ARL-TR-9204 ● June 2021

DEVCOM ARMY RESEARCH LABORATORY

A Primer for Programming and Applying the Simple Laboratory Integration Platform (SLIP)

by Thomas Kottke and Julian D Fleniken

NOTICES

Disclaimers

ARL-TR-9204 ● JUNE 2021

A Primer for Programming and Applying the Simple Laboratory Integration Platform (SLIP)

Thomas Kottke
Oak Ridge Associated Universities

Julian D Fleniken
Weapons and Materials Research Directorate,
DEVCOM Army Research Laboratory

REPORT DOCUMENTATION PAGE						Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**						
1. REPORT DATE *(DD-MM-YYYY)* June 2021		**2. REPORT TYPE** Technical Report				**3. DATES COVERED (From - To)** December 2018–December 2020
4. TITLE AND SUBTITLE A Primer for Programming and Applying the Simple Laboratory Integration Platform (SLIP)						**5a. CONTRACT NUMBER** **5b. GRANT NUMBER** **5c. PROGRAM ELEMENT NUMBER**
6. AUTHOR(S) Thomas Kottke and Julian D Fleniken						**5d. PROJECT NUMBER** **5e. TASK NUMBER** **5f. WORK UNIT NUMBER**
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) DEVCOM Army Research Laboratory ATTN: FCDD-RLW-TA Aberdeen Proving Ground, MD 21005						**8. PERFORMING ORGANIZATION REPORT NUMBER** ARL-TR-9204
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)						**10. SPONSOR/MONITOR'S ACRONYM(S)** **11. SPONSOR/MONITOR'S REPORT NUMBER(S)**
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release: distribution unlimited.						
13. SUPPLEMENTARY NOTES						
14. ABSTRACT The simple laboratory integration platform (SLIP) is a microcontroller-centric single-board processing system designed to streamline electronic integration efforts. A guide is presented with which programmers can readily access the SLIP system for application to a wide variety of laboratory automation tasks. The required hardware and software components are listed along with sources where they can be obtained. A series of obscure, but necessary, housekeeping tasks are presented to streamline the process of configuring, initializing, and activating the SLIP. Finally, a sample C program is provided and discussed to demonstrate the workflow process of writing, compiling, downloading, debugging, and executing code on the SLIP.						
15. SUBJECT TERMS microcontroller, single-board, laboratory automation, laboratory integration, programming						
16. SECURITY CLASSIFICATION OF:				**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Thomas Kottke
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified		UU	45	**19b. TELEPHONE NUMBER (Include area code)** (410) 278-2557

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures **v**

Acknowledgments **vi**

1. Introduction **1**

2. Requirements for SLIP Programming **2**

2.1 Required Hardware for SLIP Programming 2

2.2 Required Software for SLIP Programming 6

3. Preprogramming Housekeeping Details **7**

3.1 Confirming the MPLAB X IDE/ICD 4 Interface 7

3.2 Creating a New MPLAB X IDE Project 8

3.3 Creating a C Program File within the New Project 9

3.4 Loading the PIC24HJ256GP210A Register Definition File into the Project 9

3.5 Configuring Global Microcontroller Startup Parameters 10

3.6 Configuring the Microcontroller Oscillator Settings 11

3.7 Configuring Individual Microcontroller Port Pins 12

4. A Demonstration of MPLAB X IDE, ICD 4, and SLIP Functionality **13**

5. Conclusions and Path Forward **16**

6. References and Notes **18**

Appendix A. Listing of PIC24HJ256GP210A Power-On Global Parameter Configuration Code **20**

Appendix B. Listing of PIC24HJ256GP210A Run-Time Oscillator Parameters Configuration Code **23**

Appendix C. Listing of PIC24HJ256GP210A Ports Configuration Code **26**

Appendix D. Listing of a Simple Laboratory Integration Platform (SLIP) Demonstration C Code 33

List of Symbols, Abbreviations, and Acronyms 36

Distribution List 37

List of Figures

Fig. 1 Hardware required for SLIP programming 3
Fig. 2 SLIP PCB, top 4
Fig. 3 SLIP PCB, bottom 5
Fig. 4 Portion of the SLIP electronic schematic 14
Fig. 5 Example of the MPLAB X IDE user interface 15

Acknowledgments

The first author would like to thank the second author for the opportunity of an encore career and the freedom to enjoy it. The authors would like to thank Kenneth M Kwashnak (Survice Engineering) for technically reviewing this report. And finally, the authors would like to thank Mark Gatlin of the US Army Combat Capabilities Development Command Army Research Laboratory Technical Publishing for editorially reviewing this manuscript and disguising the authors' illiteracy.

1. Introduction

The Simple Laboratory Integration Platform (SLIP) is a microcontroller-centric single-board processing system developed in the Applied Physics Branch of the Terminal Effects Division of the Weapons and Materials Research Directorate at the US Army Combat Capabilities Development Command Army Research Laboratory. This device was designed to streamline electronic system integration efforts involving data acquisition, data processing, communication, control, or power management. The SLIP was originally developed almost a decade ago to facilitate the testing and evaluation of adaptive/cooperative protection system sensor and effector components in a variety of configurations and combinations. However, since that time this device has been used in many other applications including IR imaging, digital signal processing, ballistic trajectory shot-line determination, counter-munition activation timing, laser-diode modulation, and stepper-motor control. To date, the individual who primarily designed and developed the SLIP has also served as the principle programmer for this device. While the SLIP appears to be capable of providing many more years of useful service, the career of the principle programmer is in fact waning. This report is provided to encourage the continued application of this device by offering future programmers a clear and concise introduction to the requirements and methods for programming, using, and improving the SLIP.

The task of highlighting a clear path forward for future SLIP programmers is addressed by providing three useful sets of information. First, the required hardware and software components are enumerated and described along with sources where they can be obtained. Then some necessary, and perhaps obscure, housekeeping details are presented that will allow the future programmer to initially bring the SLIP "alive" with a minimal amount of frustration. Finally, an elementary program is presented that the user can use to demonstrate that the SLIP is functional and provide a template for future programming efforts.

It is the intention of the authors to generate subsequent reports that will offer additional software utilities of increasing complexity to assist potential SLIP programmers in their journey to becoming proficient users of this hardware. This effort will provide multiple benefits. Primarily, these utilities will spotlight various useful features of the SLIP and educate future programmers about the device's open-ended capabilities. A secondary goal is to effectively develop a hardware abstraction layer[1] between the programmer and the SLIP. Hardware abstractions are routines in software that provide programs with access to hardware resources through programming interfaces. In this way, much of the drudgery of using hardware resources can be alleviated by distancing the programmer from the myriad minutiae. It is not the intention of the authors to hide the inner workings of this device from potential users. Quite the contrary, every effort is made to suggest additional references and resources to expand the user's repertoire of programming skills and knowledge. Toward that end, each utility will provide all necessary source code and explanations for user inspection and improvement. The intention is to provide utilities that can assist novice programmers with a preliminary set of tools to assist and streamline their initial efforts to create useful applications.

2. Requirements for SLIP Programming

A modest collection of both hardware and software items needs to be collected and assembled to program and apply the SLIP. These items will be enumerated along with relevant features and sources where they can be obtained.

2.1 Required Hardware for SLIP Programming

The following hardware items are required for SLIP programming:

- Fully populated SLIP printed circuit board (PCB)
- SLIP PCB power source
- SLIP PCB power cable
- SLIP compatible in-circuit programmer/debugger with adapters and cables

Figure 1 displays a collection of these hardware items, which are considered individually in detail in the following.

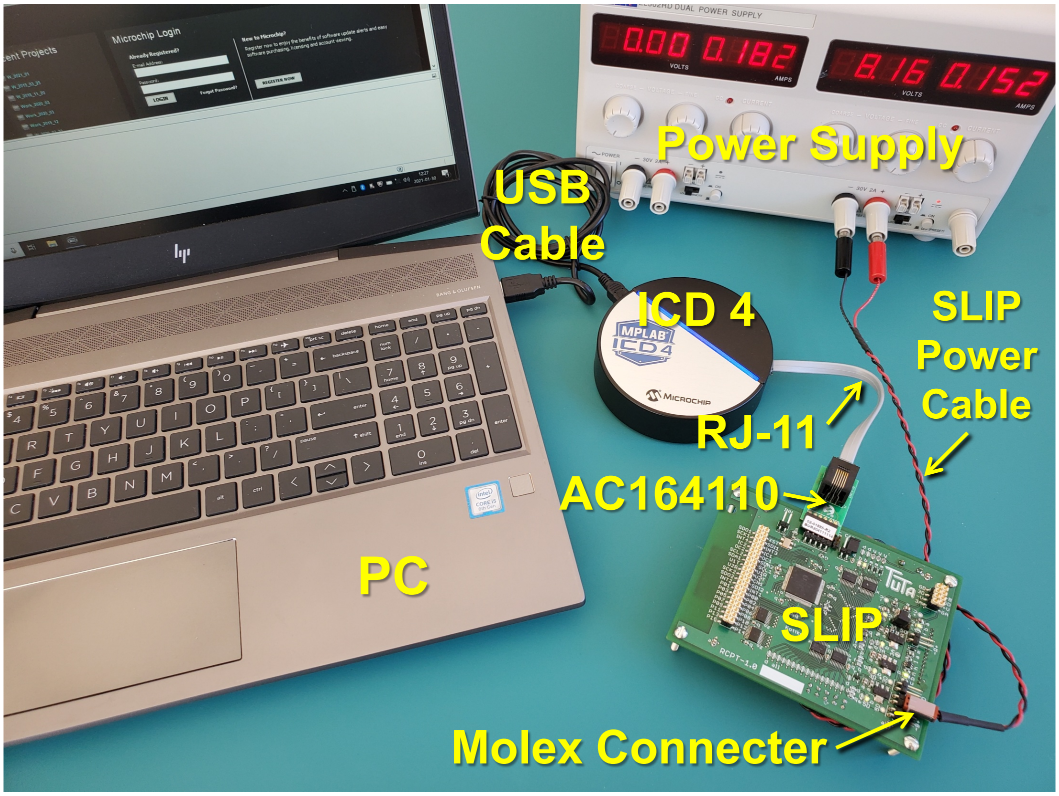

Fig. 1 Hardware required for SLIP programming

The SLIP was developed within the Applied Physics Branch of the DEVCOM Army Research Laboratory, which currently is the only source for this item. Please contact the authors for availability. This integration platform was designed around the Microchip Technology[2] PIC24HJ256GP210A microcontroller.[3] This device was selected for its multiple capabilities in addition to its ease of use and extensive, readily available support documentation.

The PIC24HJ256GP210A is a high-end 16-bit microcontroller with an abundant supply of program memory and peripheral functionality to allow hardware platform development for a wide variety of applications. This microcontroller can operate at central processor unit speeds of up to 40 million instructions per second. In addition to a multitude of timers, flexible interrupt architecture, analog-to-digital converters, digital input/output (I/O), and direct memory access support, this microcontroller also includes a powerful communications module. This module supports multiple common communication protocols. The PIC24HJ256GP210A is also serviced by a mature integrated development environment (IDE) that seamlessly enables mixed language programming, code text editing, machine code compilation, and device programming.

Details of the SLIP hardware's design, fabrication, and specific capabilities have been presented in a previous report.[4] Figures 2 and 3 of the current report illustrate a fully populated SLIP and highlight the locations of some of the major components.

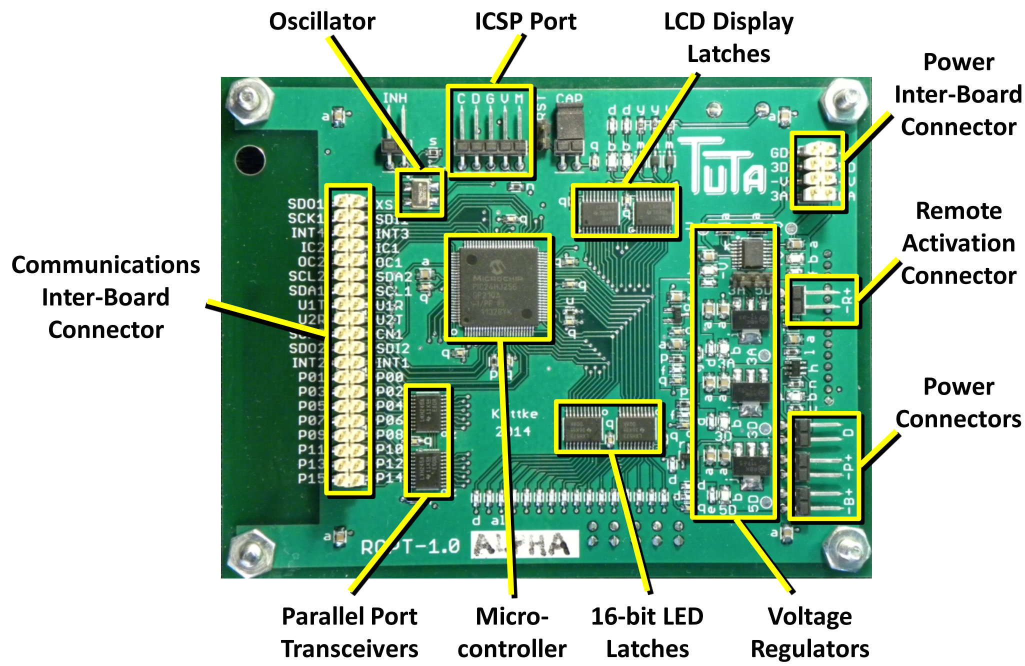

Fig. 2 SLIP PCB, top

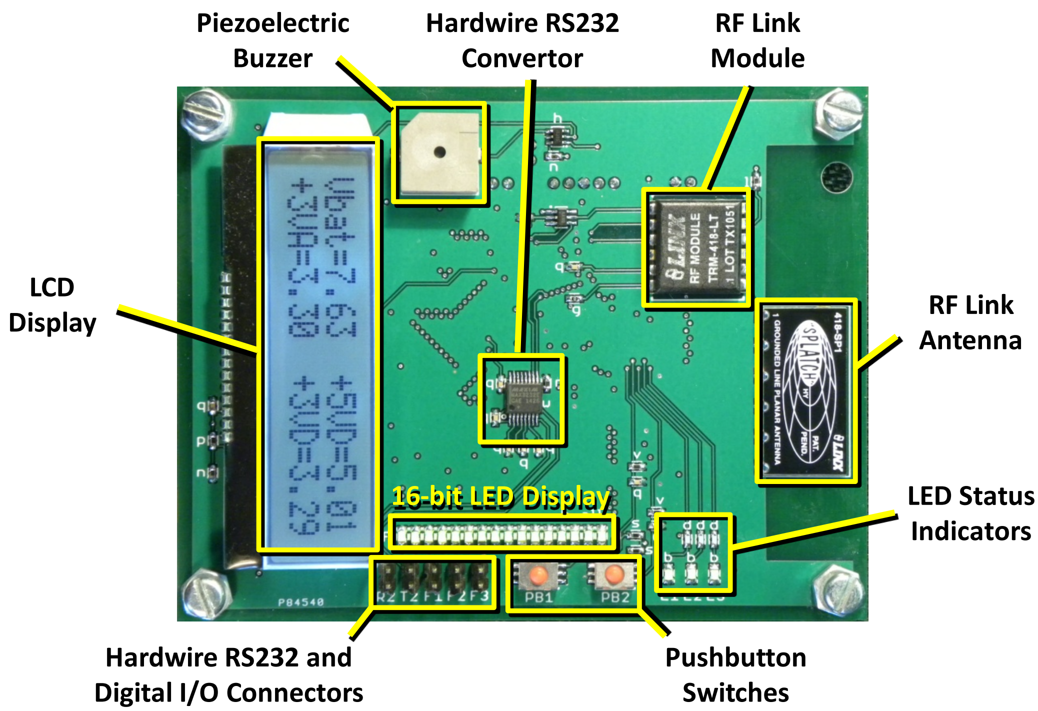

Fig. 3 SLIP PCB, bottom

A suitable power supply for SLIP programming is any unipolar voltage source capable of providing between 7 and 35 V and 250 mA of current. The laboratory power supply illustrated in Fig. 1 is ideal. For mobile applications, a two- or three-cell lithium polymer battery pack is also convenient.

The power cable for SLIP programming is a simple two-wire assembly connecting the negative and positive terminals of the power supply to the –P and +P pins, respectively, on the SLIP board. Figure 2 displays the location of the –P and +P SLIP pins in the highlighted box labeled Power Connectors in the lower right-hand corner of this figure.

Of course, the connectors on the power supply end of this cable will depend on the chosen power supply. For the example illustrated in Fig. 1, simple banana plugs are used. On the SLIP end of the power cable, a two-pin 0.100-inch pitch Molex connector[5] available from Digi-Key Electronics[6] provides a convenient and secure connection. Regardless of the chosen connectors, care should be exercised to maintain the correct voltage polarity.

The programmer/debugger serves as a hardware interface between the PC, where code is entered, developed, and compiled, and the SLIP, where the compiled code will ultimately be executed. When acting as a programmer, this interface device simply downloads the compiled code from the PC to the program memory of the

SLIP's microcontroller. Following this download, the connection between the programmer/debugger and the SLIP can be removed. Whenever power is applied to the SLIP, the downloaded program will automatically be initiated and proceed to completion.

In contrast, when the programmer/debugger is operating in the debugger mode, downloads from the PC to the SLIP consist of not only the compiled program code but additional executive executable code. This executive code allows the PC to monitor the real-time execution of the compiled code on the microcontroller. Therefore, the connection between the programmer/debugger and the SLIP board must be maintained whenever the programmer/debugger is operating in this mode. The advantage of the debug mode is program execution on the SLIP can be halted at any time, or at preset breakpoints, and all microcontroller variable and register values can be examined and altered.

The programmer/debugger must be compatible with the operation of the microcontroller on the SLIP board. A suitable choice is the Microchip Technology ICD 4[7] available from Digi-Key Electronics.[8] Figure 1 displays this programmer/debugger and its connections to both the PC and the SLIP. A standard USB cable connects the ICD 4 programmer/debugger to the PC. The ICD 4 connects to the SLIP via the in-circuit serial programming port (ICSP) highlighted in the top of Fig. 2. This connection requires a modular RJ-11[9] cable and a Microchip Technology AC164110 RJ-11 to ICSP adapter board[10] available from Digi-Key Electronics.[11] The USB cable and the modular RJ-11 cable are included with the ICD 4.

2.2 Required Software for SLIP Programming

- Conveniently, the two PC software packages required for SLIP programming are both available online and are free:
  - Microchip Technology MPLAB X Integrated Development Environment
  - Microchip Technology XC16 C Programming Language Compiler

The Microchip Technology MPLAB X IDE is a convenient, flexible software package that combines multiple resources to assist with code configuration, entry, development, programming, and debugging. This IDE works seamlessly with the ICD 4 programmer/debugger and most other Microchip Technology MPLAB development tools and software. To download this IDE software, go to https://www.microchip.com/en-us/development-tools-tools-and-software/mplab-x-ide.

One quarter of the way down this page is a click-tab labeled "Downloads" that brings up additional click-tabs for Windows, Linux, and Apple operating system–compatible downloads of the MPLAB X IDE. Each of these downloads transfer in excess of 1 gigabyte of data. Therefore, a high-speed data connection is recommended.

Use of the Microchip Technology XC16 C Programming Language Compiler allows the SLIP to be programmed in the high-level C language in addition to the native, low-level 16-bit assembly language. This will be a welcome advantage for most new programmers. To download this compiler go to https://www.microchip.com/en-us/development-tools-tools-and-software/mplab-xc-compilers and click on the "Compiler Download" tab one-third of the way down the page. Then click on "MPLAB XC16 Compiler" under the PC operating system of your choice. This compiler download is a more modest 100 megabytes in size.

3. Preprogramming Housekeeping Details

After the necessary hardware and software components have been collected, a number of housekeeping details need to be addressed before programming the SLIP can begin. These housekeeping chores include the following:

- Determining the MPLAB X IDE software and ICD 4 hardware are interfaced and communicating successfully
- Creating a new project within the MPLAB X IDE environment
- Creating a C program file within this new project
- Loading a file into the new project that defines registers and includes other useful information about the PIC24HJ256GP210A microcontroller
- Configuring various microcontroller global power-on parameters
- Configuring the microcontroller oscillator settings
- Configuring the individual microcontroller port pin types, directions, and initial values

At first introduction, these housekeeping details may appear somewhat onerous. However, in short order they will become routine. And like all chores, they are necessary and more palatable after they have been completed. In the following descriptions, references to specific text items presented within the IDE software graphical user interface will appear in quotations for clarity.

3.1 Confirming the MPLAB X IDE/ICD 4 Interface

The first housekeeping goal is to ascertain that the MPLAB X IDE program on the PC and the ICD 4 programmer/debugger hardware are interfacing properly. Begin by opening the IDE program. All necessary USB drivers for the ICD 4 were auto-loaded to the PC at the same time the IDE software was installed. The next step is to simply connect the ICD 4 to the PC using the supplied USB cable, as illustrated in Fig. 1. The status bar strip on the ICD 4 will glow purple, change to blue, blink, and finally remain a steady blue. Plug the supplied RJ-11 modular cable into the ICD 4 and connect the supplied ICD Test Interface Module AC164113 onto the other end of the RJ-11 cable. On the PC, click the IDE "Debug" pull-down tab. Near the bottom of the pull-down list click "Run Debugger/Programmer Self Test". A pop-up window will ask the user to "Please select the tool you would like to run the self test on". Under "ICD 4" click the serial number that matches the number on the back of your particular ICD 4 programmer/debugger and click the "OK" button. After a few moments, the ICD 4 will click and another pop-up window will ask you to "Please ensure the RJ-11 cable is connected to the test board before continuing". Click the "Yes" button. After the ICD 4 finishes clicking again, the IDE on the PC should display the following:

"Test interface PGC clock line write succeeded."
"Test interface PGD data line write succeeded."
"Test interface PGC clock line read succeeded."
"Test interface PGD data line read succeeded."
"Test interface LVP control line test succeeded."
"Test interface MCLR level test succeeded."
"ICD 4 is functioning properly."

At this point, the user can be confident that the MPLAB X IDE program on the PC and the ICD 4 programmer/debugger are interfacing properly.

3.2 Creating a New MPLAB X IDE Project

The next task is for the user to create a new project within the MPLAB X IDE program on the PC. To create a new project, the IDE needs to know

- Specific type of project to be created
- Specific microcontroller on which the project is meant to execute
- Which, if any, compilers are to be used
- In what directory program files are to be stored

Start this task by creating a new working folder on the PC where the SLIP program files will be stored. Under the "File" pull-down tab in the IDE program select "New Project…". A new "Choose Project" pop-up window will appear where the type of project can be selected. Under "Categories" select "Microchip Embedded", under "Projects:" select "Standalone Project" and click "Next". A second pop-up window is the “Select Device” window. Under “Family” select “16-bit MCUs (PIC24)”; under “Device” select “PIC24HJ256GP210A” near the bottom of the list; under “Tools:” select “No Tools” and click “Next”. The third pop-up window is the “Select Compiler” window. Under “XC16” select the version of the C compiler downloaded from the Microchip Technology website and click “Next”. The final pop-up window allows the user to inform the IDE of the previously created working folder where the new SLIP project and all related files are to be stored. Under “Project Name:” enter a suitable, descriptive filename. Under “Project Location:” browse to and select the previously created project working folder and click “Open”. The “Project Folder:” textbox will automatically populate with a folder name consisting of the project location folder, concatenated with the project name, concatenated with an addition sublevel folder of the same project name with “.X” suffix. This .X folder is where the MPLAB X IDE will store all the files it generates to support the newly created project. The user will have little reason to interact with this .X folder. Click “Finish”. An inspection of the recently created working folder created for this SLIP project will reveal the addition of a new subfolder with the supplied “Project Name”. This is where all the code the user generates will reside. Furthermore, this subfolder will contain an additional .X subfolder of the same name, which is the working domain of the IDE.

3.3 Creating a C Program File within the New Project

Creating a C program file within the new project is straightforward. The upper left-hand window of the IDE displays tabs labeled “Projects”, “Files”, and “Services”. Click the “Projects” tab. The displayed file hierarchy shows the project folder at the top with the selected project name. One of the underlying subfolders is labeled “Source Files”. Right-click this folder and select “New” followed by “C Main File…”. A pop-up window requests the desired C filename. Enter a descriptive filename, leave all the other fields in this pop-up with their default values, and click “Finish”. The user will see the newly created C program file consisting of a very rudimentary main C program template with documenting remarks at the top, a couple of #include statements, and an empty C function named “main” at the bottom. At this point, check to see where the newly created C program file has been created. If it has been located in the .X subfolder of the project file, cut it and paste it into the project file itself.

3.4 Loading the PIC24HJ256GP210A Register Definition File into the Project

The next housekeeping detail is to load a pre-existing file into the project that contains multiple types of information required by the C language compiler.

Primarily, this file contains the defined names of the registers and bit locations of the specified microcontroller that the IDE will use to access the information contained within them. Conveniently, these names match the Microchip Technology data sheets as closely as possible. In addition, this file contains configuration name definitions and other useful information. To retrieve this file, go to the C:\Program Files\Microchip\xc16\vX.XX\support\PIC24H\h folder, copy the file named p24HJ256GP210A.h, and paste this file into the newly created project folder. Note that "vX.XX" refers to the current version number of the downloaded XC16 C programming language compiler. As of January 2021 the current version was v1.61, but this will certainly change with time. Substitute the version number of the C compiler that you are currently using. Returning to the project window in the IDE, right-click on the "Important Files" subfolder and select "Add Item to Important Files". Select the p24HJ256GP210A.h file recently added to the project file and click "Select". The p24HJ256GP210A.h filename is added to the "Important Files" subfolder. By right-clicking on this filename and selecting "Open", the contents of this file can be examined. The remaining step in this task is to include the contents of file p24HJ256GP210A.h during compilation. This is accomplished by adding the following line of code

```
#include "p24HJ256GP210A.h"
```

to the newly created C program file below the preexisting include statements. Statements that begin with a "#" are compiler directives. All compiler directives are executed by the compiler before any C language statements are considered.

3.5 Configuring Global Microcontroller Startup Parameters

The MPLAB ICD 4 In-Circuit Debugger Quick Start Guide[12] specifies recommended microcontroller global settings that need to be configured to ensure proper communication between the microcontroller on the target SLIP board and the ICD 4. Appendix A of this report lists a header file that will properly configure the specified microcontroller settings. The details of this header file are not particularly important for the novice SLIP programmer at this time. However, inspection of this appendix reveals another collection of compiler directives. Among other tasks, these compiler directives select an initial clock source for the microcontroller on the target SLIP board. This task cannot be accomplished using straight C code because the clock source must be correctly configured before any code can be executed. Specifically, the 10-MHz oscillator on the SLIP, which is external to the microcontroller, is selected as the clock source and phase-lock loop[13] modification of this time base within which the microcontroller is enabled. Programmers with specific concerns about this configuration setting process should consult the PIC24HJXXXGPX06A/X08A/X10A data sheet[14] Section 21.1. The procedures for loading the header file of Appendix A into the program file and how to associate it with the project via the C code is as follows.

In the upper left-hand window of the IDE, click the "Projects" tab as before and then right-click the subfolder "Header Files", select "New", and select "C Header Files…". In the resulting pop-up window, enter "Config_PIC24.h" as the filename. This new file will appear within the "Header Files" folder. Right-click the new file listing and select "Open". The contents of the new header file will be listed in the IDE. Erase the current contents of this newly created header file, copy the contents of Appendix A, and paste them into the header file. Finally, open the C program file and enter the following

#include "Config_PIC24.h"

below the pre-existing include statements. This statement allows the project to access the contents of the "Config_PIC24.h" header file for global settings configuration.

3.6 Configuring the Microcontroller Oscillator Settings

Like so many faunal systems, without a good "heart beat", the SLIP cannot function. The global settings of the previous section ensure the microcontroller will have access to a valid initial clock signal. However, this initial timing source uses power-on default values that generate a clock frequency less than the maximum allowed 40-MHz clock speed. The next housekeeping task is to maximize the timing signal to the microcontroller by adjusting the phase-lock loop modification parameters. Appendix B of this report lists a header file that will properly set these phase-lock loop parameters for maximum microcontroller clock speed. Details relating to specifics of the phase-lock loop clock modification process are available in Section 7 of the Microchip Technology PIC24H Family Reference Manual.[15] The procedure for loading the header file of Appendix B into the program file and accessing it within the C code is very similar to the process used to load Appendix A, with one small addition at the end. Create a new header file named Config_oscill.h. Open this header file and replace the preexisting contents with the contents of Appendix B. Next, open the C program file and enter the following

#include "Config_oscill.h"

below the preexisting include statements. One additional new step is to add the statement

"config_osc()"

inside the "main" function of the C code. This line of code calls the function located in the associated header file.

3.7 Configuring Individual Microcontroller Port Pins

The PIC24HJ256GP210A microcontroller on the SLIP has 100 pins. Many of these pins are included in general-purpose I/O ports. Most of these I/O port pins are multiplexed with alternate functions to add flexibility and allow different functions to be performed at various times. The final housekeeping task is to configure many of these individual pins with respect to function type, direction, and initial value. To utilize a pin for a specific application, it is generally necessary to specify three characteristics about the pin:

1. Whether the pin will perform an analog or a digital function
2. Whether the pin will function as an input device or an output device
3. If the pin is configured as a digital output device, whether the output state at power-on will be high or low

This is accomplished by accessing and defining several registers within the microcontroller that control these settings. A tutorial explaining I/O port configuration details is available online.[16]

Some of the multiplexed microcontroller pins are routed to internal locations on the SLIP board that are not intended for external access. These pins that perform a single, static, internal function can be preconfigured to a static state that allows them to perform that function as required at any time. Other microcontroller pins are routed to external connection points, such as the inter-board header arrays. It may not be known in advance what sort of external devices will be attached to these external SLIP connections. Therefore, to avoid conflicting signal levels at power-on that may detrimentally affect the microcontroller or the external device, these pins generally are configured as high-impedance digital inputs. This benign power-on configuration ensures minimal interaction between the quiescent SLIP state and external devices. These initial pin configurations can be adjusted at a later time as external SLIP connections are required to interact with specific devices with known characteristics.

Appendix C of this report lists a header file that will properly configure the initial microcontroller pin settings. The procedures for loading the header file of Appendix C into the program file and accessing it within the C code should now be familiar. Create a new header file named Config_ports.h. Open this header file and replace the pre-existing contents with the contents of Appendix C. Next, Open the C program file and enter

#include “Config_ports.h”

below the preexisting include statements. Finally, add the statement

“config_ports()”

inside the “main” function of the C code to call the function located in the associated header file.

4. A Demonstration of MPLAB X IDE, ICD 4, and SLIP Functionality

With all housekeeping chores completed, the IDE can now be used to construct and compile a C program to yield executable code, this code can be downloaded to the target SLIP using the ICD 4 programmer/debugger, and finally executed. This exercise will serve two purposes. First, it will highlight a few of the SLIP’s more elementary features. Second, it will demonstrate the workflow process of writing, compiling, downloading, debugging, and executing code.

This demonstration program will access the piezoelectric buzzer, pushbutton switches, and LED status indicators on the SLIP board. Figure 3 illustrates the locations of all these devices. Figure 4 presents the portion of the SLIP schematic that displays the electrical connections between the microcontroller and these three device types. From this figure, it can be seen that the piezoelectric buzzer is controlled by microcontroller pin RD7; the 3-bit LED status indicators are controlled by pins RE0, RE1, and RE2; and the pushbutton switches are controlled by pins RE3 and RE4. All these pins are routed to internal locations on the SLIP board that are not intended for external access. Therefore, these pins are preconfigured to a static state in header file “Config_ports.h” that allows them to perform their function as required at any time.

A simple program that accesses these devices on the target SLIP board is listed in Appendix D. This code can be cut from this appendix and pasted into the IDE C code source file replacing whatever is currently there. Figure 5 illustrates the MPLAB X IDE with this code inserted. Pertinent features of the IDE user interface are labeled.

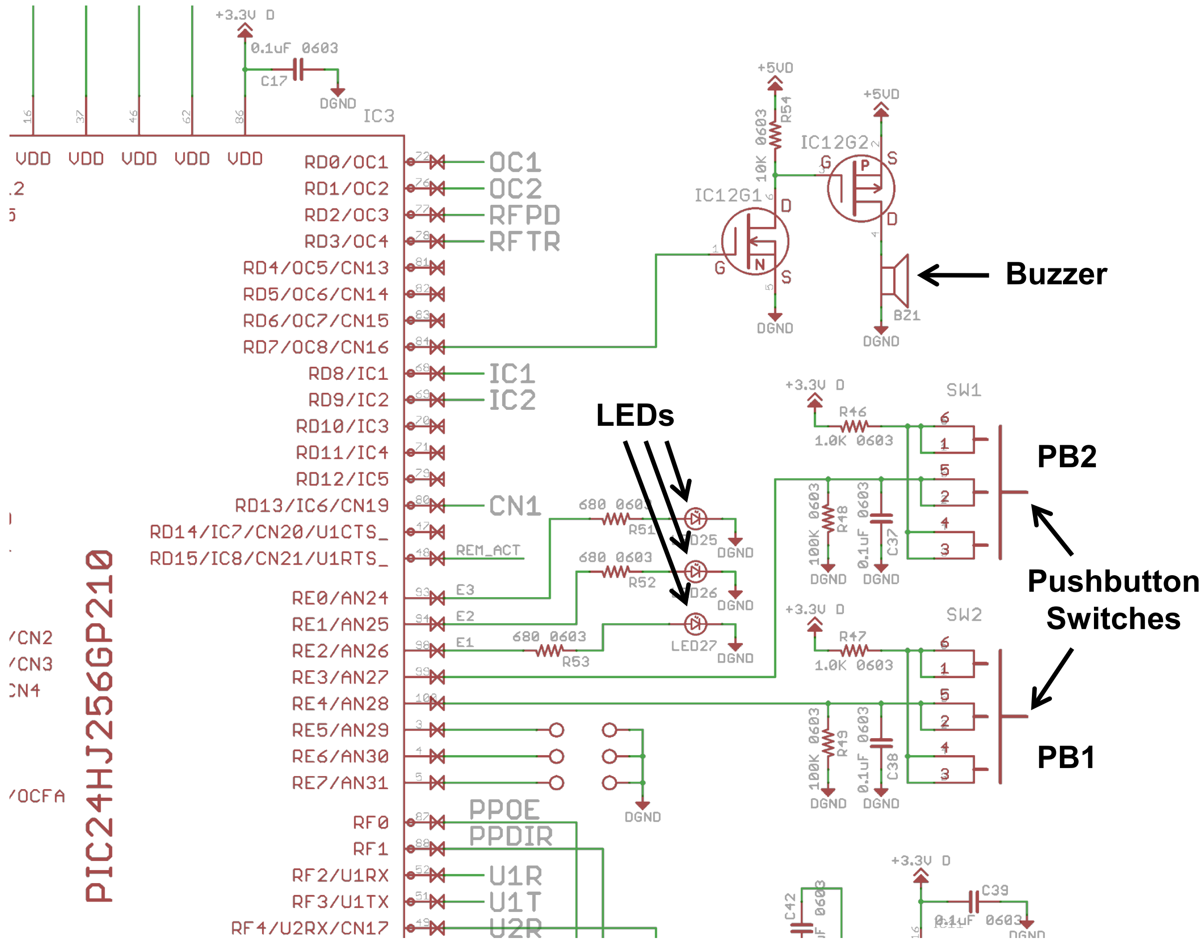

Fig. 4 **Portion of the SLIP electronic schematic**

Extensive comments in this code describe the program's operation. A very general description follows. The program begins with remarks specifying the origin and purpose of the code. This is followed by a listing of the included header files as discussed in the previous housekeeping section. Required variables are then declared. The "main" program begins by calling the functions contained in some of the included header files. This is followed by an infinite loop that first checks the status of the pushbutton switches and turns the piezoelectric buzzer on or off depending on which pushbutton switch is depressed. The program then branches to a small section of code determined by the value of the state variable "state_leds". In this snippet of code, the state variable is incremented to the next allowed value, and the 3-bit LED status indicators are updated appropriately in a four-step process. First, the current Port E value is read and stored in the variable "port_value". The three lowest bits of this stored value are then effectively stripped by ANDing the read value with the bit mask 0b1111111111111000. Next, the new 3-bit LED pattern is inserted by ORing the stripped value with 0b0000000000000XXX, where

XXX is the new desired LED pattern. Finally, the modified Port E value is reloaded into the latch register associated with this port.

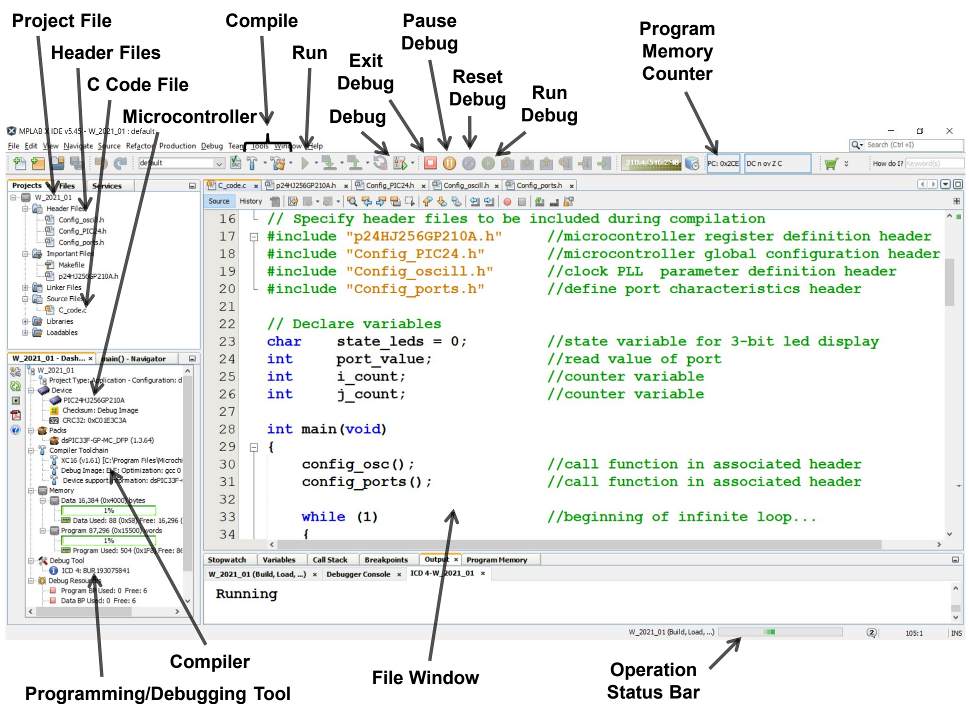

Fig. 5 Example of the MPLAB X IDE user interface

The admittedly modest result on the SLIP is the flashing of the LED status indicator and the ability to turn the buzzer on and off using the pushbutton switches. Clocking at 40 MHz, the previously described section of the while loop will execute in less than a microsecond. At this update rate, the blinking of the LEDs is imperceptible. The final task of the while loop is to effectively count up to 1,024,000. This busy work slows the execution time of the while loop down to almost 200 ms, which allows the blinking of the LEDS to be observed.

The first step toward running this C code on the target SLIP is to simply determine if it will compile successfully in the IDE environment on the PC. Clicking a compile icon in the IDE should generate a listing of compilation information ending with a "BUILD SUCCESSFUL" notification. To purposely force a compilation error, remove the semicolon from the end of any C statement in the while loop and recompile. Now the result is the listing of error code information ending with "BUILD FAILED". Clicking on the error code statement will highlight the vicinity of the C code where the compilation failed. Reinsert the semicolon and recompile.

A next logical step might be to simply execute this code on the target SLIP without debugging capability. To program the SLIP with the executable code requires the ICD 4 to be connected between the PC and the SLIP with power supplied to the SLIP. Clicking the run icon in the IDE will program the SLIP and begin execution. The SLIP can now be disconnected from the ICD 4. Whenever power is applied to the SLIP the downloaded code will be executed. The only way to halt execution is by removing power.

If the execution of the code programmed into the SLIP is not as expected or the details of the execution process wish to be studied, then the debugging capability of the ICD 4 can be employed. With the SLIP reconnected to the ICD 4 and powered, click the debug icon in the IDE. The first line of C code to be executed will be highlighted. Execution of the code can now be initiated by clicking the run debug icon, and temporarily halted by clicking the pause debug icon. In the paused state, the current value of any variable can be inspected by clicking on any white space in the program listing file window and hovering the cursor over the desired variable name. Execution can be restarted from the point where it was halted by again clicking the run debug icon. To start execution from the beginning, click the reset debug icon followed by the run debug icon. Note that any changes made to the C code in the paused state will not take effect until the debugger is exited using the exit debug icon and reentered using the debug icon.

Another useful feature of the debugger is the capability to set breakpoints at any point in the code. Breakpoints are set by clicking the line number in the left-hand gutter in the file window. The specified line will be highlighted and a pink square will be added to the gutter. Execution of the code will automatically be paused when the breakpoint is reached. Breakpoints are removed by clicking the pink square in the gutter.

5. Conclusions and Path Forward

A guide has been presented by which programmers can readily access the SLIP system for application to a wide variety of laboratory tasks. The required hardware and software components have been listed along with sources where they can be obtained. A series of obscure, but necessary housekeeping tasks were presented to streamline the process of configuring, initializing, and activating the SLIP. Finally, a sample C program was provided and discussed to demonstrate the workflow process of writing, compiling, downloading, debugging, and executing code.

The details presented in this report are introductory. A successful SLIP programmer will require additional reference material. Additional information about the SLIP hardware can be obtained from the authors and the following ARL Technical Report:

Kottke T. *An Integration Platform for Adaptive/Cooperative Protection Systems*; 2015 Sep. Report No.: ARL-TR-7422.

An overview and specific details regarding the PIC24HJ256GP210A microcontroller on the SLIP are available at the following:

https://www.microchip.com/wwwproducts/en/PIC24HJ256GP210A

https://ww1.microchip.com/downloads/en/DeviceDoc/70592d.pdf

General information concerning applicable programmer/debuggers and the ICD 4 in particular are available at the following:

https://www.microchip.com/en-us/development-tools-tools-and-software/programmers-and-debuggers

https://www.microchip.com/DevelopmentTools/ProductDetails/PartNO/DV164045

Extensive information about using the MPLAB X Integrated Development Environment can be found at

https://www.microchip.com/content/dam/mchp/documents/MCU08/ProductDocuments/UserGuides/50002027E.pdf

The MPLAB XC16 C Compiler User's Guide is available at

https://ww1.microchip.com/downloads/en/DeviceDoc/50002071K.pdf

6. References and Notes

1. Wikipedia. Hardware abstraction site [accessed 2021 Jan]. https://en.wikipedia.org/wiki/Hardware_abstraction.

2. Microchip Technology. Contact information site [accessed 2021 Jan]. https://www.microchip.com/about-us/contact-us.

3. Microchip Technology. Product information page [accessed 2021 Jan]. https://www.microchip.com/wwwproducts/en/PIC24HJ256GP210A.

4. Kottke T. An integration platform for adaptive/cooperative protection systems. Army Research Laboratory (US); 2015 Sep. Report No.: ARL-TR-7422.

5. Molex Corporation. Product information page [accessed 2021 Jan]. https://www.molex.com/molex/products/part-detail/crimp_housings/0022012027.

6. Digi-Key Electronics. Product information page [accessed 2021 Jan]. https://www.digikey.com/en/products/detail/molex/0022012027/171991.

7. Microchip Technology. Product information page [accessed 2021 Jan]. https://www.microchip.com/developmenttools/ProductDetails/DV164045.

8. Digi-Key Electronics. Product information page [accessed 2021 Jan]. https://www.digikey.com/en/products/detail/microchip-technology/DV164045/7595436?s=N4IgTCBcDaIJYGMAmACALCAugXyA.

9. Wikipedia. Registered jack site [accessed 2021 Jan]. https://en.wikipedia.org/wiki/Registered_jack.

10. Microchip Technology. Product information page [accessed 2021 Jan]. https://www.microchip.com/DevelopmentTools/ProductDetails/PartNO/DV164045.

11. Digi-Key Electronics. Product information page [accessed 2021 Jan]. https://www.digikey.com/en/products/detail/microchip-technology/AC164110/1212490?s=N4IgTCBcDaIIYGMCMA2ALEpAGEBdAvkA.

12. Microchip Technology. Product information page [accessed 2021 Jan]. https://ww1.microchip.com/downloads/en/DeviceDoc/50002538B.pdf.

13. Wikipedia. Information page [accessed 2021 Feb]. https://en.widipedia.org /wiki/Phase-locked_loop.

14. Microchip Technology. Product information page [accessed 2021 Jan]. https://ww1.microchip.com/downloads/en/DeviceDoc/70592d.pdf.

15. Microchip Technology. Product information page [accessed 2021 Feb]. http://ww1.microchip.com/downloads/en/DeviceDoc/DS70227E.pdf.

16. Microchip Technology. Developer help page [accessed 2021 Feb]. https://microchipdeveloper.com/16bit:io-ports.

Appendix A. Listing of PIC24HJ256GP210A Power-On Global Parameter Configuration Code

```
//<<<<< ---------------- 77 CHARACTER WIDTH TEMPLATE ----------------
>>>>>

/*
File:               Config_PIC24.h
Date:               Jan 2021
Language:           MPLAB XC16 C Compiler V1.61
Microprocessor:     PIC24HJ256GP210A
Author:             Tom Kottke
E-Mail:             lambiniboy@hotmail.com
Phone:              443-504-5201
/*

// For a listing of device configuration registers, allowed values, and
// descriptions see section 21.1 of PIC24HJXXXGPX06A/X08A/X10A Data Sheet
// Microchip Technology document DS70592B

// C code must include header include statement:
//     #include "p24HJ256GP210A.h"

//Boot Segment Write Protect
#pragma config BWRP = WRPROTECT_OFF

//Boot Segment Program Flash Code Protection
#pragma config BSS = NO_FLASH

//Boot Segment RAM Protection
#pragma config RBS = NO_RAM

//Secure Segment Program Write Protect
#pragma config SWRP = WRPROTECT_OFF

//Secure Segment Program Flash Code Protection
#pragma config SSS = NO_FLASH

//Secure Segment Data RAM Protection
#pragma config RSS = NO_RAM

//General Code Segment Write Protect
#pragma config GWRP = OFF

//General Segment Code Protect
#pragma config GSS = OFF

//PLL Lock Enable Bit
#pragma config PLLKEN = ON

//Oscillator Mode
#pragma config FNOSC = PRIPLL

//Two-speed Oscillator Start-Up Enable
#pragma config IESO = ON

//Primary Oscillator Source
#pragma config POSCMD = EC

//OSC2 Pin Function
#pragma config OSCIOFNC = OFF
```

```
//Clock Switching and Monitor
#pragma config FCKSM = CSDCMD

//Watchdog Timer Postscaler
#pragma config WDTPOST = PS3276

//Watchdog Timer Prescaler
#pragma config WDTPRE = PR128

//Watchdog Timer Enable
#pragma config FWDTEN = OFF

//Power On Reset Timer Value
#pragma config FPWRT = PWR128

//ICSP Communication Channel Select
#pragma config ICS = PGD2

//JTAG Port Enable
#pragma config JTAGEN = OFF
```

Appendix B. Listing of PIC24HJ256GP210A Run-Time Oscillator Parameters Configuration Code

```
//<<<<< ---------------- 77 CHARACTER WIDTH TEMPLATE ---------------- >>>>>


File:               Config_oscill.h
Date:               Jan 2021
Language:           MPLAB XC16 C Compiler V1.61
Microprocessor:     PIC24HJ256GP210A
Author:             Tom Kottke
E-Mail:             lambiniboy@hotmail.com
Phone:              443-504-5201
*/

//  PROGRAM DESCRIPTION ----------------------------------------------------
-
/*
    This file is used to set run-time oscillator parameters. Power-on reset
    global oscillator configurations must be previously set using
    configuration macros in code_Config_PIC24.h.

    C code must include header include statement:
        #include "Config_oscill.h"

    C code must include the following call to function defined in this header
        at the beginning of the "main" function:
            config_osc();

    Fosc is the frequency of the selected oscillator.

    Fcy is the device instruction clock, which for the PIC24HJX10 is Fosc/2.

    For the PIC24HJ256GP210A, Fcy up to 40MHz is supported.

    The selected oscillator can optionally use an on-chip PLL to obtain
    different speeds of operation. First, the input to the PLL unit goes
    through a division, a multiplication, and another division. The first
    division factor N1 has a value from 2 to 33. The resulting frequency
    Fin/N1 must be between 0.8MHz and 8MHz. Therefore, Fin must be greater
    than 1.6MHz. The multiplication factor M has a value from 2 to 513 and
    must generate a frequency between 100MHz and 200MHz. Finally, the second
    division factor N2 is 2, 4, or 8 and must produce a final frequency in
    the range 12.5MHz to 80MHz.

    Example:  Fin = 10MHz, N1 = 2, M = 32, N2 = 2  =>  Fosc=80MhZ, Fcy=40MHz
*/

void    __attribute__((__no_auto_psv__)) config_osc(void);  // declare func

void    config_osc(void)
{
    PLLFBDbits.PLLDIV = 30; //set M PLL multiplication factor to 32
    CLKDIVbits.PLLPOST = 0; //set N2 PLL division factor to 2
    CLKDIVbits.PLLPRE = 0;  //set N1 PLL division factor to 2
```

```
    while (OSCCONbits.COSC != 0b011);   //wait for clock switch

    while (OSCCONbits.LOCK != 1);       //wait for PLL to lock
}
```

Appendix C. Listing of PIC24HJ256GP210A Ports Configuration Code

```
//<<<<< ----------------- 77 CHARACTER WIDTH TEMPLATE ----------------- >>>>>

/*                  Config_ports.h

File:               Config_ports.h
Date:               Jan 2021
Language:           XC16 C Compiler
Microprocessor:     PIC24HJXXXGPX10
Author:             Tom Kottke
E-Mail:             lambiniboy@hotmail.com
Phone:              443-504-5201

*/

/*  PROGRAM DESCRIPTION -------------------------------------------------------
-

    This file is used to configure the data direction, initial state, and
    analog or digital nature of the individual pins of the microcontroller
    ports.  For each pin a listing of the multiplexed functions is provided
    along with the pin number and function description.

    In the main C program an include statement should be added:

        #include path + Config_ports.h

    and the included function should be called:

        config_ports();

*/

void    __attribute__((__no_auto_psv__)) config_ports(void);  // declare func

void    config_ports(void)
{
//  PORT A --------------------------------------------------------------------
-
    TRISAbits.TRISA0 = 0;   //TMS/RA0           17  LAT_16LED           TUTA
    LATAbits.LATA0 = 0;     //TMS/RA0           17  latch on high

    TRISAbits.TRISA1 = 0;   //TCK/RA1           38  EN_16LED            TUTA
    LATAbits.LATA1 = 1;     //TCK/RA1           38  enable on low

    TRISAbits.TRISA2 = 1;   //SCL2/RA2          58  SCL2 header Left 5   DAUG
    LATAbits.LATA2 = 0;     //SCL2/RA2          58

    TRISAbits.TRISA3 = 1;   //SDA2/RA3          59  SDA2 header Right 5
DAUG
    LATAbits.LATA3 = 0;     //SDA2/RA3          59

    TRISAbits.TRISA4 = 0;   //TDI/RA4           60  EN_LCD_DISP         TUTA
    LATAbits.LATA4 = 1;     //TDI/RA4           60  enable on low

    TRISAbits.TRISA5 = 0;   //TDO/RA5           61  LAT_LCD_DISP        TUTA
```

```
    LATAbits.LATA5 = 0;      //TDO/RA5            61  latch on high

    TRISAbits.TRISA6 = 1;   //AN22/CN22/RA6     91  ADC monitor +3.3V_D  TUTA
    LATAbits.LATA6 = 0;      //AN22/CN22/RA6     91  xxxx
    AD1PCFGHbits.PCFG22 = 0;//AN22/CN22/RA6     91  analog

    TRISAbits.TRISA7 = 1;   //AN23/CN23/RA7     92  ADC monitor of bat   TUTA
    LATAbits.LATA7 = 0;      //AN23/CN23/RA7     92  xxxx
    AD1PCFGHbits.PCFG23 = 0;////AN23/CN23/RA7   92  analog

    TRISAbits.TRISA9 = 1;    //VREF-/RA9          28  not used
    LATAbits.LATA9 = 0;      //VREF-/RA9          28  xxxx

    TRISAbits.TRISA10 = 1;  //VREF+/RA10         29  ADC +3V reference    TUTA
    LATAbits.LATA10 = 0;     //VREF+/RA10         29  xxxx

    TRISAbits.TRISA12 = 1;  //AN20/INT1/RA12     18  INT1 header Right 11 DAUG
    LATAbits.LATA12 = 0;     //AN20/INT1/RA12     18  xxxx
    AD1PCFGHbits.PCFG20 = 1;//AN20/INT1/RA12     18  digital

    TRISAbits.TRISA13 = 1;  //AN21/INT2/RA13     19  INT2 header Left 11  DAUG
    LATAbits.LATA13 = 0;     //AN21/INT2/RA13     19  xxxx
    AD1PCFGHbits.PCFG21 = 1;//AN21/INT2/RA13     19  digital

    TRISAbits.TRISA14 = 1;  //INT3/RA14          66  INT3 header Right 2  DAUG
    LATAbits.LATA14 = 0;     //INT3/RA14          66  xxxx

    TRISAbits.TRISA15 = 1;  //INT4/RA15          67  INT4 header Left 2   DAUG
    LATAbits.LATA15 = 0;     //INT4/RA15          67  xxxx

//  PORT B --------------------------------------------------------------
    TRISBbits.TRISB0 = 1;   //PGED3/AN0/CN2/RB0 25  P00                  TUTA
    LATBbits.LATB0 = 0;      //PGED3/AN0/CN2/RB0 25  xxxx
    AD1PCFGLbits.PCFG0 = 1; //PGED3/AN0/CN2/RB0 25  digital

    TRISBbits.TRISB1 = 1;   //PGEC3/AN1/CN3/RB1 24  P01                  TUTA
    LATBbits.LATB1 = 0;      //PGEC3/AN1/CN3/RB1 24  xxxx
    AD1PCFGLbits.PCFG1 = 1; //PGEC3/AN1/CN3/RB1 24  digital

    TRISBbits.TRISB2 = 1;   //AN2/SS1/CN4/RB2   23  P02                  TUTA
    LATBbits.LATB2 = 0;      //AN2/SS1/CN4/RB2   23  xxxx
    AD1PCFGLbits.PCFG2 = 1; //AN2/SS1/CN4/RB2   23  digital

    TRISBbits.TRISB3 = 1;   //AN3/CN5/RB3       22  P03                  TUTA
    LATBbits.LATB3 = 0;      //AN3/CN5/RB3       22  xxxx
    AD1PCFGLbits.PCFG3 = 1; //AN3/CN5/RB3       22  digital

    TRISBbits.TRISB4 = 1;   //AN4/CN6/RB4       21  P04                  TUTA
    LATBbits.LATB4 = 0;      //AN4/CN6/RB4       21  xxxx
    AD1PCFGLbits.PCFG4 = 1; //AN4/CN6/RB4       21  digital

    TRISBbits.TRISB5 = 1;   //AN5/CN7/RB5       20  P05                  TUTA
    LATBbits.LATB5 = 0;      //AN5/CN7/RB5       20  xxxx
    AD1PCFGLbits.PCFG5 = 1; //AN5/CN7/RB5       20  digital

    TRISBbits.TRISB6 = 1;   //PGEC1/AN6/OCFA/RB6 26 P06                  TUTA
    LATBbits.LATB6 = 0;      //PGEC1/AN6/OCFA/RB6 26 xxxx
    AD1PCFGLbits.PCFG6 = 1; //PGEC1/AN6/OCFA/RB6 26 digital
```

```
    TRISBbits.TRISB7 = 1;   //PGED1/AN7/RB7     27  P07                 TUTA
    LATBbits.LATB7 = 0;     //PGED1/AN7/RB7     27  xxxx
    AD1PCFGLbits.PCFG7 = 1; //PGED1/AN7/RB7     27  digital

    TRISBbits.TRISB8 = 1;   //AN8/RB8           32  P08                 TUTA
    LATBbits.LATB8 = 0;     //AN8/RB8           32  xxxx
    AD1PCFGLbits.PCFG8 = 1; //AN8/RB8           32  digital

    TRISBbits.TRISB9 = 1;   //AN9/RB9           33  P09                 TUTA
    LATBbits.LATB9 = 0;     //AN9/RB9           33  xxxx
    AD1PCFGLbits.PCFG9 = 1; //AN9/RB9           33  digital

    TRISBbits.TRISB10 = 1;  //AN10/RB10         34  P10                 TUTA
    LATBbits.LATB10 = 0;    //AN10/RB10         34  xxxx
    AD1PCFGLbits.PCFG10 = 1;//AN10/RB10         34  digital

    TRISBbits.TRISB11 = 1;  //AN11/RB11         35  P11                 TUTA
    LATBbits.LATB11 = 0;    //AN11/RB11         35  xxxx
    AD1PCFGLbits.PCFG11 = 1;//AN11/RB11         35  digital

    TRISBbits.TRISB12 = 1;  //AN12/RB12         41  P12                 TUTA
    LATBbits.LATB12 = 0;    //AN12/RB12         41  xxxx
    AD1PCFGLbits.PCFG12 = 1;//AN12/RB12         41  digital

    TRISBbits.TRISB13 = 1;  //AN13/RB13         42  P13                 TUTA
    LATBbits.LATB13 = 0;    //AN13/RB13         42  xxxx
    AD1PCFGLbits.PCFG13 = 1;//AN13/RB13         42  digital

    TRISBbits.TRISB14 = 1;  //AN14/RB14         43  P14                 TUTA
    LATBbits.LATB14 = 0;    //AN14/RB14         43  xxxx
    AD1PCFGLbits.PCFG14 = 1;//AN14/RB14         43  digital

    TRISBbits.TRISB15 = 1;  //AN15/OCFB/CN12/RB15 44 P15                TUTA
    LATBbits.LATB15 = 0;    //AN15/OCFB/CN12/RB15 44 xxxx
    AD1PCFGLbits.PCFG15 = 1;//AN15/OCFB/CN12/RB15 44 digital

//  PORT C ---------------------------------------------------------------
-
    TRISCbits.TRISC1 = 1;   //AN16/T2CK/T7CK/RC1 6  not used
    LATCbits.LATC1 = 0;     //AN16/T2CK/T7CK/RC1 6  xxxx
    AD1PCFGHbits.PCFG16 = 1;//AN16/T2CK/T7CK/RC1 6  digital

    TRISCbits.TRISC2 = 1;   //AN17/T3CK/T6CK/RC2 7  not used
    LATCbits.LATC2 = 0;     //AN17/T3CK/T6CK/RC2 7  xxxx
    AD1PCFGHbits.PCFG17 = 1;//AN17/T3CK/T6CK/RC2 7  digital

    TRISCbits.TRISC3 = 1;   //AN18/T4CK/T9CK/RC3 8  ADC monitor +5V_D    TUTA
    LATCbits.LATC3 = 0;     //AN18/T4CK/T9CK/RC3 8  xxxx
    AD1PCFGHbits.PCFG18 = 0;//AN18/T4CK/T9CK/RC3 8  analog

    TRISCbits.TRISC4 = 1;   //AN19/T5CK/T8CK/RC4 9  ADC monitor +3.3V_A  TUTA
    LATCbits.LATC4 = 0;     //AN19/T5CK/T8CK/RC4 9  xxxx
    AD1PCFGHbits.PCFG19 = 0;//analog

    TRISCbits.TRISC12 = 1;  //OSC1/CLKIN/RC12   63  OSC input           TUTA
    LATCbits.LATC12 = 0;    //OSC1/CLKIN/RC12   63  xxxx
```

```
    TRISCbits.TRISC13 = 1;  //PGED2/SOSCI/CN1/RC13 73 ICSP data         TUTA
    LATCbits.LATC13 = 0;    //PGED2/SOSCI/CN1/RC13 73 xxxx

    TRISCbits.TRISC14 = 1;  //PGEC2/SOSCO/T1CK/CN0/RC14 74 ICSP clock    TUTA
    LATCbits.LATC14 = 0;    //PGEC2/SOSCO/T1CK/CN0/RC14 74 xxxx

    TRISCbits.TRISC15 = 1;  //OSC2/CLKO/RC15    64  not used
    LATCbits.LATC15 = 0;    //OSC2/CLKO/RC15    64  xxxx

//  PORT D ---------------------------------------------------------------
-
    TRISDbits.TRISD0 = 1;   //OC1/RD0          72  OC1 header Right 4   DAUG
    LATDbits.LATD0 = 0;     //OC1/RD0           72

    TRISDbits.TRISD1 = 1;   //OC2/RD1          76  OC2 header Left 4    DAUG
    LATDbits.LATD1 = 0;     //OC2/RD1           76

    TRISDbits.TRISD2 = 0;   //OC3/RD2          77  r/f link power down  TUTA
    LATDbits.LATD2 = 0;      //OC3/RD2            77  low output => powered
down

    TRISDbits.TRISD3 = 0;   //OC4/RD3          78  r/f link trans/recv  TUTA
    LATDbits.LATD3 = 0;     //OC4/RD3           78  low output => receive

    TRISDbits.TRISD4 = 1;   //OC5/CN13/RD4      81  not used
    LATDbits.LATD4 = 0;     //OC5/CN13/RD4      81  xxxx

    TRISDbits.TRISD5 = 1;   //OC6/CN14/RD5      82  not used
    LATDbits.LATD5 = 0;     //OC6/CN14/RD5      82  xxxx

    TRISDbits.TRISD6 = 1;   //OC7/CN15/RD6      83  not used
    LATDbits.LATD6 = 0;     //OC7/CN15/RD6      83  xxxx

    TRISDbits.TRISD7 = 0;   //OC8/CN16/RD7      84  BUZZER               TUTA
    LATDbits.LATD7 = 0;     //OC8/CN16/RD7      84  low output => OFF

    TRISDbits.TRISD8 = 1;   //IC1/RD8          68  IC1 header Right 3   DAUG
    LATDbits.LATD8 = 0;     //IC1/RD8           68

    TRISDbits.TRISD9 = 1;   //IC2/RD9          69  IC2 header Left 3    DAUG
    LATDbits.LATD9 = 0;     //IC2/RD9           69  low output => high FIRE

    TRISDbits.TRISD10 = 1;  //IC3/RD10          70  not used
    LATDbits.LATD10 = 0;    //IC3/RD10          70  xxxx

    TRISDbits.TRISD11 = 1;  //IC4/RD11          71  not used
    LATDbits.LATD11 = 0;    //IC4/RD11          71  xxxx

    TRISDbits.TRISD12 = 1;  //IC5/RD12          79  not used
    LATDbits.LATD12 = 0;    //IC5/RD12          79  xxxx

    TRISDbits.TRISD13 = 1;  //IC6/CN19/RD13     80  CN1 header Right 9   DAUG
    LATDbits.LATD13 = 0;    //IC6/CN19/RD13     80

    TRISDbits.TRISD14 = 1;  //IC7/U1CTS/CN20/RD14 47 not used
    LATDbits.LATD14 = 0;    //IC7/U1CTS/CN20/RD14 47 xxxx

    TRISDbits.TRISD15 = 1;  //IC8/U1RTS/CN21/RD15 48 REMote_ACTivation   TUTA
```

```
    LATDbits.LATD15 = 0;    //IC8/U1RTS/CN21/RD15 48 high output => ON

//  PORT E ---------------------------------------------------------------
-
    TRISEbits.TRISE0 = 0;   //AN24/RE0          93  Bit 3 LED           TUTA
    LATEbits.LATE0 = 0;     //AN24/RE0          93  low output => OFF
    AD1PCFGHbits.PCFG24 = 1;//AN24/RE0          93  digital

    TRISEbits.TRISE1 = 0;   //AN25/RE1          94  Bit 2 LED           TUTA
    LATEbits.LATE1 = 0;     //AN25/RE1          94  low output => OFF
    AD1PCFGHbits.PCFG25 = 1;//AN25/RE1          94  digital

    TRISEbits.TRISE2 = 0;   //AN26/RE2          98  Bit 1 LED           TUTA
    LATEbits.LATE2 = 0;     //AN26/RE2          98  low output => OFF
    AD1PCFGHbits.PCFG26 = 1;//AN26/RE2          98  digital

    TRISEbits.TRISE3 = 1;   //AN27/RE3          99  SWitch 1 input      TUTA
    LATEbits.LATE3 = 0;     //AN27/RE3          99  xxxx
    AD1PCFGHbits.PCFG27 = 1;//AN27/RE3          99  digital

    TRISEbits.TRISE4 = 1;   //AN28/RE4          100 SWitch 2 input      TUTA
    LATEbits.LATE4 = 0;     //AN28/RE4          100 xxxx
    AD1PCFGHbits.PCFG28 = 1;//AN28/RE4          100 digital

    TRISEbits.TRISE5 = 0;   //AN29/RE5          3   Fire 3 output       TUTA
    LATEbits.LATE5 = 0;     //AN29/RE5          3   low output
    AD1PCFGHbits.PCFG29 = 1;//AN29/RE5          3   digital

    TRISEbits.TRISE6 = 0;   //AN30/RE6          4   Fire 2 output       TUTA
    LATEbits.LATE6 = 0;     //AN30/RE6          4   low output
    AD1PCFGHbits.PCFG30 = 1;//AN30/RE6          4   digital

    TRISEbits.TRISE7 = 0;   //AN31/RE7          5   Fire 1 output       TUTA
    LATEbits.LATE7 = 0;     //AN31/RE7          5   low output
    AD1PCFGHbits.PCFG31 = 1;//AN31/RE7          5   digital

//  PORT F ---------------------------------------------------------------
-
    TRISFbits.TRISF0 = 0;   //RF0               87  Par Port Output En   TUTA
    LATFbits.LATF0 = 1;     //RF0               87  high output => disable

    TRISFbits.TRISF1 = 0;   //RF1               88  Par Port Direction   TUTA
    LATFbits.LATF1 = 0;     //RF1               88  low output => B -> A

    TRISFbits.TRISF2 = 1;   //U1RX/RF2          52  Uart 1 RX, r/f link  TUTA
    LATFbits.LATF2 = 0;     //U1RX/RF2          52  header Right 7       DAUG

    TRISFbits.TRISF3 = 0;   //U1TX/RF3          51  Uart 1 TX, r/f link  TUTA
    LATFbits.LATF3 = 1;     //U1TX/RF3          51  header Left 7        DAUG

    TRISFbits.TRISF4 = 1;   //U2RX/CN17/RF4     49  Uart 2 RX, hardwire  TUTA
    LATFbits.LATF4 = 0;     //U2RX/CN17/RF4     49  header Left 8        DAUG

    TRISFbits.TRISF5 = 0;   //U2TX/CN18/RF5     50  Uart 2 TX, hardwire  TUTA
    LATFbits.LATF5 = 1;     //U2TX/CN18/RF5     50  header Right 8       DAUG

    TRISFbits.TRISF6 = 1;   //SCK1/INT0/RF6     55  SCK1 header Left 1   DAUG
    LATFbits.LATF6 = 0;     //SCK1/INT0/RF6     55
```

```
    TRISFbits.TRISF7 = 1;   //SDI1/RF7          54  SDI1 header Right 1  DAUG
    LATFbits.LATF7 = 0;     //SDI1/RF7          54

    TRISFbits.TRISF8 = 1;   //SDO1/RF8          53  SDO1 header Left 0   DAUG
    LATFbits.LATF8 = 0;     //SDO1/RF8          53

    TRISFbits.TRISF12 = 1;  //U2CTS/RF12        40  not used
    LATFbits.LATF12 = 0;    //U2CTS/RF12        40  xxxx

    TRISFbits.TRISF13 = 1;  //U2RTS/RF13        39  not used
    LATFbits.LATF13 = 0;    //U2RTS/RF13        39  xxxx

//  PORT G -------------------------------------------------------------
-
    TRISGbits.TRISG0 = 1;   //RG0               90  not used
    LATGbits.LATG0 = 0;     //RG0               90  xxxx

    TRISGbits.TRISG1 = 1;   //RG1               89  not used
    LATGbits.LATG1 = 0;     //RG1               89  xxxx

    TRISGbits.TRISG2 = 1;   //SCL1/RG2          57  SCL1 header Right 6  DAUG
    LATGbits.LATG2 = 0;     //SCL1/RG2          57

    TRISGbits.TRISG3 = 1;   //SDA1/RG3          56  SDA1 header Left 6   DAUG
    LATGbits.LATG3 = 0;     //SDA1/RG3          56

    TRISGbits.TRISG6 = 1;   //SCK2/CN8/RG6      10  SCK2 header Left 9   DAUG
    LATGbits.LATG6 = 0;     //SCK2/CN8/RG6      10

    TRISGbits.TRISG7 = 1;   //SDI2/CN9/RG7      11  SDI2 header Right 10 DAUG
    LATGbits.LATG7 = 0;     //SDI2/CN9/RG7      11

    TRISGbits.TRISG8 = 1;   //SDO2/CN10/RG8     12  SDO2 header Left 10  DAUG
    LATGbits.LATG8 = 0;     //SDO2/CN10/RG8     12

    TRISGbits.TRISG9 = 1;   //SS2/CN11/RG9      14  not used
    LATGbits.LATG9 = 0;     //SS2/CN11/RG9      14  xxxx

    TRISGbits.TRISG12 = 1;  //RG12              96  not used
    LATGbits.LATG12 = 0;    //RG12              96  xxxx

    TRISGbits.TRISG13 = 1;  //RG13              97  not used
    LATGbits.LATG13 = 0;    //RG13              97  xxxx

    TRISGbits.TRISG14 = 1;  //RG14              95  not used
    LATGbits.LATG14 = 0;    //RG14              95  xxxx

    TRISGbits.TRISG15 = 1;  //RG15              1   not used
    LATGbits.LATG15 = 0;    //RG15              1   xxxx
}
```

Appendix D. Listing of a Simple Laboratory Integration Platform (SLIP) Demonstration C Code

```
//<<<<< ----------------- 77 CHARACTER WIDTH TEMPLATE ----------------- >>>>>

/*
File:               C_code.c
Date:               Jan 2021
Language:           XC16 C Compiler
Microprocessor:     PIC24HJXXXGPX10
Author:             Tom Kottke
E-Mail:             lambiniboy@hotmail.com
Phone:              443-504-5201
*/
// This is the C code for the demonstration program presented in "A Primer
// for Programming and Applying the Simple Laboratory Integration Platform
// (SLIP)"

// Specify header files to be included during compilation
#include "p24HJ256GP210A.h"     //microcontroller register definition header
#include "Config_PIC24.h"      //microcontroller global configuration header
#include "Config_oscill.h"      //clock PLL  parameter definition header
#include "Config_ports.h"       //define port characteristics header

// Declare variables
char    state_leds = 0;         //state variable for 3-bit led display
int     port_value;             //read value of port
int     i_count;                //counter variable
int     j_count;                //counter variable

int main(void)
{
    config_osc();               //call function in associated header
    config_ports();             //call function in associated header

    while (1)                   //beginning of infinite loop...
    {
        if (PORTEbits.RE4)      //if pushbutton switch PB1 is pressed...
        {
            LATDbits.LATD7 = 1; //...then turn on audio buzzer
        }

        if (PORTEbits.RE3)      //if pushbutton switch PB2 is pressed...
        {
            LATDbits.LATD7 = 0; //...then turn off audio buzzer
        }

        switch (state_leds)     //switch branching on state of leds
        {
            case 0:                     //if state of leds is zero...
                    state_leds = 1;     //increment led state variable
                    port_value = PORTE; //read current port value
                    port_value &= 0b1111111111111000;   //strip last 3 bits
                    port_value |= 0b0000000000000000;   //replace last 3 bits
                    LATE = port_value;  //output modified port value
                     break;
            case 1:                     //if state of leds is one...
                    state_leds = 2;     //increment led state variable
                    port_value = PORTE; //read current port value
                    port_value &= 0b1111111111111000;   //strip last 3 bits
```

```
                port_value |= 0b0000000000000001;   //replace last 3 bits
                LATE = port_value;  //output modified port value
                break;
            case 2:                     //if state of leds is two...
                state_leds = 3;     //increment led state variable
                port_value = PORTE; //read current port value
                port_value &= 0b1111111111111000;   //strip last 3 bits
                port_value |= 0b0000000000000010;   //replace last 3 bits
                LATE = port_value;  //output modified port value
                break;
            case 3:                     //if state of leds is three...
                state_leds = 4;     //increment led state variable
                port_value = PORTE; //read current port value
                port_value &= 0b1111111111111000;   //strip last 3 bits
                port_value |= 0b0000000000000011;   //replace last 3 bits
                LATE = port_value;  //output modified port value
                break;
            case 4:                     //if state of leds is four...
                state_leds = 5;     //increment led state variable
                port_value = PORTE; //read current port value
                port_value &= 0b1111111111111000;   //strip last 3 bits
                port_value |= 0b0000000000000100;   //replace last 3 bits
                LATE = port_value;  //output modified port value
                break;
            case 5:                     //if state of leds is five...
                state_leds = 6;     //increment led state variable
                port_value = PORTE; //read current port value
                port_value &= 0b1111111111111000;   //strip last 3 bits
                port_value |= 0b0000000000000101;   //replace last 3 bits
                LATE = port_value;  //output modified port value
                break;
            case 6:                     //if state of leds is six...
                state_leds = 7;     //increment led state variable
                port_value = PORTE; //read current port value
                port_value &= 0b1111111111111000;   //strip last 3 bits
                port_value |= 0b0000000000000110;   //replace last 3 bits
                LATE = port_value;  //output modified port value
                break;
            case 7:                     //if state of leds is seven...
                state_leds = 0;     //increment led state variable
                port_value = PORTE; //read current port value
                port_value &= 0b1111111111111000;   //strip last 3 bits
                port_value |= 0b0000000000000111;   //replace last 3 bits
                LATE = port_value;  //output modified port value
                break;
        }                               //end of switch branch

        for(i_count=0;i_count<32000;i_count++)  //super crude time delay
        {
            for(j_count=0;j_count<32;j_count++)
            {

            }
        }                                       //end of time delay routine
    }                                           //end of infinite while loop
}                                               //end of main C function
```

List of Symbols, Abbreviations, and Acronyms

ARL	Army Research Laboratory
DEVCOM	US Army Combat Capabilities Development Command
ICSP	in-circuit serial programming port
IDE	integrated development environment
I/O	input/output
IR	infrared
LED	light-emitting diode
PC	personal computer
PCB	printed circuit board
SLIP	Simple Laboratory Integration Platform
USB	Universal Serial Bus

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	DEVCOM ARL FCDD RLD DCI TECH LIB
24 (PDF)	DEVCOM ARL FCDD RLW TA T KOTTKE J FLENIKEN S BILYK P BERNING M COPPINGER STEVEN DAVIS K KWASHNAK M MCNEIR D MALONE J NESTA W UHLIG L VANDERHOEF A VALENZUELA B WILMER C WOLFE FCDD RLW S SCHOENFELD FCDD RLW B C HOPPEL FCDD RLW C D LYON FCDD RLW TB M KLEINBERGER R KARGUS FCDD RLW TG N GNIAZDOWSKI E FIORAVANTE S TROMBETTA J PRITCHETT