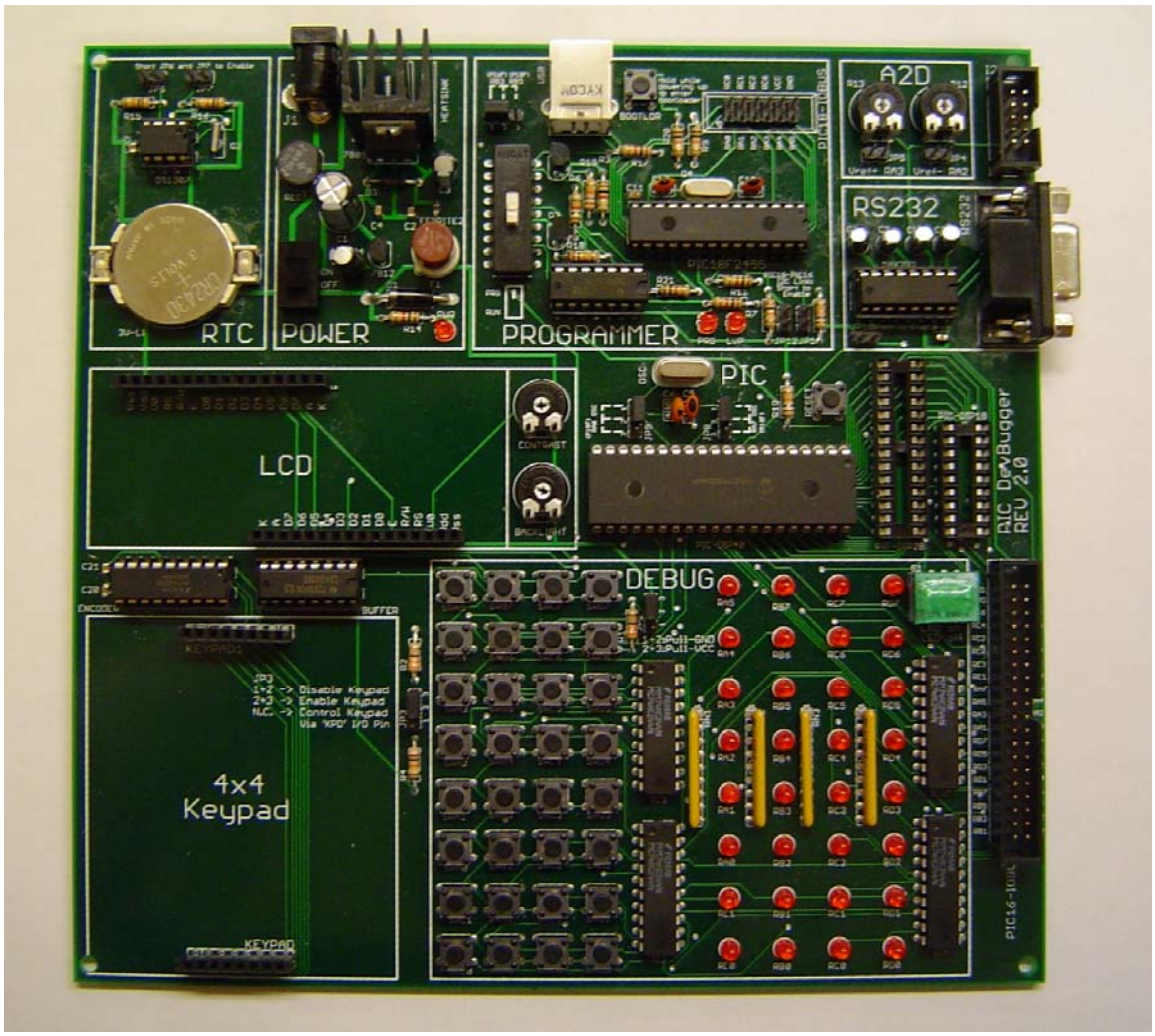
AER201 – Engineering Design

PIC *DevBugger* Manual



1 Table of Contents

2 Introduction

2.1 Overview

The *PIC DevBugger* USB development board was designed as a complete mobile solution to PIC development, including a full-speed USB programmer and a number of peripheral modules. One especially useful feature of the board is the debugging module, which monitors all pin states and allows the user to simulate inputs to the PIC. The board is ideal for students, as it can be used to quickly develop and test code. Its uses can vary from an exclusive development platform to a full-scale embedded processing/control system used in a final design.

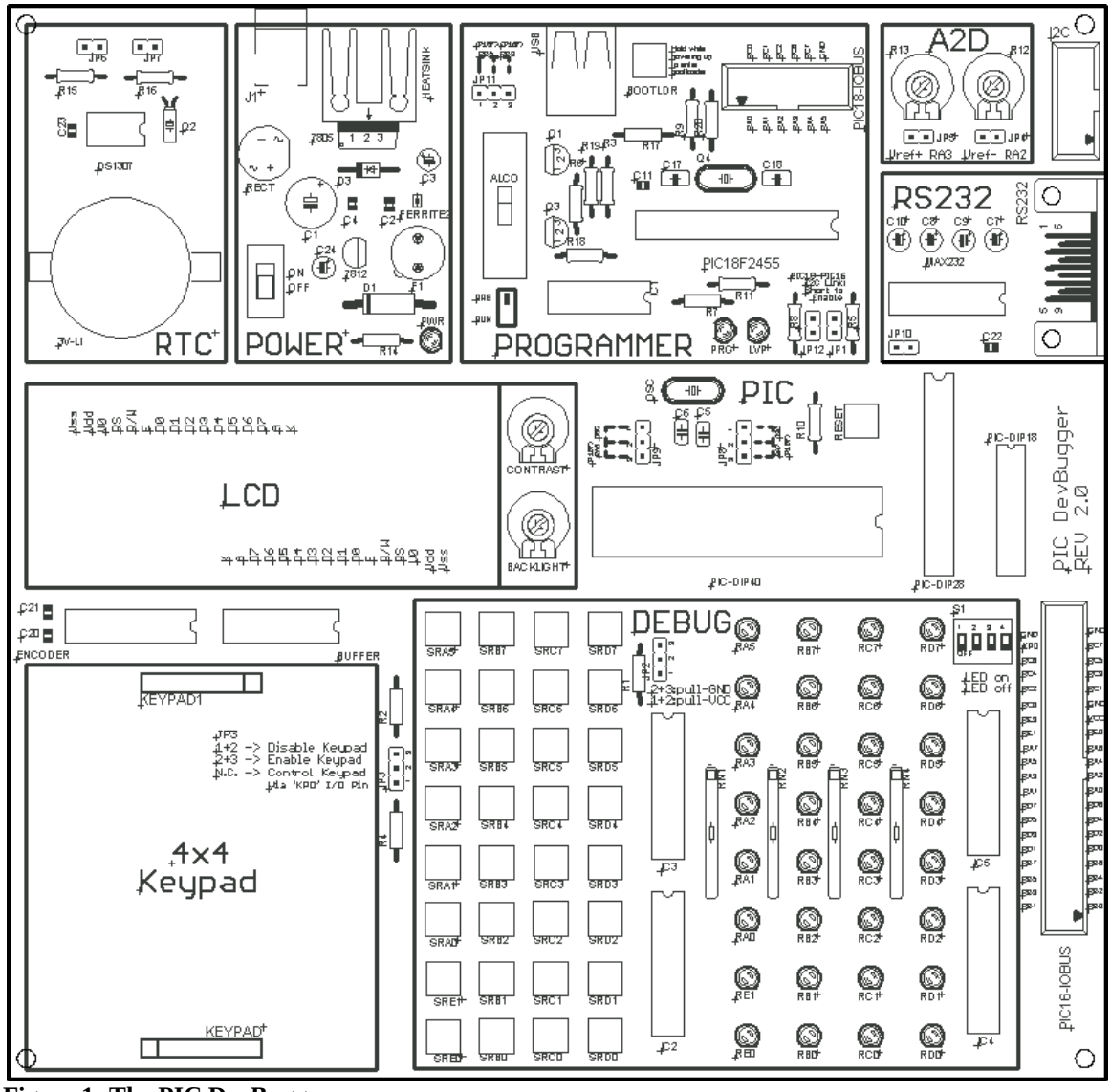


Figure 1: The PIC DevBugger

2.2 Features

- User-changeable oscillator clock (10MHz crystal included)
- In-circuit USB Programmer supporting both High and Low Voltage Programming
- On-board RS232 peripheral including female socket and level converter
- Power supply compatible with voltages ranging from 7.5VDC to 17VDC
- Real Time Clock peripheral with 32.768khz crystal and battery socket
- I2C bus expansion socket
- On-board HD44780 LCD socket with contrast and backlight controls
- On-board 4x4 keypad socket and signal encoder
- 40-pin I/O bus with ribbon cable connector
- On-board A2D voltage reference settings
- Debugging Module with 32 indicator LEDs and signal-simulation buttons
- Programmer firmware can be modified to act as extra CPU/memory

2.3 Included in the Box

The development system should come with the following items:

- *PIC DevBugger* USB development board
- CD/URL from which to download necessary software/drivers
- USB cable
- 40-pin I/O bus cable
- HD44780-controlled LCD display
- 4x4 matrix keypad
- Wall adapter unit (15V)

3 Operation

3.1 Operational Modes

The PIC *DevBugger* has three modes of operation, as follows:

Programming: Used to load compiled HEX code onto the PIC device. To enter this mode, flip the Programmer module's slide switch to the PRG position. The VDD, VPP, PGC, PGD and PGM pins of the main PIC device are disconnected from the main I/O bus, and connected instead to the Programmer module. Code may or may not execute on the target device while in this mode.

Executing: This is the primary operational mode of the board. To enter this mode, flip the Programmer's slide switch to the RUN position. All PIC device I/O pins are connected to the I/O bus, and code executes freely. In addition, if the Bootloader features are used, the programmer's user-defined code begins to execute after a delay of a fraction of a second. In this mode, the power supply only needs to receive 7.5V input for full functionality.

Bootloader: For advanced users only. This mode allows the user to re-program the firmware of the Programmer module via the USB link. To enter this mode, turn off the board's power supply, then hold the BTLDR button on the Programmer module while turning the power supply back on. More details on using the Bootloader feature can be found in Section 6. In this mode, the power supply only needs to receive 7.5V input for full functionality.

3.2 Connecting to the PC for Programming

In order to load HEX code to the PIC device, the *DevBugger* must connect to a PC:

1. Install the **PICusb** programming software before connecting the board:
    a. Run the PICusb setup program
    b. If a prompt concerning Windows Logo testing or compatibility appears, select "Continue Anyway". This is expected behaviour.
2. Connect the 15V wall adapter to the board
3. Set the board to **Programming** mode (Section 3.1)
4. Flip the power switch to the ON position (Section 4.1)
5. Connect the *DevBugger* board to the PC using the included USB cable
6. The PC should detect the device and open a wizard.
    a. Do not connect to Windows Update.
    b. On the second screen, choose the automatic installation.
    c. If a prompt concerning Windows Logo testing or compatibility appears, select "Continue Anyway". This is expected behaviour.
7. Ensure all jumpers for the Programmer module are set correctly (Section 4.2)

8. The PIC device on the board is now ready to be programmed. (Section 5)

3.3 Customizing Board Operation

The *DevBugger* was designed with versatility in mind. To customize the operation of the board, several configuration jumpers and switches have been included, which must be set by the user. Before using the board for the first time, please ensure that the jumpers for each module have been configured as desired. More information about jumper settings for individual modules can be found in Section 4.

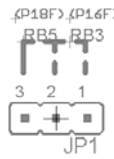


Figure 2

For those unfamiliar with jumpers, a jumper is a set of 2 or more exposed pins that can be connected adjacently in pairs, using a 'shunt'. When two pins of the jumper are connected to each other, they are 'shorted'. As a case example, if pins 1 and 2 in the following figure are connected using a shunt, then we say we have 'shorted' pins 1+2, and we have configured the board for use with a P16F type microcontroller as labeled above the jumper.

3.4 Interfacing with External Circuits

While the board is operating in **Executing** mode, all I/O pins of the PIC microcontroller are directly connected to the main I/O bus socket. Using the provided ribbon cable, this bus can be used to interface directly with external circuitry, since it provides a +5V supply and ground reference in addition to direct access to the PIC device's pins. Also note that the Keypad peripheral adds an extra pin to the bus, for on-the-fly enable/disable control, as described in its respective section. A detailed description of the ribbon cable's pin connections is given in Section 4.9.

WARNING: the 5V supply on the bus is NOT intended as an alternate method of powering the DevBugger board, and NO guarantee is made for the continued integrity of the board in cases of such usage.

4 Board Modules

4.1 Power Supply

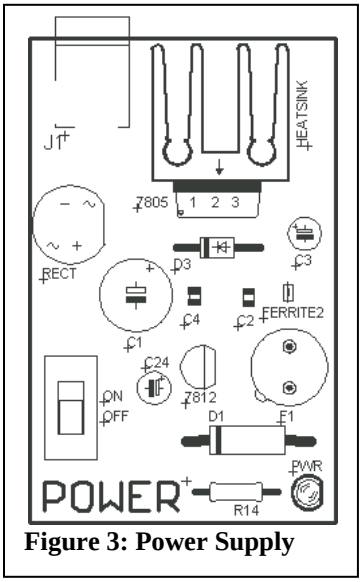


Figure 3: Power Supply

The Power Supply module is designed to take DC input of 7.5V to 17V, and output a regulated 5V for all modules to share including the I/O bus. The max current capacity for all modules combined is 1A, enforced by a small replaceable fuse (1.25A, TR5 size).

The input connector is a female 5.5x2.1mm jack, which is compatible with many commonplace adapters, and wall adapters of any polarity may be used since the input is rectified.

This module also supplies the Programmer with a +12V regulated source, for use in high-voltage programming. As such, if the input voltage is less than +13V, only the Low-Voltage programming mode may be used.

A single slide switch controls power to the entire board by interrupting the positive power terminal immediately after rectification. In the UP position, the board is powered, as indicated by a red LED; in the DOWN position, all modules are unpowered.

4.2 On-board USB Programmer

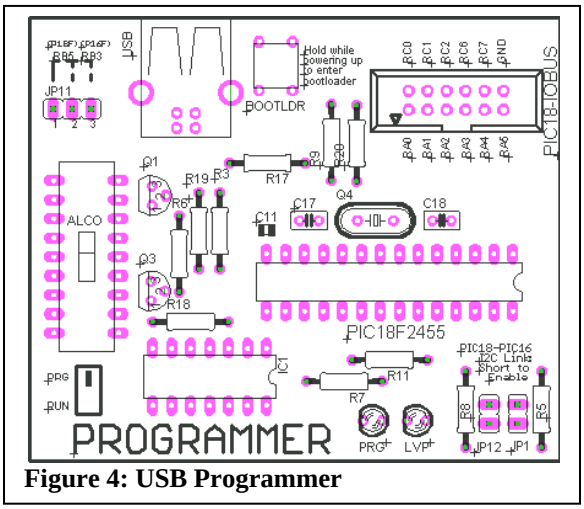


Figure 4: USB Programmer

The *DevBugger* includes an on-board USB PIC programmer. The programmer can operate in either High Voltage Programming (**HVP)** mode or Low Voltage Programming (**LVP**) mode, depending on PC-side software settings and the power supply.

For **HVP** mode, the power supply must be 13V or greater, and the state of **JP11** is ignored.

For **LVP** mode, the power supply must be 7.5V or greater. For programming PIC16F family devices, pins 1+2 of **JP11** must be shorted, and for PIC18F family devices, pins 2+3 must be shorted.

To enter Programming Mode, set the slide switch to the UP position, and to enter Executing mode, set the slide switch to the DOWN position. To enter **Bootloader** mode (see Section 6), turn off the board power supply, hold the BTLDR button, and turn the power supply back on.

For advanced users wishing to modify the programmer's source code in order to take advantage of parallel computing, an I2C link may be established between the PIC18F2455 of the Programmer module and the main PIC device. To physically establish the electrical connection between the two devices, jumpers **JP1** and **JP12** must be shorted.

Users who want to use the PIC16 register monitoring software or the PIC18's I/O pins from the PIC16 during runtime also need to short jumpers **JP1** and **JP12**. Both of these functions – runtime register monitoring and PIC18 I/O pin access – also make use of the I2C bus implemented on the DevBugger board between the PIC16 and PIC18. Please refer to the I2C section of this manual for more information on these capabilities.

4.3 Debugging Module

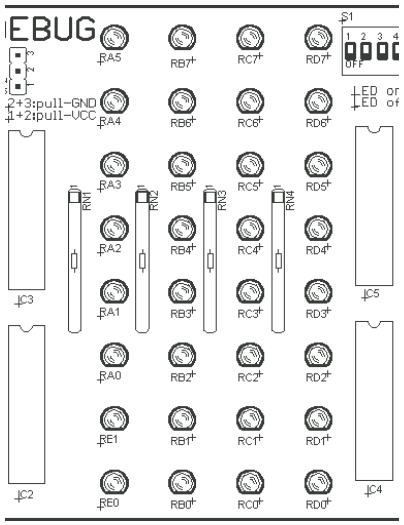


Figure 5: Indicator LEDs

The Debugging module allows the user to monitor the states of the PIC device's I/O pins via 32 indicator LED's. These indicators are fully buffered, so they do not impact the voltage levels of the signals they are monitoring – in other words, they can safely monitor the logic states of low-power sensor signals. However, it should be noted that unless specific lines are connected to signals or driven by the PIC's I/O ports, their indicators may flash unpredictably since they are floating.

The indicators can be disabled in four columns through the included DIP switch, in order to reduce current draw.

To control the LEDs:
Slide switch 1 for **PORTA/PORTE**
Slide switch 2 for **PORTB**
Slide switch 3 for **PORTC**
Slide switch 4 for **PORTD**

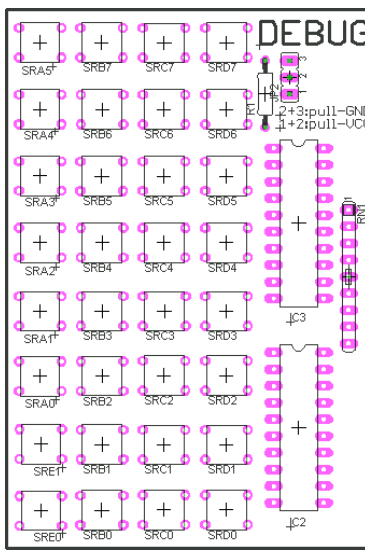


Figure 6: Signal Buttons

The *DevBugger* also comes equipped with 32 momentary pushbutton switches to simulate digital signal inputs. **JP2** determines the behaviour of the switches:

Short pins **1 and 2**: Corresponding I/O pin is pulled **low** when pushbutton is pressed
Short pins **2 and 3**: Corresponding I/O pin is pulled **high** when pushbutton is pressed

4.4 HD44780 Based LCD

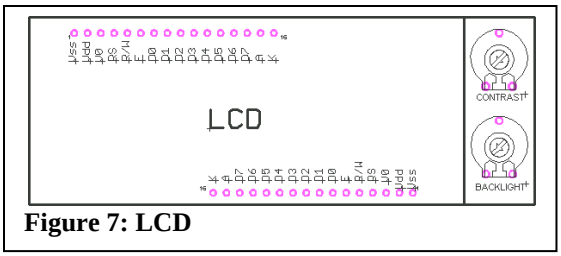


Figure 7: LCD

This peripheral module allows a HD44780-based LCD display to be easily connected to the board. Controls are provided for backlight and contrast, and two equivalent headers have been provided to support different LCD orientations.

Since the HD44780 protocol supports either 8-bit or 4-bit data transfer modes, the *DevBugger* has been configured to use 4-bit mode in the interest of conserving I/O pins. The HD44780 interface pins have been mapped to the PIC I/O ports as follows:

HD44780	PIC I/O Pin
RS	RD2
R/W	GND
E	RD3
D4	RD4
D5	RD5
D6	RD6
D7	RD7

4.5 4x4 KEYPAD

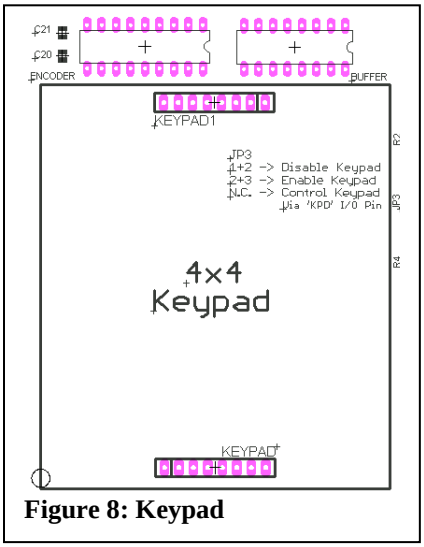


Figure 8: Keypad

Since 4x4 keypads are commonly used in microcontroller applications, this module was included to simplify the required interface. Two equivalent headers provide a socket for the keypad in different orientations, and a **MM74C922** hex encoder simplifies the polling process while reducing pin requirements. The data pins for the encoder are connected to **PORTB<7:4>** and the 'data available' pin (active high) is connected to **RB1**.

JP3 allows the user to enable or disable the keypad:
Short pins **1 and 2** to *disable* keypad
Short pins **2 and 3** to *enable* keypad

The Keypad module can also be enabled or disabled on-the-fly through the special **KPD** pin on the I/O bus. If this is desired, **JP3** should be left unconnected. If **KPD** is set high, the keypad will be *disabled*, and if set low, the keypad will be *enabled*. Note that **KPD** can be controlled either by external circuitry or directly by the PIC by connecting it to one of the ports on the I/O bus.

4.6 RS-232 DB-9 Communication

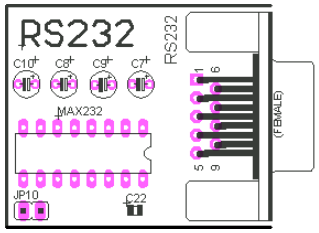


Figure 9: RS232 Module

This peripheral module allows the user to communicate with a PC through a serial port, using the PIC's USART module. A MAX232 chip is used as a level converter since the USART and the serial port use different voltage levels, and a built-in DB9 connector allows easy connection to other RS232-compatible devices. Note that this module is connected to pins RC6 and RC7 when enabled.
To enable this module, short **JP10**.

4.7 A2D Reference

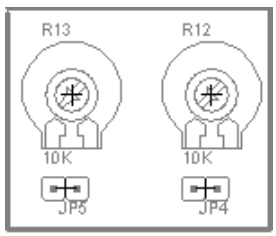


Figure 10: A2D

The *DevBugger* has been equipped with two potentiometers, **R12** and **R13** to set the voltage reference levels for the Analog to Digital Converter (**ADC**). To enable these references, short JP4 and JP5; if left unconnected, the references are disabled.

Short **JP4** to enable Vref on **RA2**
Short **JP5** to enable Vref on **RA3**

4.8 Real Time Clock

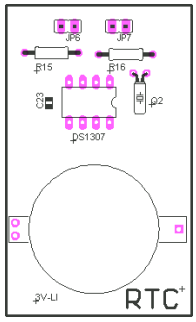


Figure 11: RTC

A Real Time Clock (**RTC**) and disk battery allow for off-chip timekeeping, even when the rest of the board is unpowered. The circuit is designed for a DS1307 RTC chip, interfaced through I^2C on pins **RC3** and **RC4**. To enable, short **JP6** and **JP7**.

Note 1: The DS1307, 3V lithium disk battery, and jumper shunts are ***not*** *included with the board.*
Note 2: The RTC module was designed to be used with backup power (the 3V disk battery) and therefore must have the battery in the socket to ensure consistent operation.
Note 3: It has been tested that the jumper shunts for I2C must be taken off when programming PIC18F2455.

4.8.1 Using the Real Time Clock

The DS1307 real time clock can be used to keep track of times in seconds, minutes, hours, days, months, and years. The numbers of days in months are automatically adjusted, including leap years. The hours function allows the chip to keep time in either 12 or 24 hour format with AM/PM indicator for 12 hour format. The advantage of the off-chip timekeeping functionality of the RTC is to free the microcontroller from the task

so that it may focus on other tasks. For more detail on the DS1307 real time clock, please refer to Chapter 7 of the AER201 course notes.

The RTC communicates with the PIC16F877/A through the I2C protocol and acts as a slave device with a 7-bit address of 1101 000X (where X denotes if the transaction is a read or a write). In order to use the RTC, PIC16F877/A must be configured as a MSSP device or master device for I2C. The master device is responsible for initiating and controlling the clock pulse for all slave devices, including the RTC. The configuration code for I2C is available to students as two files: i2c_common.asm and rtc_macros.inc. A brief description of these files:

i2c_common.asm: This source file contains the lowest level algorithm to deal directly with the I2C protocol. It also contains the algorithm to deal with communication between the microcontroller and RTC as well as for the pic16 to pic18 I2C communication. This file must be included in the user's MPLAB project.

rtc_macros: This file must be included in the MPLAB project as well as the user code wherever these macros are called. This file contains three primary macros that can access all the timekeeping functions of the RTC. These are:

Macro: rtc_resetAll

Input parameters: none
Output: none
Description: When invoked, this macro resets all the time keeping registers on the RTC memory or resets time to zero.

Macro: rtc_set *address , data*

Input parameters: Takes in 2 literal parameters, the address of the register being written to and the data which will be written
Output: none
Description: This macro will initiate a write event to the RTC. The user must specify the address which will be written to (i.e. the seconds register which holds the time in seconds) and the data which will be written to the specified address.

Macro: rtc_read *address*

Input parameters: address to be read from
Output: DOUT (0x75), dig10 (0x77), dig1 (0x78)
Description: This macro will initiate a read event to the RTC and read data from the specified address. The data from the RTC will be saved to data memory general purpose register 0x75 or DOUT as an 8-bit binary number. For the convenience of the user, this data will also be converted into a two digit ASCII number and the tens digit will be stored in 0x77 or dig10 and the ones digit will be stored in 0x78 or dig1. This is advantageous because HD447780 based LCDs only display ASCII numbers.

In order to use these files, simply add them to your existing MPLAB project (If you do not know how to make a project in MPLAB, use the MPLAB Project Wizard under *Project* to generate a project.) To do this, follow these steps:

1. Copy i2c_common.asm and rtc_macros.inc into your project directory.
2. Open MPLAB and load the project where the RTC is to be used.
3. Go to *View* and make sure Project is checked.
4. In the Project window where all the files in the project are listed. Right click Source File and select Add Files… Select and add i2c_common.asm in your project directory.
5. In the Project window where all the files in the project are listed. Right click Header Files and select Add Files… Select and add rtc_macros.inc in your project directory.
6. In any source files in which the user code calls the RTC macros, you must use the include directive at the top of the page to include rtc_macros.inc. Simply type at the top: *include <rtc_macros.inc>*.
7. Before using the RTC macros, you must enable and configure the PIC16 as a master I2C device. To do this, simply call i2c_common_setup subroutine (this subroutine is located in i2c_common.asm). Simply type: *call i2c_common_steup*. It is suggested that this subroutine to be called at the top of the main source file. You only need to call this subroutine once.
8. Invoke the RTC macros when needed in the user code to use the RTC.

Here is the memory map of the RTC registers:

ADDRESS	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	FUNCTION	RANGE
00H	CH	10 Seconds			Seconds				Seconds	00–59
01H	0	10 Minutes			Minutes				Minutes	00–59
02H	0	12	10 Hour	10 Hour	Hours				Hours	1–12 +AM/PM 00–23
		24	PM/ AM							
03H	0	0	0	0	0	DAY			Day	01–07
04H	0	0	10 Date		Date				Date	01–31
05H	0	0	0	10 Month	Month				Month	01–12
06H	10 Year				Year				Year	00–99
07H	OUT	0	0	SQWE	0	0	RS1	RS0	Control	—
08H-3FH									RAM 56 x 8	00H–FFH

0 = Always reads back as 0.

Figure 12: RTC memory map. Register 00h to 07h are used for timekeeping. 07h is used to generate square waves but this function is not included. 08h to 3fh are general purpose registers and can act as extra memory.

To reset register 00h to 07h to zero, simply invoke the rtc_resetAll macro. To set a register invoke rtc_set macro with the address of the register and the data to be set as parameters to the macro (i.e. setting seconds to zero: *rtc_set 00h , 00h)*. To read from these registers, invoke the rtc_read macro with the address of the register to be read from (i.e. reading seconds register: *rtc_read 00h)*. The result will be saved in bank0 0x75 of PIC16F877/A or as a two digit ASCII numbers in 0x77 (tens digit) and 0x78 (ones digit).

4.9 Main I/O BUS

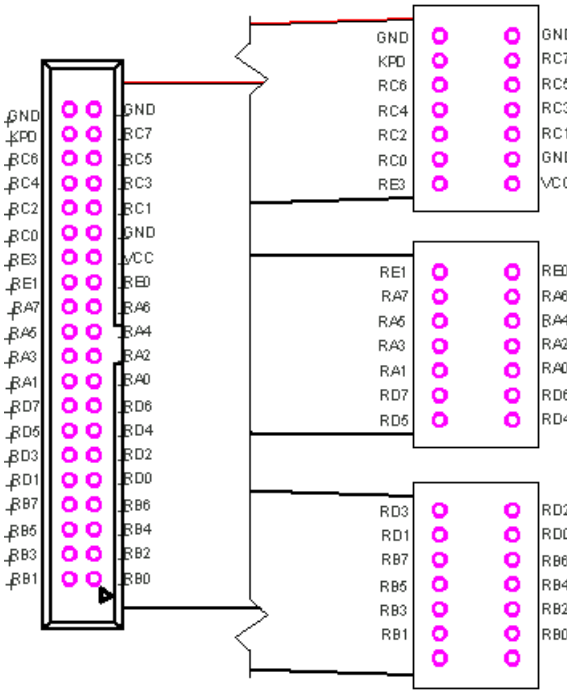


Figure 13: IO Bus Pin-Outs, Top View

A 40-pin bus has been provided to allow direct access to each I/O pin available on the PIC, as well as a special purpose pin for enabling/disabling the keypad at runtime (see Section 4.5). The pinouts for the socket and the provided protoboard adapter cable are shown to the left.

It is important to note that to access **RA6** and **RA7,** jumpers **JP8** and **JP9** must be properly set, as described in Section **Error! Reference source not found.**.

4.10 I2C BUS

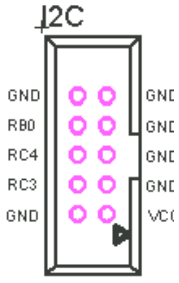


Figure 14: I2C

An I2C bus socket has also been provided to allow a separate I^2C bus to a peripheral device. A 10-bin ribbon cable connector (not supplied with board) should be used for this purpose. The pinouts of the socket are shown to the left.

4.11 Main PIC Device

This section of the board has several sockets for PIC devices of different sizes. Only one socket may be occupied at a time, otherwise bus conflicts will arise. The *DevBugger* is primarily intended to be used with a PIC16F877(A), although many other PIC devices in the **PIC16F** and **PIC18F** families are currently supported. The configuration file in the PC side programming software can also be modified to include support for many more devices.

Additionally, some PIC18F devices allow the user to employ an internal oscillator and configure **RA6** and **RA7** as general purpose I/O pins. If this is intended, jumpers **JP9** and **JP8** must be set as follows:

Short pins **1 and 2** to use external oscillator
Short pins **2 and 3** to enable **RA6, RA7**

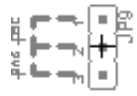


Figure 15

Of course, the appropriate configurations must also be set from within the code.

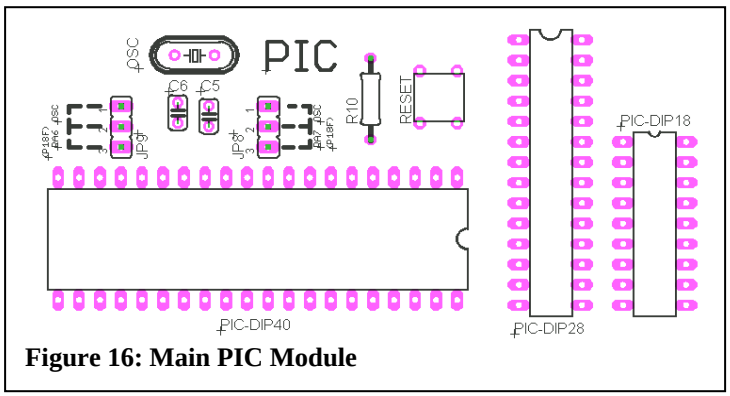


Figure 16: Main PIC Module

5 Programming Software

5.1 Overview

Because the *DevBugger* communicates with the PC through USB, popular programming applications such as **WinPic** and **PiKdev** are incompatible. Instead, an application called **PICusb** is provided, which is designed specifically to communicate with the *DevBugger* hardware.

PIC device support is stable for PIC16F family devices, but still unstable for PIC18F family devices. However, as long as "Verify After Programming" or "Read each word back after Write" is used and no errors are detected, it can always be safely assumed that the device was successfully programmed.

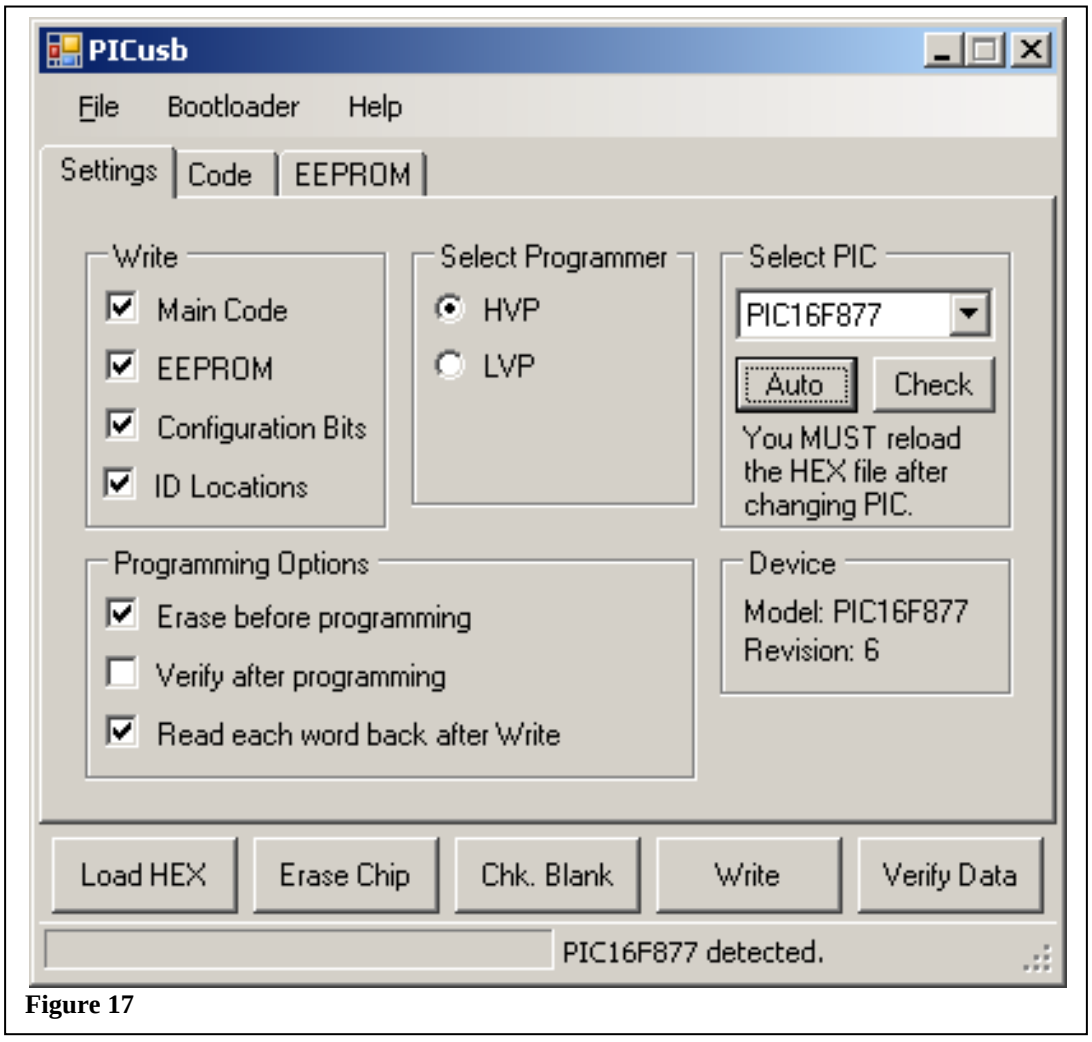


Figure 17

5.2 Quick Start

The user interface for **PICusb** is fairly intuitive, with reasonable defaults for all settings. For those without prior experience with PIC programming software, the following steps can generally be used to load the HEX code onto the device:

1. Connect the *DevBugger* to the PC as described in Section 3.2.
2. It's usually a good idea to press the 'Auto' button at this point to detect the type of PIC currently used in the DevBugger. If the detected device doesn't match one's expectation, the situation should be investigated before proceeding further. If no compatible device is detected, then the device may be damaged.
3. Load the desired HEX file into memory by clicking **Load Hex**.
4. Unless you are an advanced user and know exactly what you are doing, leave all options at their defaults except **Verify After** and **Read Back After Write**. Generally, only one of these two options needs to be checked; if the device supports readback, then the latter option will give faster programming time.
5. Press the **Write** button to load the HEX file to the PIC device. If no error message occurs, then the PIC should have been programmed successfully.

As a general usage note, if the PIC device selection is changed in the dropdown box, then the HEX file must be re-loaded as in Step 3.

5.3 Device Connections

When the *DevBugger* is connected to the PC, and is in **Programming** mode, it should automatically be detected by **PICusb**. Note that in **Executing** mode, the Programmer will not connect to the PC. If persistent errors occur, disconnect the board from the PC, unpower and re-power the *DevBugger*, and re-connect the board to the PC.

The type of PIC currently used in the board can be automatically detected by **PICusb** by pressing the **Auto** button under the device list. Alternately, if the device type is known, it may be selected manually from the dropdown box. In this case, it is usually a good idea (but not necessary) to press **Check** to ensure that the device ID matches the expected value, and to obtain the revision information of the device.

In some special cases, if a device appears to be malfunctioning and is not automatically detected, the problem may be resolved by setting the device type manually and performing a bulk erase. Alternately, if it is suspected that readback is functioning improperly, all checks may be bypassed by manually selecting the device type and unchecking both the Verify and Readback options.

5.4 Operations

The following is a description of all operations that may be performed upon the PIC device currently used in the *DevBugger*.

Load Hex loads a HEX file into memory, for loading onto a PIC device or for verification against the code already loaded onto a PIC device.

Erase Chip performs a bulk erase of the device, effectively returning it to its factory state. In some cases, this may fix a device that appears faulty.

Chk. Blank is used to verify that the chip is indeed 'blank'; this is useful after performing an **Erase Chip** operation to verify that the operation succeeded.

Write programs the currently loaded HEX code into the target device. Several options for this operation are available. By default, all aspects of the HEX file are programmed into the device; however, options under the *Write* group may be unchecked to prevent the programmer from writing specific locations. Also, **Erase before programming** instructs the programmer to perform a bulk erase prior to programming the device. This is necessary in most cases, to ensure that 'clear' locations in the device's memory are indeed cleared. The two other 'Programming Options' are for error checking, to ensure that the device has been programmed correctly. In most cases the Readback option should be used; however, if a warning or error is given, then use the Verify option instead.

Verify Data reads the contents of the PIC device and compares it against the loaded HEX file. If any differences are observed, an error is given and the operation fails. Note that this operation will still fail on differences if 'Write' options are unchecked. As such, it is recommended that all Write options remain enabled.

5.5 Sample Code

The **PICusb** package contains four sample projects, located in the **Samples** folder where the application is installed. In these code samples, it is assumed that the microcontroller used is a PIC16F877, with a 10MHz oscillator. The samples are described below:

DS1307 RTC — This project demonstrates the code necessary to interface with the DS1307 real-time clock IC (not included) on the DevBugger board, using the I^2C module of the main PIC device. Also, it demonstrates RS232 usage; the program first resets the RTC's seconds to zero, and then repeatedly reads the time and transmits over RS232 to a PC. The baud rate is 9600, with 8-bit data and no parity.

Keypad_LCD — This project demonstrates the basics of interfacing with the Keypad and LCD modules of the DevBugger; anything that is typed on the keypad is immediately displayed on the LCD.

PortTest This project is a simple test program, which can be used to quickly verify that MPLAB, PICusb and the Programmer module of the board each function correctly. If this program executes correctly, then each of the Debug LED's should flash sequentially when the board is placed in RUN mode. Note that RA4 on PIC16 devices will not turn on unless pulled up by either a debug switch or an external resistor, since it is an open-drain output.

RS232 This project demonstrates two-way RS232 communications; it sends a welcome message to the computer, and then repeatedly echoes any data received back to the computer. As with **DS1307 RTC**, the baud rate used is 9600, with 8-bit data and no parity.

6 Advanced Programming Topics

6.1 Overview

The Programmer module is fully functional out of the box, and can be used to program PIC devices of families PIC16F and PIC18F. However, since the module itself is built around a PIC18F2455, it is capable of much more than programming the main PIC device. To take advantage of this untapped potential, several extra features have been implemented to give the user full control over the device.

Bootloader A USB-enabled bootloader has been programmed into the device's firmware, allowing the user to re-load the Programmer with their own HEX code.

Mode Detect The programming firmware can detect whether the Programmer switch is set to PRG or to RUN, via pins RC7 and RB4. For more details, see Section 6.3.

I²C Bus An I²C bus connecting the PIC18F2455 to the main PIC device may be enabled by shorting **JP1** and **JP12**. This allows communication between the two devices, opening up the possibility of parallel processing.

PIC18 Bus A secondary I/O bus socket has been provided, giving access to the unused pins on the PIC18F2455. The pinouts for this socket are shown to the right. Since this is an advanced feature of the board, no cable is provided.

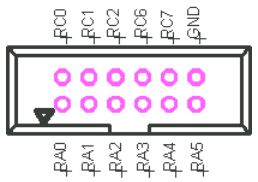


Figure 17

6.2 Required Tools

Compiling: If it is desired to program the firmware in C (for example, to modify the source code of the default programming firmware), the **MPLAB C18** compiler suite is required in addition to the **MPLAB IDE**. If only assembly is being used, then any PIC assembly compiler will suffice to generate the HEX file. The source code for the programmer's default firmware is provided with the rest of the board's software.

Loading: In order to load the newly compiled HEX firmware to the programmer, the **Bootloader** mode should be used. A Microchip application, "PDFSUSB", is installed together with PICusb, and must be used to load code. Alternately, if another *DevBugger* or other PIC18F2455-compatible programmer is available, it may also be used to reprogram the entire device.

6.3 Coding Firmware Modifications

There are three options for modifying the Programmer module's firmware:

1. Append user code to the programmer, to modify the run-mode behaviour
2. Update the programmer firmware itself via the bootloader
3. Use another HVP programmer to re-image the entire PIC18F2455 chip

6.3.1 Modifying Run-Mode

Using this option, the user can append their own code to the original Programmer firmware, which will only be run when the board is in **RUN** mode.

The first step is to write the custom code, to be appended. When writing this code, it is absolutely important to **ensure that all code addresses are offset by 0x1600**, and also **do not include config information**. For example, the reset vector would be located at 0x1600, and the interrupts would be located at 0x1608 and 0x1618. For more information on how to do this, see Section 6.3.5. Another important consideration is Programming Mode re-entry; this functionality is optional, but without it, the board can only be returned to Programming mode by unpowering/re-powering it while the Programmer's slide switch is set to PGM. In order to implement this, the user code should periodically check pin RC7; if RC7 is HIGH, then the user code should either reset the PIC (if currently within a function call) or optionally jump to **0x800** if in the main() function. Besides this, the code considerations mentioned in Section 6.3.4 should be followed.

After compiling the custom code, open **PICusb**. Under the **Bootloader** menu, choose **Append HEX to Firmware**. Open the compiled user HEX file, and when prompted, save the new HEX file to a different location. This new HEX file which you have saved can now be loaded to the Programmer by the method described in Section 6.4.

6.3.2 Updating the Programmer Firmware

In case bugs are discovered in the default Programmer firmware, new firmware may be provided in the future. In such an event, the new HEX file can be downloaded from the provider's website and then loaded using the method in Section 6.4.

6.3.3 Re-Imaging the PIC18F2455

This procedure is not recommended, as it is very easy to corrupt the DevBugger's bootloader. However, in case the bootloader itself must be updated, it must be loaded using a second HVP-capable programmer board. This procedure will most likely overwrite any Programmer firmware or User Code that was loaded previously, so a backup should be made first using the existing bootloader, as described in Section 6.4. However, it may be possible to update only the bootloader, while leaving the existing programmer and user code intact. If a second DevBugger board is being used to perform this task, then uncheck "Erase before programming"; if the new bootloader HEX code does not include new Programmer firmware, then the existing code may remain untouched.

6.3.4 User Code Considerations

Since the bootloader and Programmer firmware also reside in program memory, user code must begin at 0x1600, including reset vectors and interrupt vectors. More information on this can be found in Section 6.3.5.

Since the bootloader requires a specific set of config words (already set), no config information should be included in user-compiled code. It may be assumed that the bootloader uses the following configuration settings:

```
#pragma config    FOSC        = HSPLL_HS  //HS osc using PLL
#pragma config    PLLDIV      = 5         //20mhz osc
#pragma config    USBDIV      = 2         //USB clk from PLL
#pragma config    CPUDIV      = OSC1_PLL2 //CPU=PLL/2

#pragma config    IESO        = OFF //Ext osc only
#pragma config    FCMEN       = OFF //No fallback to int-osc
#pragma config    PWRT        = ON  //Enable power-up timer
#pragma config    BOR         = ON  //BOR in HW only
#pragma config    BORV        = 3   //BOR on 2.05V (min setting)
#pragma config    VREGEN      = ON  //USB internal vreg
#pragma config    WDT         = OFF

#pragma config    MCLRE       = ON  //enable MCLR, dis. RE3
#pragma config    LPT1OSC     = ON  //TMR1 low-power mode
#pragma config    PBADEN      = OFF //PortB A/D off on reset
#pragma config    CCP2MX      = ON  //CCP2 on RC1
#pragma config    STVREN      = ON  //Reset on stack overflow
#pragma config    LVP         = OFF
#pragma config    XINST       = OFF //Disable extended instruction set
#pragma config    DEBUG       = OFF //ICD off

#pragma config    CPB         = OFF //No code protect
#pragma config    CP0         = OFF
#pragma config    CP1         = OFF
#pragma config    CP2         = OFF
#pragma config    CPD         = OFF //EEPROM not protected

#pragma config    WRTB        = ON  //Writeprotect 000 to 7FF
#pragma config    WRT0        = OFF //No other write protect
#pragma config    WRT1        = OFF
#pragma config    WRT2        = OFF
#pragma config    WRTC        = OFF //Config not protected
#pragma config    WRTD        = OFF //EEPROM not protected

#pragma config    EBTRB       = OFF //Table reads not protected
#pragma config    EBTR0       = OFF
#pragma config    EBTR1       = OFF
#pragma config    EBTR2       = OFF
```

User code must not modify PORTB<7:2>, as these are reserved for the Programmer's operation; also, PORTC<7> is used for PRG/RUN mode detection, as described in Section 6.3.1. As such, it must ALWAYS be set as an input. Similarly, pin RA0 is used for bootloader detection, so it has a 10K pull-up resistor to VCC. During ordinary operation, this does not have an effect on all applications, so RA0 is still provided on the I/O port for the programmer.

Pins RB1 and RB0 are used for an optional Programmer-to-PIC I^2C bus, which can be enabled (connected) by shorting jumpers JP1 and JP12 in the Programmer module. 10K pull-up resistors on these lines are already provided on the board. The specifics of I^2C communication between the Programmer and the main PIC are left to the user.

6.3.5 Mapping Vectors

When using Assembly, this is just a matter of changing the value following the ORG directives. For example, instead of ORG 0x00 for the reset vector, in user code to be appended to programmer firmware, this should be ORG 0x1600.

When using C or Relocatable assembly code, the Linker script must also be modified. A sample script is shown below:

```
// Sample linker command file
LIBPATH .

FILES c018i.o
FILES clib.lib
FILES p18f2455.lib

CODEPAGE   NAME=boot       START=0x0           END=0x15FF          PROTECTED
CODEPAGE   NAME=vectors    START=0x1600        END=0x1629          PROTECTED
CODEPAGE   NAME=page       START=0x162A        END=0x7FFF
CODEPAGE   NAME=idlocs     START=0x200000      END=0x200007        PROTECTED
CODEPAGE   NAME=config     START=0x300000      END=0x30000D        PROTECTED
CODEPAGE   NAME=devid      START=0x3FFFFE      END=0x3FFFFF        PROTECTED
CODEPAGE   NAME=eedata     START=0xF00000      END=0xF000FF        PROTECTED

ACCESSBANK NAME=accessram  START=0x0           END=0x5F
DATABANK   NAME=gpr0       START=0x60          END=0xFF
DATABANK   NAME=gpr1       START=0x100         END=0x1FF
DATABANK   NAME=gpr2       START=0x200         END=0x2FF
DATABANK   NAME=gpr3       START=0x300         END=0x3FF
DATABANK   NAME=usb4       START=0x400         END=0x4FF           PROTECTED
DATABANK   NAME=usb5       START=0x500         END=0x5FF           PROTECTED
DATABANK   NAME=usb6       START=0x600         END=0x6FF           PROTECTED
DATABANK   NAME=usb7       START=0x700         END=0x7FF           PROTECTED
ACCESSBANK NAME=accesssfr  START=0xF60         END=0xFFF           PROTECTED

SECTION    NAME=CONFIG     ROM=config

STACK SIZE=0x100 RAM=gpr3
```

The vectors must then also be changed in code, from the "#pragma code" directives, in a similar fashion to that of assembly's "ORG" directive.

6.4 Using the Bootloader

For information on entering the **Bootloader** mode, refer to Section 3.1. Connect the *DevBugger* to the PC via the USB link, and open the **PDFSUSB** program that came bundled with the board software; you should see the following application:

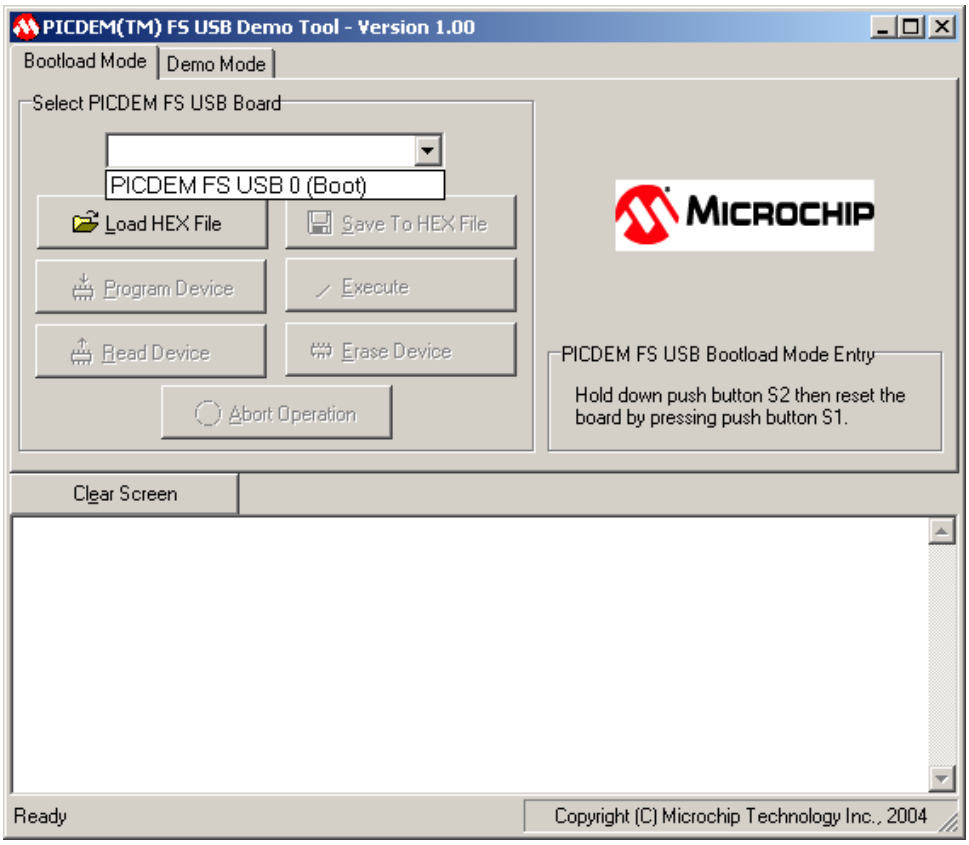


Figure 18

Note that the first time Bootloader mode is entered, when the DevBugger is connected to the PC it will be recognized as new hardware. Follow the same procedure as in Section 3.2 for device detection in order to install the appropriate drivers.

The *DevBugger* should automatically be detected when it is connected to the PC, and will appear in the dropdown list as **PICDEM FS USB 0 (Boot)**. This is because the bootloader is adapted from one of Microchip's sample applications, and **PDFSUSB** is the PC-side companion application. Ignore the *Demo Mode* tab, since it is meant for a different board. If the board is not detected, then the bootloader has likely been corrupted previously, and must be re-loaded as described in Section 6.5.

Select **PICDEM FS USB 0 (Boot)** from the dropdown list. It is recommended first to read the device and save it to a backup HEX file, in case you wish to restore the original firmware. To program the new HEX firmware, click *Load HEX File* to load it into memory, and then select *Program Device*. The status messages in the textbox at the bottom should inform you when the operation is complete.

When you are finished with loading the new HEX firmware, simply reset the board via the power supply's switch to exit the **Bootloader** mode. Alternately, selecting **Execute** from **PDFSUSB** may also successfully reset the board, although this method has not been tested extensively.

Note: **PDFSUSB** should be closed before attempting to re-compile any source code, since it may hold the old HEX file open and prevent the compiler from over-writing it.

6.5 Restoring to Original State

The **PICusb** software package includes several HEX files containing the original firmware of the DevBugger board. These are located in the **Programmer Firmware** folder of the application directory:

- **btldr_pgmr-1.6.hex** – HEX file containing both the original bootloader and programmer firmware; only to be used internally for **PICusb**'s "Append to Firmware" feature.
- **fw-complete-1.6.hex** – Contains the original programmer firmware, as well as the default run-mode firmware that facilitates switching back to the Programmer firmware when RUN mode is re-entered.
- **image-1.6.hex** – A full image of the original firmware, useful for restoring the DevBugger to its factory state.

In the event that the bootloader becomes corrupted, or the original firmware needs to be restored, or even if the PIC18F2455 chip needs to be replaced, **image-1.6.hex** may be used to restore the Programmer module to its factory state. In order to accomplish this, a second *DevBugger* or other HVP-capable and PIC18F2455-compatible board is required. Simply remove the PIC18F2455 chip from the Programmer module, insert it into the second programmer's socket, and use its software to load the original HEX image onto the chip. The chip can then be replaced in the *DevBugger*'s Programmer module, and it should be fully restored.

Alternatively, if only the Programmer firmware is corrupted (but the bootloader is still functional), the bootloader can be used to restore the *DevBugger* to its original state by loading **fw-complete-1.6.hex**.

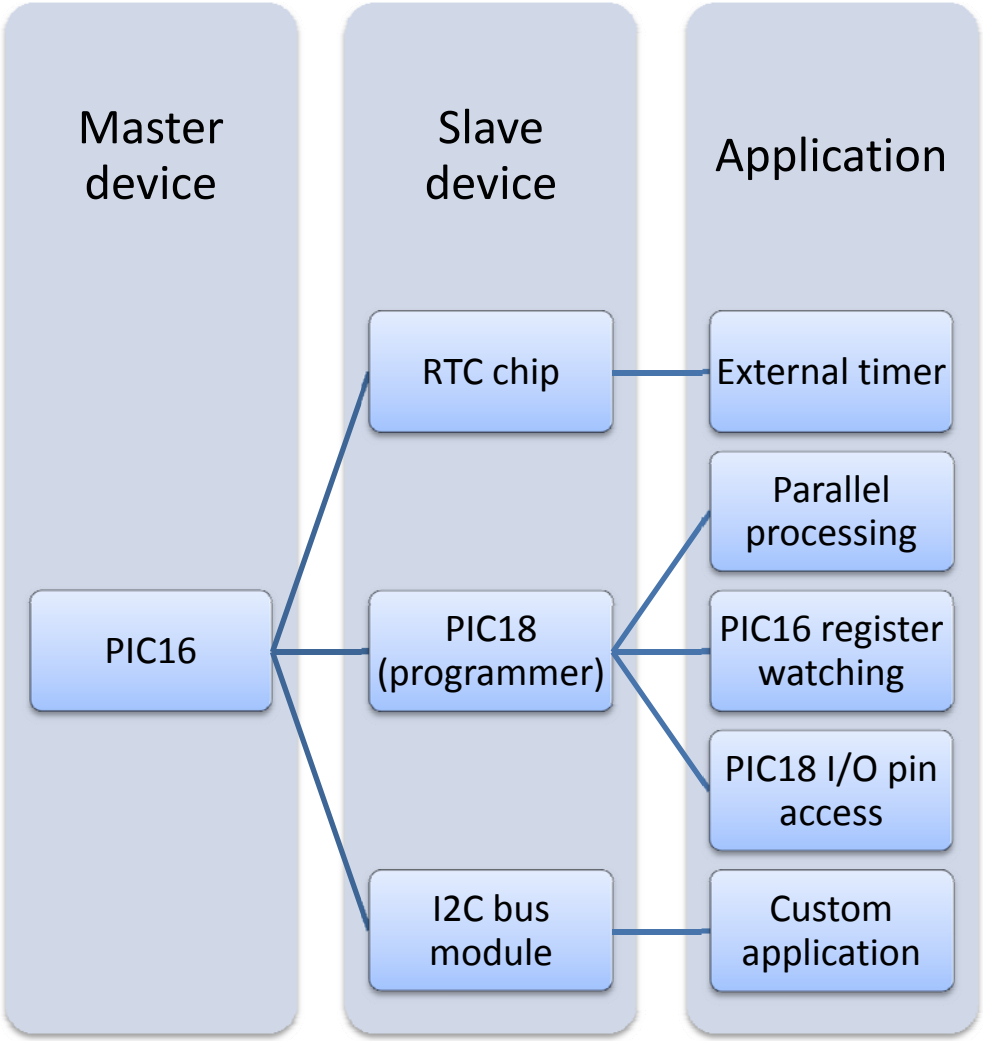
7 I2C PIC to PIC Communication

7.1 I2C Overview

The I2C is a widely used serial bus specification that allows communication between one or more master devices with one or more slave devices. The master is the device that selects the receiver by transmitting the address of the slave device and initiates all transactions (read or write) between itself and the slave. The I2C implementation on the DevBugger board includes one master – the PIC16 – and three slave devices – the RTC chip, the PIC18, and the I2C female header pins at the top right corner of the board which can connect to an external I2C slave device.

7.2 Using the I2C

The hierarchical chart below shows the possible applications of the I2C bus, as it is implemented on the DevBugger board.



As shown above, there are five ways in which the I2C bus on the DevBugger can be used. First, one can use access the RTC module for timing purposes through the I2C bus. Second, one can set up PIC16 to PIC18 communication for parallel processing. This is for advanced users only, as it requires that both the master and the slave be programmed. Third, one can take advantage of the USB capability of the PIC18 as well by using the PIC16 register watching software to monitor the PIC16's I/O ports and file registers during runtime. Fourth, one can enable PIC16 to PIC18 I2C communication to enable the usage of PIC18 I/O pins by the PIC16 during runtime, since the PIC18 is idle when the DevBugger is in RUN mode. This effectively adds 10 to the PIC16's I/O pin count. Finally, an I2C-capable slave device can be connected to the I2C header pins at the top right corner of the board, to communicate with the PIC16 master device.

7.2.1 PIC16-RTC Communication

For information on how to use the RTC from the PIC16, please refer to section 4.8.

7.2.2 PIC16-PIC18 Communication for Parallel Processing

To implement parallel processing, the user must write the necessary code not just for the PIC16 master device, but also for the PIC18 as the slave device. From the PIC16's standpoint, programming is greatly simplified by the i2c_common.asm file developed for RTC communication, the PIC16 register watching protocol, and the PIC18 I/O pin usage protocol. This file is a collection of macros that execute instructions on the hardware level for the I2C – for instance, start transaction, write, check acknowledge, read, send acknowledge, and stop transaction. From the PIC18's standpoint, the programming is a bit more involved as the user must develop the code from scratch, but the actual logic involved is simple, at least in principle. The slave code polls or waits for an interrupt signaling the start of an event (always initiated by the master device), figures out what type of event it is based on the value of an I2C status register, handles it, and then loops back to the beginning. Microchip's AN735 and AN734 documents are excellent resources for programming and understanding the I2C protocol, as is Chapter 7 of the AER201 course text, where more detailed information can be found on the I2C.

7.2.3 PIC16-PIC18 Communication for PIC16 Register Watching

This application of the I2C can be thought of as an example of how one can use PIC to PIC communication. It is a finished product which can be used without any actual programming on the user's part. The design is simple:

1) PIC16: The user calls one of a number of macros to update the contents of a port or register during the running of their PIC16 code.
2) PIC18: The data is sent to the PIC18 through I2C, which is sent to the USB.
3) PC: The application (available through the website) displays this information.

There are four easy steps to set up this functionality.

1. Add *i2c_common.asm* and *p2p_macros.inc* to the PIC16 project
2. Add *#include <p2p_mcros.inc>* in the code
3. Add *call i2c_common_setup* in the code
4. Call the macros as needed
5. Make sure the programmer PIC18 has the firmware with register watching

Below is a table listing the macros contained in *p2p_macros.inc* which must be called by the user's code.

Macro	Description
watch_PORTA	Sends the value of PORTA to the PORTA field in the application
watch_PORTB	Sends the value of PORTB to the PORTB field in the application
watch_PORTC	Sends the value of PORTC to the PORTC field in the application
watch_PORTD	Sends the value of PORTD to the PORTD field in the application
watch_PORTE	Sends the value of PORTE to the PORTE field in the application
watch_TRISA	Sends the value of TRISA to the TRISA field in the application
watch_TRISB	Sends the value of TRISB to the TRISB field in the application
watch_TRISC	Sends the value of TRISC to the TRISC field in the application
watch_TRISD	Sends the value of TRISD to the TRISD field in the application
watch_TRISE	Sends the value of TRISE to the TRISE field in the application
watch_register_0 watch_register_1 watch_register_2 watch_register_3 watch_register_4 watch_register_5 watch_register_6 watch_register_7	These macros have an input parameter representing the address of the file register the user wants to monitor. This address is displayed along with the value of the register on the PC application. One can monitor up to 8 registers (registers 0 to 7). Example: watch_register_0 0x05 - The PC application displays 00000101 and the value of PORTA beside it (0x05 is the address of PORTA).

There are two things to keep in mind. First, the USB sends a packet every 10 ms, so this is the time-resolution for data updates on the PC application. Secondly, calling one of these macros is a one-time update of the value of the register. For registers with regularly changing values, it is necessary to call the macro repeatedly either in a loop or at crucial junctures in the code. Even so, most of the processing in one's code takes place in a matter of microseconds since one operation on the PIC16 takes 400 µs – thus, one cannot actually see and monitor the contents of file registers that change this quickly.

One solution is to use breakpoints. Using the I2C, it is easy to implement a breakpoint subroutine that stops the PIC16's operation and waits for the user to press the programmer module's bootloader switch before continuing. Presented below is the implementation:

```
breakpoint
        <call whatever watching macros you want here, for e.g. watch_PORTA>
        bsf_TRISA_0  ; sets the pin the bootloader switch is connected to, to input
        btfsc_PORTA_0   ; waits for the user to press the switch and pull RA0 low
        goto $-1    ; if RA0 is not low, keep looping
        <put a bit of a delay here>    ;so that the PIC is not at the next breakpoint
                                        by the time you release the switch
        return  ; continue with the program
```

The user should call this subroutine at key junctures in the code to stop execution, transmit register values, read them from the PC, and then push the bootloader switch when ready to continue operation.

7.2.4 PIC16-PIC18 Communication for PIC18 I/O Pin Usage

The I2C can be used to enable use of the PIC18's I/O pins from the PIC16 as if they were its own pins. These pins can be controlled using macros available in an include file, which are designed to look and behave similar to regular I/O control operations in the PIC16 instruction set. This effectively adds 10 additional I/O pins for the PIC16, but the only downside is the roughly 250 µs operation time for write operations and 500 µs for read operations, compared 400 ns for regular operations. Follow the four basic steps below to use the extended I/O pin set:

1. Add *i2c_common.asm* and *p2p_macros.inc* to the project
2. Add *#include <p2p_macros.inc>* in the code
3. Add *call i2c_common_setup* in the code
4. Call the macros as needed

* The PIC18-side firmware should already be set-up to handle the I/O instructions.

Below is the list of macros available in the macros include file. The function and usage of these macros do not require explanation as they behave almost identically to their counterparts in the PIC16 instruction set, except there are no input parameters.

Write to PIC18 pins/ports	Write to PIC18 TRIS	Read from PIC18 pins/ports
• bcf_PORTA_0	• bcf_TRISA_0	• btfsc_PORTA_0
• bcf_PORTA_1	• bcf_TRISA_1	• btfsc_PORTA_1
• bcf_PORTA_2	• bcf_TRISA_2	• btfsc_PORTA_2
• bcf_PORTA_3	• bcf_TRISA_3	• btfsc_PORTA_3
• bcf_PORTA_4	• bcf_TRISA_4	• btfsc_PORTA_4
• bcf_PORTA_5	• bcf_TRISA_5	• btfsc_PORTA_5
• bcf_PORTC_0	• bcf_TRISC_0	• btfsc_PORTC_0
• bcf_PORTC_1	• bcf_TRISC_1	• btfsc_PORTC_1
• bcf_PORTC_2	• bcf_TRISC_2	• btfsc_PORTC_2
• bcf_PORTC_6	• bcf_TRISC_6	• btfsc_PORTC_6
• bsf_PORTA_0	• bsf_TRISA_0	• btfss_PORTA_0
• bsf_PORTA_1	• bsf_TRISA_1	• btfss_PORTA_1
• bsf_PORTA_2	• bsf_TRISA_2	• btfss_PORTA_2
• bsf_PORTA_3	• bsf_TRISA_3	• btfss_PORTA_3
• bsf_PORTA_4	• bsf_TRISA_4	• btfss_PORTA_4
• bsf_PORTA_5	• bsf_TRISA_5	• btfss_PORTA_5
• bsf_PORTC_0	• bsf_TRISC_0	• btfss_PORTC_0
• bsf_PORTC_1	• bsf_TRISC_1	• btfss_PORTC_1
• bsf_PORTC_2	• bsf_TRISC_2	• btfss_PORTC_2
• bsf_PORTC_6	• bsf_TRISC_6	• btfss_PORTC_6
• clrf_PORTA	• clrf_TRISA	• movf_PORTA_W
• clrf_PORTC	• clrf_TRISC	• movf_PORTC_W
• setf_PORTA	• setf_TRISA	
• setf_PORTC	• setf_TRISC	

7.2.5 PIC16-I2C Bus Communication

Implementing communication between the PIC16 as the master and an external slave device is for advanced users only. The user would have to program both the master and slave codes, in addition to ensuring that the hardware for the bus connection is set-up properly.

7.3 Important Note About i2c_common.asm and Its Macros

*In the following paragraphs, PIC-to-PIC refers to the PIC16 to PIC18 communication protocols for PIC16 register watching and PIC18 I/O pin usage from the PIC16.

The RTC and PIC-to-PIC macros call subroutines in i2c_common.asm and will need to access SFRs on the PIC16. Therefore it will require switching memory banks. Therefore, invoking these functions will not guarantee the user will remain in the same bank before the macros are invoked. It is strongly advised to re-select the memory bank after calling the macros or use the *banksel* directive in MPLAB.

The I2C communication code does take up some of the user's resources. Since the macros call subroutines, one stack level will be taken. The I2C code will be stored in program memory and will take about 200 to 300 memory slots of the available 8192 slots on the PIC16 without any macro calls but this includes both the real time clock I2C algorithm as well as the PIC-to-PIC algorithm. Keep in mind that macros are simply directives in MPLAB and macro code can be thought of as "copy and pasted" into the user's code whenever invoked. This will result in the user code being much longer than anticipated especially if these macros are invoked linearly (or not in loops) over and over again.

The RTC code will take up 8 general registers in the PIC16 data memory (0x71 to 0x78 in bank0) to store data used during operation. These registers are defined as a cblock in i2c_common.asm and the corresponding names are only for convenience sake; the actual code uses the actual register address. The main ones that are relevant to the user are 0x75 (DOUT), 0x77 (dig10), and 0x78 (dig1) where data from the RTC are stored. It is important not to overwrite the data in these registers or else one cannot expect proper operation.

Likewise, the PIC-to-PIC communication code uses a register, 0x70, which should not be accessed or modified by the user.

Sample code is provided to the students which use *i2c_common.asm* and *rtc_macros.inc* as well as *p2p_macros.inc*.