timer loop.

A full listing of software functions is provided in Table I.

Many of these features were designed to be configurable; that is, the hardware and memory resources they use can be manipulated by a programmer without affecting the function of the program. For example, the LCD interface can be programmed to use any of the device's available output pins. Configurable features can be set up by the programmer to suit a given application, and are then written into the code when it is compiled. These design choices are said to comprise compile-time options.

An LCD screen was required to provide a convenient way to operate devices free of computer control, and to streamline the debugging process. An ASCII display driven by the common and easily interfaced Hitachi HD44780 driver chip was chosen for the design. The chip provides a built-in library of 200 common characters, plus the capacity for 8 user-defined custom characters. The library includes the functions for writing single characters or strings of characters to any location on a screen driven by the Hitachi chip, as well as writing numeric data in either decimal or hexadecimal.

The LCD hardware interface requires the use of four pins as data lines and three pins as control lines, the selection of which is fully configurable at compile time. Because of the library's construction, only one routine—the routine that drives the external interface—is hardware-dependent; users who wish to modify the interface, or use a different piece of hardware, need only modify the interface routine and otherwise maintain the higher-level functionality of the rest of the library. There are provisions for including multiple low-level interface routines, with selection of the desired one at compile time.

User input is provided through a set of three pushbuttons and an optical quadrature encoder which can be used to incrementally adjust numerical values. The pushbuttons are utilized to implement a 'cellular phone'-style menu-driven user interface, the details of whose function are described in the section "Operating modes and the Extensible User Interface" below. It is notable that both edge-detecting interrupts available natively on the PIC microprocessors are used for managing the quadrature input from the optical encoder, and device designers may wish to forgo the use of the encoder as a user interface in favor of using it for feedback control.

The included math library allows abstracted operation on 16- and 32-bit fixed point integers. Supported operations include signed and unsigned addition, subtraction, negation, comparison, sign-extension, multiplication, and bit-shifting by 4 bits (multiplication and division by 16). Multiplication of two signed 16-bit fixed-point numbers, or one signed and one unsigned 16-bit number, to produce a signed 32-bit result, is supported. The math library also includes routines to convert 8-bit and 16-bit hexadecimal to packed binary-coded-decimal for the purposes of displaying internal numerical results in decimal form on the LCD screen. See Table I for a full list of Math Library functions.

Finally, the platform provides a configurable timer-interrupt-driven servo loop, which also forms the backbone of the main operating loop. At compile time, the programmer can select the interrupt frequency. When the interrupt occurs, an Interrupt Service Routine (ISR) is executed, which is responsible for sending the control signal derived from the prior period of computation to its output. The ISR also updates a millisecond-timer used to schedule events. The millisecond-timer is designed to keep correct time regardless of servo frequency, provided that the actual servo frequency is an integer division of 1000 Hz. The operation of the timer ISR is referred to as 'Foreground' operation, because the occurrence of an interrupt takes precedence over other operations occurring in the 'Background.' The design of the timer ISR should make it robust to timer overruns, as described below in the section titled, "Background Loop Timing."

III. Software Details

The software platform has been designed with the primary focus on stability, usability and extensibility. Below the primary structural features of the platform that provide these characteristics are described.

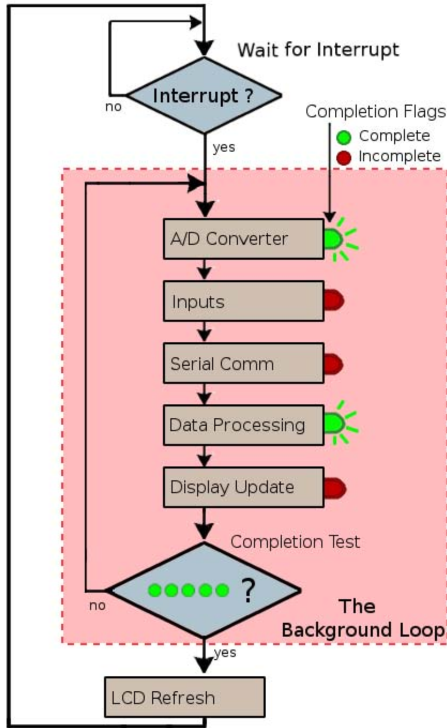

Fig. 1. The structure of the software's background operation. The processor idles until the timer interrupt occurs, then runs through the background processes sequentially in the Background Loop.

A. Background Loop Structure

Almost all of the program operates in what is called the 'Background Loop,' the set of routines that are executed in between timer interrupts. The CPU sits idle until the timer interrupt occurs. The timer ISR sets a flag which instructs the Background Loop to begin as soon as the ISR exits. The processor then enters the Background Loop, in which all data collection and computation is performed. The loop runs through a series of routines in sequence. Each routine sets a flag when it is complete, but may also return without setting its completion flag. Only when all flags have been marked as 'complete' does the background loop terminate. All routines can inspect the flags, allowing complex sequencing of events. For example, a programmer may wish to delay computation of the next servo command signal until the Analog to Digital routine is complete (Figure 1).

The timer ISR was designed to be robust to occasional dramatic overruns of the timer interrupt by the background processes, such as the delays that might be caused by periodic updates to the LCD display. After the background loop exits, LCD updates are performed, and then the processor returns to wait for the timer interrupt to occur. If the timer interrupt has already occured, indicating a timer overrun, the processor will immediately enter the next background loop, ensuring that the next timer interrupt will be performed with the most recent data (Figure 2). Because all computation and data acquisition is performed

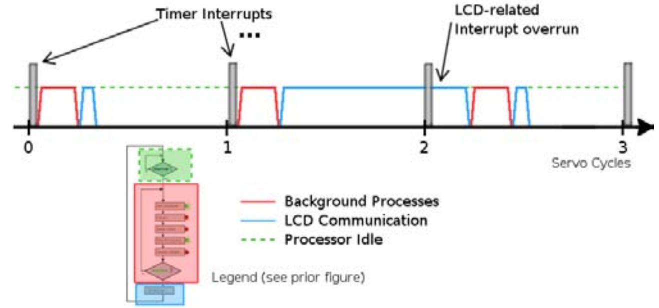

Fig. 2. A schematic depicting the relative durations of the timer interrupt and background processes. Because the timer interrupt routine is very brief, interrupt stacking does not occur, and the software is robust to background processes overrunning the timer interrupt.

during background operation, the timer ISR will always complete before the next timer interrupt occurs, and so timer interrupts will not 'stack,' and the period between timer interrupts will remain constant.

B. Operating Modes and the Extensible User Interface

User interactivity is an important feature in certain applications. The platform includes support for user input through pushbuttons, and user output through an LCD display. The software platform includes a programming scheme for easily implementing a menu-driven user interface, wherein each button corresponds to a menu option, and pressing that button can execute some function or provide another menu. The menu structure, the set of button presses and the function calls that they generate, is fully flexible and can be easily manipulated by the programmer. This functionality is described in detail below.

The software platform provides for up to thirty-two user-definable "Operating Modes," each of which is mapped to its own set of function calls to associate with the three pushbuttons. When a button press is detected, the program enters a routine which computes an index into a table of function calls. Depending on the current Operating Mode and the particular button pressed, one of those functions is selected and called by means of a jump table. That function can then perform any task, such as changing the state of the program to a new Operating Mode. A programmer must specify only which function should be called for each situation, and then write the various functions referred to, when designing the menu structure.

Figure 3 shows the structure of the configuration file and a simplified model for the operation of the code that makes this system work. The configuration file takes the form of a set of macros, which are replaced by the assembler automatically in the jump table, forming a sequential list of function calls. The jump table routine takes a numerical input from 0 to 2, which represents which button was pushed by the user. When the routine runs, it considers the current mode number, multiplies that mode by 3 and adds the numerical input. The resulting value is used as an index

```
; Configuration of the User Interface
; MODE_X_Y maps a button-press of button 'Y' in mode 'X'
; to the specified function.  The programmer can use
; this structure to generate a "menu" interface.  Flow
; through the menu choices can be changed merely by
; changing these labels.
;
; In this example, the named functions 'GoIntoMode1',
; 'GoIntoMode2', 'AnotherFunction', 'ReturnToMode0',
; and 'DoNothing' would need to be written by the
; programmer.
#define MODE_0_0 GoIntoMode1
#define MODE_0_1 GoIntoMode2
#define MODE_0_2 AnotherFunction

#define MODE_1_0 GoIntoMode2
#define MODE_1_1 ReturnToMode0
#define MODE_1_2 DoNothing
.
.
.
```

(a) Menu Interface Configuration

```
; Jump Table Implementation
; (in simplified PIC pseudo-assembly)
; This code runs every time a button is pressed, and
; does not need to be modified by the programmer.
...
  MOVF   ModeNumber,W    ; mode number into WREG
  MULLW  0x3             ; multiply by 3
  ADDWF  ButtonPress,W   ; Which button pressed? (0-2)
  ADDWF  ProgramCounter  ; jump forward by the result

; Jump Table begins here
; when current mode is 0...
  BRANCH MODE_0_0        ; button 0 calls MODE_0_0
  BRANCH MODE_0_1        ; button 1 calls MODE_0_1
  BRANCH MODE_0_2        ; button 2 calls MODE_0_2
; when current mode is 1...
  BRANCH MODE_1_0        ; button 0 calls MODE_1_0
  BRANCH MODE_1_1        ; button 1 calls MODE_1_1
  BRANCH MODE_1_2        ; button 2 calls MODE_1_2
.
.
.
```

(b) Jump Table Implementation of the Interface

Fig. 3. A description of the programming design of the flexible user interface. Figure 3a represents a configuration file; figure 3b shows a simplified implementation of the jump table that implements the menu. A programmer may reconfigure the menu simply by changing the macros defined in the upper portion.

into the sequential list of function calls. The program then 'jumps' forward by the amount of the index and lands on the function call that corresponds to the current mode and the given button press. Simply changing the macros is sufficient to modify the menu structure, because it changes which function calls get loaded into the jump table at compile time.

The mode switching machinery provides a few other basic functions. Any mode can be configured to either use or not use the optical encoder input. If the device is in an operating mode that is programmed to ignore the optical encoder, then all encoder counts that are received will be discarded. Any mode (except for the primary mode, known as the "Main Mode" or "Mode 0") can also be configured as a "Waiting Mode," during which an absence of user input after a set amount of time will cause the device to revert to the Main Mode.

Another supported function of the user interface is to

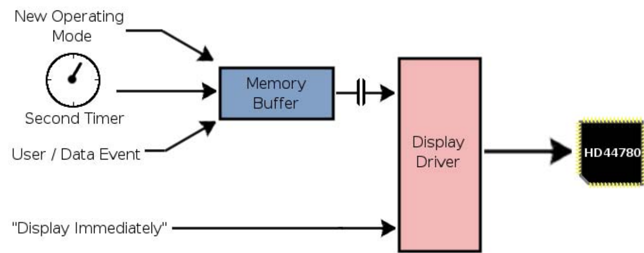

Fig. 4. The different software interfaces for writing to the LCD display. Contents of the Memory Buffer that have changed since the last update get updated once per timer interrupt, after other background processes have completed. "Display Immediately" commands get sent to the LCD screen right away.

recognize when the user presses a button and holds it down. If properly configured, the software can perform one event upon the pressing of the button, and another event upon the button's release.

The sample application described below makes use of this menu interface in concert with the LCD screen; however, this behavior is not required in general. The menu interface can be adapted to use any form of output, or no output.

C. LCD Display Routines

Because the LCD interface requires asynchronous communication with an external device, it can be expensive in terms of processor time, and prohibitively slow in some applications. In laboratory tests, the communication time with the LCD driver was observed to be as much as 10-50 microseconds per character transmitted, depending on the clock speed of the LCD driver[1] To minimize the time cost of LCD communication, the software provides a variety of methods for displaying data. The primary method is a local buffer of LCD memory space to which user programs can write only as frequently as output is desired. The LCD library then transmits only nonempty data in bulk once per timer interrupt to minimize overhead.

The display software contains three pipelines for writing to this memory. The first is executed only upon a change of Operating Mode, and is useful for costly, complete redisplays of the screen. The second is executed once every 256 milliseconds, which is convenient for periodically updating sensor data. The third can be executed at any time, and is useful for updating display information that does not change in a predictable fashion, such as responses to user input. In addition, routines which display immediately, bypassing the local memory buffer and bulk data transmission, are provided, an extension that is particularly useful for debugging. Figure 4 shows these different pathways schematically.

Finally, the entire display mechanism can be easily switched on or off during runtime if it is not intended to be used, or used only during software development. For

[1] The LCD Driver's clock speed can be configured in hardware by changing the external resistance across the HD44780's internal oscillator. Please see the driver chip's documentation [5] for details.

example, an application can be designed in which the LCD display is disabled until user input is detected, at which point it is switched on, allowing for interactive operation.

D. Receiving and Responding to commands over the Serial Port

The hardware functionality of the serial port is merely to receive and transmit messages over a serial connection using the RS232 format [6]. Conversion to a standards-compliant voltage must be done using separate hardware. Interpretation of the messages is left to the software, but a basic framework is provided.

The serial port was designed with the intention of primarily receiving commands from a host computer—the software does not initiate communication except to respond to a command. At the default baud rate of 38,400 baud and sampling frequency of 1 kHz, twenty-four bits of data can be comfortably received and transmitted every timer interrupt. The size of the command word was designed to be 16 bits. Data reception is interrupt-driven, with the interrupt routine capturing each byte of data as it arrives. If the software receive buffer fills up, the interrupt routine will set the serial CTS bit[2], telling a host computer with enabled hardware-handshaking to postpone transmission of further data.

Once two bytes have been received, the serial background routine processes them as a command. A jump-table architecture is used, providing an essentially limitless functionality. Depending on the numerical value of the command, any function in program memory can be called. Preprocessing of commands is also possible, to allow some or all commands to bypass the jump table architecture.

Whereas data reception is interrupt driven, and only one command is permitted to be received per timer interrupt, transmission is a background process. Any time the serial background routine detects that the hardware transmission buffer is empty and there is data to transmit, it will transmit the next byte. Because of the repeating nature of the Background Loop, this approach is highly efficient and causes minimal idle time when there is data to be transmitted.

E. Code Modularity

The software's source code is divided into three groups of files: 'core' files, which are invariant from project to project, 'project' files, which are modified or written entirely by the project programmer, and 'header' files, which contain compile-time options that the programmer can adjust. By selectively including or omitting certain blocks of code, the programmer can select which 'modules' should be used in the application. A complete description of the process of application programming that is facilitated by this platform is beyond the scope of this paper, and is included instead as part of the documentation that accompanies the published code (see Section V).

[2]See the TIA-232 standard [6]

IV. A SAMPLE APPLICATION

This section describes an application that was developed using this platform. The program implements PID compensation of a system whose feedback is read through an analog sensor. The application allows the programmer to calibrate the sensor, setting both its 'zero' value and a proportionality constant that scales A/D counts to real-world units. The user can also select the reference to be one of three sources: another analog input, an internal reference that can be adjusted by the user, or a command received over the serial communication interface. Finally, the user can set the gains for the proportional, integral, and backward difference components of the compensator.

The device is meant to be operated interactively and independently of a computer, unless the serial communications interface is to be used as the reference source. In order to fully realize this device, therefore, a certain set of hardware devices, including an LCD screen driven by the Hitachi HD44780 chip or equivalent, and a directional motor amplifier such as an H-Bridge, must be used. Tested, functional schematics and a parts list for such a device are available from the authors.

The following description demonstrates the features of the software platform from the perspective of the user interface. A reading of the code for this sample application (see Section V) will show how the user interface and the platform's features are connected.

Figure 5 illustrates the structure of the menu interface used in this design. The left side of the figure is the code listing for the macros that define the menu jump table, as discussed in Figure 3. The right side shows a block diagram representing the nine different Operating Modes. At the bottom of each box are three labels which correspond to the functions of the three pushbuttons in that particular mode. Mode 0, the "MainMode", is a branching off point into one of three modes: the first pushbutton, labeled "Config", allows the user to enter a mode to configure device operation; the second pushbutton, "Gains", enables the user to tweak the compensation design; and the third button, "Run", operates the device. In the code listing to the left, the entries for Mode 0 are listed as "EnterConfig," "EnterCtrl," and "EnterRun," and button presses trigger execution of functions bearing those names.

In the "Config" mode, the user can choose to select the reference source, calibrate the sensor, or surrender control of the device to the serial communication port. The press of the first button takes the user into the "Reference" mode, where he or she can select an internal reference, an external reference from an analog input, or a reference received over the serial port. In the "Cal" mode, the user calibrates the sensor, a process which has two stages, a zero stage and a gain stage. In the zero stage, the sensor should be placed into a zero-input state. When ready, the user presses the button labeled "Gain" and the controller reads the current sensor value to subtract from future readings

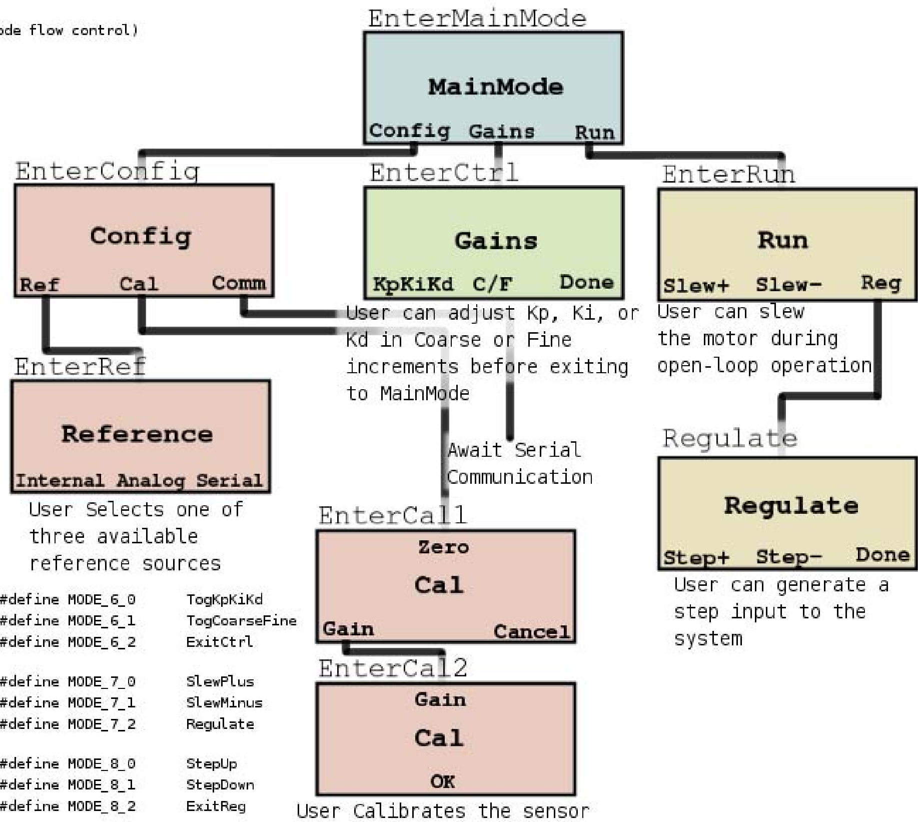

Fig. 5. The menu structure for a sample application produced with the platform. The application is a PID compensator which uses an analog to digital converter to read sensor input and PWM to generate control output. The user can adjust the compensator gains, as well as select the source of the reference signal at runtime. The menu interface configuration file is shown at left; a schematic of the menu structure is shown at right.

before entering the gain phase of calibration. In this stage, the user should supply a known input to the sensor and adjust the controller's internal numerical gain using the optical encoder until the displayed value matches the actual value. Should the user enter the "Comm" mode, the device will idle until it receives a command over the serial port or until the user cancels.

In the "Gains" mode, the user can use the optical encoder to adjust the proportional, integral, and backward-difference gains of the internal PID compensator. In this mode, the first button allows the user to cycle through the three gains, and the second button toggles selection of the less-significant byte and the more-significant byte of the gains' values. As long as the user is in this mode, rotation of the optical encoder will cause an adjustment to the selected byte of the selected gain. The third button exits.

In the "Run" mode, the user can use the first two buttons to output a fixed PWM signal with the chosen direction pin outputting a 'positive' or 'negative' sense. This enables the user to slew the motor forward and backward in order to prepare the device being controlled. The third button enters the "Regulate" mode, in which the device reads from the sensor, compares the value to the selected reference, performs PID compensation on the error signal and outputs the signed result to the PWM pin to run a motor, the sign being interpreted to set the sense of the direction pin. In this mode, if the user has selected the internal reference, then he or she can step its value up or down by a fixed value, and the optical encoder adjusts the reference in a more continuous fashion. The third button exits.

The sample program is designed to work with a pressure sensor, and so on-screen displays refer to input values in PSI, indicating the units of Pounds per Square Inch. The software is fully functional and used extensively in the authors' research.

V. CODE AVAILABILITY

The code is available under the Gnu General Public License on an author's home page on the World Wide Web at Carnegie Mellon University, and on the open-source clearinghouse, Sourceforge, at `http://sourceforge.net/`, under the project name "TyPIC."

VI. CONCLUSION

The flexibility of writing one's own operating software for a given mechatronic application often far outweighs the simplicity of using pre-packaged tools. The authors have

TABLE I

A FULL LIST OF SOFTWARE FUNCTIONS.

Background Processes	
`AtoD`	Poll A/D converter to read analog inputs in succession
`InputProc`	Read user input and change Modes
`SerialProc`	Process commands received over the serial port
`DataProc`	Compute new servo command; log data
`DisplayRefresh`	Interface with the LCD Memory Buffer
LCD Library Functions	*544 bytes = 272 code words*
`LCDSetup`	*Utility* – Initialize the LCD screen and display a short message
`HexToAscii`	*Utility* – Convert numeric data to ASCII format
`LCDClearBuffer`	*Buffer* – Zero the contents of the Memory Buffer
`LCDWriteChar`	*Buffer* – Write a character to a specified location
`LCDHexToBuffer`	*Buffer* – Write a number to the buffer in ASCII format
`LCDPutWord`	*Buffer* – Write a string of characters to the memory buffer
`LCDClearHome`	*Display-Immediately* – Clear the screen and home the cursor
`LCDWriteHex`	*Display-Immediately* – Write the current hex value to the screen for debugging
`LCDPutCharNow`	*Display-Immediately* – Write the specified character to the screen for debugging
`LCDRefresh`	*Utility* – Copy non-zero entries in the Memory Buffer to the LCD screen
Math Library Functions	*802 bytes = 401 code words*
`Compare16U`	16-bit unsigned compare
`Compare16S`	16-bit signed compare
`Add16`	Add two 16-bit numbers – signed or unsigned
`Add16C`	Add two 16-bit numbers and retain addends
`Add16Sat`	Add two signed 16-bit numbers with saturation
`Add32Sat`	Add two signed 32-bit numbers with saturation
`SignExtend8`	Sign-extend an 8-bit number to 16 bits
`Subtract16`	Subtract one 16-bit value from another – Signed or Unsigned
`Subtract16C`	Subtract two 16-bit numbers and retain subtractands
`Neg2Comp16`	Twos-complement negation of 16-bit number
`Neg2Comp32`	Twos-complement negation of 32-bit number
`Abs16`	Absolute Value of 16-bit number
`Abs32`	Absolute Value of 32-bit number
`LShift16`	Left-shift 16-bit number by 4 bits
`RShift16`	Right-shift 16-bit number by 4 bits
`RShift24`	Right-shift 24-bit number by 4 bits
`LShift32`	Left-shift 32-bit number by 4 bits
`RShift32`	Right-shift 32-bit number by 4 bits
`HexToDec8`	Convert 8-bit number to 3-place decimal
`HexToDec16`	Convert signed 16-bit number to signed 5-place decimal
`Multiply16S_un`	Multiply a 16-bit signed and 16-bit unsigned number to produce 32-bit signed result
`Multiply16S`	Multiply two signed 16-bit numbers

attempted to create a software package that strikes a balance between ease of use and availability of the full functionality of the processor. The resulting software platform has been demonstrated to be a valuable tool for rapid development of complex applications involving the PIC microcontroller, and may provide utility to other researchers who use embedded controllers in research.

REFERENCES

[1] (2004) The microchip website. [Online]. Available: http://www.microchip.com/

[2] B. Kuczenski, W. C. Messner, and P. R. LeDuc, "An automated system for controlling the laminar flow interface in a microfluidic system," IMECE Congress, November 2004.

[3] (1991, June) The GNU general public license. Free Software Foundation, Inc. Boston, MA. [Online]. Available: http://www.fsf.org/licenses/gpl.html

[4] *28/40 pin Enhanced FLASH Microcontrollers (PIC18Fxx8)*, Microchip Technology, Inc., 2003, doc. no. DS41159C.

[5] *Dot Matrix Liquid Crystal Display Controller/Driver (HD44780U)*, Hitachi, 1999.

[6] *Interface between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange*, TIA Std. ANSI/TIA-232-F-1997, September 1997.