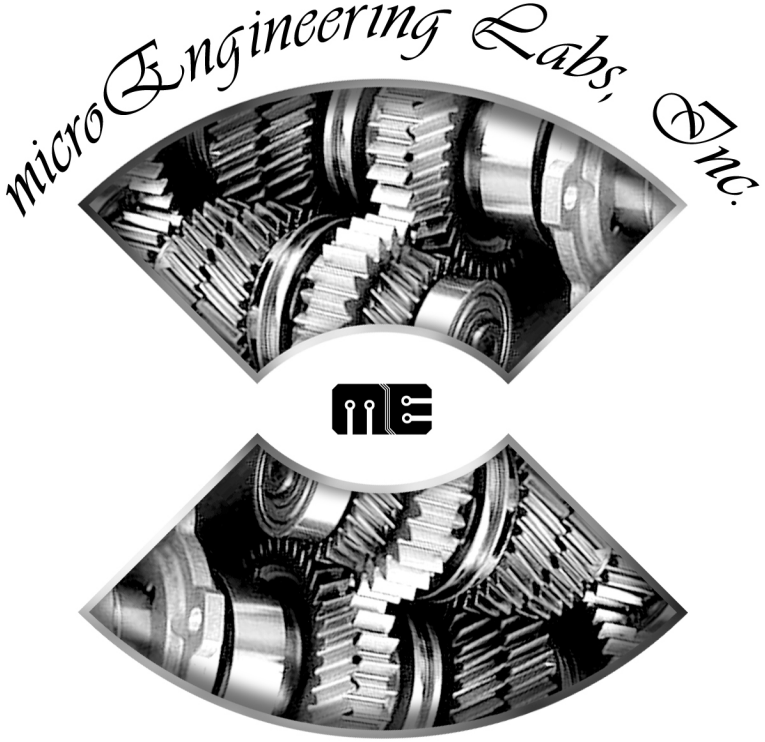


PICBASIC PRO™ Compiler

COPYRIGHT NOTICE

This manual describes the use and operation of the PICBASIC PRO™ Compiler from microEngineering Labs, Inc. Use of the PICBASIC PRO Compiler without first obtaining a license is a violation of law. To obtain a license, along with the latest version of the product and documentation, contact microEngineering Labs, Inc.

microEngineering Labs, Inc.
Box 60039
Colorado Springs CO 80960-0039
(719) 520-5323
(719) 520-1867 fax
email: support@melabs.com
web: www.melabs.com

TRADEMARKS

PICBASIC PRO™ Compiler

microEngineering Labs, Inc.

6/09

TABLE OF CONTENTS

1. Introduction

The PICBASIC PRO™ Compiler (PBP) makes it even quicker and easier for you to program Microchip Technology's powerful PIC® microcontrollers (MCUs). The English-like BASIC language is much easier to read and write than assembly language.

The PICBASIC PRO Compiler is "BASIC Stamp II like" and has most of the libraries and functions of both the BASIC Stamp I and II. Being a true compiler, programs execute much faster and may be longer than their Stamp equivalents.

PBP is not quite as compatible with the BASIC Stamps as our original PICBASIC™ Compiler is with the BS1. Decisions were made that we hope improve the language overall. These differences are spelled out later in this manual.

PBP defaults to create files that run on a PIC16F84 clocked at 4MHz. Only a minimum of other parts are necessary: 2 22pf capacitors for the 4MHz crystal, a 4.7K pull-up resistor tied to the /MCLR pin and a suitable 5- volt power supply. PIC MCUs other than the PIC16F84, as well as oscillators of frequencies other than 4MHz, may be used with the PICBASIC PRO Compiler.

1.1. The PIC® MCUs

The PICBASIC PRO Compiler produces code that may be programmed into a wide variety of PIC microcontrollers having from 8 to 100 pins and various on-chip features including A/D converters, hardware timers and serial ports.

The current version of the PICBASIC PRO Compiler supports most of the Microchip Technology PIC MCUs, including the 12-bit core, 14-bit core and both 16-bit core series, the PIC17 and PIC18 devices, as well as the Micromint PicStics. Support is limited for the 12-bit core PIC MCUs as they have a limited set of resources including a smaller stack and smaller code page size. See the `README.TXT` file for the very latest PIC MCU support list.

For general purpose PIC MCU development using the PICBASIC PRO Compiler, the PIC12F683, PIC16F690, 16F88, 16F876A, 16F877A, PIC18F2620 and 18F4620 are the current PIC MCUs of choice. These microcontrollers use flash technology to allow rapid erasing and

reprogramming to speed program debugging. With the click of the mouse in the programming software, the flash PIC MCU can be instantly erased and then reprogrammed again and again. Other PIC MCUs in the PIC12C5xx, 12C67x, PIC14000, PIC16C4xx, 16C5x, 16C55x, 16C6xx, 16C7xx, 16C9xx, PIC17 and PIC18 series are either one-time programmable (OTP) or have a quartz window in the top (JW) to allow erasure by exposure to ultraviolet light for several minutes.

Most PIC12F6xx, PIC16F6xx, 16F8xx and PIC18F devices also contain between 64 and 1024 bytes of non-volatile data memory that can be used to store program data and other parameters even when the power is turned off. This data area can be accessed simply by using the PICBASIC PRO Compiler's **READ** and **WRITE** commands. (Program code is always permanently stored in the PIC MCU's code space whether the power is on or off.)

By using a flash PIC MCU for initial program testing, the debugging process may be sped along. Once the main routines of a program are operating satisfactorily, a PIC MCU with more capabilities or expanded features of the compiler may be utilized.

While many PIC MCU features will be discussed in this manual, for full PIC MCU information it is necessary to obtain the appropriate PIC MCU data sheets or CD-ROM from Microchip Technology. Refer to Appendix E for contact information.

1.2. About This Manual

This manual cannot be a full treatise on the BASIC language. It describes the PICBASIC PRO Compiler instruction set and provides examples on how to use it. If you are not familiar with BASIC programming, you should acquire a book on the topic. Or just jump right in. BASIC is designed as an easy-to-use language. Try a few simple commands to see how they work. Or start with the examples and then build on them.

The next section of this manual covers installing the PICBASIC PRO Compiler and writing your first program. Following is a section that describes different options for compiling programs.

Programming basics are covered next, followed by a reference section listing each PICBASIC PRO command in detail. The reference section

shows each command prototype, a description of the command and some examples. Curly brackets, {}, indicate optional parameters.

The remainder of the manual provides information for advanced programmers - the inner workings of the compiler.

1.3. Sample Programs

Example programs to help get you started can be found in the SAMPLES subdirectory. Additional example programs can be found in the sample programs section of the microEngineering Labs, Inc. web site.

2. Getting Started

2.1. Software Installation

The PICBASIC PRO Compiler files are compressed into a setup file on the included disk. They must be installed before use.

To install the software, execute SETUP.EXE on the disk and follow the setup instructions presented.

All of the necessary files will be installed to a subdirectory named C:\PBP on the hard drive. The README.TXT file in that subdirectory has the latest information about the PICBASIC PRO Compiler.

2.2. Your First Program

For operation of the PICBASIC PRO Compiler you will use the included IDE or a text editor or word processor for creation of your program source file, some sort of PIC MCU programmer such as our EPIC Programmer™ or melabs U2 Programmer, and the PICBASIC PRO Compiler itself. Of course you also need a PC to run it all on.

The sequence of events goes something like this:

First, start the included or one of the other available IDEs/editors. Select the PIC MCU you intend to use from the IDE's drop-down list. Next, create the BASIC source file for the program or open one of the BASIC source files included with PBP. The source file name usually (but isn't required to) ends with the extension .BAS or .PBP.

The text file that is created must be pure ASCII text. It must not contain any special codes that might be inserted by word processors for their own purposes. You are usually given the option of saving the file as pure DOS or ASCII text by most word processors.

The following program provides a good first test of a PIC MCU in the real world. You may type it in or you can simply copy it from the SAMPLES subdirectory included with the PICBASIC PRO Compiler. The file is named BLINK.BAS. The BASIC source file should be created in or moved to the same directory where the PBP.EXE file is located.

```
' Example program to blink an LED connected to PORTB.0
about once a second
```

```
mainloop: High PORTB.0   ' Turn on LED
      Pause 500          ' Delay for .5 seconds

      Low PORTB.0        ' Turn off LED
      Pause 500          ' Delay for .5 seconds

      Goto mainloop      ' Go back to mainloop and
                        blink LED forever
      End
```

Once you are satisfied that the program you have written will work flawlessly, you can execute the PICBASIC PRO Compiler by clicking on the IDE's build or compile button. If you are using DOS, enter PBPW followed by the name of your text file at a DOS prompt. For example, if the text file you created is named BLINK.BAS, at the DOS command prompt enter:

```
PBPW blink
```

If you don't tell it otherwise, the PICBASIC PRO Compiler defaults to creating code for the PIC16F84. To compile code for PIC MCUs other than the PIC16F84, simply use the **-P** command line option described later in the manual to specify a different target processor. For example, if you intend to run the above program, BLINK.BAS, on a PIC16F877A, compile it using the command:

```
PBPW -p16f877a blink
```

The compiler will display an initialization (copyright) message and process your file. If it likes your file, it will create an assembler source code file (in this case named BLINK.ASM) and automatically invoke an assembler (PM or MPASMWIN) to complete the task. If all goes well, the final PIC MCU code file will be created (in this case, BLINK.HEX). If you have made the compiler unhappy, it will issue a string of errors that will need to be corrected in your BASIC source file before you try compilation again.

To help ensure that your original file is flawless, it is best to start by writing and testing a short piece of your program, rather than writing an entire 100,000 line monolith all at once and then trying to debug it from end to end.

2.3. Program That MCU

There are two steps left - putting your compiled program into the PIC microcontroller and testing it.

The PICBASIC PRO Compiler generates standard 8-bit Merged Intel HEX (.HEX) files that may be used with any PIC MCU programmer including our EPIC Programmer™ and melabs U2 Programmer. PIC MCUs cannot be programmed with BASIC Stamp programming cables.

The following is an example of how a PIC MCU may be programmed using one of our programmers.

Make sure there are no PIC MCUs installed in the programmer programming socket or any attached adapters.

If you are using the melabs U2 Programmer, plug it into the PC USB port using a USB cable.

If you are using the EPIC Programmer, hook it to the PC parallel printer port using a DB25 male to DB25 female printer extension cable.

Plug the AC adapter into the wall and then into the programmer (or attach 2 fresh 9-volt batteries to the EPIC Programmer and connect the “Batt ON” jumper. Using an AC adapter instead of batteries is highly recommended.) A separate power supply is not required for the melabs U2 Programmer.

The LED(s) on the EPIC Programmer may be on or off at this point. Do not insert a PIC MCU into the programming socket when an LED is on or before the programming software has been started. The LED should glow green on the melabs U2 Programmer indicating it is ready.

Launch the programmer software. Once the programming screen is displayed, select the PIC MCU you will be programming. Next, use the mouse to click on Open file. Select BLINK.HEX or another file you would like to program into the PIC MCU from the dialog box.

Once the file has been loaded, you can look at the Code or Memory window to see your PIC MCU program code. You should also look at the Configuration window and verify that it is as desired before proceeding.

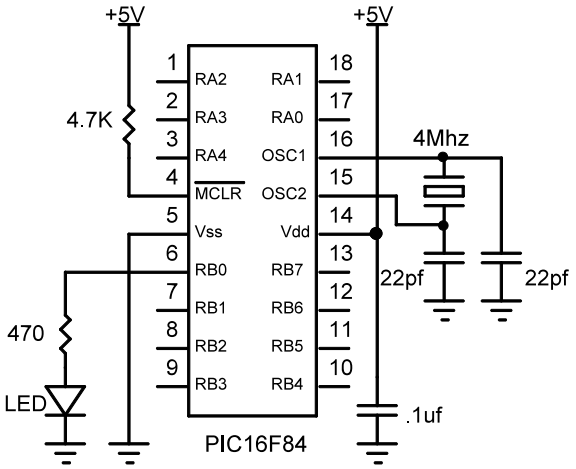
In general, the Oscillator should be set to XT or HS for a 4MHz crystal and the Watchdog Timer should be set to ON for PICBASIC PRO programs. Most importantly, **Code Protect** must be **OFF** when programming any windowed (JW) PIC MCUs. You may not be able to erase a windowed PIC MCU that has been code protected. You can find more information on these configuration fuses in the Microchip data sheet for the device you are using.

When it all looks marvelous, it is time to insert a PIC MCU into the programming socket and click on Program. The PIC MCU will first be checked to make sure it is blank and then your code will be programmed into it.

Once the programming is complete and the LED is no longer red, it is time to test your program.

2.4. It's Alive

The sample schematic below gives you an idea of the few things that need to be connected to the PIC MCU to make it work. Basically, all you need is a pull-up resistor on the /MCLR line, a 4MHz crystal with 2 capacitors, and some kind of 5-volt power supply. We have added an LED and resistor to provide the output from the `BLINK` program.



Build and double check this simple circuit on a protoboard and plug in the PIC MCU you just programmed. Our line of **PICPROTO**™ prototyping boards is perfect for this kind of thing.

Connect a power supply. Your PIC MCU should come to life and start blinking the LED about once a second. If it does not blink, check all of the connections and make sure 5 volts is present at the appropriate pins on the PIC MCU.

From these simple beginnings, you can create your own world-conquering application.

2.5. I've Got Troubles

The most common problems with getting PIC MCUs running involve making sure the few external components are of the appropriate value and properly connected to the PIC MCU. Following are some hints to help get things up and running.

Make sure the /MCLR pin is connected to 5 volts either through some kind of voltage protected reset circuit or simply with a 4.7K resistor. If you leave the pin unconnected, its level floats around and sometimes the PIC MCU will work but usually it won't. The PIC MCU has an on-chip power-on-reset circuit so in general just an external pull-up resistor is adequate. But in some cases the PIC MCU may not power up properly and an external circuit may be necessary. See the Microchip data books for more information.

Be sure you have a good crystal with the proper value capacitors connected to it. Capacitor values can be hard to read. If the values are off by too much, the oscillator won't start and run properly. A 4MHz crystal with two 22pf (picofarad) ceramic disk capacitors is a good start for most PIC MCUs. Once again, check out the Microchip data books for additional thoughts on the matter.

Make sure your power supply is up to the task. While the PIC MCU itself consumes very little power, the power supply must be filtered fairly well. If the PIC MCU is controlling devices that pull a lot of current from your power supply, as they turn on and off they can put enough of a glitch on the supply lines to cause the PIC MCU to stop working properly. Even an LED display can create enough of an instantaneous drain to momentarily clobber a small power supply (like a 9-volt battery) and cause the PIC MCU to lose its mind.

Start small. Write short programs to test features you are unsure of or might be having trouble with. Once these smaller programs are working properly, you can build on them.

Try doing things a different way. Sometimes what you are trying to do looks like it should work but doesn't, no matter how hard you pound on it. Usually there is more than one way to skin a program. Try approaching the problem from a different angle and maybe enlightenment will ensue.

2.5.1. PIC® MCU Specific Issues

It is imperative that you read the Microchip data sheet for the PIC MCU device you are using. Some devices have features that can interfere with expected pin operations. Many PIC MCUs have analog comparators on PORTA or another port. When these chips start up, PORTA is set to analog mode. This makes the pin functions on PORTA work in an unexpected manner. To change the pins to digital, simply add the line:

```
CMCON = 7
```

near the front of your program.

The register names of some PIC MCUs may be different than the examples above (or below). Be sure to check the Microchip data sheet for the device you are using so that you can choose the appropriate register name.

Any PIC MCU with analog inputs will come up in analog mode. You must set them to digital if that is how you intend to use them:

```
ADCON1 = 7
```

For many of the PIC MCUs, including the PIC12F675 and PIC16F676, a different register must be set instead:

```
ANSEL = 0
```

While these settings work for many devices, you will need to check the data sheet for the specific device to verify the exact settings.

Another example of potential disaster is that PORTA, pin 4 exhibits unusual behavior when used as an output. This is because the pin has an open drain output rather than the usual bipolar stage of the rest of the output pins. This means it can pull to ground when set to 0, but it will simply float when set to a 1, instead of going high. To make this pin act in the expected manner, add a pull-up resistor between the pin and 5 volts. The value of the resistor may be between 1K and 33K, depending

on the drive necessary for the connected input. This pin acts as any other pin when used as an input.

Some PIC MCUs, such as the PIC16F62x(A), 16F87x(A) and PIC18F allow low-voltage programming. This function takes over one of the PORTB pins and can cause the device to act erratically if this pin is not pulled low. It is best to make sure that low-voltage programming is not enabled at the time the PIC MCU is programmed.

All of the PIC MCU pins are set to inputs on power-up. If you need a pin to be an output, set it to an output before you use it, or use a PICBASIC PRO command that does it for you. Once again, review the PIC MCU data sheets to become familiar with the idiosyncrasies of a particular part.

There is no data direction (TRIS) register for PORTA on PIC17 devices. Therefore, commands that rely on the TRIS register for their operation, such as **I2CREAD** and **I2CWRITE**, may not be used on PORTA.

The name of the port pins on most 8-pin PIC12 devices is GPIO. The name for the TRIS register is TRISIO.

```
GPIO.0 = 1
TRISIO = %101010
```

On the PIC10 and some PIC12 devices, pin GPIO.2 is forced to an input regardless of the setting of the TRIS register. To allow this pin to be used as a standard I/O pin, add the following line to the beginning of the program:

```
OPTION_REG.5 = 0
```

As hinted at above, the name of the OPTION register that PICBASIC PRO uses for all PIC MCUs is OPTION_REG.

Certain PIC MCUs have on-chip non-volatile data storage implemented like an I2C interfaced serial EEPROM. **READ** and **WRITE** will not work on devices with on-chip I2C interfaced serial EEPROM like the PIC12CE51x, 12CE67x and PIC16CE62x parts. Use the **I2CREAD** and **I2CWRITE** instructions instead.

Some PIC MCUs, such as the PIC12C67x, 12CE67x, 12F6xx and PIC16F6xx, have on-chip RC oscillators. Some of these devices contain an oscillator calibration factor in the last location of code space. The on-chip oscillator may be fine-tuned by acquiring the data from this

location and moving it into the OSCCAL register. Two **DEFINEs** have been created to perform this task automatically each time the program starts:

```
Define OSCCAL_1K 1        ' Set OSCCAL for 1K
                          device
Define OSCCAL_2K 1        ' Set OSCCAL for 2K
                          device
```

Add one of these 2 **DEFINE**s near the beginning of the PICBASIC PRO program to perform the setting of OSCCAL.

If a UV erasable device has been erased, the calibration value is no longer in memory. If one of these **DEFINE**s is used on an erased part, it will cause the program to loop endlessly. To set the OSCCAL register on an erased part, near the beginning of the program, add the line:

```
OSCCAL = $a0        ' Set OSCCAL register to $a0
```

The $a0 is merely an example. The part would need to be read before it is erased to obtain the actual OSCCAL value for that particular device.

PICBASIC PRO will automatically load the OSCCAL value for the 12-bit core devices, if it is available. It is unnecessary to use the above **DEFINE**s with these devices.

Some PIC MCUs with internal oscillators, like the PIC16F88, include an oscillator control register that allows the selection of the oscillator frequency. On power-up or reset, this register may default to a slow oscillator setting like 32kHz. This slow speed may make it look as if the program is not running at all. To set the oscillator to a faster frequency such as 4MHz, set:

```
OSCCON = $60        ' Set OSCCON to 4MHz
```

Please see the Microchip data sheets for more information on OSCCON and any of the other PIC MCU registers.

2.5.2. PICBASIC and BASIC Stamp Compatiblity

There are some differences between the standard PICBASIC Compiler, the BASIC Stamps and the PICBASIC PRO Compiler. See section 10 for information on these differences.

2.5.3. Code Crosses Page Boundary Messages

Many PIC MCUs contain code space memory that is segmented into 512, 2K or 8K word pages. As large files are compiled and then assembled, they may start to use more than the first page. As each page is used, PM, the assembler, will issue a message that the code is crossing a particular boundary. This is normal and no cause for alarm. PBP will make sure to take care of most of the issues for you.

The only thing that you must be aware of is the **BRANCH** instruction. If a **BRANCH** tries to access a label on the other side of a boundary, it will not work properly. **BRANCHL** should be used instead. It can address labels in any code page.

2.5.4. Out of Memory Errors

Compiling large PICBASIC PRO source code files can tax the memory of the PC running the compiler. If an Out of Memory error is issued and the FILES and BUFFERS are set as recommended, an alternate version of PBP can be used. PBPW.EXE has been compiled to make use of all of the memory available to Windows 95, 98, ME, NT, 2000, XP and Vista. You must, of course, be running in a DOS shell from one of these 32-bit Windows environments or be within Microchip's MPLAB or another Windows IDE. To execute the Windows version from the DOS command line, simply substitute PBPW for PBP.

```
PBPW blink
```

2.6. Coding Style

Writing readable and maintainable programs is an art. There are a few simple techniques you can follow that may help you become an artist.

2.6.1. Comments

Use lots of comments. Even though it may be perfectly obvious to you what the code is doing as you write it, someone else looking at the program (or even yourself when you are someone else later in life) may not have any idea of what you were trying to achieve. While comments take up space in your BASIC source file, they do not take up any additional space in the PIC MCU so use them freely.

Make the comments tell you something useful about what the program is doing. A comment like “Set Pin0 to 1" simply explains the syntax of the language but does nothing to tell you why you have the need to do this. Something like “Turn on the Battery Low LED” might be a lot more useful.

A block of comments at the beginning of the program and before each section of code can describe what is about to happen in more detail than just the space remaining after each statement. But don’t include a comment block instead of individual line comments - use both.

At the beginning of the program describe what the program is intended to do, who wrote it and when. It may also be useful to list revision information and dates. Specifying what each pin is connected to can be helpful in remembering what hardware this particular program is designed to run on. If it is intended to be run with a non-standard crystal or special compiler options, be sure to list those as well.

2.6.2. Pin and Variable Names

Make the name of a pin or variable something more coherent than **Pin0** or **B1.** In addition to the liberal use of comments, descriptive pin and variable names can greatly enhance readability. The following code fragment demonstrates:

```
BattLED Var PORTB.0      ' Low battery LED
level Var   Byte         ' Variable will contain the
                         battery level

      If level < 10 Then       ' If batt level is low
            High BattLED       ' Turn on the LED
      Endif
```

2.6.3. Labels

Labels should also be more meaningful than “label1:" or “here:”. Usually the line or routine you are jumping to does something unique. Try and give at least a hint of its function with the label, and then follow up with a comment.

2.6.4. GOTO

Try not to use too many **GOTO**s. While **GOTO**s may be a necessary evil, try to minimize their use as much as possible. Try to write your code in

logical sections and not jump around too much. GOSUBs can be helpful in achieving this.

3. Command Line Options

3.1. Usage

The PICBASIC PRO Compiler can be invoked from the DOS command line using one of the following command formats:

```
PBP Options Filename
PBPW Options Filename
PBPL Options Filename
```

PBP is the DOS version of the compiler. PBPW is the Windows version of the compiler and can take advantage of all the PC's memory. PBPL is the long version of the compiler and can use 32-bit variables, as well as all of the PC's memory under Windows. PBPL can only be used with the PIC18 devices.

Zero or more *Options* can be used to modify the manner in which PBP compiles the specified file. *Options* begin with either a minus (-) or a forward slash (/). The character following the minus or slash is a letter which selects the *Option*. Additional characters may follow if the *Option* requires more information. Each *Option* must be separated by a space and no spaces may occur within an *Option*.

Multiple *Options* may be used at the same time. For example the command line:

```
PBPW -p16f877a blink
```

will cause the file BLINK.BAS to be compiled targeted for a PIC16F877a processor.

The first item not starting with a minus is assumed to be the *Filename*. If no extension is specified, the default extension .BAS is used. If a path is specified, that directory is searched for the named file. Regardless of where the source file is found, files generated by PBP are placed in the current directory.

By default, PBP automatically launches the assembler (PM.EXE) if the compilation has no errors. PBP expects to find PM.EXE in the same directory as PBP.EXE. If the compilation has errors or the **-s** option is used, the assembler is not launched.

If PBP is invoked with no parameters or filename, a brief help screen is displayed.

3.2. Options

Option	Description
A	Use a different Assembler
C	Insert source lines as Comments into assembler file
E	Output errors to a file
H or **?**	Display Help screen
K+	Add source level debugging information (COFF)
K-	Add assembler level debugging information (COFF)
L	Use a different Library file
O	Pass Option to assembler
P	Specify target Processor
S	Skip execution of assembler after compilation
V	Verbose mode
Z	Add source level debugging information (COD)

3.2.1. Option -A

PBP has the capability to use either PM, which is included with PBP, or Microchip's MPASMWIN as its assembler. When using MPASMWIN, PBPW or PBPL must be specified instead of PBP. PBPW is the Windows executable version of PBP. To specify MPASMWIN as the assembler, use **-ampasmwin** on the command line:

```
PBPW -ampasmwin filename
PBPL -ampasmwin filename
```

MPASMWIN must be acquired from Microchip and set up in the system path. See the file MPLAB.TXT on the disk for more details.

If no assembler is specified on the command line, PM is used.

3.2.2. Option -C

The -C option causes PBP to insert the PICBASIC PRO source file lines as comments into the assembly language source file. This can be useful as a debugging tool or learning tool as it shows the PICBASIC PRO instruction followed by the assembly language instructions it generates.

```
PBPW -c filename
```

3.2.3. Option -E

The -E option causes PBP to send all the errors to a file, filename.er.

```
PBPW -e filename
```

3.2.4. Option -H or -?

The -H or -?option causes PBP to display a brief help screen. This help screen is also displayed if no option and filename is specified on the command line.

3.2.5. Option -K+

The -K+ option tells PBPW to add BASIC source level simulation and debugging information to the COFF file that is generated during compilation. For an example of how to use this within MPLAB 8.20 and later, see the MPLAB.TXT file.

```
PBPW -k+ -ampasmwin filename
```

3.2.6. Option -K-

The -K- option tells PBPW to add assembler level simulation and debugging information to the COFF file that is generated during compilation. For an example of how to use this within MPLAB 8.20 and later, see the MPLAB.TXT file.

```
PBPW -k- -ampasmwin filename
```

3.2.7. Option -L

The -L option lets you select the library used by PICBASIC PRO. This option is generally unnecessary as the default library file is set in a configuration file for each microcontroller. For more information on PICBASIC PRO libraries, see the advanced sections later in this manual.

```
PBPW -lpbpps2 filename
```

This example tells PBP to compile `filename` using the PicStic2 library.

3.2.8. Option -O

The -O option causes the letters following it to be passed to the assembler on its command line as options.

The PM assembler's manual on disk contains information about the assembler and its options.

```
PBPW -ol filename
```

This example tells PBP to generate a `filename.lst` file after a successful compilation.

More than one -O option may be passed to the assembler at a time.

3.2.9. Option -P

If not told otherwise, PBP compiles programs for the PIC16F84. If the program requires a different processor as its target, its name must be specified on the command line use the **-P** option.

For example, if the desired target processor for the PBP program is a PIC16F877a, the command line should look something like the following:

```
PBPW -p16F877a filename
```

3.2.10. Option -S

Normally, when PBP successfully compiles a program, it automatically launches the assembler. This is done to convert the assembler output of PBP to a `.HEX` file. The **-S** option prevents this, leaving PBP's output in

the generated .ASM file.

Since -s prevents the assembler from being invoked, options that are simply passed to the assembler using the -o command line switch are effectively ignored.

```
PBPW -s filename
```

3.2.11. Option -v

The -v option turns on PBP's verbose mode which presents more information during program compilation.

```
PBPW -v filename
```

3.2.12. Option -z

The -z option tells PBPW to add source level simulation and debugging information to the COD file that is generated during compilation. For an example of how to use this within MPLAB versions prior to 8.20, see the MPLAB.TXT file.

```
PBPW -z -ampasmwin filename
```

4. PICBASIC PRO Basics

4.1. Identifiers

An identifier is, quite simply, a name. Identifiers are used in PBP for line labels and variable names. An identifier is any sequence of letters, digits, and underscores, although it must not start with a digit. Identifiers are not case sensitive, thus label, LABEL, and Label are all treated as equivalent. And while labels might be any number of characters in length, PBP only recognizes the first 31.

4.2. Line Labels

In order to mark statements that the program might wish to reference with **GOTO** or **GOSUB** commands, PBP uses line labels. Unlike many older BASICs, PBP doesn't allow line numbers and doesn't require that each line be labeled. Rather, any PBP line may start with a line label, which is simply an identifier followed by a colon (**:**).

```
here: Serout 0,N2400,["Hello, World!",13,10]
      Goto here
```

4.3. Variables

Variables are where temporary data is stored in a PICBASIC PRO program. They are created using the **VAR** keyword. Variables may be bit-, byte- and word-sized for PBP and PBPW, and bit-, byte-, word- and long-sized for PBPL. Space for each variable is automatically allocated in the microcontroller's RAM by PBP. The format for creating a variable is as follows:

```
Label VAR   Size{.Modifiers}
```

Label is any unique identifier, excluding keywords, as described above. *Size* is **BIT**, **BYTE**, **WORD** or, for PBPL, **LONG**.

Some examples of creating variable are:

```
dog   VAR   BYTE
cat   VAR   BIT
w0    VAR   WORD
big   VAR   LONG        ' PBPL only
```

Optional *Modifiers* add additional control over how the variable is created and are listed in the section on Aliases, below.

The size and range of each variable type is detailed in the following table:

Size	# of bits	Range
BIT	1	0 to 1
BYTE	8	0 to 255
WORD	16	0 to 65535
LONG *	32	-2147483648 to 2147483647

* PBPL only.

As the table shows, bit-, byte- and word-sized variables are always unsigned, i.e. positive numbers. Long-sized variables, which are only available in PBPL, are always signed, twos-complement numbers, including positive and negative values.

PBPL interprets only long variable types as signed numbers. Words, bytes, and of course bits are always interpreted as positive, unsigned integers when used as terms in a PBP math operation.

If the result of an operation could possibly be negative, it should be stored to a long-sized variable type to preserve the sign. If a negative result is placed in a variable type other than long, subsequent calculations using this value will interpret it as a positive number.

There are no predefined user variables in PICBASIC PRO. For compatibility sake, two files have been provided that create the standard variables used with the BASIC Stamps: `BS1DEFS.BAS` and `BS2DEFS.BAS`. To use one of these files, add the line:

```
Include "bs1defs.bas"
```

or

```
Include "bs2defs.bas"
```

near the top of the PICBASIC PRO program. These files contain numerous **VAR** statements that create all of the BASIC Stamp variables and pin definitions.

However, instead of using these "canned" files, we recommend you

create your own variables using names that are meaningful to you.

The number of variables available depends on the amount of RAM on a particular device and the size of the variables and arrays. PBP reserves approximately 24 RAM locations for its own use. It may also create temporary variables that use additional RAM locations that are used for sorting out complex equations.

4.4. Aliases

VAR can also be used to create an alias (another name) for a variable. This is most useful for accessing the innards of a variable.

```
fido   VAR   dog          ' fido is another name
                          for dog
b0     VAR   w0.BYTE0     ' b0 is the first byte
                          of word w0
b1     VAR   w0.BYTE1     ' b1 is the second byte
                          of word w0
flea   VAR   dog.0        ' flea is bit 0 of dog
```

These variable modifiers may also be used in statements:

```
b = w0.BYTE0
OPTION_REG.7 = 0
```

Modifier	Description
BIT0 or **0**	Create alias to bit 0 of byte or word
BIT1 or **1**	Create alias to bit 1 of byte or word
BIT2 or **2**	Create alias to bit 2 of byte or word
BIT3 or **3**	Create alias to bit 3 of byte or word
BIT4 or **4**	Create alias to bit 4 of byte or word
BIT5 or **5**	Create alias to bit 5 of byte or word
BIT6 or **6**	Create alias to bit 6 of byte or word
BIT7 or **7**	Create alias to bit 7 of byte or word
BIT8 or **8**	Create alias to bit 8 of word or long
BIT9 or **9**	Create alias to bit 9 of word or long
BIT10 or **10**	Create alias to bit 10 of word or long

BIT11 or **11**	Create alias to bit 11 of word or long
BIT12 or **12**	Create alias to bit 12 of word or long
BIT13 or **13**	Create alias to bit 13 of word or long
BIT14 or **14**	Create alias to bit 14 of word or long
BIT15 or **15**	Create alias to bit 15 of word or long
BIT16 or **16***	Create alias to bit 16 of long
BIT17 or **17***	Create alias to bit 17 of long
BIT18 or **18***	Create alias to bit 18 of long
BIT19 or **19***	Create alias to bit 19 of long
BIT20 or **20***	Create alias to bit 20 of long
BIT21 or **21***	Create alias to bit 21 of long
BIT22 or **22***	Create alias to bit 22 of long
BIT23 or **23***	Create alias to bit 23 of long
BIT24 or **24***	Create alias to bit 24 of long
BIT25 or **25***	Create alias to bit 25 of long
BIT26 or **26***	Create alias to bit 26 of long
BIT27 or **27***	Create alias to bit 27 of long
BIT28 or **28***	Create alias to bit 28 of long
BIT29 or **29***	Create alias to bit 29 of long
BIT30 or **30***	Create alias to bit 30 of long
BIT31 or **31***	Create alias to bit 31 of long
BYTE0 or **LOWBYTE**	Create alias to low byte of word or long
BYTE1 or **HIGHBYTE**	Create alias to high byte of word or long
BYTE2*	Create alias to upper byte of long
BYTE3*	Create alias to top byte of long
WORD0*	Create alias to low word of long
WORD1*	Create alias to high word of long

* PBPL only.

4.5. Arrays

Variable arrays can be created in a similar manner to variables.

```
Label VAR   Size[Number of elements]
```

Label is any identifier, excluding keywords, as described above. *Size* is **BIT**, **BYTE** or **WORD** (or **LONG** for PBPL.) *Number of elements* is how many array locations is desired. Some examples of creating arrays are:

```
sharks VAR  BYTE[10]
fish   VAR  BIT[8]
```

The first array location is element 0. In the `fish` array defined above, the elements are numbered `fish[0]` to `fish[7]` yielding 8 elements in total.

Because of the way arrays are accessed and allocated in memory, there are size limits for each type:

Size	Maximum Number of elements
BIT	256
BYTE	96*
WORD	48*
LONG	**

* Processor family dependent.
** PIC18 only - no set limit.

Arrays must fit entirely within one RAM bank on 12-bit, 14-bit or PIC17 devices. Arrays may span banks on PIC18 devices. On PIC18 devices, byte, word and long-sized arrays are only limited in length by the amount of available memory. The compiler will assure that arrays, as well as simple variables, will fit in memory before successfully compiling.

4.6. Symbols

SYMBOL provides yet another method for aliasing variables and constants. It is included for BS1 compatibility. **SYMBOL** cannot be used to create a variable. Use **VAR** to create a variable.

```
SYMBOL lion = cat ' cat was previously created
                  using VAR
SYMBOL mouse = 1  ' Same as mouse CON 1
```

4.7. Constants

Named constants may be created in a similar manner to variables. It may be more convenient to use a name for a constant instead of using a constant number. If the number needs to be changed, it may be changed in only one place in the program; where the constant is defined. Variable data cannot be stored in a constant.

```
Label CON   Constant expression
```

Some examples of constants are:

```
mice  CON   3
traps CON   mice * 1000
```

4.8. Numeric Constants

PBP allows numeric constants to be specified in one of three bases: decimal, binary and hexadecimal. Binary values are specified using the prefix '**%**' and hexadecimal values using the prefix '**$**'. Decimal values are the default and require no prefix.

```
100   ' Decimal value 100
%100  ' Binary value for decimal 4
$100  ' Hexadecimal value for decimal 256
```

For ease of programming, single characters are converted to their ASCII equivalents. Character constants must be quoted using double quotes and must contain only one character (otherwise, they are string constants, see below).

```
"A"   ' ASCII value for decimal 65
"d"   ' ASCII value for decimal 100
```

4.9. String Constants

PBP doesn't provide string handling capabilities per se, but strings can be used with some commands. A string contains one or more characters and is delimited by double quotes. No escape sequences are supported for non-ASCII characters (although most PBP commands have this handling built-in).

```
Lcdout "Hello"      ' Output String (Short for
                    "H","e","l","l","o")
```

Strings are usually treated as a list of individual character values.

4.10. Ports and Other Registers

All of the PIC MCU registers, including the ports, can be accessed just like any other byte-sized variable in PICBASIC PRO. This means that they can be read from, written to or used in equations directly:

```
PORTA = %01010101       ' Write value to PORTA
anyvar = PORTB & $0f    ' Isolate lower 4 bits
                        of PORTB and place
                        result into anyvar
```

4.11. Pins

Pins may be accessed in a number of different ways. The simplest way to specify a pin for an operation is to simply use its PORT name and bit number:

```
PORTB.1 = 1 ' Set PORTB, bit 1 to a 1
```

To make it easier to remember what a pin is used for, it may be assigned a name using the **VAR** command. In this manner, the name may then be used in any operation:

```
led   Var   PORTA.0      ' Rename PORTA.0 as led
      High led           ' Set led (PORTA.0) high
```

Another way to create an alias to a pin is with the **PIN** command:

```
led   Pin   0
```

PIN uses the following constant, 0 - 15, to set an alias to a pin on PORTL (0 - 7) or PORTH (8 - 15), as shown in the table below. While **VAR**,

above, may be used to specify any pin on any port, **PIN** may only be used with constants 0 - 15 to specify up to 16 pins on PORTL and PORTH. The command example above is the equivalent of:

```
led   Var   PORTL.0
```

For compatibility with the BASIC Stamp, pins used in PICBASIC PRO Compiler commands may also be referred to by a number, 0 - 15. This number references different physical pins on the PIC MCU hardware ports dependent on how many pins the microcontroller has.

No. PIC MCU Pins	0 - 7 (PORTL)	8 - 15 (PORTH)
8-pin	GPIO	GPIO
14 and 20-pin	PORTA	PORTC
18-pin	PORTB	PORTA
28-pin (except 14000)	PORTB	PORTC
14000	PORTC	PORTD
40-pin and up	PORTB	PORTC

If a port does not have 8 pins, such as PORTA, only the pin numbers that exist may be used, i.e. 8 - 12. Using pin numbers 13 - 15 will have no discernable effect.

This pin number, 0 - 15, has nothing to do with the physical pin number of a PIC MCU. Depending on the particular PIC MCU, pin number 0 could be physical pin 6, 21 or 33, but in each case it maps to PORTB.0 (or GPIO.0 for 8-pin devices, or PORTA.0 for 14 and 20-pin devices, or PORTC.0 for a PIC14000).

```
High 0       ' Set PORTB.0 (or GPIO.0) high

B0 = 9       ' Select PORTC.1 (or PORTA.1)
Toggle B0    ' Toggle PORTC.1 (or PORTA.1)
```

Pins may be referenced by number (0 - 15), name (e.g. **Pin0**, if BS1DEFS.BAS or BS2DEFS.BAS is included or you have defined them yourself), or full bit name (e.g. PORTA.1). Any pin or bit of the microcontroller can be accessed using the last method.

The pin names (i.e. **Pin0**) are not automatically included in your program. In most cases, you would define pin names as you see fit using

the **VAR** command:

```
led   Var   PORTB.3
```

However, two definition files have been provided to enhance BASIC Stamp compatibility. The files BS1DEFS.BAS or BS2DEFS.BAS may be included in the PICBASIC PRO program to provide pin and bit names that match the BASIC Stamp names.

```
Include "bs1defs.bas"
```

or

```
Include "bs2defs.bas"
```

BS1DEFS.BAS defines **Pins**, **B0-B13**, **W0-W6** and most of the other BS1 pin and variable names.

BS2DEFS.BAS defines **Ins**, **Outs**, **InL, Inh, OutL, Outh, B0-B25**, **W0-W12** and most of the other BS2 pin and variable names.

PORTL and **PORTH** are also defined in PBP. **PORTL** encompasses **Pin0** - **Pin7** and **PORTH** encompasses **Pin8** - **Pin15.**

When a PIC MCU powers-up, all of the pins are set to input. To use a pin as an output, the pin or port must be set to an output or a command must be used that automatically sets a pin to an output.

To set a pin or port to an output (or input), set its TRIS register. Setting a TRIS bit to 0 makes its corresponding port pin an output. Setting a TRIS bit to 1 makes its corresponding port pin an input. For example:

```
TRISA = %00000000        ' Or TRISA = 0
```

sets all the PORTA pins to outputs.

```
TRISB = %11111111        ' Or TRISB = 255
```

sets all the PORTB pins to inputs.

```
TRISC = %10101010
```

Sets all the even pins on PORTC to outputs, and the odd pins to inputs. Individual bit directions may be set in the same manner.

```
TRISA.0 = 0
```

sets PORTA, pin 0 to an output. All of the other pin directions on PORTA are unchanged.

The BASIC Stamp variable names **Dirs**, **Dirh**, **Dirl** and **Dir0-Dir15** are not defined and must not be used with the PICBASIC PRO Compiler. TRIS must be used instead, but has the opposite state of **Dirs**.

This **does not** work in PICBASIC PRO:

```
Dir0 = 1     ' Doesn't set pin PORTB.0 to output
```

Do this instead:

```
TRISB.0 = 0 ' Set pin PORTB.0 to output
```

4.12. Comments

A PBP comment starts with either the **REM** keyword, the single quote (') or the semi-colon (;). All following characters on this line are ignored.

Unlike many BASICs, **REM** is a unique keyword and not an abbreviation for REMark. Thus, variables names may begin with **REM**.

4.13. Multi-statement Lines

In order to allow more compact programs and logical grouping of related commands, PBP supports the use of the colon (:) to separate statements placed on the same line. Thus, the following two examples are equivalent:

```
W2 = W0
W0 = W1
W1 = W2
```

is the same as:

```
W2 = W0 : W0 = W1 : W1 = W2
```

This does not, however, change the size of the generated code.

4.14. Line-extension Character

The maximum number of characters that may appear on one PBP line is

256. Longer statements may be extended to the next line using the line-extension character (_) at the end of each line to be continued.

```
Branch B0,[label0,label1,label2,_
label3,label4]
```

4.15. INCLUDE

Other BASIC source files may be added to a PBP program by using **INCLUDE**. You may have standardized subroutines, definitions or other files that you wish to keep separate. The Stamp and serial mode definition files are examples of this. These files may be included in programs when they are necessary, but kept out of programs where they are not needed.

The included file's source code lines are inserted into the program exactly where the **INCLUDE** is placed.

```
INCLUDE "modedefs.bas"
```

4.16. DEFINE

Some elements, like the clock oscillator frequency and the LCD pin locations, are predefined in PBP. **DEFINE** allows a PBP program to change these definitions, if desired.

DEFINE may be used to change the predefined oscillator value, the **DEBUG** pins and baud rate and the LCD pin locations, among other things.

These definitions must be in all upper case, exactly as shown. If not, the compiler may not recognize them. No error message will be produced for **DEFINE**s the compiler does not recognize.

See the appropriate sections of the manual for specific information on these definitions. A list of **DEFINE**s is shown in Appendix B.

```
DEFINE OSC 4        ' Oscillator speed in MHz:
                    3(3.58) 4 8 10 12 16 20 24 25
                    32 33 40 48 64
```

4.17. Math Operators

Unlike the BASIC Stamp, the PICBASIC PRO Compiler performs all math

and comparison operations in full hierarchal order. This means that there is precedence to the operators. Multiplies and divides are performed before adds and subtracts, for example. To ensure the operations are carried out in the order you would like, use parenthesis to group the operations:

```
A = (B + C) * (D - E)
```

All math operations when compiling with PBP and PBPW are unsigned and performed with 16-bit precision. Math operations for PBPL use 32-bit precision.

Bitwise operators, including the shift operators, always operate in an unsigned fashion, regardless of the variable type they are acting on, signed or unsigned.

Math Operators	Description
+	Addition
-	Subtraction
*	Multiplication
**	Top 16 Bits of Multiplication
*/	Middle 16 Bits of Multiplication
/	Division
//	Remainder (Modulus)
<<	Shift Left
>>	Shift Right
ABS	Absolute Value*
ATN	Arctangent
COS	Cosine
DCD	2n Decode
DIG	Digit
DIV32	31-bit x 15-bit Divide
HYP	Hypotenuse
MAX	Maximum*
MIN	Minimum*
NCD	Encode
REV	Reverse Bits
SIN	Sine
SQR	Square Root
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive OR
~	Bitwise NOT
&/	Bitwise NOT AND
|/	Bitwise NOT OR
^/	Bitwise NOT Exclusive OR

*Implementation differs from BASIC Stamp.

4.17.1. Multiplication

PBP and PBPW perform unsigned 16-bit x 16-bit multiplication, while PBPL performs signed 32-bit x 32-bit multiplication.

```
W1 = W0 * 1000      ' Multiply value in W0 by 1000
                    and place the result in W1
```

PBPL interprets only long variable types as signed numbers. Words, bytes, and of course bits are always interpreted as positive, unsigned integers when used as terms in a PBP math operation.

If the result of a multiplication could possibly be negative, it should be stored to a long-sized variable type to preserve the sign. If a negative result is placed in a variable type other than long, subsequent calculations using this value will interpret it as a positive number.

```
B0 = 4
L0 = B0 * -1        ' Result is -4 in L0
W0 = B0 * -1        ' Result is 65,532 in W0
```

'***/**' and '******' Operators

There are two special multiplication operators that allow large result values to be handled in a special way. These operators ignore some of the least-significant bytes of the result and return higher order bytes instead. With PBP and PBPW, this allows you to work (in a limited way) with 32-bit multiplication results. With PBPL, the top 32 bits of a 48-bit result are available.

The '***/**' operator discards the least-significant byte of the result (byte0), and returns the 4 higher bytes to the result variable. If the result variable is a word or byte, the value will be truncated to fit.

```
W3 = W1 */ W0       ' Multiply W1 by W0, ignore
                    byte0 of the result, return
                    byte1 and byte2 in W3

L3 = L1 */ L0       ' Multiply L1 by L0, ignore
                    byte0 of the result, return
                    byte1 through byte4 in L3
```

A simple way to think about '***/**' is that it shifts the result 8 places to the

right, resulting in an automatic division by 256. (This does not hold true if the result is a negative number.) This is useful for multiplying by non-integer constants.

If you wished to convert miles to kilometers, for example, you would need to multiply by a constant 1.6. PBP's integer math won't allow you to write "1.6" in an equation, but you can use '***/**' to accomplish the same result:

```
kilometers = miles */ 410      ' Same as
                               kilometers =
                               (miles * 410) /
                               256
```

The '******' operator is similar, but ignores two bytes instead of one. When using PBPL with long variable types, it returns byte2 through byte5 of the 48-bit result value. This gives a result that is shifted 16 places to the right, an inherent divide by 65536.

```
W2 = W0 ** 1000    ' Multiply W0 by 1000 and
                   place the high order 16 bits
                   (which may be 0) in W2
```

4.17.2. Division

PBP and PBPW perform unsigned 16-bit x 16-bit division. The '**/**' operator returns the 16-bit result.

PBPL performs signed 32-bit x 32-bit division. The '**/**' operator returns the 32-bit result.

The '**//**' operator returns the remainder. This is sometimes referred to as the modulus of the number.

```
W1 = W0 / 1000     ' Divide value in W0 by 1000
                   and place the result in W1
W2 = W0 // 1000    ' Divide value in W0 by 1000
                   and place the remainder in W2
```

PBPL interprets only long variable types as signed numbers. Words, bytes, and of course bits are always interpreted as positive, unsigned integers when used as terms in a PBP math operation.

If the result of a division could possibly be negative, it should be stored to a long-sized variable type to preserve the sign. If a negative result is

placed in a variable type other than long, subsequent calculations using this value will interpret it as a positive number.

```
B0 = 4
L0 = B0 / -1        ' Result is -4 in L0
W0 = B0 / -1        ' Result is 65,532 in W0
```

The same applies to the '//' operator:

```
B0 = 23
L0 = B0 // -4       ' Result is -3 in L0
W0 = B0 // -4       ' Result is 65,533 in W0
```

4.17.3. Shift

The '<<' and '>>' operators shift a value left or right, respectively, 0 to 31 times. The newly shifted-in bits are set to 0.

```
B0 = B0 << 3        ' Shifts B0 left 3 places
                    (same as multiply by 8)
W1 = W0 >> 1        ' Shifts W0 right 1 position
                    and places result in W1 (same
                    as divide by 2)
```

4.17.4. ABS

ABS returns the absolute value of a number. If a byte is greater than 127 (high bit set), **ABS** will return 256 - value. If a word is greater than 32767 (high bit set), **ABS** will return 65536 - value. If a long is negative, **ABS** will return 4294967296 - value.

```
B1 = ABS B0
```

4.17.5. ATN

ATN returns the 8-bit arctangent of 2 twos-complement 8-bit values. If a byte is greater than 127 (high bit set), it is treated as a negative value. The arctangent returned is in binary radians, 0 to 255, representing a range of 0 to 359 degrees.

```
ang = x ATN y
```

4.17.6. COS

COS returns the 8-bit cosine of a value. The result is in two's complement form (i.e. -127 to 127). It uses a quarter-wave lookup table to find the result. Cosine starts with a value in binary radians, 0 to 255, as opposed to the usual 0 to 359 degrees.

```
B1 = COS B0
```

4.17.7. DCD

DCD returns the decoded value of a bit number. It changes a bit number (0 - 31) into a binary number with only that bit set to 1. All other bits are set to 0.

```
B0 = DCD 2          ' Sets B0 to %00000100
```

4.17.8. DIG

DIG returns the value of a decimal digit. Simply tell it the digit number (0 - 9 with 0 being the rightmost digit) you would like the value of, and voila.

```
B0 = 123            ' Set B0 to 123
B1 = B0 DIG 2       ' Sets B1 to 1 (digit 2 of
                    123)
```

4.17.9. DIV32

PBP and PBPW's multiply (*) function operates as a 16-bit x 16-bit multiply yielding a 32-bit internal result. However, since the compiler only supports a maximum variable size of 16 bits, access to the result had to happen in 2 steps: c = b * a returns the lower 16 bits of the multiply while d = b ** a returns the upper 16 bits. There was no way to access the 32-bit result as a unit.

In many cases it is desirable to be able to divide the entire 32-bit result of the multiply by a 16-bit number for averaging or scaling. A new function has been added for this purpose: **DIV32**. **DIV32** is actually limited to dividing a 31-bit unsigned integer (max 2147483647) by a 15-bit unsigned integer (max 32767). This should suffice in most circumstances.

As the compiler only allows a maximum variable size of 16 bits, **DIV32**

relies that a multiply was just performed and that the internal compiler variables still contain the 32-bit result of the multiply. No other operation may occur between the multiply and the **DIV32** or the internal variables may be altered, destroying the 32-bit multiplication result.

This means, among other things, that **ON INTERRUPT** must be **DISABLE**d from before the multiply until after the **DIV32**. If **ON INTERRUPT** is not used, there is no need to add **DISABLE** to the program. Interrupts in assembler should have no effect on the internal variables so they may be used without regard to **DIV32**.

The following code fragment shows the operation of **DIV32**:

```
a      Var   Word
b      Var   Word
c      Var   Word
dummy  Var   Word

       b = 500
       c = 1000

       Disable      ' Necessary if On Interrupt used

       dummy = b * c      ' Could also use ** or */
       a = DIV32 100

       Enable       ' Necessary if On Interrupt used
```

This program assigns b the value 500 and c the value 1000. When multiplied together, the result would be 500000. This number exceeds the 16-bit word size of a variable (65535). So the dummy variable contains only the lower 16 bits of the result. In any case, it is not used by the **DIV32** function. **DIV32** uses variables internal to the compiler as the operands.

In this example, **DIV32** divides the 32-bit result of the multiplication b * c by 100 and stores the result of this division, 5000, in the word-sized variable a.

DIV32 is not supported by PBPL as that version of the compiler always uses a 32-bit x 32-bit divide.

4.17.10. HYP

HYP returns the hypotenuse of a right triangle, or the length of the side opposite the right angle. It simply calculates the square root of the sum of the squares of the length of the 2 sides adjacent to the right angle.

For PBPW, the input values are treated as twos-complement numbers representing a range of +127 to -128. For PBPL, a long variable or constant must be used if a negative value is to be represented. In any case, the value returned is always positive.

```
B2 = B0 HYP B1      ' Same as B2 = SQR ((B0 * B0)
                    + (B1 * B1))
```

4.17.11. MAX and MIN

MAX and **MIN** returns the maximum and minimum, respectively, of two numbers. It is usually used to limit numbers to a value.

```
B1 = B0 MAX 100     ' Set B1 to the larger of B0
                    and 100 (B1 will be between
                    100 & 255)
B1 = B0 MIN 100     ' Set B1 to the smaller of B0
                    and 100 (B1 can't be bigger
                    than 100)
```

4.17.12. NCD

NCD returns the priority encoded bit number (1 - 32) of a value. It is used to find the highest bit set in a value. It returns 0 if no bit is set.

```
B0 = NCD %01001000        ' Sets B0 to 7
```

4.17.13. REV

REV reverses the order of the lowest bits in a value. The number of bits to be reversed is from 1 to 32.

```
B0 = %10101100 REV 4      ' Sets B0 to %00000011
```

4.17.14. SIN

SIN returns the 8-bit sine of a value. The result is in two's complement form (i.e. -127 to 127). It uses a quarter-wave lookup table to find the result. Sine starts with a value in binary radians, 0 to 255, as opposed to the usual 0 to 359 degrees.

```
B1 = SIN B0
```

4.17.15. SQR

SQR returns the square root of a value. Since PICBASIC PRO only works with integers, the result will always be an 8-bit integer (16-bits for PBPL) no larger than the actual result.

```
B0 = SQR W1 ' Sets B0 to square root of W1
```

4.17.16. Bitwise Operators

Bitwise operators act on each bit of a value in boolean fashion. They can be used to isolate bits or add bits into a value.

```
B0 = B0 & %00000001     ' Isolate bit 0 of B0
B0 = B0 | %00000001     ' Set bit 0 of B0
B0 = B0 ^ %00000001     ' Flip state of bit 0 of
                        B0
```

4.18. Comparison Operators

Comparison operators are used in **IF..THEN** statements to compare one expression with another. These comparisons for bytes and words are unsigned. They cannot be used to test if a number is less than 0. Long variables in PBPL are signed and can be tested for less than 0.

Comparison Operator	Description
= or ==	Equal
<> or !=	Not Equal
<	Less Than
>	Greater Than
<=	Less Than or Equal
>=	Greater Than or Equal

```
If i > 10 Then Exit
```

4.19. Logical Operators

Logical operators differ from bitwise operations. They yield a true/false result from their operation. Values of 0 are treated as false. Any other value is treated as true. They are mostly used in conjunction with the comparison operators in an **IF..THEN** statement.

Logical Operator	Description
AND or **&&**	Logical AND
OR or **||**	Logical OR
XOR or **^^**	Logical Exclusive OR
NOT or **!**	Logical NOT
ANDNOT	Logical NAND
ORNOT	Logical NOR
XORNOT	Logical NXOR

```
If (A == big) AND (B > mean) Then run
```

Be sure to use parenthesis to tell PBP the exact order of operation.

5. PICBASIC PRO Statement Reference

@	Insert one line of assembly language code.
ADCIN	Read on-chip analog to digital converter.
ARRAYREAD	Parse array (string) and fill variables.
ARRAYWRITE	Send variables and constants to array (string).
ASM..ENDASM	Insert assembly language code section.
BRANCH	Computed GOTO (equivalent to ON GOTO).
BRANCHL	BRANCH out of page (long BRANCH).
BUTTON	Debounce and auto-repeat input on specified pin.
CALL	Call assembly language subroutine.
CLEAR	Zero all variables.
CLEARWDT	Clear (tickle) Watchdog Timer.
COUNT	Count number of pulses on a pin.
DATA	Define initial contents of on-chip EEPROM.
DEBUG	Asynchronous serial output with fixed pin and baud.
DEBUGIN	Asynchronous serial input with fixed pin and baud.
DISABLE	Disable ON DEBUG and ON INTERRUPT processing.
DISABLE DEBUG	Disable ON DEBUG processing.
DISABLE INTERRUPT	Disable ON INTERRUPT processing.
DO..LOOP	Repeatedly execute a block of statements.
DTMFOUT	Produce touch-tone frequencies on a pin.
EEPROM	Define initial contents of on-chip EEPROM.
ENABLE	Enable ON DEBUG and ON INTERRUPT processing.
ENABLE DEBUG	Enable ON DEBUG processing.
ENABLE INTERRUPT	Enable ON INTERRUPT processing.
END	Stop program execution and enter low power mode.
ERASECODE	Erase block of code memory.
EXIT	Exit the current block structure.
FOR..NEXT	Repeatedly execute statements in a counted loop.
FREQOUT	Produce 1 or 2 frequencies on a pin.
GOSUB	Call BASIC subroutine at specified label.
GOTO	Continue execution at specified label.
HIGH	Make pin output high.
HPWM	Output hardware pulse width modulated pulse train.
HSERIN	Hardware asynchronous serial input.
HSERIN2	Hardware asynchronous serial input, second port.
HSEROUT	Hardware asynchronous serial output.
HSEROUT2	Hardware asynchronous serial output, second port.
I2CREAD	Read from I^2C device.

I2CWRITE Write to I^2C device.
IF..THEN..ELSE..ENDIF Conditionally execute statements.
INPUT Make pin an input.
LCDIN Read from LCD RAM.
LCDOUT Display characters on LCD.
{**LET**} Assign result of an expression to a variable.
LOOKDOWN Search constant table for value.
LOOKDOWN2 Search constant / variable table for value.
LOOKUP Fetch constant value from table.
LOOKUP2 Fetch constant / variable value from table.
LOW Make pin output low.
NAP Power down processor for short period of time.
ON DEBUG Execute BASIC debug monitor.
ON GOSUB Computed **GOSUB**.
ON GOTO Computed **GOTO** (equivalent to **BRANCHL**).
ON INTERRUPT Execute BASIC subroutine on an interrupt.
OWIN 1-wire input.
OWOUT 1-wire output.
OUTPUT Make pin an output.
PAUSE Delay (1 millisecond resolution).
PAUSEUS Delay (1 microsecond resolution).
PEEK Read byte from register.
PEEKCODE Read byte from code space.
POKE Write byte to register.
POKECODE Write byte to code space when programming device.
POT Read potentiometer on specified pin.
PULSIN Measure pulse width on a pin.
PULSOUT Generate pulse on a pin.
PWM Output pulse width modulated pulse train to pin.
RANDOM Generate pseudo-random number.
RCTIME Measure pulse width on a pin.
READ Read byte from on-chip EEPROM.
READCODE Read word from code memory.
REPEAT..UNTIL Execute statements until condition is true.
RESUME Continue execution after interrupt handling.
RETURN Continue at statement following last **GOSUB**.
REVERSE Make output pin an input or an input pin an output.
SELECT CASE Compare a variable with different values.
SERIN Asynchronous serial input (BS1 style).
SERIN2 Asynchronous serial input (BS2 style).
SEROUT Asynchronous serial output (BS1 style).

SEROUT2	Asynchronous serial output (BS2 style).
SHIFTIN	Synchronous serial input.
SHIFTOUT	Synchronous serial output.
SLEEP	Power down processor for a period of time.
SOUND	Generate tone or white-noise on specified pin.
STOP	Stop program execution.
SWAP	Exchange the values of two variables.
TOGGLE	Make pin output and toggle state.
USBIN	USB input.
USBINIT	Initialize USB.
USBOUT	USB output.
USBSERVICE	USB service loop.
WHILE..WEND	Execute statements while condition is true.
WRITE	Write byte to on-chip EEPROM.
WRITECODE	Write word to code memory.
XIN	X-10 input.
XOUT	X-10 output.

5.1. @

```
@ Statement
```

When used at the beginning of a line, @ provides a shortcut for inserting one assembly language *Statement* into your BASIC program. You can use this shortcut to mix assembly language code with PICBASIC PRO statements.

```
i      Var   Byte
rollme Var   Byte

       For i = 1 To 4
@      rlf   _rollme, F  ; Rotate byte left once
       Next i
```

The @ shortcut can also be used to include assembly language routines in another file. For example:

```
@      Include "fp.asm"
```

@ resets the register page to 0 before executing the assembly language instruction. The register page should not be altered using @.

See the section on assembly language programming for more information.

5.2. ADCIN

ADCIN *Channel,Var*

Read the on-chip analog to digital converter *Channel* and store the result in *Var*. While the ADC registers can be accessed directly, **ADCIN** makes the process a little easier.

Before **ADCIN** can be used, the appropriate TRIS register must be set to make the desired pins inputs. The ADCON, ANCON and/or ANSEL registers must also be set to assign the desired pins to analog inputs and in some cases to set the result format and clock source (set the clock source the same as the **DEFINE** specified for it, below). See the Microchip data sheets for more information on these registers and things like the clock source and how to set them for the specific device. Note: The PIC14000 ADC is not compatible with the **ADCIN** instruction.

Depending on the device, it may have an 8-, 10- or 12-bit ADC. For many PIC MCUs, the high bit of ADCON0 or ADCON1 controls whether the result is left or right justified. In most cases, 8-bit results should be left justified (ADCON1.7 = 0) and 10- and 12-bit results should be right justified (ADCON1.7 = 1).

Several **DEFINE**s may also be used. The defaults are shown below:

```
DEFINE ADC_BITS 8        ' Set number of bits in
                         result (8, 10 or 12)
DEFINE ADC_CLOCK 3       ' Set clock source (rc =
                         3)
DEFINE ADC_SAMPLEUS 50   ' Set sampling time in
                         microseconds
```

ADC_SAMPLEUS is the number of microseconds the program waits between setting the *Channel* and starting the analog to digital conversion. This is the sampling time.

```
TRISA = 255          ' Set PORTA to all input
ADCON1 = 0           ' PORTA is analog
ADCIN 0, B0          ' Read channel 0 to B0
```

5.3. ARRAYREAD

ARRAYREAD *ArrayVar*,{*Maxlength*,*Label*,}**[***Item...***]**

Read one or more *Item*s from byte array *ArrayVar* using **SERIN2** modifiers. **ARRAYREAD** can be used to scan a byte array for values and other strings and move these elements to other variables.

An optional *Maxlength* and *Label* may be included to allow the program to limit the number of characters read from the array so as not to read past its allocated length. *Maxlength* must be less than 256, though longer arrays may be read using either multiple **ARRAYREAD**s or by leaving off this optional parameter. If the count exceeds the *Maxlength*, the program will exit the **ARRAYREAD** command and jump to *Label*.

ARRAYREAD supports many different data modifiers which may be mixed and matched freely within a single **ARRAYREAD** statement to provide various input formatting. These are the same modifiers used in **DEBUGIN**, **HSERIN** and **SERIN2**.

Modifier	Operation
BIN{1..32}	Read binary digits
DEC{1..10}	Read decimal digits
HEX{1..8}	Read upper case hexadecimal digits
SKIP *n*	Skip *n* characters
STR ArrayVar2*n*{*c*}	Read string of *n* characters optionally ended in character *c*
WAIT ( )	Compare sequence of characters
WAITSTR ArrayVar2{*n*}	Compare for character string

Notes:

1) A variable preceded by **BIN** will read the ASCII representation of its binary value. For example, if **BIN B0** is specified and "1000" is read, B0 will be set to 8.
2) A variable preceded by **DEC** will read the ASCII representation of its decimal value. For example, if **DEC B0** is specified and "123" is read, B0 will be set to 123.
3) A variable preceded by **HEX** will read the ASCII representation of

its hexadecimal value. For example, if **HEX B0** is specified and "FE" is read, B0 will be set to 254.

4) **SKIP** followed by a count less than 256, will skip that many characters in the array. For example, **SKIP 4** will skip 4 characters.

5) **STR** followed by a byte array variable, count and optional ending character will read a string of characters. The length is determined by the count, less than 256, or when the optional character is encountered in the input.

6) The list of data items to be read may be preceded by one or more qualifiers between parenthesis after **WAIT**. **ARRAYREAD** must read these bytes in exact order before reading the data items. If any byte read does not match the next byte in the qualifier sequence, the qualification process starts over (i.e. the next read byte is compared to the first item in the qualifier list). A *Qualifier* can be a constant, variable or a string constant. Each character of a string is treated as an individual qualifier.

7) **WAITSTR** can be used as **WAIT** above to force **ARRAYREAD** to compare a string of characters of an optional length, less than 256, before proceeding.

Once any **WAIT** or **WAITSTR** comparisons are satisfied, **ARRAYREAD** begins storing data in the variables associated with each *Item*. If the variable name is used alone, the value of the read ASCII character is stored in the variable. If variable is preceded by **BIN**, **DEC** or **HEX**, then **ARRAYREAD** converts a binary, decimal or hexadecimal value in ASCII and stores the result in that variable. All non-digits read prior to the first digit of the decimal value are ignored and discarded. The non-digit character which terminates the value is also discarded.

BIN, **DEC** and **HEX** may be followed by a number. Normally, these modifiers read as many digits as are in the input. However, if a number follows the modifier, **ARRAYREAD** will always read that number of digits, skipping additional digits as necessary.

ARRAYREAD is not supported on 12-bit core PIC MCUs due to RAM and stack constraints.

```
' Get first 2 bytes from array A
ARRAYREAD A,[B0,B1]

' Skip 2 chars and grab a 4 digit decimal number
ARRAYREAD A,[SKIP 2,DEC4 W0]
```

```
' Find "x" in array A and then read a string
ARRAYREAD A,20,tlabel,[WAIT("x"),STR ar\10]
```

5.4. ARRAYWRITE

ARRAYWRITE *ArrayVar*,{*Maxlength*,*Label*,}**[***Item...***]**

Write one or more *Item*s to byte array *ArrayVar* using **SEROUT2** modifiers. **ARRAYWRITE** allows the writing of formatted data to a byte array which can then be output by other compiler commands, such as **I2CWRITE**, to write to a serial LCD, for example.

An optional *Maxlength* and *Label* may be included to allow the program to limit the number of characters written to the array so as not to exceed its allocated length. *Maxlength* must be less than 256, though longer arrays may be written using either multiple **ARRAYWRITE**s or by leaving off this optional parameter. If the count exceeds the *Maxlength*, the program will exit the **ARRAYWRITE** command and jump to *Label*.

ARRAYWRITE supports many different data modifiers which may be mixed and matched freely within a single **ARRAYWRITE** statement to provide various string formatting. These are the same modifiers used in **DEBUG**, **HSEROUT**, **LCDOUT** and **SEROUT2**.

Modifier	Operation
{I}{S}BIN{1..32}	Write binary digits
{I}{S}DEC{1..10}	Write decimal digits
{I}{S}HEX{1..8}	Write hexadecimal digits
REP *cn***	Write character *c* repeated *n* times
STR ArrayVar2{*n*}	Write string of *n* characters

Notes:

1) A string constant is written as a literal string of characters, for example “Hello”.
2) A numeric value preceded by **BIN** will write the ASCII representation of its binary value. For example, if **B0** = 8, then **BIN B0** (or **BIN 8**) will write “1000".
3) A numeric value preceded by **DEC** will write the ASCII representation of its decimal value. For example, if **B0** = 123, then **DEC B0** (or **DEC 123**) will write “123".
4) A numeric value preceded by **HEX** will write the ASCII representation of its hexadecimal value. For example, if **B0** = 254, then **HEX B0** (or **HEX 254**) will write “FE”.

5) **REP** followed by a character and a count less than 256, will repeat the character, count times. For example, **REP "0"\4** will write "0000".
6) **STR** followed by a byte array variable and optional count will write a string of characters. The length is determined by the count, less than 256, or when 0 is encountered in the string.

BIN, **DEC** and **HEX** may be preceded or followed by several optional parameters. If any of them are preceded by an **I** (for indicated), the values written will be preceded by either a "**%**", "**#**" or "**$**" to indicate the following value is binary, decimal or hexadecimal.

If any are preceded by an **S** (for signed), the values will be written preceded by a "**-**" if the high order bit of the data is set. Keep in mind that all of the math and comparisons in PBP and PBPW are unsigned (PBPL is signed). However, unsigned math can yield signed results. For example, take the case of **B0 = 9 - 10**. The result of **DEC B0** would be "255". Writing **SDEC B0** would give "-1" since the high order bit is set. So with a little trickery, the unsigned math of PBP can yield signed results.

BIN, **DEC** and **HEX** may also be followed by a number. Normally, these modifiers write exactly as many digits as are necessary, zero blanked (leading zeros are not sent). However, if a number follows the modifier, **ARRAYWRITE** will always write that number of digits, adding leading zeros as necessary. It will also trim off any extra high order digits. For example, **BIN6 8** would write "001000" and **BIN2 8** would write "00".

Any or all of the modifier combinations may be used at once. For example, **ISDEC4 B0.**

ARRAYWRITE is not supported on 12-bit core PIC MCUs due to RAM and stack constraints.

```
' Send the ASCII value of B0 followed by a
period and 2 digit ASCII value of B1
ARRAYWRITE A,[DEC B0,".",DEC2 B1]

' Send "B0 =" followed by the binary value of B0
ARRAYWRITE A,20,tlabel,["B0=",IBIN8 B0]
```

5.5. ASM..ENDASM

ASM
ENDASM

The **ASM** and **ENDASM** instructions tells PBP that the code between these two lines is in assembly language and should not be interpreted as PICBASIC PRO statements. You can use these two instructions to mix assembly language code with BASIC statements.

The maximum size for an assembler text section is 8K characters. This is the maximum size for the actual source, including comments, not the generated code. If the text block is larger than this, you must break it into multiple **ASM..ENDASM** sections or simply include it in a separate file.

ASM resets the register page to 0. You must ensure that the register page is reset to 0 before **ENDASM**, if the assembly language code has altered it.

ENDASM must not appear in a comment in the assembly language section of the program. As the compiler cannot discern what is happening in the assembly section, an **ENDASM** anywhere in an **ASM** section will cause the compiler to revert to BASIC parsing.

See the section on assembly language programming for more information.

```
ASM
      bsf PORTA, 0       ; Set bit 0 on PORTA
      bcf PORTB, 0       ; Clear bit 0 on PORTB
ENDASM
```

5.6. BRANCH

BRANCH *Index*,[*Label*{,*Label...*}]

BRANCH causes the program to jump to a different location based on a variable index. Also see **BRANCHL** and **ON GOTO**.

Index selects one of a list of *Label*s. Execution resumes at the indexed *Label*. For example, if *Index* is zero, the program jumps to the first *Label* specified in the list, if *Index* is one, the program jumps to the second *Label*, and so on. If *Index* is greater than or equal to the number of *Label*s, no action is taken and execution continues with the statement following the **BRANCH**. Up to 255 (256 for PIC18) *Label*s may be used in a **BRANCH**.

For 12- and 14-bit core and PIC17 devices, *Label* must be in the same code page as the **BRANCH** instruction. If you cannot be sure they will be in the same code page, use **BRANCHL** below.

For PIC18 devices, the *Label* must be within 1K of the **BRANCH** instruction as it uses a relative jump. If the *Label* is out of this range, use **BRANCHL**.

```
BRANCH B4,[dog,cat,fish]
' Same as:
'  If B4=0 Then dog (goto dog)
'  If B4=1 Then cat (goto cat)
'  If B4=2 Then fish (goto fish)
```

5.7. BRANCHL

BRANCHL *Index*,[*Label*{,*Label...*}]

BRANCHL (**BRANCH** long) works very similarly to **BRANCH** in that it causes the program to jump to a different location based on a variable index. The main difference is that it can jump to a *Label* that is in a different code page than the **BRANCHL** instruction for 12- and 14-bit core and PIC17 devices, or further away than 1K for PIC18 devices. It also generates code that is about twice the size as code generated by the **BRANCH** instruction. If you are sure the labels are in the same page as the **BRANCH** instruction or if the microcontroller does not have more than one code page, using **BRANCH** instead of **BRANCHL** will minimize memory usage. **BRANCHL** is a different syntax of **ON GOTO**.

Index selects one of a list of *Label*s. Execution resumes at the indexed *Label*. For example, if *Index* is zero, the program jumps to the first *Label* specified in the list, if *Index* is one, the program jumps to the second *Label*, and so on. If *Index* is greater than or equal to the number of *Label*s, no action is taken and execution continues with the statement following the **BRANCHL**. Up to 127 (1024 for PIC18) *Label*s may be used in a **BRANCHL**.

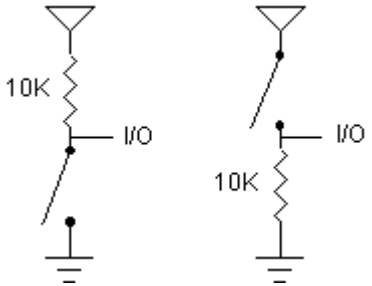
```
BRANCHL B4,[dog,cat,fish]
' Same as:
'  If B4=0 Then dog (goto dog)
'  If B4=1 Then cat (goto cat)
'  If B4=2 Then fish (goto fish)
```

5.8. BUTTON

BUTTON *Pin,Down,Delay,Rate,BVar,Action,Label*

Read *Pin* and optionally perform debounce and auto-repeat. *Pin* is automatically made an input. *Pin* may be a constant, 0-15, or a variable that contains a number 0-15 (e.g. B0) or a pin name (e.g. PORTA.0).

Down	State of pin when button is pressed (0..1).
Delay	Cycle count before auto-repeat starts (0..255). If 0, no debounce or auto-repeat is performed. If 255, debounce, but no auto-repeat, is performed.
Rate	Auto-repeat rate (0..255).
BVar	Byte-sized variable used internally for delay/repeat countdown. It must be initialized to 0 prior to use and not used elsewhere in the program.
Action	State of button to act on (0 if not pressed, 1 if pressed).
Label	Execution resumes at this label if *Action* is true.




```
' Goto notpressed if button not pressed on Pin2
BUTTON PORTB.2,0,100,10,B2,0,notpressed
```

BUTTON needs to be used within a loop for auto-repeat to work properly.

BUTTON accomplishes debounce by delaying program execution for a period of milliseconds to wait for the contacts to settle down. The default debounce delay is 10ms. To change the debounce to another value, use **DEFINE**:

```
' Set button debounce delay to 50ms
DEFINE BUTTON_PAUSE 50
```

Be sure that BUTTON_PAUSE is all in upper case. The maximum delay for 12-bit core devices is 65ms.

In general, it is easier to simply read the state of the pin in an **IF..THEN** than to use the **BUTTON** command as follows:

```
      If PORTB.2 = 1 Then notpressed
```

Example program:

```
INCLUDE "modedefs.bas"  ' Include serial modes

SO    Con   0           ' Define serial output pin
Bpin  Con   2           ' Define Button input pin
B0    Var   Byte

      B0 = 0            ' Zero Button working buffer

mainloop: BUTTON Bpin,1,10,5,B0,0,notp    ' Check
                                          button (skip
                                          if not
                                          pressed)
      Serout SO,N2400,["Press",13,10]     ' Indicate
                                          button
                                          pressed
notp: Serout SO,N2400,[#B0,13,10]    ' Show working
                                     variable

      Pause 100         ' Wait a little
      Goto mainloop     ' Do it forever
```

5.9. CALL

CALL *Label*

Execute the assembly language subroutine named *_Label*.

GOSUB is normally used to execute a PICBASIC PRO subroutine. The main difference between **GOSUB** and **CALL** is that with **CALL**, *Label*'s existence is not checked until assembly time. A *Label* in an assembly language section can be accessed using **CALL** that is otherwise inaccessible to PBP. The assembly *Label* should be preceeded by an underscore (_).

See the section on assembly language programming for more information on **CALL**.

```
CALL pass    ' Execute assembly language
             subroutine named _pass
```

5.10. CLEAR

CLEAR

Set all RAM registers to zero.

CLEAR zeroes all the RAM registers in each bank. This will set all variables, including the internal system variables (but not PIC MCU hardware registers) to zero. This is not automatically done when a PBP program starts as it is on a BASIC Stamp. In general, the variables should be initialized in the program to an appropriate state rather than using **CLEAR**.

CLEAR does not zero bank 0 registers on 12-bit core devices.

```
CLEAR          ' Clear all variables to 0
```

5.11. CLEARWDT

CLEARWDT

Clear (tickle) the Watchdog Timer.

The Watchdog Timer is used in conjunction with the **SLEEP** and **NAP** instructions to wake the PIC MCU after a certain period of time. Assembler instructions (clrwdt) to keep the Watchdog Timer from timing out under normal circumstances and resetting the PIC MCU are automatically inserted at appropriate places throughout the program.

CLEARWDT allows the placement of additional clrwdt instructions in the program.

```
CLEARWDT            ' Clear Watchdog Timer
```

A **DEFINE** can be used to remove all of the clrwdt instructions the compiler automatically adds. In many cases, the clrwdt instruction is replaced with nop to preserve the timing of the routine.

```
DEFINE NO_CLRWDT 1      ' Don't insert CLRWDTs
```

5.12. COUNT

COUNT *Pin,Period,Var*

Count the number of pulses that occur on *Pin* during the *Period* and stores the result in *Var*. *Pin* is automatically made an input. *Pin* may be a constant, 0-15, or a variable that contains a number 0-15 (e.g. B0) or a pin name (e.g. PORTA.0).

The resolution of *Period* is in milliseconds. It tracks the oscillator frequency based on the **DEFINE**d OSC.

COUNT checks the state of *Pin* in a tight loop and counts the low to high transitions. With a 4MHz oscillator it checks the pin state every 20us. With a 20MHz oscillator it checks the pin state every 4us. From this, it can be determined that the highest frequency of pulses that can be counted is 25KHz with a 4MHz oscillator and 125KHz with a 20MHz oscillator, if the frequency has a 50% duty cycle (the high time is the same as the low time).

```
' Count # of pulses on Pin1 in 100 milliseconds
COUNT PORTB.1,100,W1

' Determine frequency on a pin
COUNT PORTA.2, 1000, W1 ' Count for 1 second
Serout PORTB.0,N2400,[W1]
```

5.13. DATA

{*Label*} **DATA** {@*Location*,}*Constant*{,*Constant...*}

Store constants in on-chip non-volatile EEPROM when the device is first programmed. If the optional *Location* value is omitted, the first **DATA** statement starts storing at address 0 and subsequent statements store at the following locations. If the *Location* value is specified, it denotes the starting location where these values are stored. An optional *Label* (not followed by a colon) can be assigned to the starting EEPROM address for later reference by the program.

Constant can be a numeric constant or a string constant. Only the least significant byte of numeric values are stored unless the **WORD** or **LONG** (PBPL only) modifier is used. Strings are stored as consecutive bytes of ASCII values. No length or terminator is automatically added.

DATA only works with microcontrollers with on-chip EEPROM such as the PIC12F683, PIC16F84 and the PIC16F87x(A) series. It will not work on devices with on-chip I2C interfaced serial EEPROM like the PIC12CE67x and PIC16CE62x parts. Since EEPROM is non-volatile memory, the data will remain intact even if the power is turned off.

The data is stored in the EEPROM space only once at the time the microcontroller is programmed, not each time the program is run. **WRITE** can be used to set the values of the on-chip EEPROM at runtime. **READ** is used to retrieve these stored **DATA** values at runtime.

```
      ' Store 10, 20 and 30 starting at location 4
      DATA @4,10,20,30

      ' Assign a label to a word at the next location
dlabel DATA word $1234  ' Stores $34, $12

      ' Assign a label to a long at the next location
llabel DATA long $12345678     ' Stores $78, $56, $34,
                               $12 (PBPL only)

      ' Skip 4 locations and store 10 0s
      DATA (4),0(10)
```

5.14. DEBUG

DEBUG *Item*{,*Item*...}

Send one or more *Item*s on a predefined pin at a predefined baud rate in standard asynchronous format using 8 data bits, no parity and 1 stop bit (8N1). The pin is automatically made an output.

If a pound sign (**#**) precedes an *Item*, the ASCII representation for each digit is sent serially. **DEBUG** (on all devices except 12-bit core) also supports the same data modifiers as **SEROUT2**. Refer to the section on **SEROUT2** for this information.

Modifier	Operation
{I}{S}BIN{1..32}	Send binary digits
{I}{S}DEC{1..10}	Send decimal digits
{I}{S}HEX{1..8}	Send hexadecimal digits
REP *cn***	Send character *c* repeated *n* times
STR ArrayVar{*n*}	Send string of *n* characters

DEBUG is one of several built-in asynchronous serial functions. It is the smallest and fastest of the software generated serial routines. It can be used to send debugging information (variables, program position markers, etc.) to a terminal program like Hyperterm. It can also be used anytime serial output is desired on a fixed pin at a fixed baud rate.

The serial pin and baud rate are specified using **DEFINE**s:

```
' Set Debug pin port
DEFINE DEBUG_REG  PORTB

' Set Debug pin bit
DEFINE DEBUG_BIT  0

' Set Debug baud rate
DEFINE DEBUG_BAUD 2400


' Set Debug mode: 0 = true, 1 = inverted
DEFINE DEBUG_MODE 1
```

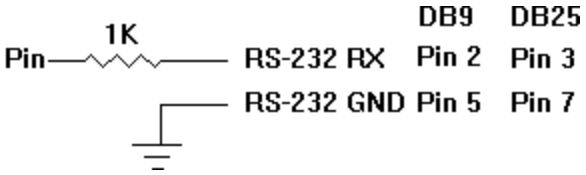
DEBUG assumes a 4MHz oscillator when generating its bit timing. To maintain the proper baud rate timing with other oscillator values, be sure to **DEFINE** the OSC setting to any different oscillator value.

In some cases, the transmission rates of **DEBUG** instructions may present characters too quickly to the receiving device. A **DEFINE** adds character pacing to the serial output transmissions. This allows additional time between the characters as they are transmitted. The character pacing **DEFINE** allows a delay of up to 65,535 microseconds (65.535 milliseconds) between each character transmitted.

For example, to pause 1 millisecond between the transmission of each character:

```
DEFINE DEBUG_PACING 1000
```

While single-chip RS-232 level converters are common and inexpensive, thanks to current RS-232 implementation and the excellent I/O specifications of the PIC MCU, most applications may not require level converters. Rather, inverted TTL (DEBUG_MODE 1) may be used. A current limiting resistor is suggested (RS-232 is suppose to be short-tolerant).




```
' Send the text "B0=" followed by the decimal
value of B0 and a linefeed out serially
DEBUG "B0=",DEC B0,10
```

5.15. DEBUGIN

DEBUGIN {*Timeout*,*Label*,}[*Item*{,*Item*...}]

Receive one or more *Item*s on a predefined pin at a predefined baud rate in standard asynchronous format using 8 data bits, no parity and 1 stop bit (8N1). The pin is automatically made an input.

An optional *Timeout* and *Label* may be included to allow the program to continue if a character is not received within a certain amount of time. *Timeout* is specified in 1 millisecond units. If the serial input pin stays idle during the *Timeout* time, the program will exit the **DEBUGIN** command and jump to *Label*.

DEBUGIN (on all devices except 12-bit core) supports the same data modifiers as **SERIN2**. Refer to the section on **SERIN2** for this information.

Modifier	Operation
BIN{1..32}	Receive binary digits
DEC{1..10}	Receive decimal digits
HEX{1..8}	Receive upper case hexadecimal digits
SKIP *n*	Skip *n* received characters
STR ArrayVar*n*{*c*}	Receive string of *n* characters optionally ended in character *c*
WAIT ( )	Wait for sequence of characters
WAITSTR ArrayVar{*n*}	Wait for character string

DEBUGIN is one of several built-in asynchronous serial functions. It is the smallest and fastest of the software generated serial routines. It can be used to receive debugging information from a terminal program like Hyperterm. It can also be used anytime serial input is desired on a fixed pin at a fixed baud rate.

The serial pin and baud rate are specified using **DEFINE**s:

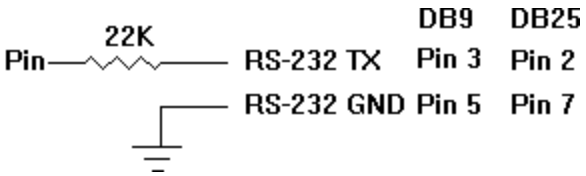
```
' Set Debugin pin port
DEFINE DEBUGIN_REG      PORTB

' Set Debugin pin bit
```

```
DEFINE DEBUGIN_BIT      0

' Set Debugin baud rate (same as Debug baud)
DEFINE DEBUG_BAUD       2400

' Set Debugin mode: 0 = true, 1 = inverted
DEFINE DEBUGIN_MODE     1
```

If any of these **DEFINE**s are not included in a program, the **DEBUGIN** port, pin or mode is set to the same values as they are for **DEBUG**. The **DEBUGIN** baud rate is always the same as **DEBUG**'s. It cannot be **DEFINE**d differently.

DEBUGIN assumes a 4MHz oscillator when generating its bit timing. To maintain the proper baud rate timing with other oscillator values, be sure to **DEFINE** the OSC setting to any different oscillator value.

While single-chip RS-232 level converters are common and inexpensive, thanks to current RS-232 implementation and the excellent I/O specifications of the PIC MCU, most applications may not require level converters. Rather, inverted TTL (DEBUGIN_MODE 1) may be used. A current limiting resistor is necessary to dissipate the higher and sometimes negative RS-232 voltage.




```
' Wait until the character "A" is received
serially and put next character into B0
DEBUGIN [WAIT("A"),B0]

' Skip 2 chars and grab a 4 digit decimal number
DEBUGIN [SKIP 2,DEC4 B0]

' Wait for value with timeout
DEBUGIN 100, timesup, [B0]
```

5.16. DISABLE

DISABLE

DISABLE both debug and interrupt processing following this instruction. Interrupts can still occur but the BASIC interrupt handler in the PICBASIC PRO program and the debug monitor will not be executed until an **ENABLE** is encountered.

DISABLE and **ENABLE** are pseudo-ops in that they give the compiler directions, rather than actually generate code. See **ON DEBUG** and **ON INTERRUPT** for more information.

```
      DISABLE       ' Disable interrupts in handler
myint: led = 1      ' Turn on LED when interrupted
      Resume        ' Return to main program
      Enable        ' Enable interrupts after handler
```

5.17. DISABLE DEBUG

```
DISABLE DEBUG
```

DISABLE DEBUG processing following this instruction. The debug monitor will not be called between instructions until an **ENABLE** or **ENABLE DEBUG** is encountered.

DISABLE DEBUG and **ENABLE DEBUG** are pseudo-ops in that they give the compiler directions, rather than actually generate code. See **ON DEBUG** for more information.

```
DISABLE DEBUG      ' Disable debug monitor calls
```

5.18. DISABLE INTERRUPT

DISABLE INTERRUPT

DISABLE INTERRUPT processing following this instruction. Interrupts can still occur but the BASIC interrupt handler in the PICBASIC PRO program will not be executed until an **ENABLE** or **ENABLE INTERRUPT** is encountered.

DISABLE INTERRUPT and **ENABLE INTERRUPT** are pseudo-ops in that they give the compiler directions, rather than actually generate code. See **ON INTERRUPT** for more information.

```
      DISABLE INTERRUPT ' Disable interrupts in
                        handler
myint: led = 1      ' Turn on LED when interrupted
      Resume        ' Return to main program
      Enable Interrupt  ' Enable interrupts after
                        handler
```

5.19. DO..LOOP

```
DO {UNTIL Condition} {WHILE Condition}
      Statement...
LOOP {UNTIL Condition} {WHILE Condition}
```

Repeatedly execute *Statement*s in a loop. An optional **UNTIL** or **WHILE** may be added to check a *Condition*.

The addition of **DO..LOOP** to the compiler means that "loop" can no longer be used as a label. If "loop" is used as a label, the compiler will issue an error. These "loop" labels will need to be changed to some other, hopefully more meaningful, name such as "Mainloop", "Buttonloop" or "Waitloop", for examples. Any **GOTO**s or other instructions that reference a "loop" label will, of course, need to be changed as well.

```
DO
      Pwm PORTC.2,127,100
LOOP

i = 1
DO UNTIL i > 10
      Serout 0,N2400,["No:",#i,13,10]
      i = i + 1
LOOP

i = 1
DO
      Serout 0,N2400,["No:",#i,13,10]
      i = i + 1
LOOP UNTIL i > 10

i = 1
DO WHILE i <= 10
      Serout 0,N2400,["No:",#i,13,10]
      i = i + 1
LOOP

i = 1
DO
      Serout 0,N2400,["No:",#i,13,10]
      i = i + 1
LOOP WHILE i <= 10
```

5.20. DTMFOUT

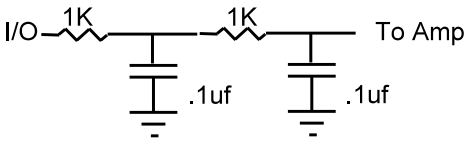
DTMFOUT *Pin*,{*Onms*,*Offms*,}**[***Tone*{,*Tone*...}**]**

Produce DTMF touch *Tone* sequence on *Pin*. *Pin* is automatically made an output. *Pin* may be a constant, 0-15, or a variable that contains a number 0-15 (e.g. B0) or a pin name (e.g. PORTA.0).

Onms is the number of milliseconds to sound each tone and *Offms* is the number of milliseconds to pause between each tone. If they are not specified, *Onms* defaults to 200ms and *Offms* defaults to 50ms.

Tones are numbered 0-15. *Tones* 0-9 are the same as on a telephone keypad. *Tone* 10 is the ***** key, *Tone* 11 is the **#** key and *Tones* 12-15 correspond to the extended keys **A**-**D**.

DTMFOUT uses **FREQOUT** to generate the dual tones. **FREQOUT** generates tones using a form of pulse width modulation. The raw data coming out of the pin looks pretty scary. Some kind of filter is usually necessary to smooth the signal to a sine wave and get rid of some of the harmonics that are generated:



DTMFOUT works best with a 20MHz or 40MHz oscillator. It can also work with a 10MHz or 8MHz oscillator and even at 4MHz, although it will start to get very hard to filter and be of fairly low amplitude. Any other frequency may not be used with **DTMFOUT**.

DTMFOUT is not supported on 12-bit core PIC MCUs due to RAM and stack constraints.

```
' Send DTMF tones for 212 on Pin1
DTMFOUT PORTB.1,[2,1,2]
```

5.21. EEPROM

EEPROM {*Location*,}**[***Constant*{,*Constant*...}**]**

Store constants in on-chip EEPROM. If the optional *Location* value is omitted, the first **EEPROM** statement starts storing at address 0 and subsequent statements store at the following locations. If the *Location* value is specified, it denotes the starting location where these values are stored.

Constant can be a numeric constant or a string constant. Only the least significant byte of numeric values are stored. Strings are stored as consecutive bytes of ASCII values. No length or terminator is automatically added.

EEPROM only works with microcontrollers with on-chip EEPROM such as the PIC12F683, PIC16F84 and the PIC16F87x(A) series. It will not work on devices with on-chip I2C interfaced serial EEPROM like the PIC12CE67x and PIC16CE62x parts. Since EEPROM is non-volatile memory, the data will remain intact even if the power is turned off.

The data is stored in the EEPROM space only once at the time the microcontroller is programmed, not each time the program is run. **WRITE** can be used to set the values of the on-chip EEPROM at runtime. **READ** is used to retrieve these stored **DATA** values at runtime.

```
' Store 10, 20 and 30 starting at location 4
EEPROM 4,[10,20,30]
```

5.22. ENABLE

ENABLE

ENABLE debug and interrupt processing that was previously **DISABLE**d following this instruction.

DISABLE and **ENABLE** are pseudo-ops in that they give the compiler directions, rather than actually generate code. See **ON DEBUG** and **ON INTERRUPT** for more information.

```
      Disable      ' Disable interrupts in handler
myint: led = 1     ' Turn on LED when interrupted
      Resume       ' Return to main program
      ENABLE       ' Enable interrupts after handler
```

5.23. ENABLE DEBUG

ENABLE DEBUG

ENABLE DEBUG processing that was previously **DISABLE**d following this instruction.

DISABLE DEBUG and **ENABLE DEBUG** are pseudo-ops in that they give the compiler directions, rather than actually generate code. See **ON DEBUG** for more information.

```
ENABLE DEBUG        ' Enable calls to the debug
                    monitor
```

5.24. ENABLE INTERRUPT

ENABLE INTERRUPT

ENABLE INTERRUPT processing that was previously **DISABLE**d following this instruction.

DISABLE INTERRUPT and **ENABLE INTERRUPT** are pseudo-ops in that they give the compiler directions, rather than actually generate code. See **ON INTERRUPT** for more information.

```
      Disable Interrupt ' Disable interrupts in
                        handler
myint: led = 1     ' Turn on LED when interrupted
      Resume       ' Return to main program
      ENABLE INTERRUPT  ' Enable interrupts after
                        handler
```

5.25. END

END

Stop program execution and enter low power mode. All of the I/O pins remain in their current state. **END** works by executing a Sleep instruction continuously in a loop.

An **END** or **STOP** or **GOTO** should be placed at the end of every program to keep it from falling off the end of memory and starting over.

```
END
```

5.26. ERASECODE

ERASECODE *Block*

Some flash PIC MCUs, like the PIC18F series, require a portion of the code space to be erased before it can be rewritten with **WRITECODE**. On these devices, an erase is performed a block at a time. An erase block may be 32 words (64 bytes) or another size, depending on the device. This size is usually larger than the write block size. See the Microchip data sheet for information on the size of the erase block for the particular PIC MCU you are using.

The first location of the block to be erased is specified by *Block*. For PIC18F devices, *Block* is a byte address rather than a word address. Be careful not to specify a *Block* that contains program code.

For 12-bit core devices that support flash data memory, like the PIC12F519 and PIC16F526, **ERASECODE** must be used to erase the rows of memory before it can be rewritten using **WRITE**.

If only a portion of a block is to be changed, for example only the first byte in the block, the entire block must be read before it is erased and all of the data rewritten, including the new data and the original data that needs to be preserved.

Flash program writes must be enabled in the configuration for the PIC MCU at device programming time for **ERASECODE** to be able to erase.

Using this instruction on devices that do not support block erase may cause a compilation error.

```
ERASECODE $1000     ' Erase code block starting at
                    location $1000
```

5.27. EXIT

EXIT

EXIT jumps to the line after the end of the current block structure and continues program execution from there. This allows the current **FOR..NEXT** or other loop construct to be left without having to satisfy a specific condition. As none of the block structures in PBP use the stack, there is no problem with the stack and **EXIT**.

```
      For x = 0 To 255
            If x = 27 Then EXIT      ' Same as Goto l
      Next x
l:
```

5.28. FOR..NEXT

```
FOR Count = Start TO End {STEP {-} Inc}
      {Body}
NEXT {Count}
```

The **FOR..NEXT** loop allows programs to execute a number of statements (the *Body*) some number of times using a variable as a counter. Due to its complexity and versatility, **FOR..NEXT** is best described step by step:

1) The value of *Start* is assigned to the index variable, *Count*. *Count* can be a variable of any type.
2) The *Body* is executed. The *Body* is optional and can be omitted (perhaps for a delay loop).
3) The value of *Inc* is added to (or subtracted from if "-" is specified) *Count*. If no **STEP** clause is defined, *Count* is incremented by one.
4) If *Count* has not passed *End* or overflowed the variable type, execution returns to Step 2.

If the loop needs to *Count* to more than 255, a word- or long-sized (PBPL only) variable must be used.

```
FOR i = 1 TO 10           ' Count from 1 to 10
Serout 0,N2400,[#i," "]   ' Send each number to
                          Pin0 serially
NEXT i                    ' Go back to and do next
                          count
Serout 0,N2400,[10]       ' Send a linefeed
```

```
FOR B2 = 20 TO 10 STEP -2      ' Count from 20 to
                               10 by 2
Serout 0,N2400,[#B2," "]       ' Send each number
                               to Pin0 serially
NEXT B2                   ' Go back to and do next
                          count
Serout 0,N2400,[10]       ' Send a linefeed
```

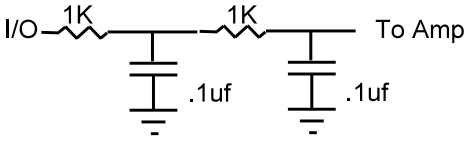
5.29. FREQOUT

FREQOUT *Pin*,*Onms*,*Frequency1*{,*Frequency2*}

Produce the *Frequency*(s) on *Pin* for *Onms* milliseconds. *Pin* is automatically made an output. *Pin* may be a constant, 0-15, or a variable that contains a number 0-15 (e.g. B0) or a pin name (e.g. PORTA.0).

One or two different frequencies from 0 to 32767 Hertz may be produced at a time.

FREQOUT generates tones using a form of pulse width modulation. The raw data coming out of the pin looks pretty scary. Some kind of filter is usually necessary to smooth the signal to a sine wave and get rid of some of the harmonics that are generated:



FREQOUT works best with a 20MHz or 40MHz oscillator. It can also work with a 10MHz or 8MHz oscillator and even at 4MHz, although it will start to get very hard to filter and be of fairly low amplitude. Any other frequency will cause **FREQOUT** to generate a frequency that is a ratio of the actual oscillator used and 20MHz.

FREQOUT is not supported on 12-bit core PIC MCUs due to RAM and stack constraints.

```
' Send 1KHz tone on Pin1 for 2 seconds
FREQOUT PORTB.1,2000,1000

' Send 350Hz / 440Hz (Dial Tone) for 2 seconds
FREQOUT PORTB.1,2000,350,440
```

5.30. GOSUB

GOSUB *Label*

Jump to the subroutine at *Label* saving its return address on the stack. Unlike **GOTO**, when a **RETURN** statement is reached after executing a **GOSUB**, execution resumes with the statement following that last executed **GOSUB** statement.

An unlimited number of subroutines may be used in a program. Subroutines may also be nested. In other words, it is possible for a subroutine to **GOSUB** to another subroutine. Such subroutine nesting must be restricted to no more than four nested levels for 12- and 14-bit core devices, 12 levels for 14-bit enhanced core and PIC17 parts and 27 levels for PIC18 parts. Interrupts cause additional locations to be used on the stack , reducing the number of possible nested **GOSUB**s. See the section on interrupts later in the manual for more information.

```
      GOSUB beep   ' Execute subroutine named beep
      ...
beep: High 0       ' Turn on LED connected to Pin0
      Sound 1,[80,10]    ' Beep speaker connected to
                         Pin1
      Low 0        ' Turn off LED connected to Pin0
      Return       ' Go back to main routine that
                   called us
```

5.31. GOTO

GOTO *Label*

Program execution continues with the statements at *Label*.

```
      GOTO send    ' Jump to statement labeled send
      ...
send: Serout 0,N2400,["Hi"]    ' Send "Hi" out Pin0
                               serially
```

5.32. HIGH

HIGH *Pin*

Make the specified *Pin* high. *Pin* is automatically made an output. *Pin* may be a constant, 0-15, or a variable that contains a number 0-15 (e.g. B0) or a pin name (e.g. PORTA.0).

```
      HIGH 0              ' Make Pin0 an output and set
                          it high (~5 volts)
      HIGH PORTA.0        ' Make PORTA, pin 0 an output
                          and set it high (~5 volts)

led   Var   PORTB.0       ' Define LED pin
      HIGH led            ' Make LED pin an output and
                          set it high (~5 volts)
```

Alternatively, if the pin is already an output, a much quicker and shorter way (from a generated code standpoint) to set it high would be:

```
      PORTB.0 = 1         ' Set PORTB pin 0 high
```

5.33. HPWM

HPWM *Channel,Dutycycle,Frequency*

Output a pulse width modulated pulse train using PWM hardware available on some PIC MCUs. It can run continuously in the background while the program is executing other instructions.

Channel specifies which hardware PWM channel to use. Some devices have between 1 and 5 PWM *Channel*s that can be used with **HPWM**. The Microchip data sheet for the particular device shows the fixed hardware pin for each *Channel*. For example, for a PIC16F877a, *Channel* 1 is CCP1 which is pin PORTC.2. *Channel* 2 is CCP2 which is pin PORTC.1.

Some devices, such as the PIC18F452, have alternate pins that may be used for **HPWM**. The following **DEFINE**s allow using these pins:

```
DEFINE CCP1_REG   PORTC        ' Channel 1 port
DEFINE CCP1_BIT   2            ' Channel 1 bit
DEFINE CCP2_REG   PORTC        ' Channel 2 port
DEFINE CCP2_BIT   1            ' Channel 2 bit
DEFINE CCP3_REG   PORTG        ' Channel 3 port
DEFINE CCP3_BIT   0            ' Channel 3 bit
DEFINE CCP4_REG   PORTG        ' Channel 4 port
DEFINE CCP4_BIT   3            ' Channel 4 bit
DEFINE CCP5_REG   PORTG        ' Channel 5 port
DEFINE CCP5_BIT   4            ' Channel 5 bit
```

Dutycycle specifies the on/off (high/low) ratio of the signal. It ranges from 0 to 255, where 0 is off (low all the time) and 255 is on (high) all the time. A value of 127 gives a 50% duty cycle (square wave).

Frequency is the desired frequency of the PWM signal. On devices with 2 channels, the *Frequency* must be the same on both channels. Not all frequencies are available at all oscillator settings. For the non-long versions of PBP (PBP and PBPW), the highest frequency at any oscillator speed is 32767Hz. The lowest usable **HPWM** *Frequency* at each oscillator setting is shown in the following table:

osc	14-bit core and PIC18	PIC17
4MHz	245Hz	3907Hz
8MHz	489Hz	7813Hz
10MHz	611Hz	9766Hz
12MHz	733Hz	11719Hz
16MHz	977Hz	15625Hz
20MHz	1221Hz	19531Hz
24MHz	1465Hz	23437Hz
25MHz	1527Hz	24415Hz
32MHz	1953Hz	31249Hz
33MHz	2015Hz	32227Hz
40MHz	2441Hz	na
48MHz	2929Hz	na
64MHz	3905Hz	na

The following **DEFINE**s specify which timer, 1 or 2, to use with PWM channel 2 and PWM channel 3 for the PIC17C7xx devices. The default is timer 1 if no **DEFINE** is specified.

```
DEFINE HPWM2_TIMER 1     ' Hpwm channel 2 timer
DEFINE HPWM3_TIMER 1     ' Hpwm channel 3 timer
```

After an **HPWM** command, the CCP control register is left in PWM mode. If the CCP pin is to be used as a normal I/O pin after an **HPWM** command, the CCP control register will need to be set to PWM off. See the Microchip data sheet for the particular device for more information.

```
HPWM 1,127,1000     ' Send a 50% duty cycle PWM
                    signal at 1kHz

HPWM 1,64,2000      ' Send a 25% duty cycle PWM
                    signal at 2kHz
```

5.34. HSERIN

HSERIN {*ParityLabel*,}{*Timeout*,*Label*,}**[***Item*{,...}**]**

Receive one or more *Item*s from the hardware serial port on devices that support asynchronous serial communications in hardware.

HSERIN is one of several built-in asynchronous serial functions. It can only be used with devices that have a hardware USART. See the device data sheet for information on the serial input pin and other parameters. The serial parameters and baud rate are specified using **DEFINE**s:

```
' Set receive register to receiver enabled
DEFINE HSER_RCSTA 90h

' Set transmit register to transmitter enabled
DEFINE HSER_TXSTA 20h

' Set baud rate
DEFINE HSER_BAUD  2400

' Set SPBRG, SPBRGH directly
' (better to set HSER_BAUD instead)
DEFINE HSER_SPBRG 25
DEFINE HSER_SPBRGH      0
```

HSER_RCSTA, HSER_TXSTA, HSER_SPBRG, and HSER_SPBRGH simply set each respective PIC MCU register, RCSTA, TXSTA, SPBRG and SPBRGH to the hexadecimal value **DEFINE**d, once, at the beginning of the program. See the Microchip data sheet for the device for more information on each of these registers.

The TXSTA register BRGH bit (bit 2) controls the high speed mode for the baud rate generator. Certain baud rates at certain oscillator speeds require this bit to be set to operate properly. To do this, set HSER_TXSTA to 24h instead of 20h. All baud rates at all oscillator speeds may not be supported by the device. See the Microchip data sheet for the hardware serial port baud rate tables and additional information.

HSERIN assumes a 4MHz oscillator when calculating the baud rate. To maintain the proper baud rate timing with other oscillator values, be sure to **DEFINE** the OSC setting to the new oscillator value.

An optional *Timeout* and *Label* may be included to allow the program to continue if a character is not received within a certain amount of time.

Timeout is specified in 1 millisecond units. If no character is received during the *Timeout* time, the program will exit the **HSERIN** command and jump to *Label*.

The serial data format defaults to 8N1, 8 data bits, no parity bit and 1 stop bit. 7E1 (7 data bits, even parity, 1 stop bit) or 7O1 (7data bits, odd parity, 1 stop bit) can be enabled using one of the following **DEFINE**s:

```
' Use only if even parity desired
DEFINE HSER_EVEN  1

' Use only if odd parity desired
DEFINE HSER_ODD   1

' Use 8 bits + parity
DEFINE HSER_BITS  9
```

The parity setting, along with all of the other HSER **DEFINE**s, affect both **HSERIN** and **HSEROUT**.

An optional *ParityLabel* may be included in the statement. The program will continue at this location if a character with a parity error is received. It should only be used if parity is enabled using one of the preceding **DEFINE**s.

As the hardware serial port only has a 2 byte input buffer, it can easily overflow if characters are not read from it often enough. When this happens, the USART stops accepting new characters and needs to be reset. This overflow error can be reset by toggling the CREN bit in the RCSTA register. A **DEFINE** can be used to automatically clear this error. However, you will not know that an error has occurred and characters may have been lost.

```
DEFINE HSER_CLROERR     1
```

To manually clear an overrun error:

```
RCSTA.4 = 0
RCSTA.4 = 1
```

Since the serial reception is done in hardware, it is not possible to set the levels to an inverted state to eliminate an RS-232 driver. Therefore a suitable driver should be used with **HSERIN**.

On devices with 2 hardware serial ports, **HSERIN** will only use the first port. The second port may read using **HSERIN2**.

HSERIN supports the same data modifiers that **SERIN2** does. Refer to the section on **SERIN2** for this information.

Modifier	Operation
BIN{1..32}	Receive binary digits
DEC{1..10}	Receive decimal digits
HEX{1..8}	Receive upper case hexadecimal digits
SKIP ***n***	Skip *n* received characters
STR ArrayVar*n*{*c*}	Receive string of *n* characters optionally ended in character *c*
WAIT ( )	Wait for sequence of characters
WAITSTR ArrayVar{*n*}	Wait for character string

```
HSERIN [B0,DEC W1]

HSERIN 100, timesup, [B0]
```

5.35. HSERIN2

HSERIN2 {*ParityLabel*,}{*Timeout*,*Label*,}**[***Item*{,...}**]**

Receive one or more *Item*s from the second hardware serial port on devices that support asynchronous serial communications in hardware.

HSERIN2 works the same as **HSERIN** with the exception that is uses the second hardware serial port on devices that have 2 ports such as the PIC18F8720 It can only be used with devices that have 2 hardware USARTs. See the device data sheet for information on the serial output pin and other parameters and the above section on **HSERIN** for more command details. The serial parameters and baud rate are specified using **DEFINE**s:

```
' Set receive register to receiver enabled
DEFINE HSER2_RCSTA      90h

' Set transmit register to transmitter enabled
DEFINE HSER2_TXSTA      20h

' Set baud rate
DEFINE HSER2_BAUD 2400

' Set SPBRG2, SPBRGH2 directly
' (better to set HSER2_BAUD instead)
DEFINE HSER2_SPBRG      25
DEFINE HSER2_SPBRGH     0

' Use only if even parity desired
DEFINE HSER2_EVEN 1

' Use only if odd parity desired
DEFINE HSER2_ODD  1

' Use 8 bits + parity
DEFINE HSER2_BITS 9

' Automatically clear overflow errors
DEFINE HSER2_CLROERR    1

HSERIN2 [B0,DEC W1]

HSERIN2 100, timesup, [B0]
```

5.36. HSEROUT

HSEROUT [*Item*{,*Item...*}**]**

Send one or more *Item*s to the hardware serial port on devices that support asynchronous serial communications in hardware.

HSEROUT is one of several built-in asynchronous serial functions. It can only be used with devices that have a hardware USART. See the device data sheet for information on the serial output pin and other parameters. The serial parameters and baud rate are specified using **DEFINE**s:

```
' Set receive register to receiver enabled
DEFINE HSER_RCSTA 90h

' Set transmit register to transmitter enabled
DEFINE HSER_TXSTA 20h

' Set baud rate
DEFINE HSER_BAUD  2400

' Set SPBRG, SPBRGH directly
' (better to set HSER_BAUD instead)
DEFINE HSER_SPBRG 25
DEFINE HSER_SPBRGH      0
```

HSER_RCSTA, HSER_TXSTA, HSER_SPBRG, and HSER_SPBRGH simply set each respective PIC MCU register, RCSTA, TXSTA, SPBRG and SPBRGH to the hexadecimal value **DEFINE**d, once, at the beginning of the program. See the Microchip data sheet for the device for more information on each of these registers.

The TXSTA register BRGH bit (bit 2) controls the high speed mode for the baud rate generator. Certain baud rates at certain oscillator speeds require this bit to be set to operate properly. To do this, set HSER_TXSTA to 24h instead of 20h. All baud rates at all oscillator speeds may not be supported by the device. See the Microchip data sheet for the hardware serial port baud rate tables and additional information.

HSEROUT assumes a 4MHz oscillator when calculating the baud rate. To maintain the proper baud rate timing with other oscillator values, be sure to **DEFINE** the OSC setting to the new oscillator value.

The serial data format defaults to 8N1, 8 data bits, no parity bit and 1 stop bit. 7E1 (7 data bits, even parity, 1 stop bit) or 7O1 (7data bits, odd parity,

1 stop bit) can be enabled using one of the following **DEFINE**s:

```
' Use only if even parity desired
DEFINE HSER_EVEN  1

' Use only if odd parity desired
DEFINE HSER_ODD   1

' Use 8 bits + parity
DEFINE HSER_BITS  9
```

The parity setting, along with all of the other HSER **DEFINE**s, affect both **HSERIN** and **HSEROUT**.

Since the serial transmission is done in hardware, it is not possible to set the levels to an inverted state to eliminate an RS-232 driver. Therefore a suitable driver should be used with **HSEROUT**.

On devices with 2 hardware serial ports, **HSEROUT** will only use the first port. The second port may be accessed using **HSEROUT2**.

HSEROUT supports the same data modifiers that **SEROUT2** does. Refer to the section on **SEROUT2** for this information.

Modifier	Operation
{I}{S}BIN{1..32}	Send binary digits
{I}{S}DEC{1..10}	Send decimal digits
{I}{S}HEX{1..8}	Send hexadecimal digits
REP *cn***	Send character *c* repeated *n* times
STR ArrayVar{*n*}	Send string of *n* characters

```
' Send the decimal value of B0 followed by a
linefeed out the hardware USART
HSEROUT [DEC B0,10]
```

5.37. HSEROUT2

HSEROUT2 [*Item*{,*Item...*}]

Send one or more *Item*s to the second hardware serial port on devices that support asynchronous serial communications in hardware.

HSEROUT2 works the same as **HSEROUT** with the exception that is uses the second hardware serial port on devices that have 2 ports such as the PIC18F8720. It can only be used with devices that have 2 hardware USARTs. See the device data sheet for information on the serial output pin and other parameters and the above section on **HSEROUT** for more command details. The serial parameters and baud rate are specified using **DEFINE**s:

```
' Set receive register to receiver enabled
DEFINE HSER2_RCSTA      90h

' Set transmit register to transmitter enabled
DEFINE HSER2_TXSTA      20h

' Set baud rate
DEFINE HSER2_BAUD 2400

' Set SPBRG2, SPBRGH2 directly
' (better to set HSER2_BAUD instead)
DEFINE HSER2_SPBRG      25
DEFINE HSER2_SPBRGH     0

' Use only if even parity desired
DEFINE HSER2_EVEN 1

' Use only if odd parity desired
DEFINE HSER2_ODD  1

' Use 8 bits + parity
DEFINE HSER2_BITS 9


' Send the decimal value of B0 followed by a
linefeed out the hardware USART
HSEROUT2 [DEC B0,10]
```

5.38. I2CREAD

I2CREAD *DataPin,ClockPin,Control,*{*Address,*}
[*Var*{*,Var...*}**]**{*,Label*}

Send *Control* and optional *Address* bytes out the *ClockPin* and *DataPin* and store the byte(s) received into *Var*. *ClockPin* and *DataPin* may be a constant, 0-15, or a variable that contains a number 0-15 (e.g. B0) or a pin name (e.g. PORTA.0).

I2CREAD and **I2CWRITE** can be used to read and write data to a serial EEPROM with a 2-wire I^2C interface such as the Microchip 24LC01B and similar devices. This allows data to be stored in external non-volatile memory so that it can be maintained even after the power is turned off. These commands operate in the I^2C master mode and may also be used to talk to other devices with an I^2C interface like temperature sensors and A/D converters.

For 12-bit core PIC MCUs only, the I2C clock and data pins are fixed at compile time by **DEFINE**s. They still must be specified in the **I2CREAD** statements, though this information is ignored by the compiler.

```
DEFINE I2C_SCL PORTA,1   ' For 12-bit core only
DEFINE I2C_SDA PORTA,0   ' For 12-bit core only
```

The upper 7 bits of the *Control* byte contain the control code along with chip select or additional address information, depending on the particular device. The low order bit is an internal flag indicating whether it is a read or write command and should be kept clear.

This format for the *Control* byte is different than the format used by the original PICBASIC Compiler. Be sure to use this format with PBP I^2C operations.

For example, when communicating with a 24LC01B, the control code is %1010 and the chip selects are unused so the *Control* byte would be %10100000 or $A0. Formats of *Control* bytes for some of the different parts follows:

Device	Capacity	Control	Address size
24LC01B	128 bytes	%1010xxx0	1 byte
24LC02B	256 bytes	%1010xxx0	1 byte
24LC04B	512 bytes	%1010xxb0	1 byte
24LC08B	1K bytes	%1010xbb0	1 byte
24LC16B	2K bytes	%1010bbb0	1 byte
24LC32B	4K bytes	%1010ddd0	2 bytes
24LC65	8K bytes	%1010ddd0	2 bytes

bbb = block select (high order address) bits
ddd = device select bits
xxx = don't care

The *Address* size sent (byte or word) is determined by the size of the variable that is used. If a byte-sized variable is used for the *Address*, an 8-bit address is sent. If a word-sized variable is used, a 16-bit address is sent. Be sure to use the proper sized variable for the device you wish to communicate with. Constants should not be used for the *Address* as the size can vary dependent on the size of the constant. Also, expressions should not be used as they can cause an improper *Address* size to be sent.

Once *Control* and/or *Address* has been sent to the device, the data specified between the square brackets is read from the device. If a word- or long-sized *Var* is specified, the bytes are read and stored into the *Var* highest byte first, followed by the lower byte(s). This order is different than the way variables are normally stored, low byte first.

A modifier, **STR**, may be included before the variable name. This can load an entire array (string) at once. If **STR** is specified, the following variable must be the name of a word or byte array, followed by a backslash (\) and a count:

```
a       Var     Byte[8]
```

```
addr  Var   Byte
      addr = 0
      I2CREAD PORTC.4,PORTC.3,$a0,addr,[STR a\8]
```

If a word- or long-sized array is specified, the bytes that comprise each element are read lowest byte first. This is the opposite of how simple words and longs are read and is consistent with the way the compiler normally stores word- and long-sized variables.

If the optional *Label* is included, this label will be jumped to if an acknowledge is not received from the I^2C device.

The I^2C instructions can be used to access the on-chip serial EEPROM on the PIC12CE and PIC16CE devices. Simply specify the pin names for the appropriate internal lines as part of the I^2C command and place the following **DEFINE** at the top of the program:

```
DEFINE I2C_INTERNAL 1
```

For the PIC12CE67x devices, the data line is GPIO.6 and the clock line is GPIO.7. For the PIC16CE62x devices, the data line is EEINTF.1 and the clock line is EEINTF.2. See the Microchip data sheets for these devices for more information.

The timing of the I^2C instructions is set so that standard speed devices (100kHz) will be accessible at clock speeds up to 8MHz. Fast mode devices (400kHz) may be used up to 20MHz. If it is desired to access a standard speed device at above 8MHz, the following **DEFINE** should be added to the program:

```
DEFINE I2C_SLOW    1
```

Because of memory and stack constraints, this **DEFINE** for 12-bit core PIC MCUs does not do anything. Low-speed (100 kHz) I2C devices may be used up to 4MHz. Above 4MHz, high-speed (400kHz) devices should be used.

Transfer on the I2C bus can be paused by the receiving device by its holding the clock line low (not supported on 12-bit core PIC MCUs). To enable this the following **DEFINE** should be added to the program:

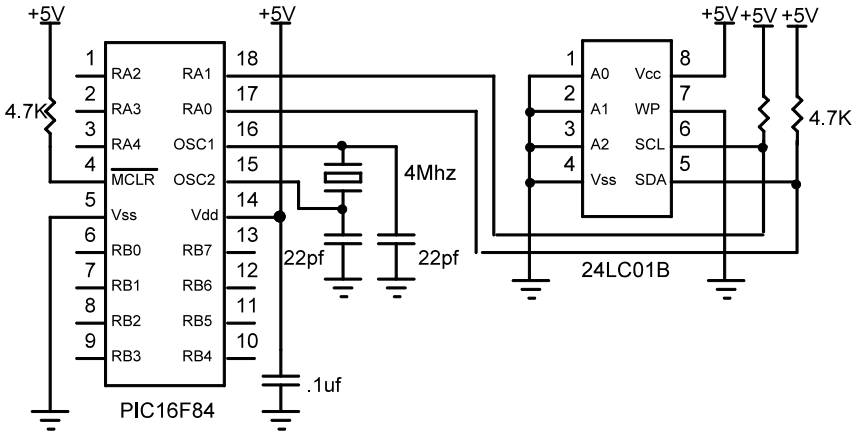
```
DEFINE I2C_HOLD    1
```

The I^2C clock and data lines should be pulled up to Vcc with a 4.7K

resistor per the following schematic as they are both run in a bi-directional open-collector manner.

To make the I2C clock line bipolar instead of open-collector the following **DEFINE** may be added to the program:

```
DEFINE I2C_SCLOUT  1
```




```
addr  Var   Byte
cont  Con   %10100000
      addr = 17          ' Set address to 17
      ' Read data at address 17 into B2
      I2CREAD PORTA.0,PORTA.1,cont,addr,[B2]
```

See the Microchip “Non-Volatile Memory Products Data Book” for more information on these and other devices that may be used with the **I2CREAD** and **I2CWRITE** commands.

5.39. I2CWRITE

I2CWRITE *DataPin,ClockPin,Control,*{*Address,*}
[*Value*{**,***Value...*}**]**{**,***Label*}

I2CWRITE sends *Control* and optional *Address* out the I^2C *ClockPin* and *DataPin* followed by *Value*. *ClockPin* and *DataPin* may be a constant, 0-15, or a variable that contains a number 0-15 (e.g. B0) or a pin name (e.g. PORTA.0).

For 12-bit core PIC MCUs only, the I2C clock and data pins are fixed at compile time by **DEFINE**s. They still must be specified in the **I2CWRITE** statements, though this information is ignored by the compiler.

```
DEFINE I2C_SCL PORTA,1   ' For 12-bit core only
DEFINE I2C_SDA PORTA,0   ' For 12-bit core only
```

The *Address* size sent (byte or word) is determined by the size of the variable that is used. If a byte-sized variable is used for the *Address*, an 8-bit address is sent. If a word-sized variable is used, a 16-bit address is sent. Be sure to use the proper sized variable for the device you wish to communicate with. Constants should not be used for the *Address* as the size can vary dependent on the size of the constant. Also, expressions should not be used as they can cause an improper *Address* size to be sent.

When writing to a serial EEPROM, it is necessary to wait 10ms (device dependent - check its data sheet) for the write to complete before attempting communication with the device again. If a subsequent **I2CREAD** or **I2CWRITE** is attempted before the write is complete, the access will be ignored.

While a single **I2CWRITE** statement may be used to write multiple bytes at once, doing so may violate the above write timing requirement for serial EEPROMs. Some serial EEPROMS let you write multiple bytes into a single page before necessitating the wait. Check the data sheet for the specific device you are using for these details. The multiple byte write feature may also be useful with I^2C devices other than serial EEPROMs that don't have to wait between writes.

If a word- or long-sized *Value* is specified, the bytes are sent highest byte first, followed by the lower byte(s). This order is different than the way variables are normally stored, low byte first.

A modifier, **STR**, may be included before the variable name. This can be used to write an entire array (string) at once and take advantage of a serial EEPROM's page mode. The data must fit into a single SEEPROM page. The page size is dependent on the particular SEEPROM device. If **STR** is specified, the following variable must be the name of a word or byte array, followed by a backslash (\) and a count:

```
a      Var   Byte[8]
addr   Var   Byte
       addr = 0
       I2CWRITE PORTC.4,PORTC.3,$a0,addr,[STR a\8]
```

If a word- or long-sized array is specified, the bytes that comprise each element are written lowest byte first. This is the opposite of how simple words and longs are written and is consistent with the way the compiler normally stores word- and long-sized variables.

If the optional *Label* is included, this label will be jumped to if an acknowledge is not received from the I^2C device.

The I^2C instructions can be used to access the on-chip serial EEPROM on the PIC12CE and PIC16CE devices. Simply specify the pin names for the appropriate internal lines as part of the I^2C command and place the following **DEFINE** at the top of the program:

```
DEFINE I2C_INTERNAL 1
```

For the PIC12CE67x devices, the data line is GPIO.6 and the clock line is GPIO.7. For the PIC16CE62x devices, the data line is EEINTF.1 and the clock line is EEINTF.2. See the Microchip data sheets for these devices for more information.

The timing of the I^2C instructions is set so that standard speed devices (100kHz) will be accessible at clock speeds up to 8MHz. Fast mode devices (400kHz) may be used up to 20MHz. If it is desired to access a standard speed device at above 8MHz, the following **DEFINE** should be added to the program:

```
DEFINE I2C_SLOW    1
```

Because of memory and stack constraints, this **DEFINE** for 12-bit core PIC MCUs does not do anything. Low-speed (100 kHz) I2C devices may be used up to 4MHz. Above 4MHz, high-speed (400kHz) devices should be used.

Transfer on the I2C bus can be paused by the receiving device by its holding the clock line low (not supported on 12-bit core PIC MCUs). To enable this the following **DEFINE** should be added to the program:

```
DEFINE I2C_HOLD    1
```

To make the I2C clock line bipolar instead of open-collector the following **DEFINE** may be added to the program:

```
DEFINE I2C_SCLOUT  1
```

See the **I2CREAD** command above for the rest of the story.

```
addr  Var   Byte
cont  Con   %10100000

      addr = 17          ' Set address to 17
      ' Send the byte 6 to address 17
      I2CWRITE PORTA.0,PORTA.1,cont,addr,[6]
      Pause 10           ' Wait 10ms for write to
                         complete
      addr = 1           ' Set address to 1
      ' Send the byte in B2 to address 1
      I2CWRITE PORTA.0,PORTA.1,cont,addr,[B2]
      Pause 10           ' Wait 10ms for write to
                         complete
```

5.40. IF..THEN

```
IF Comp {AND/OR Comp...} THEN Label

IF Comp {AND/OR Comp...} THEN Statement...

IF Comp {AND/OR Comp...} THEN
      Statement...
{ELSEIF Comp {AND/OR Comp...} THEN
      Statement...}
{ELSE
      Statement...}
ENDIF
```

Performs one or more comparisons. Each *Comp* term can relate a variable to a constant or other variable and includes one of the comparison operators listed previously.

IF..THEN evaluates the comparison terms for true or false. If it evaluates to true, the operation after the **THEN** is executed. If it evaluates to false, the operation after the **THEN** is not executed. Comparisons that evaluate to 0 are considered false. Any other value is considered true.

For PBP and PBPW, all comparisons are unsigned since they only supports unsigned types. **IF..THEN** cannot be used to check if a number is less than 0. Using PBPL, signed comparisons, including less than zero, may be performed.

It is essential to use parenthesis to specify the order in which the operations should be tested. Otherwise, operator precedence will determine it for you and the result may not be as expected.

IF..THEN can operate in 2 manners. In one form, the **THEN** in an **IF..THEN** is essentially a **GOTO**. If the condition is true, the program will **GOTO** the label after the **THEN**. If the condition evaluates to false, the program will continue at the next line after the **IF..THEN**.

```
If Pin0 = 0 Then pushd  ' If button connected to
                        Pin0 is pushed (0), jump
                        to label pushd
```

```
If B0 >= 40 Then old      ' If the value in
                          variable B0 is greater
                          than or equal to 40,
                          jump to old

If PORTB.0 Then itson     ' If PORTB, pin 0 is
                          high (1), jump to itson

If (B0 = 10) And (B1 = 20) Then mainloop
```

In the second form, **IF..THEN** can conditionally execute a group of *Statement*s following the **THEN**. The *Statement*s may be placed directly after the **THEN** or may be on another line and followed by an optional **ELSEIF** or **ELSE** and non-optional **ENDIF** to complete the structure.

```
If B0 <> 10 Then B0 = B0 + 1: B1 = B1 - 1

If B0 <> 10 Then
      B0 = B0 + 1
      B1 = B1 - 1
Endif

If B0 = 20 Then
      led = 1
Else
      led = 0
Endif

If B0 = 20 Then
      led = 1
Elseif B0 = 40 Then
      led = 1
Else
      led = 0
Endif
```

5.41. INPUT

INPUT *Pin*

Makes the specified *Pin* an input. *Pin* may be a constant, 0-15, or a variable that contains a number 0-15 (e.g. B0) or a pin name (e.g. PORTA.0).

```
INPUT 0             ' Make Pin0 an input
INPUT PORTA.0       ' Make PORTA, pin 0 an input
```

Alternatively, the pin may be set to an input in a much quicker and shorter way (from a generated code standpoint):

```
TRISB.0 = 1         ' Set PORTB, pin 0 to an input
```

All of the pins on a port may be set to inputs by setting the entire TRIS register at once:

```
TRISB = %11111111 ' Set all of PORTB to inputs
```

5.42. LCDIN

LCDIN {*Address*,}**[***Var*{,*Var...*}**]**

Read LCD RAM at *Address* and store data to *Var*.

LCDs have RAM onboard that is used for character memory. Most LCDs have more RAM available that is necessary for the displayable area. This RAM can be written using the **LCDOUT** instruction. The **LCDIN** instruction allows this RAM to be read.

CG (character generator) RAM runs from address $40 to $7f. Display data RAM starts at address $80. See the data sheet for the specific LCD for these addresses and functions.

It is necessary to connect the LCD read/write line to a PIC MCU pin so that it may be controlled to select either a read (**LCDIN**) or write (**LCDOUT**) operation. Two **DEFINE**s control the pin address:

```
DEFINE LCD_RWREG PORTE   ' LCD read/write pin
                         port
DEFINE LCD_RWBIT 2       ' LCD read/write pin bit
```

See **LCDOUT** for information on connecting an LCD to a PIC MCU.

```
LCDIN [B0]
```

5.43. LCDOUT

LCDOUT *Item*{,Item...}

Display *Item*s on an intelligent Liquid Crystal Display. PBP supports LCD modules with a Hitachi 44780 controller or equivalent. These LCDs usually have a 14- or 16-pin single- or dual-row header at one edge.

If a pound sign (**#**) precedes an *Item*, the ASCII representation for each digit is sent to the LCD. **LCDOUT** (on all devices except 12-bit core) can also use any of the modifiers used with **SEROUT2**. See the section on **SEROUT2** for this information.

Modifier	Operation
{I}{S}BIN{1..32}	Send binary digits
{I}{S}DEC{1..10}	Send decimal digits
{I}{S}HEX{1..8}	Send hexadecimal digits
REP *cn***	Send character *c* repeated *n* times
STR ArrayVar{*n*}	Send string of *n* characters

A program should wait for up to half a second before sending the first command to an LCD. It can take quite a while for an LCD to start up.

The LCD is initialized the first time any character or command is sent to it using **LCDOUT**. If it is powered down and then powered back up for some reason during operation, an internal flag can be reset to tell the program to reinitialize it the next time it uses **LCDOUT**:

```
FLAGS = 0
```

Commands are sent to the LCD by sending a $FE followed by the command. Some useful commands are listed in the following table:

Command	Operation
$FE, 1	Clear display
$FE, 2	Return home
$FE, $0C	Cursor off
$FE, $0E	Underline cursor on
$FE, $0F	Blinking cursor on
$FE, $10	Move cursor left one position
$FE, $14	Move cursor right one position
$FE, $80	Move cursor to beginning of first line
$FE, $C0	Move cursor to beginning of second line
$FE, $94	Move cursor to beginning of third line
$FE, $D4	Move cursor to beginning of fourth line

Note that there are commands to move the cursor to the beginning of the different lines of a multi-line display. For most LCDs, the displayed characters and lines are not consecutive in display memory - there can be a break in between locations. For most 16x2 displays, the first line starts at $80 and the second line starts at $C0. The command:

```
LCDOUT $FE, $80 + 4
```

sets the display to start writing characters at the forth position of the first line. 16x1 displays are usually formatted as 8x2 displays with a break between the memory locations for the first and second 8 characters. 4-line displays also have a mixed up memory map, as shown in the table above.

See the data sheet for the particular LCD device for the character memory locations and additional commands..

```
LCDOUT $FE,1,"Hello"      ' Clear display and show
                          "Hello"
LCDOUT $FE,$C0,"World"    ' Jump to second line
                          and show "World"
LCDOUT B0,#B1             ' Display B0 and decimal
                          ASCII value of B1
```

The LCD may be connected to the PIC MCU using either a 4-bit bus or an 8-bit bus. If an 8-bit bus is used, all 8 bits must be on one port. If a 4-

bit bus is used, the top 4 LCD data bits must be connected to either the bottom 4 or top 4 bits of one port. Enable and Register Select may be connected to any port pin. R/W may be tied to ground if the **LCDIN** command is not used.

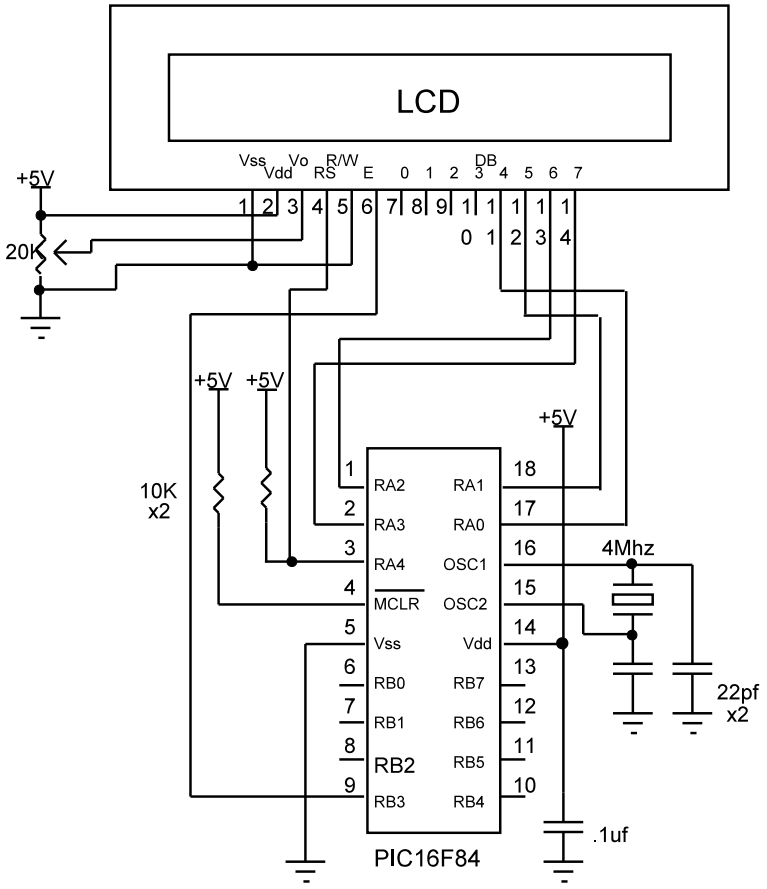
PBP assumes the LCD is connected to specific pins unless told otherwise using **DEFINE**s. It assumes the LCD will be used with a 4-bit bus with data lines DB4 - DB7 connected to PIC MCU PORTA.0 - PORTA.3, Register Select to PORTA.4 and Enable to PORTB.3.

It is also preset to initialize the LCD to a 2 line display.

To change this setup, place one or more of the following **DEFINEs**, all in upper-case, at the top of your PICBASIC PRO program:

```
' Set LCD Data port
DEFINE LCD_DREG   PORTA
' Set starting Data bit (0 or 4) if 4-bit bus
DEFINE LCD_DBIT   0
' Set LCD Register Select port
DEFINE LCD_RSREG  PORTA
' Set LCD Register Select bit
DEFINE LCD_RSBIT  4
' Set LCD Enable port
DEFINE LCD_EREG   PORTB
' Set LCD Enable bit
DEFINE LCD_EBIT   3
' Set LCD bus size (4 or 8 bits)
DEFINE LCD_BITS   4
' Set number of lines on LCD
DEFINE LCD_LINES  2
' Set command delay time in us
DEFINE LCD_COMMANDUS    1500
' Set data delay time in us
DEFINE LCD_DATAUS 44
```

The following schematic shows one way to connect an LCD to a PIC MCU, using the defaults for the PIC16F84:


LCD
Vss Vdd Vo RS R/W E DB 0 1 2 3 4 5 6 7
1 2 3 4 5 6 7 8 9 10 11 12 13 14
+5V
20K
+5V +5V
+5V
10K x2
1 RA2 RA1 18
2 RA3 RA0 17
3 RA4 OSC1 16
4 MCLR OSC2 15
5 Vss Vdd 14
6 RB0 RB7 13
7 RB1 RB6 12
8 RB2 RB5 11
9 RB3 RB4 10
PIC16F84
4Mhz
22pf x2
.1uf

5.44. {LET}

{**LET**} *Var* **=** *Value*

Assign a *Value* to a *Var*iable. The *Value* may be a constant, another variable or the result of an expression. Refer to the previous section on operators for more information. The keyword **LET** itself is optional.

```
LET B0 = B1 * B2 + B3
B0 = Sqr W1
```

5.45. LOOKDOWN

LOOKDOWN *Search*,[*Constant*{,*Constant...*}],*Var*

The **LOOKDOWN** statement searches a list of 8-bit *Constant* values for the presence of the *Search* value. If found, the index of the matching constant is stored in *Var*. Thus, if the value is found first in the list, *Var* is set to zero. If second in the list, *Var* is set to one. And so on. If not found, *Var* remains unchanged.

The constant list can be a mixture of numeric and string constants. Each character in a string is treated as a separate constant with the character's ASCII value. Array variables with a variable index may not be used in **LOOKDOWN** although array variables with a constant index are allowed. Up to 255 (256 for PIC18) constants are allowed in the list.

```
Serin 1,N2400,B0   ' Get hexadecimal character
                   from Pin1 serially
LOOKDOWN B0,["0123456789ABCDEF"],B1 ' Convert
                                    hexadecimal
                                    character in
                                    B0 to
                                    decimal
                                    value B1
Serout 0,N2400,[#B1]     ' Send decimal value to
                         Pin0 serially
```

5.46. LOOKDOWN2

LOOKDOWN2 *Search*,{*Test*}**[***Value*{,*Value...*}**]**,*Var*

The **LOOKDOWN2** statement searches a list of *Values* for the presence of the *Search* value. If found, the index of the matching constant is stored in *Var*. Thus, if the value is found first in the list, *Var* is set to zero. If second in the list, *Var* is set to one. And so on. If not found, *Var* remains unchanged.

The optional parameter *Test* can be used to perform a test for other than equal to ("**=**") while searching the list. For example, the list could be searched for the first instance where the *Search* parameter is greater than the *Value* by using "**>**" as the *Test* parameter. If *Test* is left out, "**=**" is assumed.

The *Value* list can be a mixture of 8- and 16-bit (and 32-bit for PBPL) numeric and string constants and variables. Each character in a string is treated as a separate constant equal to the character's ASCII value. Expressions may not be used in the *Value* list, although they may be used as the *Search* value.

Array variables with a variable index may not be used in **LOOKDOWN2** although array variables with a constant index are allowed. Up to 85 (256 for PIC18) values are allowed in the list.

LOOKDOWN2 generates code that is about 3 times larger than **LOOKDOWN**. If the search list is made up only of 8-bit constants and strings, use **LOOKDOWN**.

```
LOOKDOWN2 W0,[512,W1,1024],B0
LOOKDOWN2 W0,>[1000,100,10],B0
```

5.47. LOOKUP

LOOKUP *Index*,**[***Constant*{,*Constant...*}**]**,*Var*

The **LOOKUP** statement can be used to retrieve values from a table of 8-bit constants. If *Index* is zero, *Var* is set to the value of the first *Constant*. If *Index* is one, *Var* is set to the value of the second *Constant*. And so on. If *Index* is greater than or equal to the number of entries in the constant list, *Var* remains unchanged.

The constant list can be a mixture of numeric and string constants. Each character in a string is treated as a separate constant equal to the character's ASCII value. Array variables with a variable index may not be used in **LOOKUP** although array variables with a constant index are allowed. Up to 255 (1024 for PIC18) constants are allowed in the list.

```
For B0 = 0 To 5          ' Count from 0 to 5
      LOOKUP B0,["Hello!"],B1 ' Get character
                              number B0 from
                              string to variable
                              B1
      Serout 0,N2400,[B1]     ' Send character
                              in B1 to Pin0
                              serially
Next B0                  ' Do next character
```

5.48. LOOKUP2

LOOKUP2 *Index*,[*Value*{,*Value...*}],*Var*

The **LOOKUP2** statement can be used to retrieve entries from a table of *Values*. If *Index* is zero, *Var* is set to the first *Value*. If *Index* is one, *Var* is set to the second *Value*. And so on. If *Index* is greater than or equal to the number of entries in the list, and *Var* remains unchanged.

The *Value* list can be a mixture of 8-bit and 16-bit (and 32-bit for PBPL) numeric and string constants and variables. Each character in a string is treated as a separate constant equal to the character's ASCII value. Expressions may not be used in the *Value* list, although they may be used as the *Index* value.

Array variables with a variable index may not be used in **LOOKUP2** although array variables with a constant index are allowed. Up to 85 (1024 for PIC18) values are allowed in the list.

LOOKUP2 generates code that is about 3 times larger than **LOOKUP**. If the *Value* list is made up of only 8-bit constants and strings, use **LOOKUP**.

```
LOOKUP2 B0,[256,512,1024],W1
```

5.49. LOW

LOW *Pin*

Make the specified *Pin* low. *Pin* is automatically made an output. *Pin* may be a constant, 0 - 15, or a variable that contains a number 0 - 15 (e.g. B0) or a pin name (e.g. PORTA.0).

```
      LOW 0              ' Make Pin0 an output and set
                         it low (0 volts)
      LOW PORTA.0        ' Make PORTA, pin 0 an output
                         and set it low (0 volts)

led   Var   PORTB.0      ' Define LED pin
      LOW led            ' Make LED pin an output and
                         set it low (0 volts)
```

Alternatively, if the pin is already an output, a much quicker and shorter way (from a generated code standpoint) to set it low would be:

```
      PORTB.0 = 0        ' Set PORTB, pin 0 low
```

5.50. NAP

NAP *Period*

Places the microcontroller into low power mode for a short period of time. During this **NAP**, power consumption is reduced. To achieve minimum current draw it may be necessary to turn off other peripherals on the device, such as the ADC, before executing the **NAP** command. See the Microchip data sheet for the specific device for information about these register settings.

NAP puts the processor to sleep for one Watchdog Timer period. If the Watchdog Timer is not enabled, the processor will sleep forever or until an enabled interrupt or reset is received.

The *Period* is used to set the Watchdog timer prescaler for devices that have a prescaler including the 12- and 14-bit core devices. The 16-bit core devices, including the PIC17 and PIC18 parts use a postscaler set at programming time to configure the Watchdog timeout period. The compiler will disregard the *Period* set in the **NAP** instruction for the 16-bit core devices.

The listed *Period*s for the 12- and 14-bit core devices are only approximate because the timing derived from the Watchdog Timer is R/C driven and can vary greatly from chip to chip and over temperature.

Period	Delay (Approx.)
0	18 milliseconds
1	36 milliseconds
2	72 milliseconds
3	144 milliseconds
4	288 milliseconds
5	576 milliseconds
6	1.152 seconds
7	2.304 seconds

```
NAP 7 ' Low power pause for about 2.3 seconds
```

5.51. ON DEBUG

ON DEBUG GOTO *Label*

ON DEBUG allows a debug monitor routine to be executed between each PICBASIC PRO instruction

The method by which this happens is similar to the method used by **ON INTERRUPT GOTO**. Once **ON DEBUG GOTO** is encountered, a call to the specified debug label is inserted before each PICBASIC PRO instruction in the program. **DISABLE DEBUG** prevents the insertion of this call while **ENABLE DEBUG** resumes the insertion of the call.

A monitor routine may be written that is activated before each instruction. This routine can send data to an LCD or to a serial comm program. Any program information may be displayed or even altered in this manner. A small monitor program example is posted on our web site.

A word-sized system variable that resides in BANK0 is required to provide a place to store the address the program was at before the monitor routine was called by **ON DEBUG GOTO**. An additional byte-sized system variable is required for PIC18 parts.

```
DEBUG_ADDRESS Var Word Bank0 System
DEBUG_ADDRESSU Var Byte Bank0 System 'PIC18 only
```

Another byte-sized variable may be used to return the level of the current program stack:

```
DEBUG_STACK Var   Byte Bank0 System
```

This level should never be greater than 4 for 12- and 14-bit core PIC MCUs, 12 for PIC17 devices or 27 for PIC18 devices in a PICBASIC PRO program. The supplied variable will be incremented at each **GOSUB** and decremented at each **RETURN**. This variable should be set to 0 at the beginning of the program.

Adding this variable to a program does add overhead in that the value of the variable must be incremented and decremented at each **GOSUB** and **RETURN**.

5.52. ON GOSUB

ON *Index* **GOSUB** *Label*{,*Label...*}

ON GOSUB causes the program to jump to a different subroutine based on a variable index. Once the subroutine is complete and a **RETURN** is encountered, the program continues execution at the line following the **ON GOSUB**.

Index selects one of a list of *Label*s. A subroutine call is made to the indexed *Label*. For example, if *Index* is zero, the program does a **GOSUB** to the first *Label* specified in the list, if *Index* is one, the program does a **GOSUB** to the second *Label*, and so on. If *Index* is greater than or equal to the number of *Label*s, no action is taken and execution continues with the statement following the **ON GOSUB**. Up to 127 (1024 for PIC18) *Label*s may be used in a **ON GOSUB**.

An unlimited number of subroutines may be used in a program. Subroutines may also be nested. In other words, it is possible for a subroutine to **GOSUB** to another subroutine. Such subroutine nesting must be restricted to no more than four nested levels for 14-bit core devices, 12 levels for 14-bit enhanced core and PIC17 parts and 27 levels for PIC18 parts. Interrupts cause additional locations to be used on the stack, reducing the number of possible nested **GOSUB**s. See the section on interrupts later in the manual for more information.

ON GOSUB is not supported on 12-bit core PIC MCUs due to RAM and stack constraints.

```
ON B4 GOSUB dog,cat,fish
‘ Same as:
‘  If B4=0 Then Gosub dog : Goto after
‘  If B4=1 Then Gosub cat : Goto after
‘  If B4=2 Then Gosub fish
‘after:
```

5.53. ON GOTO

ON *Index* **GOTO** *Label*{,*Label...*}

ON GOTO causes the program to jump to a different location based on a variable index. It can jump to a *Label* that is in a different code page than the **ON GOTO** instruction for 12- and 14-bit core and PIC17 devices, or further away than 1K for PIC18 devices. It generates code that is about twice the size as code generated by the **BRANCH** instruction. If you are sure the labels are in the same page as the **BRANCH** instruction or if the microcontroller does not have more than one code page, using **BRANCH** instead of **ON GOTO** will minimize memory usage. **ON GOTO** is a different syntax of **BRANCHL**.

Index selects one of a list of *Label*s. Execution resumes at the indexed *Label*. For example, if *Index* is zero, the program jumps to the first *Label* specified in the list, if *Index* is one, the program jumps to the second *Label*, and so on. If *Index* is greater than or equal to the number of *Label*s, no action is taken and execution continues with the statement following the **ON GOTO**. Up to 127 (1024 for PIC18) *Label*s may be used in a **ON GOTO**.

```
ON B4 GOTO dog,cat,fish
‘ Same as:
‘  If B4=0 Then dog (goto dog)
‘  If B4=1 Then cat (goto cat)
‘  If B4=2 Then fish (goto fish)
```

5.54. ON INTERRUPT

```
ON INTERRUPT GOTO Label
```

ON INTERRUPT allows the handling of microcontroller interrupts by a PICBASIC PRO subroutine.

There are 2 ways to handle interrupts using the PICBASIC PRO Compiler. The first is to write an assembly language interrupt routine. This is the way to handle interrupts with the shortest latency and lowest overhead, but must contain only assembly language, not BASIC, code. This method is discussed under advanced topics in a later section.

The second method is to write a PICBASIC PRO interrupt handler. This looks just like a PICBASIC PRO subroutine but ends with a **RESUME** instead of a **RETURN**.

When an interrupt occurs, it is flagged. As soon as the current PICBASIC PRO statement's execution is complete, the program jumps to the BASIC interrupt handler at *Label*. Once the interrupt handler is complete, a **RESUME** statement sends the program back to where it was when the interrupt occurred, picking up where it left off.

DISABLE and **ENABLE** allow different sections of a PICBASIC PRO program to execute without the possibility of being interrupted. The most notable place to use **DISABLE** is right before the actual interrupt handler. Or the interrupt handler may be placed before the **ON INTERRUPT** statement as the interrupt flag is not checked before the first **ON INTERRUPT** in a program.

Latency is the time it takes from the time of the actual interrupt to the time the interrupt handler is entered. Since PICBASIC PRO statements are not re-entrant (i.e. you cannot execute another PICBASIC PRO statement while one is being executed), there can be considerable latency before the interrupt routine is entered.

PBP will not enter the BASIC interrupt handler until it has finished executing the current statement. If the statement is a **PAUSE** or **SERIN**, it could be quite a while before the interrupt is acknowledged. The program must be designed with this latency in mind. If it is unacceptable and the interrupts must be handled more quickly, an assembly language interrupt routine must be used.

Overhead is another issue. **ON INTERRUPT** will add instructions before every statement to check whether or not an interrupt has occurred. **DISABLE** turns off the addition of these instructions. **ENABLE** turns it back on again. Usually the additional instructions will not be much of a problem, but long programs in small microcontrollers could suffer.

More than one **ON INTERRUPT** may be used in a program.

```
ON INTERRUPT GOTO myint ' Interrupt handler is
                        myint
INTCON = %10010000      ' Enable RB0 interrupt

. . .

DISABLE                 ' Disable interrupts in
                        handler
myint: led = 1          ' Turn on LED when interrupted
INTCON.1 = 0            ' Clear interrupt flag
RESUME                  ' Return to main program
ENABLE                  ' Enable interrupts after
                        handler
```

To turn off interrupts permanently (or until needed again) once **ON INTERRUPT** has been used, set INTCON to $80:

```
INTCON = $80
```

5.55. OUTPUT

OUTPUT *Pin*

Make the specified *Pin* an output. *Pin* may be a constant, 0 - 15, or a variable that contains a number 0 - 15 (e.g. B0) or a pin name (e.g. PORTA.0).

```
OUTPUT 0            ' Make Pin0 an output
OUTPUT PORTA.0      ' Make PORTA, pin 0 an output
```

Alternatively, the pin may be set to an output in a much quicker and shorter way (from a generated code standpoint):

```
TRISB.0 = 0         ' Set PORTB, pin 0 to an
                    output
```

All of the pins on a port may be set to outputs by setting the entire TRIS register at once:

```
TRISB = %00000000 ' Set all of PORTB to outputs
```

5.56. OWIN

OWIN *Pin*,*Mode*,[*Item...*]{,*Label*}

Optionally send a reset pulse to a one-wire device and then read one or more bits or bytes of data from it, optionally ending with another reset pulse.

Pin may be a constant, 0 - 15, or a variable that contains a number 0 - 15 (e.g. B0) or a pin name (e.g. PORTA.0).

Mode specifies whether a reset is sent before and/or after the operation and the size of the data items, either bit or byte.

Mode bit number	Effect
0	1 = send reset pulse before data
1	1 = send reset pulse after data
2	0 = byte-sized data, 1 = bit-sized data

Some *Mode* examples would be: *Mode* of %000 (decimal 0) means no reset and byte-sized data, *Mode* of %001 (decimal 1) means reset before data and byte-sized data, *Mode* of %100 (decimal 4) means no reset and bit-sized data.

Item is one or more variables or modifiers separated by commas. The allowable modifiers are **STR** for reading data into a byte array variable and **SKIP** for skipping a number of input values.

The **SKIP** and **STR** modifiers are not supported for the 12-bit core PIC MCUs because of RAM and stack size limits.

If a device is not present, **OWIN** can jump to an optional *Label*.

```
OWIN PORTC.0,%000,[STR temperature\2,SKIP 4,
count_remain, count_per_c]
```

This statement would receive bytes from a one-wire device on PORTC pin 0 with no reset pulse being sent. It would receive 2 bytes and put them into the byte array temperature, skip the next 4 bytes and then read the final 2 bytes into separate variables.

5.57. OWOUT

OWOUT *Pin*,*Mode*,[*Item...*]{,*Label*}

Optionally send a reset pulse to a one-wire device and then writes one or more bits or bytes of data to it, optionally ending with another reset pulse.

Pin may be a constant, 0 - 15, or a variable that contains a number 0 - 15 (e.g. B0) or a pin name (e.g. PORTA.0).

Mode specifies whether a reset is sent before and/or after the operation and the size of the data items, either bit or byte.

Mode bit number	Effect
0	1 = send reset pulse before data
1	1 = send reset pulse after data
2	0 = byte-sized data, 1 = bit-sized data

Some *Mode* examples would be: *Mode* of %000 (decimal 0) means no reset and byte-sized data, *Mode* of %001 (decimal 1) means reset before data and byte-sized data, *Mode* of %100 (decimal 4) means no reset and bit-sized data.

Item is one or more constants, variables or modifiers separated by commas. The allowable modifiers are **STR** for sending data from a byte array variable and **REP** for sending a number of repeated values.

The **REP** and **STR** modifiers are not supported for the 12-bit core PIC MCUs because of RAM and stack size limits.

If a device is not present, **OWOUT** can jump to an optional *Label*.

```
OWOUT PORTC.0,%001,[$cc,$be]
```

This statement would send a reset pulse to a one-wire device on PORTC pin 0 followed by the bytes $cc and $be.

5.58. PAUSE

PAUSE *Period*

Pause the program for *Period* milliseconds. *Period* is 16-bits using PBP and PBPW, so delays can be up to 65,535 milliseconds (a little over a minute). Using PBPL, *Period* is 32-bits. This will allow delays of over 49 days. Long values are interpreted as unsigned. This may result in a longer pause than expected. If a long variable is used and it could go negative, it should be limited to greater than or equal to 0 using a function like MAX, for example.

Unlike the other delay functions (**NAP** and **SLEEP**), **PAUSE** doesn't put the microcontroller into low power mode. Thus, **PAUSE** consumes more power but is also much more accurate. It has the same accuracy as the system clock.

PAUSE assumes an oscillator frequency of 4MHz. If an oscillator other that 4MHz is used, PBP must be told using a **DEFINE** OSC command. See the section on speed for more information.

```
PAUSE 1000  ' Delay for 1 second
```

5.59. PAUSEUS

PAUSEUS *Period*

Pause the program for *Period* microseconds. *Period* is 16-bits, so delays can be up to 65,535 microseconds. Unlike the other delay functions (**NAP** and **SLEEP**), **PAUSEUS** doesn't put the microcontroller into low power mode. Thus, **PAUSEUS** consumes more power but is also much more accurate.

Because **PAUSEUS** takes a minimum number of cycles to operate, depending on the frequency of the oscillator, delays of less than a minimum number of microseconds are not possible using **PAUSEUS**. To obtain shorter delays, use an assembly language routine.

OSC	Minimum delay	Minimum delay PIC18
3 (3.58)	20us	20us**
4	24us	19us**
8	12us	9us**
10	8us	7us**
12	7us	5us**
16	5us	4us**
20	3us	3us**
24	3us	2us**
25,32,33	2us*	2us**
40,48,64	-	1us**

* PIC17 only.
** PIC18 only.

PAUSEUS assumes an oscillator frequency of 4MHz. If an oscillator other that 4MHz is used, PBP must be told using a **DEFINE** OSC command. See the section on speed for more information.

```
PAUSEUS 1000        ' Delay for 1 millisecond
```

5.60. PEEK

PEEK *Address*,*Var*

Read the microcontroller register at the specified *Address* and stores the result in *Var*. Special PIC MCU features such as A/D converters and additional I/O ports may be read using **PEEK**.

If *Address* is a constant, the contents of this register number are placed into *Var*. If *Address* is the name of a special function register, e.g. PORTA, the contents of this register will be placed into *Var*. If *Address* is a RAM location, the value of the RAM location will first be read, then the contents of the register specified by that value will be placed into *Var*.

However, all of the PIC MCU registers can be and should be accessed without using **PEEK** and **POKE**. All of the PIC MCU registers are considered 8-bit variables by PICBASIC PRO and may be used as you would any other byte-sized variable. They can be read directly or used directly in equations.

```
B0 = PORTA          ' Get current PORTA pin states
                    to B0
```

5.61. PEEKCODE

PEEKCODE *Address*,*Var*

Read a value from the code space at the specified *Address* and store the result in *Var*.

PEEKCODE can be used to read data stored in the code space of a PIC MCU. It executes a call to the specified *Address* and places the returned value in *Var*. The specified location should contain a `retlw` and the data value. **POKECODE** may be used to store this value at the time the device is programmed.

```
PEEKCODE $3ff, OSCCAL    ' Get OSCCAL value for
                         PIC12C671/12CE673

PEEKCODE $7ff, OSCCAL    ' Get OSCCAL value for
                         PIC12C672/12CE674
```

5.62. POKE

POKE *Address*,*Value*

Write *Value* to the microcontroller register at the specified *Address*. Special PIC MCU features such as A/D converters and additional I/O ports may be written using **POKE**.

If *Address* is a constant, *Value* is placed into this register number. If *Address* is the name of a special function register, e.g. PORTA, *Value* will be placed into this register. If *Address* is a RAM location, the contents of the RAM location will first be read, then *Value* is placed into the register specified by those contents.

However, all of the PIC MCU registers can be and should be accessed without using **PEEK** and **POKE**. All of the PIC MCU registers are considered 8-bit variables by PICBASIC PRO and may be used as you would any other byte-sized variable. They can be written directly or used directly in equations.

```
TRISA = 0    ' Set PORTA to all outputs
PORTA.0 = 1  ' Set PORTA bit 0 high
```

5.63. POKECODE

POKECODE {@*Address*,}*Value*{,*Value...*}

Store *Value*s to the code space at the current program address or optional specified *Address* at the time the microcontroller is programmed.

POKECODE can be used to generate tables in the code space of the PIC MCU. It generates a return with the data in W. This data can be accessed using the **PEEKCODE** instruction.

If the optional *Address* is not specified, data storage will be located immediately after the preceding program instruction written.

To avoid interruption of program flow, **POKECODE** should be the last line of your program. It should be placed after the **END** or **STOP** command.

```
      POKECODE 10, 20, 30       ' Store 10, 20, and 30
                                in code space
```

Generates:

```
      retlw 10
      retlw 20
      retlw 30
```

```
      POKECODE @$7ff, $94       ' Set OSCCAL value for
                                PIC12C672/12CE674
```

Generates:

```
      org   7ffh
      retlw 94h
```

5.64. POT

POT *Pin*,*Scale*,*Var*

Reads a potentiometer (or some other resistive device) on *Pin*. *Pin* may be a constant, 0 - 15, or a variable that contains a number 0 - 15 (e.g. B0) or a pin name (e.g. PORTA.0).

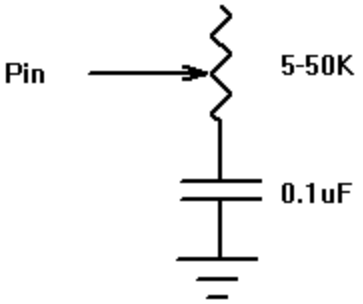
The resistance is measured by timing the discharge of a capacitor through the resistor (typically 5K to 50K). *Scale* is used to adjust for varying RC constants. For larger RC constants, *Scale* should be set low (a minimum value of one). For smaller RC constants, *Scale* should be set to its maximum value (255). If *Scale* is set correctly, *Var* should be zero near minimum resistance and 255 near maximum resistance.

Unfortunately, *Scale* must be determined experimentally. To do so, set the device under measure to maximum resistance and read it with *Scale* set to 127. Adjust Scale until the Pot command returns 254. If 255, decrease the scale. If 253 or lower, increase the scale. (Note: This is similar to the process performed by the **Alt-P** option of the BS1 environment).

Use the following code to automate the process. Make sure that you set the pot to maximum resistance.

```
B0     Var    Byte
scale  Var    Byte

       For scale = 1 To 255
              POT 0,scale,B0
              If (B0 > 253) Then calibrated
       Next scale

       Serout 2,0,["Increase R or C.",10,13]
       Stop

calibrated:
       Serout 2,0,["Scale= ",#scale,10,13]
```

Potentiometer wiring example:



5.65. PULSIN

PULSIN *Pin*,*State*,*Var*

Measures pulse width on *Pin*. If *State* is zero, the width of a low pulse is measured. If *State* is one, the width of a high pulse is measured. The measured width is placed in *Var*. If the pulse edge never happens or the width of the pulse is too great to measure, *Var* is set to zero.

Pin is automatically made an input. *Pin* may be a constant, 0 - 15, or a variable that contains a number 0 - 15 (e.g. B0) or a pin name (e.g. PORTA.0).

The resolution of **PULSIN** is dependent upon the oscillator frequency. If a 4MHz oscillator is used, the pulse width is returned in 10us increments. If a 20MHz oscillator is used, the pulse width will have a 2us resolution. Defining an OSC value has no effect on **PULSIN**. The resolution always changes with the actual oscillator speed.

PULSIN normally waits a maximum of 65535 counts before it determines there is no pulse. If it is desired to wait fewer or more counts before it stops looking for a pulse or the end of a pulse, a **DEFINE** can be added:

```
DEFINE PULSIN_MAX 1000
```

This **DEFINE** also affects **RCTIME** in the same manner.

```
' Measure high pulse on Pin4 stored in W3
PULSIN PORTB.4,1,W3
```

5.66. PULSOUT

PULSOUT *Pin*,*Period*

Generates a pulse on *Pin* of specified *Period*. The pulse is generated by toggling the pin twice, thus the initial state of the pin determines the polarity of the pulse. *Pin* is automatically made an output. *Pin* may be a constant, 0 - 15, or a variable that contains a number 0 - 15 (e.g. B0) or a pin name (e.g. PORTA.0).

The resolution of **PULSOUT** is dependent upon the oscillator frequency. If a 4MHz oscillator is used, the *Period* of the generated pulse will be in 10us increments. If a 20MHz oscillator is used, *Period* will have a 2us resolution. Defining an OSC value has no effect on **PULSOUT**. The resolution always changes with the actual oscillator speed.

```
' Send a pulse 1mSec long (at 4MHz) to Pin5
PULSOUT PORTB.5,100
```

5.67. PWM

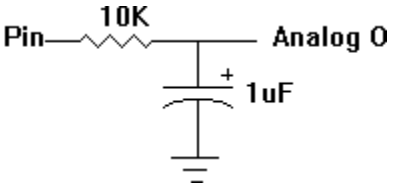
PWM *Pin*,*Duty*,*Cycle*

Outputs a pulse width modulated pulse train on *Pin*. Each cycle of PWM consists of 256 steps. The *Duty* cycle for each PWM cycle ranges from 0 (0%) to 255 (100%). This PWM cycle is repeated *Cycle* times. *Pin* may be a constant, 0 - 15, or a variable that contains a number 0 - 15 (e.g. B0) or a pin name (e.g. PORTA.0).

The *Cycle* time of **PWM** is dependent upon the oscillator frequency. If a 4MHz oscillator is used, each *Cycle* is about 5ms long. If a 20MHz oscillator is used, each *Cycle* is about 1ms in length. Defining an OSC value has no effect on **PWM**. The *Cycle* time always changes with the actual oscillator speed.

If you want continuous PWM output and the PIC MCU has PWM hardware, **HPWM** may be used instead of **PWM**. See the section on for **HPWM** more information about it.

Pin is made an output just prior to pulse generation and reverts to an input after generation stops. The **PWM** output on a pin looks like so much garbage, not a beautiful series of square waves. A filter of some sort is necessary to turn the signal into something useful. An RC circuit can be used as a simple D/A converter:




```
PWM PORTB.7,127,100      ' Send a 50% duty cycle
                         PWM signal out Pin7 for
                         100 cycles
```

5.68. RANDOM

RANDOM *Var*

Perform one iteration of pseudo-randomization on *Var*. *Var* should be a 16-bit variable. Array variables with a variable index may not be used in **RANDOM** although array variables with a constant index are allowed. *Var* is used both as the seed and to store the result. The pseudo-random algorithm used has a walking length of 65535 (only zero is not produced).

```
RANDOM W4    ' Randomize value in W4
```

5.69. RCTIME

RCTIME *Pin*,*State*,*Var*

RCTIME measures the time a *Pin* stays in a particular *State*. It is basically half a **PULSIN**. *Pin* is automatically made an input. *Pin* may be a constant, 0 - 15, or a variable that contains a number 0 - 15 (e.g. B0) or a pin name (e.g. PORTA.0).

RCTIME may be used to read a potentiometer (or some other resistive device). Resistance can be measured by discharging and timing the charge (or vice versa) of a capacitor through the resistor (typically 5K to 50K).

The resolution of **RCTIME** is dependent upon the oscillator frequency. If a 4MHz oscillator is used, the time in state is returned in 10us increments. If a 20MHz oscillator is used, the time in state will have a 2us resolution. Defining an OSC value has no effect on **RCTIME**. The resolution always changes with the actual oscillator speed.

If the pin never changes state, 0 is returned. **RCTIME** normally waits a maximum of 65535 counts before it determines there is no change of state. If it is desired to wait fewer or more counts before it stops looking for this change, a **DEFINE** can be added:

```
DEFINE PULSIN_MAX 1000
```

This **DEFINE** also affects **PULSIN** in the same manner.

```
Low PORTB.3              ' Discharge cap to start
Pause 10                 ' Discharge for 10ms
RCTIME PORTB.3,0,W0      ' Read potentiometer on
                         Pin3
```

5.70. READ

READ *Address*,{**WORD**}{**LONG**}*Var*{,*Var...*}

Read bytes, words and longs (if PBPL used) from the on-chip EEPROM at the specified *Address* and stores the result in *Var*. This instruction may only be used with a PIC MCU that has an on-chip EEPROM data area such as the PIC12F683, PIC16F84 and the 16F87x(A) series.

READ will not work on devices with on-chip I2C interfaced serial EEPROM like the PIC12CE67x and PIC16CE62x parts. Use the **I2CREAD** instruction instead.

```
READ 5,B0    ' Put the value at EEPROM location 5
             into B0
READ  0,Word W1,Word W2,B6
READ  10,Long L0   ' PBPL only
```

5.71. READCODE

READCODE *Address*,*Var*

Read the value at location *Address* in code space into *Var*.

Some PIC16F and PIC18 devices allow program code to be read at run-time. This may be useful for additional data storage or to verify the validity of the program code.

For PIC16F devices, 14-bit-sized data can be read from word code space *Address*es.

For PIC18 devices, byte or word-sized data can be read from byte (rather than word) code space *Address*es.

The listing file may be examined to determine program addresses.

```
READCODE $100,w     ' Put the code word at
                    location $100 into W
```

5.72. REPEAT..UNTIL

```
REPEAT
      Statement...
UNTIL Condition
```

REPEATedly execute *Statement*s **UNTIL** the specified *Condition* is true. When the *Condition* is true, execution continues at the statement following the **UNTIL**. *Condition* may be any comparison expression.

```
i = 0
REPEAT
      PORTB.0[i] = 0
      i = i + 1
UNTIL i > 7
```

5.73. RESUME

RESUME {*Label*}

Pick up where program left off after handling an interrupt. **RESUME** is similar to **RETURN** but is used at the end of a PICBASIC PRO interrupt handler.

If the optional *Label* is used, program execution will continue at *Label* instead of where it was when it was interrupted. This can, however, cause problems with the stack. If the device has a stack pointer that is not accessible by the program, like a PIC16F877A, any other return addresses on the stack will no longer be accessible. If the stack pointer is accessible as it is on the 14-bit enhanced core and the PIC18 devices, it is cleared to 0 before the jump to *Label* is executed. If you would rather manipulate the stack pointer yourself, the following **DEFINE** keeps the compiler from clearing it:

```
DEFINE NO_CLEAR_STKPTR 1
```

See **ON INTERRUPT** for more information.

```
clockint:   seconds = seconds + 1    ' Count time
            RESUME                   ' Return to program after
                                     interrupt

error:      High errorled            ' Turn on error LED
            RESUME restart           ' Resume somewhere else
```

5.74. RETURN

RETURN

Return from subroutine. **RETURN** resumes execution at the statement following the **GOSUB** which called the subroutine.

```
      Gosub sub1   ' Go to subroutine labeled sub1
            ...
sub1: Serout 0,N2400,["Lunch"]       ' Send "Lunch" out
                                     Pin0 serially
      RETURN       ' Return to main program after Gosub
```

5.75. REVERSE

REVERSE *Pin*

If *Pin* is an input, it is made an output. If *Pin* is an output, it is made an input. *Pin* may be a constant, 0 - 15, or a variable that contains a number 0 - 15 (e.g. B0) or a pin name (e.g. PORTA.0).

```
Output 4     ' Make Pin4 an output
REVERSE 4    ' Change Pin4 to an input
```

5.76. SELECT CASE

```
SELECT CASE Var
      CASE Expr1 {, Expr...}
            Statement...
      CASE Expr2 {, Expr...}
            Statement...
      {CASE ELSE
            Statement...}
END SELECT
```

CASE statements are sometimes easier to use than multiple **IF..THEN**s. These statements are used to compare a variable with different values or ranges of values, and take action based on the value.

The *Var*iable to be used in all of the comparisons is specified in the **SELECT CASE** statement. Each **CASE** is followed by the *Statement*s to be executed if the **CASE** is true. **IS** may be used to specify a comparison other than equal to. If none of the **CASE**s are true, the *Statement*s under the optional **CASE ELSE** statement are executed. An **END SELECT** closes the **SELECT CASE**.

```
SELECT CASE x
      CASE 1
            y = 10
      CASE 2, 3
            y = 20
      CASE IS > 5
            y = 100
      CASE ELSE
            y = 0
END SELECT
```

5.77. SERIN

SERIN
Pin,*Mode*,{*Timeout*,*Label*,}{**[***Qual...***]**,}{*Item...*}

Receive one or more *Item*s on *Pin* in standard asynchronous format using 8 data bits, no parity and one stop bit (8N1). **SERIN** is similar to the BS1 Serin command with the addition of a *Timeout*. *Pin* is automatically made an input. *Pin* may be a constant, 0 - 15, or a variable that contains a number 0 - 15 (e.g. B0) or a pin name (e.g. PORTA.0).

The *Mode* names (e.g. **T2400**) are defined in the file MODEDEFS.BAS. To use them, add the line:

```
Include "modedefs.bas"
```

to the top of the PICBASIC PRO program. BS1DEFS.BAS and BS2DEFS.BAS already includes MODEDEFS.BAS. Do not include it again if one of these files is already included. The *Mode* numbers may be used without including this file.

Mode	*Mode* No.	Baud Rate	State
T2400	0	2400	True
T1200	1	1200	
T9600	2	9600	
T300	3	300	
N2400	4	2400	Inverted
N1200	5	1200	
N9600	6	9600	
N300	7	300	

An optional *Timeout* and *Label* may be included to allow the program to continue if a character is not received within a certain amount of time. *Timeout* is specified in 1 millisecond units. If the serial input pin stays in the idle state during the *Timeout* time, the program will exit the **SERIN** command and jump to *Label*.

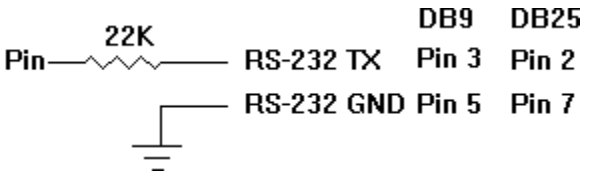
The list of data items to be received may be preceded by one or more

qualifiers enclosed within brackets. **SERIN** must receive these bytes in exact order before receiving the data items. If any byte received does not match the next byte in the qualifier sequence, the qualification process starts over (i.e. the next received byte is compared to the first item in the qualifier list). A *Qualifier* can be a constant, variable or a string constant. Each character of a string is treated as an individual qualifier.

Once the qualifiers are satisfied, **SERIN** begins storing data in the variables associated with each *Item*. If the variable name is used alone, the value of the received ASCII character is stored in the variable. If variable is preceded by a pound sign (#), **SERIN** converts a decimal value in ASCII and stores the result in that variable. All non-digits received prior to the first digit of the decimal value are ignored and discarded. The non-digit character which terminates the decimal value is also discarded. The decimal value received may not be greater than 65535, even when a long variable is specified.

SERIN assumes a 4MHz oscillator when generating its bit timing. To maintain the proper baud rate timing with other oscillator values, be sure to **DEFINE** the OSC setting to the new oscillator value.

While single-chip RS-232 level converters are common and inexpensive, the excellent I/O specifications of the PIC MCU allow most applications to run without level converters. Rather, inverted input (**N300**..**N9600**) can be used is conjunction with a current limiting resistor.




```
' Wait until the character "A" is received
serially on Pin1 and put next character into B0
SERIN 1,N2400,["A"],B0
```

5.78. SERIN2

SERIN2 *DataPin*{*FlowPin*},*Mode*,{*ParityLabel*,}
{*Timeout*,*Label*,}[*Item...*]

Receive one or more *Item*s on *Pin* in standard asynchronous format. **SERIN2** is similar to the BS2 Serin command. *DataPin* is automatically made an input. The optional *FlowPin* is automatically made an output. *DataPin* and *FlowPin* may be a constant, 0 - 15, or a variable that contains a number 0 - 15 (e.g. B0) or a pin name (e.g. PORTA.0).

The optional flow control pin, *FlowPin*, may be included to help keep data from overrunning the receiver. If it is used, *FlowPin* is automatically set to the enabled state to allow transmission of each character. This enabled state is determined by the polarity of the data specified by *Mode*.

Mode is used to specify the baud rate and operating parameters of the serial transfer. The low order 13 bits select the baud rate. Bit 13 selects parity or no parity. Bit 14 selects inverted or true level. Bit 15 is not used.

The baud rate bits specify the bit time in microseconds - 20. To find the value for a given baud rate, use the equation:

(1000000 / baud) - 20

Some standard baud rates are listed in the following table.

Baud Rate	Bits 0 - 12
300	3313
600	1646
1200	813
2400	396
4800	188
9600*	84
19200*	32
38400*	6

*Oscillator faster than 4MHz may be required.

Bit 13 selects parity (bit 13 = 1) or no parity (bit 13 = 0). Normally, the serial transmissions are 8N1 (8 data bits, no parity and 1 stop bit). If parity is selected, the data is received as 7E1 (7 data bits, even parity and 1 stop bit). To receive odd parity instead of even parity, include the following **DEFINE** in the program:

```
DEFINE SER2_ODD   1
```

Bit 14 selects the level of the data and flow control pins. If bit 14 = 0, the data is received in true form for use with RS-232 drivers. If bit14 = 1, the data is received inverted. This mode can be used to avoid installing RS-232 drivers.

Some examples of *Mode* are: *Mode* = 84 (9600 baud, no parity, true), *Mode* = 16780 (2400 baud, no parity, inverted), *Mode* = 27889 (300 baud, even parity, inverted). Appendix A shows more *Mode* examples.

If *ParityLabel* is included, this label will be jumped to if a character with bad parity is received. It should only be used if parity is selected (bit 13 = 1).

An optional *Timeout* and *Label* may be included to allow the program to continue if a character is not received within a certain amount of time. *Timeout* is specified in 1 millisecond units. If the serial input pin stays in the idle state during the *Timeout* time, the program will exit the **SERIN2** command and jump to *Label*.

A **DEFINE** allows the use of data bits other than 8 (or 7 with parity). SER2_BITS data bits may range from 4 bits to 8 (the default if no **DEFINE** is specified). Enabling parity uses one of the number of bits specified.

Defining SER2_BITS to 9 allows 8 bits to be read and written along with a 9th parity bit.

With parity disabled (the default):

```
DEFINE SER2_BITS 4      ' Set Serin2 and Serout2
                        data bits to 4
DEFINE SER2_BITS 5      ' Set Serin2 and Serout2
                        data bits to 5
DEFINE SER2_BITS 6      ' Set Serin2 and Serout2
                        data bits to 6
```

```
DEFINE SER2_BITS 7      ' Set Serin2 and Serout2
                        data bits to 7
DEFINE SER2_BITS 8      ' Set Serin2 and Serout2
                        data bits to 8 (default)
```

With parity enabled:

```
DEFINE SER2_BITS 5      ' Set Serin2 and Serout2
                        data bits to 4
DEFINE SER2_BITS 6      ' Set Serin2 and Serout2
                        data bits to 5
DEFINE SER2_BITS 7      ' Set Serin2 and Serout2
                        data bits to 6
DEFINE SER2_BITS 8      ' Set Serin2 and Serout2
                        data bits to 7 (default)
DEFINE SER2_BITS 9      ' Set Serin2 and Serout2
                        data bits to 8
```

SERIN2 supports many different data modifiers which may be mixed and matched freely within a single **SERIN2** statement to provide various input formatting.

Modifier	Operation
BIN{1..32}	Receive binary digits
DEC{1..10}	Receive decimal digits
HEX{1..8}	Receive upper case hexadecimal digits
SKIP *n*	Skip *n* received characters
STR ArrayVar*n***{***c***}**	Receive string of *n* characters optionally ended in character *c*
WAIT ( )	Wait for sequence of characters
WAITSTR ArrayVar{*n***}**	Wait for character string

1) A variable preceded by **BIN** will receive the ASCII representation of its binary value. For example, if **BIN B0** is specified and "1000" is received, B0 will be set to 8.
2) A variable preceded by **DEC** will receive the ASCII representation of its decimal value. For example, if **DEC B0** is specified and "123" is received, B0 will be set to 123.
3) A variable preceded by **HEX** will receive the ASCII representation of its hexadecimal value. For example, if **HEX B0** is specified and "FE" is received, B0 will be set to 254.

4) **SKIP** followed by a count less than 256, will skip that many characters in the input stream. For example, **SKIP 4** will skip 4 characters.
5) **STR** followed by a byte array variable, count and optional ending character will receive a string of characters. The length is determined by the count, less than 256, or when the optional character is encountered in the input.
6) The list of data items to be received may be preceded by one or more qualifiers between parenthesis after **WAIT**. **SERIN2** must receive these bytes in exact order before receiving the data items. If any byte received does not match the next byte in the qualifier sequence, the qualification process starts over (i.e. the next received byte is compared to the first item in the qualifier list). A *Qualifier* can be a constant, variable or a string constant. Each character of a string is treated as an individual qualifier.
7) **WAITSTR** can be used as **WAIT** above to force **SERIN2** to wait for a string of characters of an optional length before proceeding.

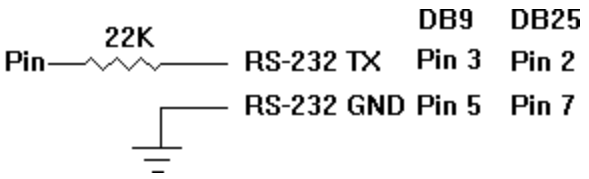
Once any **WAIT** or **WAITSTR** qualifiers are satisfied, **SERIN2** begins storing data in the variables associated with each *Item*. If the variable name is used alone, the value of the received ASCII character is stored in the variable. If variable is preceded by **BIN**, **DEC** or **HEX**, then **SERIN2** converts a binary, decimal or hexadecimal value in ASCII and stores the result in that variable. All non-digits received prior to the first digit of the decimal value are ignored and discarded. The non-digit character which terminates the value is also discarded.

BIN, **DEC** and **HEX** may be followed by a number. Normally, these modifiers receive as many digits as are in the input. However, if a number follows the modifier, **SERIN2** will always receive that number of digits, skipping additional digits as necessary.

SERIN2 assumes a 4MHz oscillator when generating its bit timing. To maintain the proper baud rate timing with other oscillator values, be sure to **DEFINE** the OSC setting to the new oscillator value. An oscillator speed faster than 4MHz may be required for reliable operation at 9600 baud and above.

While single-chip RS-232 level converters are common and inexpensive, thanks to current RS-232 implementation and the excellent I/O specifications of the PIC MCU, most applications don't require level converters. Rather, inverted TTL (*Mode* bit 14 = 1) can be used. A

current limiting resistor is suggested (RS-232 is suppose to be short-tolerant).



SERIN2 is not supported on 12-bit core PIC MCUs due to RAM and stack constraints.

```
' Wait until the character "A" is received
serially on Pin1 and put next character into B0
SERIN2 1,16780,[WAIT("A"),B0]

' Skip 2 chars and grab a 4 digit decimal number
SERIN2 PORTA.1,84,[SKIP 2,DEC4 B0]

SERIN2 PORTA.1\PORTA.0,84,100,tlabel,[WAIT("x",
b0),STR ar]
```

5.79. SEROUT

SEROUT *Pin*,*Mode*,[*Item*{,*Item...*}]

Sends one or more items to *Pin* in standard asynchronous format using 8 data bits, no parity and one stop (8N1). **SEROUT** is similar to the BS1 Serout command. *Pin* is automatically made an output. *Pin* may be a constant, 0 - 15, or a variable that contains a number 0 - 15 (e.g. B0) or a pin name (e.g. PORTA.0).

The *Mode* names (e.g. **T2400**) are defined in the file MODEDEFS.BAS. To use them, add the line:

```
Include "modedefs.bas"
```

to the top of the PICBASIC PRO program. BS1DEFS.BAS and BS2DEFS.BAS already includes MODEDEFS.BAS. Do not include it again if one of these files is already included. The *Mode* numbers may be used without including this file.

Mode	*Mode* No.	Baud Rate	State
T2400	0	2400	Driven True
T1200	1	1200	
T9600	2	9600	
T300	3	300	
N2400	4	2400	Driven Inverted
N1200	5	1200	
N9600	6	9600	
N300	7	300	
OT2400	8	2400	Open True*
OT1200	9	1200	
OT9600	10	9600	
OT300	11	300	
ON2400	12	2400	Open Inverted*
ON1200	13	1200	
ON9600	14	9600	
ON300	15	300	

* Open modes not supported on 12-bit core PIC MCUs.

SEROUT supports three different data types which may be mixed and matched freely within a single **SEROUT** statement.

1) A string constant is output as a literal string of characters.
2) A numeric value (either a variable or a constant) will send the corresponding ASCII character. Most notably, 13 is carriage return and 10 is line feed.
3) A numeric value preceded by a pound sign (#) will send the ASCII representation of its decimal value, up to 65535. For example, if W0 = 123, then #W0 (or #123) will send "1", "2", "3".

SEROUT assumes a 4MHz oscillator when generating its bit timing. To maintain the proper baud rate timing with other oscillator values, be sure to **DEFINE** the OSC setting to the new oscillator value.

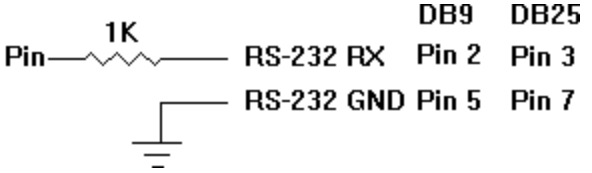
In some cases, the transmission rates of **SEROUT** instructions may present characters too quickly to the receiving device. A **DEFINE** adds

character pacing to the serial output transmissions. This allows additional time between the characters as they are transmitted. The character pacing **DEFINE** allows a delay of 1 to 65,535 microseconds (.001 to 65.535 milliseconds) between each character transmitted.

For example, to pause 1 millisecond between the transmission of each character:

```
DEFINE CHAR_PACING 1000
```

While single-chip RS-232 level converters are common and inexpensive, thanks to current RS-232 implementation and the excellent I/O specifications of the PIC MCU, most applications don't require level converters. Rather, inverted TTL (**N300**..**N9600**) can be used. A current limiting resistor is suggested (RS-232 is suppose to be short-tolerant).




```
SEROUT 0,N2400,[#B0,10] ' Send the ASCII value
                        of B0 followed by a
                        linefeed out Pin0
                        serially
```

5.80. SEROUT2

SEROUT2 *DataPin*{*FlowPin*},*Mode*,{*Pace*,}
{*Timeout*,*Label*,}[*Item...*]

Send one or more *Item*s to *DataPin* in standard asynchronous serial format. **SEROUT2** is similar to the BS2 Serout command. *DataPin* is automatically made an output. The optional *FlowPin* is automatically made an input. *DataPin* and *FlowPin* may be a constant, 0 - 15, or a variable that contains a number 0 - 15 (e.g. B0) or a pin name (e.g. PORTA.0).

The optional flow control pin, *FlowPin*, may be included to help keep data from overrunning the receiver. If it is used, the serial data will not be sent until *FlowPin* is in the proper state. This state is determined by the polarity of the data specified by *Mode*.

An optional *Timeout* and *Label* may be included to allow the program to continue if *FlowPin* does not change to the enabled state within a certain amount of time. *Timeout* is specified in units of 1 millisecond. If *FlowPin* stays disabled during the *Timeout* time, the program will exit the **SEROUT2** command and jump to *Label*.

In some cases, the transmission rates of **SEROUT2** instructions may present characters too quickly to the receiving device. It may not be desirable to use an extra pin for flow control. An optional *Pace* can be used to add character pacing to the serial output transmissions. This allows additional time between the characters as they are transmitted. The character pacing allows a delay of 1 to 65,535 milliseconds between each character transmitted.

Mode is used to specify the baud rate and operating parameters of the serial transfer. The low order 13 bits select the baud rate. Bit 13 selects parity or no parity. Bit 14 selects inverted or true level. Bit 15 selects whether it is driven or open.

The baud rate bits specify the bit time in microseconds - 20. To find the value for a given baud rate, use the equation:

(1000000 / baud) - 20

Some standard baud rates are listed in the following table.

Baud Rate	Bits 0 - 12
300	3313
600	1646
1200	813
2400	396
4800	188
9600*	84
19200*	32
38400*	6

*Oscillator faster than 4MHz may be required.

Bit 13 selects parity (bit 13 = 1) or no parity (bit 13 = 0). Normally, the serial transmissions are 8N1 (8 data bits, no parity and 1 stop bit). If parity is selected, the data is sent as 7E1 (7 data bits, even parity and 1 stop bit). To transmit odd parity instead of even parity, include the following **DEFINE** in the program:

```
DEFINE SER2_ODD   1
```

Bit 14 selects the level of the data and flow control pins. If bit 14 = 0, the data is sent in true form for use with RS-232 drivers. If bit14 = 1, the data is sent inverted. This mode can be used to avoid installing RS-232 drivers.

Bit 15 selects whether the data pin is always driven (bit 15 = 0), or is open in one of the states (bit 15 = 1). The open mode can be used to chain several devices together on the same serial bus.

See Appendix A for a table of *Mode* examples.

A **DEFINE** allows the use of data bits other than 8 (or 7 with parity). SER2_BITS data bits may range from 4 bits to 8 (the default if no **DEFINE** is specified). Enabling parity uses one of the number of bits specified. Defining SER2_BITS to 9 allows 8 bits to be read and written along with a 9th parity bit.

With parity disabled (the default):

```
DEFINE SER2_BITS 4        ' Set Serin2 and Serout2
                          data bits to 4
```

```
DEFINE SER2_BITS 5      ' Set Serin2 and Serout2
                        data bits to 5
DEFINE SER2_BITS 6      ' Set Serin2 and Serout2
                        data bits to 6
DEFINE SER2_BITS 7      ' Set Serin2 and Serout2
                        data bits to 7
DEFINE SER2_BITS 8      ' Set Serin2 and Serout2
                        data bits to 8 (default)
```

With parity enabled:

```
DEFINE SER2_BITS 5      ' Set Serin2 and Serout2
                        data bits to 4
DEFINE SER2_BITS 6      ' Set Serin2 and Serout2
                        data bits to 5
DEFINE SER2_BITS 7      ' Set Serin2 and Serout2
                        data bits to 6
DEFINE SER2_BITS 8      ' Set Serin2 and Serout2
                        data bits to 7 (default)
DEFINE SER2_BITS 9      ' Set Serin2 and Serout2
                        data bits to 8
```

SEROUT2 supports many different data modifiers which may be mixed and matched freely within a single **SEROUT2** statement to provide various output formatting.

Modifier	Operation
{I}{S}BIN{1..32}	Send binary digits
{I}{S}DEC{1..10}	Send decimal digits
{I}{S}HEX{1..8}	Send hexadecimal digits
REP *cn***	Send character *c* repeated *n* times
STR ArrayVar{*n*}	Send string of *n* characters

Notes:

1) A string constant is output as a literal string of characters, for example “Hello”.
2) A numeric value (either a variable or a constant) will send the corresponding ASCII character. Most notably, 13 is carriage return and 10 is line feed.
3) A numeric value preceded by **BIN** will send the ASCII representation of its binary value. For example, if **B0** = 8, then **BIN B0** (or **BIN 8**) will send “1000".
4) A numeric value preceded by **DEC** will send the ASCII

representation of its decimal value. For example, if **B0** = 123, then **DEC B0** (or **DEC 123**) will send "123".

5) A numeric value preceded by **HEX** will send the ASCII representation of its hexadecimal value. For example, if **B0** = 254, then **HEX B0** (or **HEX 254**) will send "FE".

6) **REP** followed by a character and count less than 256, will repeat the character, count times. For example, **REP "0"\4** will send "0000".

7) **STR** followed by a byte array variable and optional count will send a string of characters. The length is determined by the count, less than 256, or when 0 is encountered in the string.

BIN, **DEC** and **HEX** may be preceded or followed by several optional parameters. If any of them are preceded by an **I** (for indicated), the output will be preceded by either a "**%**", "**#**" or "**$**" to indicate the following value is binary, decimal or hexadecimal.

If any are preceded by an **S** (for signed), the output will be sent preceded by a "**-**" if the high order bit of the data is set. This allows the transmission of negative numbers. Keep in mind that all of the math and comparisons in PBP and PBPW are unsigned (PBPL is signed). However, unsigned math can yield signed results. For example, take the case of **B0 = 9 - 10**. The result of **DEC B0** would be "255". Sending **SDEC B0** would give "-1" since the high order bit is set. So with a little trickery, the unsigned math of PBP can yield signed results.

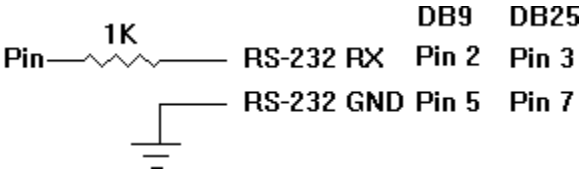
BIN, **DEC** and **HEX** may also be followed by a number. Normally, these modifiers display exactly as many digits as are necessary, zero blanked (leading zeros are not sent). However, if a number follows the modifier, **SEROUT2** will always send that number of digits, adding leading zeros as necessary. It will also trim off any extra high order digits. For example, **BIN6 8** would send "001000" and **BIN2 8** would send "00".

Any or all of the modifier combinations may be used at once. For example, **ISDEC4 B0.**

SEROUT2 assumes a 4MHz oscillator when generating its bit timing. To maintain the proper baud rate timing with other oscillator values, be sure to **DEFINE** the OSC setting to the new oscillator value. An oscillator speed faster than 4MHz may be required for reliable operation at 9600 baud and above.

While single-chip RS-232 level converters are common and inexpensive,

thanks to current RS-232 implementation and the excellent I/O specifications of the PIC MCU, most applications don't require level converters. Rather, inverted TTL (*Mode* bit 14 = 1) can be used. A current limiting resistor is suggested (RS-232 is suppose to be short-tolerant).



SEROUT2 is not supported on 12-bit core PIC MCUs due to RAM and stack constraints.

```
' Send the ASCII value of B0 followed by a
linefeed out Pin0 serially at 2400 baud
SEROUT2 0,16780,[DEC B0,10]

' Send "B0 =" followed by the binary value of B0
out PORTA pin 1 serially at 9600 baud
SEROUT2 PORTA.1,84,["B0=", IHEX4 B0]
```

5.81. SHIFTIN

SHIFTIN *DataPin*,*ClockPin*,*Mode*,[*Var*{*Bits*}...]

Clock *ClockPin*, synchronously shift in bits on *DataPin* and store the data received into *Var*. *ClockPin* and *DataPin* may be a constant, 0-15, or a variable that contains a number 0-15 (e.g. B0) or a pin name (e.g. PORTA.0).

Bits optionally specifies the number of bits to be shifted in. If it is not specified, 8 bits are shifted in, independent of the variable type. The *Bits* shifted in are always the low order bits, regardless of the *Mode* used, LSB or MSB.

The *Mode* names (e.g. **MSBPRE**) are defined in the file MODEDEFS.BAS. To use them, add the line:

```
Include "modedefs.bas"
```

to the top of the PICBASIC PRO program. BS1DEFS.BAS and BS2DEFS.BAS already includes MODEDEFS.BAS. Do not include it again if one of these files is already included. The *Mode* numbers may be used without including this file.. Some *Mode*s do not have a name.

For *Mode*s 0-3, the clock idles low, toggles high to clock in a bit, and then returns low. For *Mode*s 4-7, the clock idles high, toggles low to clock in a bit, and then returns high.

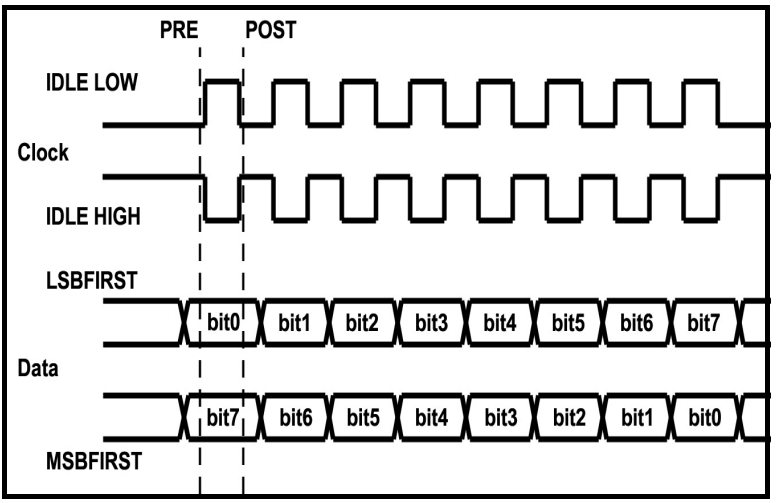
Mode	*Mode* No.	Operation
MSBPRE	**0**	Shift data in highest bit first, Read data before sending clock. Clock idles low.
LSBPRE	**1**	Shift data in lowest bit first, Read data before sending clock. Clock idles low.
MSBPOST	**2**	Shift data in highest bit first, Read data after sending clock. Clock idles low.
LSBPOST	**3**	Shift data in lowest bit first, Read data after sending clock. Clock idles low.
	4	Shift data in highest bit first, Read data before sending clock. Clock idles high.
	5	Shift data in lowest bit first, Read data before sending clock. Clock idles high.
	6	Shift data in highest bit first, Read data after sending clock. Clock idles high.
	7	Shift data in lowest bit first, Read data after sending clock. Clock idles high.

The shift clock runs at about 50kHz, dependent on the oscillator. The active state is held to a minimum of 2 microseconds. A **DEFINE** allows the active state of the clock to be extended by an additional number of microseconds up to 65,535 (65.535 milliseconds) to slow the clock rate. The minimum additional delay is defined by the **PAUSEUS** timing. See its section for the minimum for any given oscillator. This **DEFINE** is not available on 12-bit core PIC MCUs.

For example, to slow the clock by an additional 100 microseconds:

```
DEFINE SHIFT_PAUSEUS 100
```

The following diagram shows the relationship of the clock to the data for the various modes:




```
SHIFTIN 0,1,MSBPRE,[B0,B1\4]
```

5.82. SHIFTOUT

SHIFTOUT *DataPin*,*ClockPin*,*Mode*,[*Var*{*Bits*}...]

Synchronously shift out *Var* on *ClockPin* and *DataPin*. *ClockPin* and *DataPin* may be a constant, 0-15, or a variable that contains a number 0-15 (e.g. B0) or a pin name (e.g. PORTA.0).

Bits optionally specifies the number of bits to be shifted out. If it is not specified, 8 bits are shifted out, independent of the variable type. The *Bits* shifted out are always the low order bits, regardless of the *Mode* used, LSB or MSB. Up to 32 *Bits* can be shifted out of a single (long) variable. If more than 32 *Bits* are required, multiple variables or constants may be included between the square brackets.

The *Mode* names (e.g. **LSBFIRST**) are defined in the file MODEDEFS.BAS. To use them, add the line:

```
Include "modedefs.bas"
```

to the top of the PICBASIC PRO program. BS1DEFS.BAS and BS2DEFS.BAS already includes MODEDEFS.BAS. Do not include it again if one of these files is already included. The *Mode* numbers may be used without including this file. Some *Mode*s do not have a name.

For *Mode*s 0-1, the clock idles low, toggles high to clock in a bit, and then returns low. For *Mode*s 4-5, the clock idles high, toggles low to clock in a bit, and then returns high.

Mode	*Mode* No.	Operation
LSBFIRST	**0**	Shift data out lowest bit first. Clock idles low.
MSBFIRST	**1**	Shift data out highest bit first. Clock idles low.
	4	Shift data out lowest bit first. Clock idles high.
	5	Shift data out highest bit first. Clock idles high.

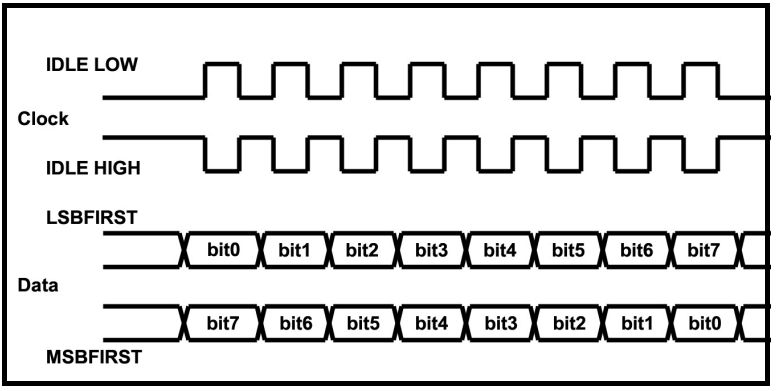
The shift clock runs at about 50kHz, dependent on the oscillator. The active state is held to a minimum of 2 microseconds. A **DEFINE** allows

the active state of the clock to be extended by an additional number of microseconds up to 65,535 (65.535 milliseconds) to slow the clock rate. The minimum additional delay is defined by the **PAUSEUS** timing. See its section for the minimum for any given oscillator. This **DEFINE** is not available on 12-bit core PIC MCUs.

For example, to slow the clock by an additional 100 microseconds:

```
DEFINE SHIFT_PAUSEUS 100
```

The following diagram shows the relationship of the clock to the data for the various modes:




```
SHIFTOUT 0,1,MSBFIRST,[B0,B1]
SHIFTOUT PORTA.1,PORTA.2,1,[wordvar\4]
SHIFTOUT PORTC.1,PORTB.1,4,[$1234\16, $56]
```

5.83. SLEEP

SLEEP *Period*

Place microcontroller into low power mode for *Period* seconds. *Period* is 16-bits using PBP and PBPW, so delays can be up to 65,535 seconds (just over 18 hours). For PBPL, *Period* is 32-bits so delays can be quite, quite long.

To achieve minimum current draw it may be necessary to turn off other peripherals on the device, such as the ADC, before executing the **SLEEP** command. See the Microchip data sheet for the specific device for information about these register settings.

SLEEP wakes up periodically using the Watchdog Timer to check to see if its time is up. If time is not up, it goes back to sleep until the next Watchdog Timer timeout and checks again. If a program is sleeping and waiting for some other event to wake it up, such as an interrupt, it may be more desirable to use the **NAP** command as it does not operate in **SLEEP**'s looped fashion.

SLEEP uses the Watchdog Timer so it is independent of the actual oscillator frequency. The granularity is about 2 seconds and may vary based on device specifics and temperature. This variance is unlike the BASIC Stamp. The change was necessitated because when the PIC MCU executes a Watchdog Timer reset, it resets many of the internal registers to predefined values. These values may differ greatly from what your program may expect. By running the **SLEEP** command uncalibrated, this issue is sidestepped.

```
SLEEP 60     ' Sleep for about 1 minute
```

5.84. SOUND

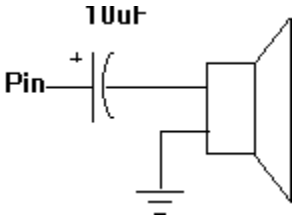
SOUND *Pin*,**[***Note*,*Duration*{,*Note*,*Duration...*}**]**

Generates tone and/or white noise on the specified *Pin*. *Pin* is automatically made an output. *Pin* may be a constant, 0 - 15, or a variable that contains a number 0 - 15 (e.g. B0) or a pin name (e.g. PORTA.0).

Note 0 is silence. *Notes* 1-127 are tones. *Notes* 128-255 are white noise. Tones and white noises are in ascending order (i.e. 1 and 128 are the lowest frequencies, 127 and 255 are the highest). *Note* 1 is about 78.74Hz and *Note* 127 is about 10,000Hz.

Duration is 0-255 and determines how long the *Note* is played in about 12 millisecond increments. *Note* and *Duration* needn't be constants.

SOUND outputs TTL-level square waves. Thanks to the excellent I/O characteristics of the PIC MCU, a speaker can be driven through a capacitor. The value of the capacitor should be determined based on the frequencies of interest and the speaker load. Piezo speakers can be driven directly.




```
SOUND PORTB.7,[100,10,50,10]   ' Send 2 sounds
                               consecutively to
                               Pin7
```

5.85. STOP

STOP

Stop program execution by executing an endless loop. This does not place the microcontroller into low power mode. The microcontroller is still working as hard as ever. It is just not getting much done.

```
STOP  ' Stop program dead in its tracks
```

5.86. SWAP

SWAP *Variable,Variable*

Exchange the values between 2 variables. Usually, it is a tedious process to swap the value of 2 variables. **SWAP** does it in one statement without using any intermediate variables. It can be used with bit, byte, word and long variables. Array variables with a variable index may not be used in **SWAP** although array variables with a constant index are allowed.

```
temp = B0          ' Old way
B0 = B1
B1 = temp

SWAP  B0,B1        ' One line way
```

5.87. TOGGLE

TOGGLE *Pin*

Invert the state of the specified *Pin*. *Pin* is automatically made an output. *Pin* may be a constant, 0 - 15, or a variable that contains a number 0 - 15 (e.g. B0) or a pin name (e.g. PORTA.0).

```
Low 0          ' Start Pin0 as low
TOGGLE 0       ' Change state of Pin0 to high
```

5.88. USBIN

USBIN *Endpoint*,*Buffer*,*Countvar*,*Label*

Get any available USB data for the *Endpoint* and place it in the *Buffer*. *Buffer* must be a byte array of suitable length to contain the data. *Countvar* should be set to the size of the buffer before **USBIN** is executed. It will contain the number of bytes transferred to the buffer. *Label* will be jumped to if no data is available.

This instruction may only be used with a PIC MCU that has an on-chip USB port such as the low-speed PIC16C745 and 765, and the full-speed PIC18F2550 and 4550.

The USB and USB18 subdirectories contain the modified Microchip USB libraries as well as example programs. USB programs require several additional files to operate (which are in the USB or USB18 subdirectory), some of which will require modification for your particular application. See the text file in the subdirectory for more information on the USB commands. The USB subdirectory is for the low-speed PIC16C devices and the USB18 subdirectory is for the full-speed PIC18F devices.

USB communications is much more complicated than synchronous (**SHIFTIN** and **SHIFTOUT**) and asynchronous (**SERIN**, **SEROUT** and so forth) communications. There is much more to know about USB operation that can possibly be described here. The USB information on the Microchip web site needs to be studied. Also, the book "USB Complete" by Jan Axelson may be helpful.

```
cnt = 8
USBIN 1, buffer, cnt, idleloop
```

5.89. USBINIT

```
USBINIT
```

USBINIT needs to be one of the first statements in a program that uses USB communications. It will initialize the USB portion of the PIC MCU.

This instruction may only be used with a PIC MCU that has an on-chip USB port such as the low-speed PIC16C745 and 765, and the full-speed PIC18F2550 and 4550.

The USB and USB18 subdirectories contain the modified Microchip USB libraries as well as example programs. USB programs require several additional files to operate (which are in the USB or USB18 subdirectory), some of which will require modification for your particular application. See the text file in the subdirectory for more information on the USB commands. The USB subdirectory is for the low-speed PIC16C devices and the USB18 subdirectory is for the full-speed PIC18F devices.

USB communications is much more complicated than synchronous (**SHIFTIN** and **SHIFTOUT**) and asynchronous (**SERIN**, **SEROUT** and so forth) communications. There is much more to know about USB operation that can possibly be described here. The USB information on the Microchip web site needs to be studied. Also, the book "USB Complete" by Jan Axelson may be helpful.

```
USBINIT
```

5.90. USBOUT

USBOUT *Endpoint*,*Buffer*,*Count*,*Label*

Take *Count* number of bytes from the array variable *Buffer* and send them to the USB *Endpoint*. If the USB buffer does not have room for the data because of a pending transmission, no data will be transferred and program execution will continue at *Label*.

This instruction may only be used with a PIC MCU that has an on-chip USB port such as the low-speed PIC16C745 and 765, and the full-speed PIC18F2550 and 4550.

The USB and USB18 subdirectories contain the modified Microchip USB libraries as well as example programs. USB programs require several additional files to operate (which are in the USB or USB18 subdirectory), some of which will require modification for your particular application. See the text file in the subdirectory for more information on the USB commands. The USB subdirectory is for the low-speed PIC16C devices and the USB18 subdirectory is for the full-speed PIC18F devices.

USB communications is much more complicated than synchronous (**SHIFTIN** and **SHIFTOUT**) and asynchronous (**SERIN**, **SEROUT** and so forth) communications. There is much more to know about USB operation that can possibly be described here. The USB information on the Microchip web site needs to be studied. Also, the book "USB Complete" by Jan Axelson may be helpful.

```
USBOUT 1, buffer,4,outloop
```

5.91. USBSERVICE

USBSERVICE

USBSERVICE needs to be executed repeatedly in the program. Since the USB code provided for the full-speed PIC18F devices is polled rather than interrupt driven, **USBSERVICE** needs to be executed at least every 10ms throughout the program. If it is not, the device may drop off the USB bus.

When interacting with Windows, at the beginning of the program after **USBINIT**, it is required that **USBSERVICE** be polled at about 250 us per loop. Even 1ms may be too slow. It can take up to 5 seconds to complete the initial interaction to get to the state of usb_device_state == CONFIGURED_STATE, but many times it will complete much more quickly. Then, you have to continue to poll **USBSERVICE** to complete the passing of the HID data to the host. Doing it for another 0.5 seconds seems to be adequate.

This instruction may only be used with a PIC MCU that has an on-chip full-speed USB port such as the PIC18F2550 and 4550.

The USB18 subdirectory contains the modified Microchip USB libraries as well as example programs. USB programs require several additional files to operate (which are in the USB18 subdirectory), some of which will require modification for your particular application. See the text file in the subdirectory for more information on the USB commands.

USB communications is much more complicated than synchronous (**SHIFTIN** and **SHIFTOUT**) and asynchronous (**SERIN**, **SEROUT** and so forth) communications. There is much more to know about USB operation that can possibly be described here. The USB information on the Microchip web site needs to be studied. Also, the book "USB Complete" by Jan Axelson may be helpful.

USBSERVICE

5.92. WHILE..WEND

```
WHILE Condition
      Statement...
WEND
```

Repeatedly execute *Statement*s **WHILE** *Condition* is true. When the *Condition* is no longer true, execution continues at the statement following the **WEND**. *Condition* may be any comparison expression.

```
i = 1
WHILE i <= 10
      Serout 0,N2400,["No:",#i,13,10]
      i = i + 1
WEND
```

5.93. WRITE

WRITE *Address*,{**WORD**}{**LONG**}*Value* {,*Value...*}

Write byte, word or long (if PBPL used) *Value*s to the on-chip EEPROM at the specified *Address*. This instruction may only be used with a PIC MCU that has an on-chip EEPROM data area such as the PIC12F683, PIC16F84 and the 16F87x(A) series.

WRITE is used to set the values of the on-chip EEPROM at runtime. To set the values of the on-chip EEPROM at device programming-time, use the **DATA** or **EEPROM** statement.

Each **WRITE** is self-timed and takes up to 10 milliseconds to execute on a PIC MCU.

For 12-bit core devices that support flash data memory, like the PIC12F519 and PIC16F526, **ERASECODE** must be used to erase the rows of memory before it can be rewritten using **WRITE**. See **ERASECODE** for more information.

If interrupts are used in a program, they must be turned off (masked, not **DISABLE**d) before executing a **WRITE**, and turned back on (if desired) after the **WRITE** instruction is complete. An interrupt occurring during a **WRITE** may cause it to fail. The following **DEFINE** turns interrupts off and then back on within a **WRITE** command. Do not use this **DEFINE** if interrupts are not used in the program.

```
DEFINE WRITE_INT 1
```

WRITE will not work on devices with on-chip I2C interfaced serial EEPROM like the PIC12CE67x and PIC16CE62x parts. Use the **I2CWRITE** instruction instead.

```
WRITE 5,B0  ' Send value in B0 to EEPROM
            location 5
WRITE 0,Word W1,Word W2,B6
WRITE 10,Long L0  ' PBPL only
```

5.94. WRITECODE

WRITECODE *Address*,*Value*

Write *Value* to the code space at location *Address*.

Some PIC16F and PIC18F devices allow program code to be written at run-time. While writing self-modifying code can be a dangerous technique, it does allow non-volatile data storage in devices that do not have on-chip EEPROM or when more than the 64 - 1024 bytes that on-chip EEPROM provides is not enough. However, one must be very careful not to write over active program memory.

The listing file may be examined to determine program addresses.

For PIC16F devices, 14-bit-sized data can be written to word code space *Address*es.

For PIC18F devices, byte or word-sized data can be written to byte (rather than word) code space *Address*es. The variable size of *Value* determines the number of bytes written. Bit- and byte-sized variables write 1 byte. Word- and long-size variables write 2 bytes to 2 sequential locations.

For block accessed devices, like the PIC16F877a and PIC18F452, a complete block must be written at once. This write block size is different for different PIC MCUs. See the Microchip data sheet for the particular device for information on the block size.

Additionally, some flash PIC MCUs, like the PIC18F series, require a portion of the code space to be erased before it can be rewritten with **WRITECODE**. See the section on **ERASECODE** for more information.

If interrupts are used in a program, they must be turned off (masked, not **DISABLE**d) before executing a **WRITECODE**, and turned back on (if desired) after the **WRITECODE** instruction is complete. An interrupt occurring during a **WRITECODE** may cause it to fail. The following **DEFINE** turns interrupts off and then back on within a **WRITECODE** command. Do not use this **DEFINE** if interrupts are not used in the program.

```
DEFINE WRITE_INT 1
```

Flash program writes must be enabled in the configuration for the PIC MCU at device programming time for **WRITECODE** to be able to write.

```
WRITECODE $100,w   ' Send value in W to code
                   space location $100
```

5.95. XIN

XIN *DataPin*,*ZeroPin*,{*Timeout*,*Label*,}[*Var*{,...}]

Receive X-10 data and store the House Code and Key Code in *Var*.

XIN is used to receive information from X-10 devices that can send such information. X-10 modules are available from a wide variety of sources under several trade names. An interface is required to connect the microcontroller to the AC power line. The TW-523 for two-way X-10 communications is required by **XIN**. This device contains the power line interface and isolates the microcontroller from the AC line. Since the X-10 format is patented, this interface also covers the license fees.

DataPin is automatically made an input to receive data from the X-10 interface. *ZeroPin* is automatically made an input to receive the zero crossing timing from the X-10 interface. Both pins should be pulled up to 5 volts with 4.7K resistors. *DataPin* and *ZeroPin* may be a constant, 0-15, or a variable that contains a number 0-15 (e.g. B0) or a pin name (e.g. PORTA.0).

An optional *Timeout* and *Label* may be included to allow the program to continue if X-10 data is not received within a certain amount of time. *Timeout* is specified in AC power line half-cycles (approximately 8.33 milliseconds).

XIN only processes data at each zero crossing of the AC power line as received on *ZeroPin*. If there are no transitions on this line, **XIN** will effectively wait forever.

If *Var* is word-sized, each House Code received is stored in the upper byte of the word. Each received Key Code is stored in the lower byte of the word. If *Var* is a byte, only the Key Code is stored.

The House Code is a number between 0 and 15 that corresponds to the House Code set on the X-10 module A through P.

The Key Code can be either the number of a specific X-10 module or the function that is to be performed by a module. In normal practice, first a command specifying the X-10 module number is sent, followed by a command specifying the function desired. Some functions operate on all modules at once so the module number is unnecessary. Hopefully, later examples will clarify things. Key Code numbers 0-15 correspond to

module numbers 1-16.

These Key Code numbers are different from the actual numbers sent and received by the X10 modules. This difference is to match the Key Codes in the BS2. To remove this Stamp translation, the following **DEFINE** may be used:

```
DEFINE XINXLAT_OFF 1
```

XIN is not supported on 12-bit core PIC MCUs due to RAM and stack constraints.

XOUT below lists the functions as well as the wiring information.

```
housekey    Var   Word

      ' Get X-10 data
mianloop: XIN PORTA.2,PORTA.0,[housekey]

      ' Display X-10 data on LCD
      Lcdout $fe,1,"House=",#housekey.byte1,
      "Key=",#housekey.byte0

      Goto mainloop     ' Do it forever


      ' Check for X-10 data, go to nodata if none
      XIN PORTA.2,PORTA.0,1,nodata,[housekey]
```

5.96. XOUT

XOUT *DataPin,ZeroPin,*
[*HouseCode\KeyCode{\Repeat}{,...}***]**

Send *HouseCode* followed by *KeyCode*, *Repeat* number of times in X-10 format. If the optional *Repeat* is left off, 2 times (the minimum) is assumed. *Repeat* is usually reserved for use with the Bright and Dim commands.

XOUT is used to send control information to X-10 modules. These modules are available from a wide variety of sources under several trade names. An interface is required to connect the microcontroller to the AC power line. Either the PL-513 for send only, or the TW-523 for two-way X-10 communications are required. These devices contain the power line interface and isolate the microcontroller from the AC line. Since the X-10 format is patented, these interfaces also cover the license fees.

DataPin is automatically made an output to send data to the X-10 interface. *ZeroPin* is automatically made an input to receive the zero crossing timing from the X-10 interface. It should be pulled up to 5 volts with a 4.7K resistor. *DataPin* and *ZeroPin* may be a constant, 0-15, or a variable that contains a number 0-15 (e.g. B0) or a pin name (e.g. PORTA.0).

XOUT only processes data at each zero crossing of the AC power line as received on *ZeroPin*. If there are no transitions on this line, **XOUT** will effectively wait forever.

HouseCode is a number between 0 and 15 that corresponds to the House Code set on the X-10 module A through P. The proper *HouseCode* must be sent as part of each command.

The *KeyCode* can be either the number of a specific X-10 module or the function that is to be performed by a module. In normal practice, first a command specifying the X-10 module number is sent, followed by a command specifying the function desired. Some functions operate on all modules at once so the module number is unnecessary. Hopefully, later examples will clarify things. *KeyCode* numbers 0-15 correspond to module numbers 1-16.

The *KeyCode* (function) names (e.g. **unitOn**) are defined in the file MODEDEFS.BAS. To use them, add the line:

```
Include "modedefs.bas"
```

to the top of the PICBASIC PRO program. BS1DEFS.BAS and BS2DEFS.BAS already includes MODEDEFS.BAS. Do not include it again if one of these files is already included. The *KeyCode* numbers may be used without including this file.

KeyCode	*KeyCode* No.	Operation
unitOn	**%10010**	Turn module on
unitOff	**%11010**	Turn module off
unitsOff	**%11100**	Turn all modules off
lightsOn	**%10100**	Turn all light modules on
lightsOff	**%10000**	Turn all light modules off
bright	**%10110**	Brighten light module
dim	**%11110**	Dim light module

These *Keycode* numbers are different from the actual numbers sent and received by the X10 modules. This difference is to match the *Keycode*s in the BS2. To remove this Stamp translation, the following **DEFINE** may be used:

```
DEFINE XOUTXLAT_OFF 1
```

XOUT is not supported on 12-bit core PIC MCUs due to RAM and stack constraints.

Wiring to the X-10 interfaces requires 4 connections. Output from the X-10 interface (zero crossing and receive data) are open-collector and require a pull up resistor of around 4.7K to 5 volts. Wiring tables for each interface is shown below:

PL-513 Wiring

Wire No.	Wire Color	Connection
1	Black	Zero crossing output
2	Red	Zero crossing common
3	Green	X-10 transmit common
4	Yellow	X-10 transmit input

TW-523 Wiring

Wire No.	Wire Color	Connection
1	Black	Zero crossing output
2	Red	Common
3	Green	X-10 receive output
4	Yellow	X-10 transmit input

```
house Var   Byte
unit  Var   Byte

      Include "modedefs.bas"

      house = 0               ' Set house to 0 (A)
      unit = 8                ' Set unit to 8 (9)
      ' Turn on unit 8 in house 0
      XOUT PORTA.1,PORTA.0,[house\unit,house\unitOn]

      ' Turn off all the lights in house 0
      XOUT PORTA.1,PORTA.0,[house\lightsOff]

      ' Blink light 0 on and off every 10 seconds
      XOUT PORTA.1,PORTA.0,[house\0]

blinkloop: XOUT PORTA.1,PORTA.0,[house\unitOn]
      Pause 10000       ' Wait 10 seconds

      XOUT PORTA.1,PORTA.0,[house\unitOff]
      Pause 10000       ' Wait 10 seconds

      Goto blinkloop
```

6. Structure of a Compiled Program

PBP is designed to be easy to use. Programs can be compiled and run with little thought to PBP's internal workings. Some people, however, only have confidence in a product when they understand its internal workings. Others are just plain curious.

This section is for them. It describes the files used and output generated by PBP and gives some idea of what is going on.

6.1. Target Specific Headers

Three target (PIC MCU) specific header files are used when a program is compiled. One is used by PBP, the other two are included for use by the assembler.

A file with the name of the microcontroller followed by the extension `.BAS` contains chip specific information needed by PBP. This includes the memory profile of the chip, which library it uses, and includes for the definition of the variables it needs. For the PIC16F84, the default microcontroller, the file is named `16F84.BAS`.

A file with the name of the microcontroller followed by the extension `.INC` is included in the generated `.ASM` file to give the assembler information about the chip, including the default configuration parameters (oscillator mode, Watchdog Timer setting, etc.). For the PIC16F84, for example, the file is named `16F84.INC`.

Finally, the assembler has its own include file that defines the addresses of the microcontroller registers. This file is usually named something on the order of `M16F8x.INC` for PM and is in the INC subdirectory.

6.2. The Library Files

PBP includes a set of library files that contain all of the code and definition files for a particular group of microcontrollers. In the case of 14-bit core PIC MCUs, for example, these files start with the name `PBPPIC14`.

`PBPPIC14.LIB` contains all of the assembly language subroutines used by the compiler. `PBPPIC14.MAC` contains all of the macros that call these subroutines. Most PICBASIC PRO commands consist of a macro and, usually, an associated library subroutine.

PBPPIC14.RAM contains the **VAR** statements that allocate the memory needed by the library.

PIC14EXT.BAS contains the external definitions that tells PBP all of the 14-bit core PIC MCU register names.

6.3. PICBASIC PRO Generated Code

A PICBASIC PRO compiled program is built in several stages. First PBP creates the .ASM file. It then builds a custom .MAC file that contains only the macros from the macro library that are used in the .ASM file. If everything is error free up to this point, it launches the assembler.

The assembler generates its own set of files. These include the .HEX final output file and possibly listing and debugging files.

6.4. .ASM File Structure

The .ASM file has a very specific structure. Things must be done in a particular order for everything to work out properly.

The first item placed in the file is an equate defining which assembler is to be used, followed by an **INCLUDE** to tell the assembler which microprocessor is the target and give it some basic information, such as the configuration data.

Next, all of the variable allocations and aliasing is listed. EEPROM initialization is next, if called for.

An **INCLUDE** for the macro file is then placed in the file, followed by an **INCLUDE** for the library subroutines.

Finally, the actual program code is incorporated. This program code is simply a list of macros that were generated from the PICBASIC PRO lines.

7. Other PICBASIC PRO Considerations

7.1. How Fast is Fast Enough?

By default, the PICBASIC PRO Compiler generates programs intended to be run on a PIC MCU with a 4MHz crystal or ceramic resonator. All of the time-sensitive instructions assume a 1 microsecond instruction time for their delays. This allows a **PAUSE** 1000, for example, to wait 1 second and the **SERIN** and **SEROUT** command's baud rates to be accurate.

There are times, however, when it would be useful to run the PIC MCU at a frequency other than 4MHz. Even though the compiled programs move along at a pretty good clip, it might be nice to run them even faster. Or maybe it is desirable to do serial input or output at 19,200 baud or higher.

PICBASIC PRO programs may be run at clock frequencies other than 4MHz in a couple of different ways. The first is to simply use an oscillator other than 4MHz and don't tell PBP. This can be a useful technique if you pay attention to what happens to the time dependent instructions.

If you wish to run the serial bus at 19,200 as described above, you would simply clock the microcontroller with an 8MHz crystal rather than a 4MHz crystal. This, in effect, makes everything run twice as fast, including the **SERIN** and **SEROUT** commands. If you tell **SERIN** or **SEROUT** to run at 9600 baud, the doubling of the crystal speed will double the actual baud rate to 19,200 baud.

However, keep in mind commands such as **PAUSE** and **SOUND** will also run twice as fast. The **PAUSE** 1000 mentioned above would only wait .5 seconds with an 8MHz crystal before allowing program execution to continue.

The preferred technique is to use a different oscillator frequency and tell PBP of your intentions. This is done through the use of a **DEFINE**. **DEFINE** is used to tell PBP to use something other than its defaults.

Normally, PBP defaults to using a 4MHz oscillator. Adding the statement:

```
DEFINE OSC 8
```

near the beginning of the PICBASIC PRO program tells PBP an 8MHz oscillator will be used instead. The acceptable oscillator definitions are:

OSC	Minimum delay	Minimum delay PIC18
3 (3.58)	20us	20us**
4	24us	19us**
8	12us	9us**
10	8us	7us**
12	7us	5us**
16	5us	4us**
20	3us	3us**
24	3us	2us**
25,32,33	2us*	2us**
40,48,64	-	1us**

* PIC17 only.
** PIC18 only.

Telling PBP the oscillator frequency allows it to compensate and produce the correct timing for **COUNT**, **DEBUG**, **DEBUGIN**, **DTMFOUT**, **FREQOUT**, **HPWM**, **HSERIN**, **HSERIN2**, **HSEROUT**, **HSEROUT2**, **I2CREAD**, **I2CWRITE**, **LCDIN**, **LCDOUT**, **OWIN**, **OWOUT**, **PAUSE**, **PAUSEUS**, **SERIN**, **SERIN2**, **SEROUT**, **SEROUT2**, **SHIFTIN**, **SHIFTOUT**, **SOUND**, **XIN** and **XOUT**.

Changing the oscillator frequency may also be used to enhance the resolution of the **PULSIN**, **PULSOUT** and **RCTIME** instructions. At 4MHz these instructions operate with a 10 microsecond resolution. If a 20MHz crystal is used, the resolution is increased 5 times to 2 microseconds. For PBP and PBPW (not PBPL), there is a tradeoff, however. The pulse width is still measured to a 16-bit word variable. With a 2 microsecond resolution, the maximum measurable pulse width would be 131,070 microseconds. For PBPL, a 32-bit variable may be used to allow measurement of very long pulses.

Going the other direction and running with a 32.768kHz oscillator is problematic. It may be desirable to attempt this for reduced power consumption reasons and it will work to some extent. The **SERIN** and **SEROUT** commands will be unusable and the Watchdog Timer may cause the program to restart periodically. Experiment to find out if your particular application is possible at this clock speed. It doesn't hurt to try.

7.2. Configuration Settings

As mentioned earlier, the default configuration settings for a particular device is set in the `.INC` file with the same name as the device, e.g. `16F84.INC`. These settings can be changed at the time the device is physically programmed.

The oscillator defaults to XT on most devices. This is the setting for the default 4MHz oscillator. If a faster oscillator is used, this setting must be changed to HS.

The Watchdog Timer is enabled by PBP. It is used, along with the TMR0 prescaler, to support the **NAP** and **SLEEP** instructions. If neither of the instructions are used in a program, the Watchdog Timer may be disabled and the prescaler used for something else.

Code Protect defaults to off but may be set to on when the device is physically programmed. Do not code protect a windowed device.

See the Microchip data sheet for the particular device for the configuration data specific to that part.

7.3. RAM Usage

In general it is not necessary to know how RAM is allocated by PBP in the microcontroller. PBP takes care of all the details so the programmer doesn't have to. However there are times when this knowledge could be useful.

Variables are stored in the PIC MCU's RAM registers. The first available RAM location is $0C for the PIC16F84 and some of the smaller PIC MCUs, and $20 for the PIC16C74 and other larger PIC MCUs. Refer to the Microchip PIC MCU data books for the actual location of the start of the RAM registers for a given microcontroller.

The variables are assigned to RAM sequentially in a particular order. The order is long arrays first (if any), followed by word, byte and bit arrays. Then space is allocated for longs, words, bytes and finally individual bits. Bits are packed into bytes as possible. This order makes the best use of available RAM. (For PIC18 devices, arrays are allocated last.)

Arrays must fit entirely within one RAM bank on 12-bit, 14-bit or PIC17

devices. Arrays may span banks on PIC18 devices. Byte-, word- and long-sized arrays are only limited in length by the amount of available memory on PIC18 devices. The compiler will assure that arrays, as well as simple variables, will fit in memory before successfully compiling.

You can suggest to PBP a particular bank to place the variable in:

```
penny       VAR   WORD BANK0
nickel      VAR   BYTE BANK1
```

If specific bank requests are made, those are handled first. If there is not enough room in a requested bank, the first available space is used and a warning is issued.

You can even set specific addresses for variables. In most cases, it is better to let PBP handle the memory mapping for you. But in some cases, such as storage of the W register in an interrupt handler, it is necessary to define a fixed address. This may be done in a similar manner to bank selection:

```
w_store     VAR   BYTE $20
```

Several system variables, using about 24 bytes of RAM, are automatically allocated by the compiler for use by library subroutines. These variables are allocated in the file PBPPIC14.RAM and must be in bank 0 (bank A on PIC18 devices).

In the generated code, user variables are prepended with an underscore (_) while system variables have no underscore so that they do not interfere with each other.

```
R0    VAR   WORD SYSTEM
```

BASIC Stamp variables B0 - B25 and W0 - W12 are not automatically allocated. It is best to create your own variables using the **VAR** instruction. However if you want these variables to be created for you, simply include the appropriate file, BS1DEFS.BAS or BS2DEFS.BAS, at the beginning of the PICBASIC PRO program. These variables allocate space separate and apart from any other variables you may later create. This is different than the BS2 where using the canned variables and user created variables can get you into hot water.

Additional temporary variables may be generated automatically by the

compiler to help it sort out equations. A listing of these variables, as well as the entire memory map, may be seen in the generated .ASM or .LST file.

If there is not enough RAM memory available for the variables, an unable to fit variable in memory error message will be issued.

7.4. Reserved Words

Reserved words are simply that - words that are reserved for use by the compiler and may not be defined as either variable names or labels. These reserved words may be the names of commands, pseudo-ops, variable types, variables that the compiler uses internally or the names of the PIC MCU registers. An error will be generated if an attempt is made to re-declare any of these reserved words.

The pseudo-ops, variable types and commands keywords are listed in their appropriate sections and in Appendix C. The names of the PIC MCU registers are defined in the file PIC??EXT.BAS, where ?? is the core type. The internal registers used by the compiler are defined in files ending with .RAM. If the files BS1DEFS.BAS, BS2DEFS.BAS or MODEDEFS.BAS are included, the definitions inside essentially become reserved words and may not be redefined.

7.5. Life After 2K

Yes, there is life after 2K using the PICBASIC PRO Compiler.

PIC MCUs have a segmented code space. PIC MCU instructions in 14-bit core parts such as Call and Goto only have enough bits within them to address 2K of program space. To get to code outside the 2K boundary, the PCLATH register must be set before each Call or Goto.

PBP automatically sets these PCLATH bits for you. There are a few restrictions imposed, however. The PICBASIC PRO library must fit entirely into page 0 of code space (the first half of page 0 for 12-bit core devices). Normally this is not an issue as the library is the first thing in a PICBASIC PRO program and the entire library is smaller than 2K. However, attention must be payed to this issue if additional libraries are used.

Assembly language interrupt handlers must also fit into page 0 of code space. Putting them at the beginning of the PICBASIC PRO program

should make this work. See the upcoming section on assembly language for more information.

The addition of instructions to set the PCLATH bits does add overhead to the produced code. PBP will set the PCLATH bits before any Call or Goto instruction on 12-bit core PIC MCUs with more than 512 words of code space, 14-bit core devices with more than 2K of code space and PIC17 devices with more than 8K of code space.

There are specific PICBASIC PRO instructions to assist with the 2K issues.

BRANCHL was created to allow branching to labels that may be further than 1K locations away on PIC18 devices or on the other side of a page boundary for all other devices. If the PIC MCU has only one code page of program space, **BRANCH** may be used as it takes up less space than **BRANCHL**. If the microcontroller has more than one page of code space, and you cannot be certain that **BRANCH** will always act within the same page, use **BRANCHL**.

The assembler may issue a warning that a page boundary has been crossed. This is normal and is there to suggest that you check for any **BRANCH**es that may cross a page boundary.

7.6. 12-Bit Core Considerations

Because of the architecture of the 12-bit core PIC MCUs, programs compiled for them by PBP will, in general, be larger and slower than programs compiled for the other PIC MCU families. In many cases, choosing a device from one of these other families will be more appropriate. However, many useful programs can be written and compiled for the 12-bit core devices.

The two main programming limitations that will most likely occur are running out of RAM memory for variables and running past the first 256 word limit for the library routines. These limitations have made it necessary to eliminate some compiler commands and modify the operation of some others.

The compiler for 12-bit core PIC MCUs uses between 20 and 22 bytes of RAM for its internal variables, with additional RAM used for any necessary temporary variables. This RAM allocation includes a 4 level software stack so that the BASIC program can still nest **GOSUB**s up to 4

levels deep. Some PIC MCU devices only have 24 or 25 bytes of RAM so there is very little space for user variables on those devices. If the Unable to Fit Variable error message occurs during compilation, choose another PIC MCU with more general purpose RAM.

12-bit core PIC MCUs can call only into the first half (256 words) of a code page. Since the compiler's library routines are all accessed by calls, they must reside entirely in the first 256 words of the PIC MCU code space. Many library routines, such as **I2CREAD**, are fairly large. It may only take a few routines to overrun the first 256 words of code space. If it is necessary to use more library routines that will fit into the first half of the first code page, it will be necessary to move to a 14- or 16-bit core PIC MCU instead of the 12-bit core device.

8. Assembly Language Programming

Assembly language routines can be a useful adjunct to a PICBASIC PRO Compiler program. While in general most tasks can be done completely in BASIC, there are times when it might be necessary to do a particular task faster, or using a smaller amount of code space, or just differently than the compiler does it. At those times it is useful to have the capabilities of an in-line assembler.

It can be beneficial to write most of a program quickly using the PICBASIC PRO language and then sprinkle in a few lines of assembly code to increase the functionality. This additional code may be inserted directly into the PBP program or included as another file.

8.1. Two Assemblers - No Waiting

Upon execution, PBP first compiles the program into assembly language and then automatically launches an assembler. This converts the assembler output into the final `.HEX` file which can be programmed into a microcontroller.

Two different assemblers may be used with PBP: PM, our assembler, and MPASM, Microchip's assembler. There are benefits and drawbacks to using each assembler. PM is handy because it can be faster than MPASM and can assemble much larger programs in DOS. PM includes an 8051-style instruction set that is more intuitive than the Microchip mnemonics. For complete information on the PM Assembler, see the `PM.TXT` file on disk.

MPASM, on the other hand, has the capability of creating debugging files. These files contains additional information that can be very useful with simulators and emulators. MPASM is also more compatible with the wide variety of assembly language examples found on the web and in Microchip's data books.

PBP defaults to using PM. To use MPASM with PBP, install all of the MPASM files into their own subdirectory. This subdirectory must also be in the DOS `PATH`. See the file `MPLAB.TXT` on the disk and the microEngineering Labs, Inc. web site for more details.

If the command line option "-ampasmwin" is used, MPASM will be launched following compilation to complete the process. MPASM will display its own screen with its progress.

```
PBPW -ampasmwin filename
```

8.2. Programming in Assembly Language

PBP programs may contain a single line of assembly language preceded by an “at” symbol (@), or one or more lines of assembly code preceded by the **ASM** keyword and ended by the **ENDASM** keyword. Both keywords appear on their lines alone.

```
@       bsf     PORTA,0

Asm
        bsf     STATUS,RP0
        bcf     TRISA,0
        bcf     STATUS,RP0
Endasm
```

The lines of assembly are copied verbatim into the assembly output file. This allows the PBP program to use all of the facilities of the assembler. This also, however, requires that the programmer have some familiarity with the PBP libraries. PBP’s notational conventions are similar to other commercial compilers and should come as no shock to programmers experienced enough to attempt in-line assembly.

All identifier names defined in a PBP program are similarly defined in assembly, but with the name preceded with an underscore (_). This allows access to user variables, constants, and even labeled locations, in assembly:

```
B0      Var     Byte

Asm
        movlw 10
        movwf _B0
Endasm
```

Thus, any name defined in assembly starting with an underscore has the possibility of conflicting with a PBP generated symbol. If conflict is avoided, can these underscored assembly values be accessed from PBP? No. Remember, the underscored names generated by PBP are only shadows of the actual information defined in the compiler. Since in-line assembly is copied directly to the output file and not processed by the compiler, the compiler not only lacks any type or value information about assembly symbols, it is completely unaware that they exist. If variables are to be shared between assembly and PBP, you must define

the variables in PBP.

Just as underscored symbols have possible conflicts, so do symbols not starting with underscores. The problem is internal library identifiers. Luckily, most library identifiers contain a '**?**' or make reference to one of the working registers (such as **R0**). Avoiding such names should reduce problems. If you should have a name collision, the assembler will report the duplicate definitions as an error.

In assembly language the comment designator changes from the single quote (') in PICBASIC PRO to a semicolon (;).

```
' PICBASIC PRO comment
; Assembly language comment
```

8.3. Placement of In-line Assembly

PBP statements execute in order of their appearance in the source. The organization of the code is as follows: Starting at location 0, the reset vector, PBP inserts some startup code followed by a jump to INIT. Next, the called-for library subroutines are stuffed in. At the end of the library is INIT, where any additional initialization is completed. Finally, at the label MAIN, the compiled PICBASIC PRO statement code is added.

The first executable line that appears in the PICBASIC PRO source is where the program starts execution. That statement literally appears in memory right behind the controller's startup and library code, right after the MAIN label.

The tendency of programmers is to place their own library functions written using the in-line assembler either before or after their code. In light of the above explanation, this could create some obvious problems. If they appear early in the program, the assembly routines execute prior to any PBP instructions (some programmers will invariably exploit this feature). If they appear at the tail of the program, execution which "falls off the end" of the PBP statements may mysteriously find themselves unintentionally executing assembly routines.

There are a couple of deciding factors as to where might be the best place to insert assembly language subroutines. If the entire program fits into one code page, place your assembly routines after your PBP code. If you need to terminate your program, explicitly place an **END** or **STOP** statement at the end of your code rather than floating off into space.

If the program is longer than one code page, it could make more sense to put the assembly language routines at the beginning of the PBP program. This should ensure them of being in the first code page so that you know where to find them. This is the way assembly language interrupt routines should be handled.

If the routines are placed at the front, you must include a **GOTO** (or JMP) around the code to the first executable PBP statement. See the section on interrupts for an example of this.

The actual code for the assembly language routines may be included in your program or in a separate file. If a routine is used by only one particular PICBASIC PRO program, it would make sense to include the assembler code within the PBP source file. This routine can then be accessed using the **CALL** command.

If it is used by several different PBP programs, a separate file containing the assembly routines can simply be included at the appropriate place in the PICBASIC PRO source:

```
Asm
      Include "myasm.inc"
Endasm
```

8.4. Another Assembly Issue

PIC MCU registers are banked. PBP keeps track of which register bank it is pointing to at all times. It knows if it is pointing to a TRIS register, for example, it needs to change the bank select bits before it can access a PORT.

It also knows to reset the bank select bits to 0 before making a Call or a Goto. It does this because it can't know the state of the bank select bits at the new location. So anytime there is a change of locale or a label that can be called or jumped to, the bank select bits are zeroed.

It also resets the bank select bits before each **ASM** and the @ assembler shortcut. Once again, the assembler routine won't know the current state of the bits so they are set to a known state. The assembler code must be sure to reset the bank select bits before it exits, if it has altered them.

9. Interrupts

Interrupts can be a scary and useful way to make your program really difficult to debug.

Interrupts are triggered by hardware events, either an I/O pin changing state or a timer timing out and so forth. If enabled (which by default they aren't), an interrupt causes the processor to stop whatever it is doing and jump to a specific routine in the microcontroller called an interrupt handler.

Interrupts are not for the faint of heart. They can be very tricky to implement properly, but at the same time they can provide very useful functions. For example, an interrupt could be used to buffer serial input data behind the scenes while the main PICBASIC PRO program is off doing something else. (This particular usage would require a microcontroller with a hardware serial port.)

There are many ways to avoid using interrupts. Quickly polling a pin or register bit instead is usually fast enough to get the job done. Or you can check the value of an interrupt flag without actually enabling interrupts.

However, if you just gotta do it, here are some hints on how to go about it.

The PICBASIC PRO Compiler has two different mechanisms to handle interrupts. The first is simply to write the interrupt handler in assembler and tack it onto the front of a PBP program. The second method is to use the PICBASIC PRO statement **ON INTERRUPT**. Each method will be covered separately, after we talk about interrupts in general.

9.1. Interrupts in General

When an interrupt occurs, the PIC MCU stores the address of the next instruction it was supposed to execute on the stack and jumps to location 4. The first thing this means is that you need an extra location on the hardware stack, which is only 8 deep on the 14-bit core devices to begin with.

The PICBASIC PRO library routines can use up to 4 stack locations themselves. The remaining 4 (12 for 14-bit enhanced core and PIC17 and 27 for PIC18) are reserved for **CALL**s and nested BASIC **GOSUB**s. You must make sure that your **GOSUB**s are only nested 3 (11 for 14-bit

enhanced core and PIC17 and 26 for PIC18) deep at most with no **CALL**s within them in order to have a stack location available for the return address. If your interrupt handler uses the stack (by doing a Call or **GOSUB** itself for example), you'll need to have additional stack space available.

Once you have dealt with the stack issues, you need to enable the appropriate interrupts. This usually means setting the INTCON register. Set the necessary enable bits along with Global Interrupt Enable. For example:

```
INTCON = %10010000
```

enables the interrupt for RB0/INT. Depending on the actual interrupt desired, you may also need to set one of the PIE registers.

Refer to the Microchip PIC MCU data books for additional information on how to use interrupts. They give examples of storing processor context as well as all the necessary information to enable a particular interrupt. This data is invaluable to your success.

Finally, select the best technique with which to handle your particular interrupts.

9.2. Interrupts in BASIC

The easiest way to write an interrupt handler is to write it in PICBASIC PRO using the **ON INTERRUPT** statement. **ON INTERRUPT** tells PBP to activate its internal interrupt handling and to jump to your BASIC interrupt handler as soon as it can after receiving an interrupt. Which brings us the first issue.

Using **ON INTERRUPT**, when an interrupt occurs PBP simply flags the event and immediately goes back to what it was doing. It does not immediately vector to your interrupt handler. Since PBP statements are not re-entrant (PBP must finish the statement that is being executed before it can begin a new one) there could be considerable delay (latency) before the interrupt is handled.

As an example, lets say that the PICBASIC PRO program just started execution of a `Pause 10000` when an interrupt occurs. PBP will flag the interrupt and continue with the **PAUSE**. It could be up to 10 seconds later before the interrupt handler is executed. If it is buffering characters from

a serial port, many characters will be missed.

To minimize the problem, use only statements that don't take very long to execute. For example, instead of `Pause 10000`, use `Pause 1` in a long **FOR..NEXT** loop. This will allow PBP to complete each statement more quickly and handle any pending interrupts.

If interrupt processing needs to occur more quicky than can be provided by **ON INTERRUPT**, interrupts in assembly language should be used.

Exactly what happens when **ON INTERRUPT** is used is this: A short interrupt handler is placed at location 4 in the PIC MCU. This interrupt handler is simply a Return. What this does is send the program back to what it was doing before the interrupt occurred. It doesn't require any processor context saving. What it doesn't do is re-enable Global Interrupts as happens using an Retfie.

A Call to a short subroutine is placed before each statement in the PICBASIC PRO program once an **ON INTERRUPT** is encountered. This short subroutine checks the state of the Global Interrupt Enable bit. If it is off, an interrupt is pending so it vectors to the users interrupt handler. If it is still set, the program continues with the next BASIC statement, after which, the GIE bit is checked again, and so forth.

When the **RESUME** statement is encountered at the end of the BASIC interrupt handler, it sets the GIE bit to re-enable interrupts and returns to where the program was before the interrupt occurred. If **RESUME** is given a label to jump to, execution will continue at that location instead. All previous return addresses will be lost in this case.

DISABLE stops PBP from inserting the Call to the interrupt checker after each statement. This allows sections of code to execute without the possibility of being interrupted. **ENABLE** allows the insertion to continue.

A **DISABLE** should be placed before the interrupt handler so that it will not keep getting restarted by checking the GIE bit.

If it is desired to turn off interrupts for some reason after **ON INTERRUPT** is encountered, you must not turn off the GIE bit. Turning off this bit tells PBP an interrupt has happened and it will execute the interrupt handler forever. Instead set:

```
INTCON = $80
```

This disables all the individual interrupts but leaves the Global Interrupt Enable bit set.

9.3. Interrupts in Assembler

Interrupts in assembly language are a little trickier.

Since you have no idea of what the processor was doing when it was interrupted, you have no idea of the state of the W register, the STATUS flags, PCLATH or even what register page you are pointing to. If you need to alter any of these, and you probably will, you must save the current values so that you can restore them before allowing the processor to go back to what it was doing before it was so rudely interrupted. This is called saving and restoring the processor context.

If the processor context, upon return from the interrupt, is not left exactly the way you found it, all kinds of subtle bugs and even major system crashes can and will occur.

This of course means that you cannot even safely use the compiler's internal variables for storing the processor context. You cannot tell which variables are in use by the library routines at any given time.

You should create variables in the PICBASIC PRO program for the express purpose of saving W, the STATUS register and any other register that may need to be altered by the interrupt handler. These variables should not be otherwise used in the BASIC program.

While it seems a simple matter to save W in any RAM register, it is actually somewhat more complicated. The problem occurs in that you have no way of knowing what register bank you are pointing to when the interrupt happens. If you have reserved a location in Bank0 and the current register pointers are set to Bank1, for example, you could overwrite an unintended location. Therefore you must reserve a RAM register location in each bank of the device at the same offset.

As an example, let's choose the PIC16C74(A). It has 2 banks of RAM registers starting at $20 and $A0 respectively. To be safe, we need to reserve the same location in each bank. In this case we will choose the first location in each bank. A special construct has been added to the **VAR** command to allow this:

```
wsave Var   Byte $20 System
wsave1 Var  Byte $a0 System
```

This instructs the compiler to place the variable at a particular location in RAM. In this manner, if the save of W "punches through" to another bank, it will not corrupt other data.

The interrupt routine should be as short and fast as you can possibly make it. If it takes too long to execute, the Watchdog Timer could timeout and really make a mess of things.

The routine should end with an Retfie instruction to return from the interrupt and allow the processor to pick up where it left off in your PICBASIC PRO program.

A good place to put the assembly language interrupt handler is at the very beginning of your PICBASIC PRO program. A **GOTO** needs to be inserted before it to make sure it won't be executed when the program starts. See the example below for a demonstration of this.

If a 14-bit core PIC MCU has more than 2K of code space, an interrupt stub is automatically added that saves the W, STATUS and PCLATH registers into the variables wsave, ssave and psave, before going to your interrupt handler. Storage for these variables must be allocated in the BASIC program:

```
wsave Var   Byte $20 System
wsave1 Var  Byte $a0 System    ' If device has
                               RAM in bank1
wsave2 Var  Byte $120 System   ' If device has
                               RAM in bank2
wsave3 Var  Byte $1a0 System   ' If device has
                               RAM in bank3
ssave Var   Byte Bank0 System
psave Var   Byte Bank0 System
```

In any case, you must restore these registers at the end of your assembler interrupt handler. If the 14-bit core PIC MCU has 2K or less of code space, or it is an PIC18 device, the registers are not saved. Your interrupt handler must save and restore any used registers.

Finally, you need to tell PBP that you are using an assembly language interrupt handler and where to find it. This is accomplished with a **DEFINE**:

```
DEFINE INTHAND Label
```

For PIC18 parts, an additional **DEFINE** allows assigning the low priority interrupt handler label:

```
DEFINE INTLHAND Label
```

Label is the beginning of your interrupt routine. PBP will place a jump to this *Label* at location 4 in the PIC MCU.

```
' Assembly language interrupt example

led   Var   PORTB.1

wsave Var   Byte $20 System
ssave Var   Byte Bank0 System
psave Var   Byte Bank0 System


Goto start  ' Skip around interrupt handler

' Define interrupt handler
define INTHAND myint

' Assembly language interrupt handler
Asm
; Save W, STATUS and PCLATH registers
myint movwf wsave       ; <= 2K only
      swapf STATUS, W   ; <= 2K only
      clrf  STATUS      ; <= 2K only
      movwf ssave       ; <= 2K only
      movf  PCLATH, W   ; <= 2K only
      movwf psave       ; <= 2K only

; Insert interrupt code here
; Save and restore FSR if used

      bsf   _led  ; Turn on LED (for example)

; Restore PCLATH, STATUS and W registers
      movf  psave, W
      movwf PCLATH
      swapf ssave, W
      movwf STATUS
      swapf wsave, F
      swapf wsave, W
```

```
      retfie
Endasm

' PICBASIC PRO program starts here
start: Low led     ' Turn LED off

' Enable interrupt on PORTB.0
      INTCON = %10010000

waitloop: Goto waitloop ' Wait here till
                        interrupted
```

10. PICBASIC PRO / PICBASIC / Stamp Differences

Compatibility is a two-edged sword. And then there is the pointy end. PICBASIC PRO has made some concessions to usability and code size. Therefore we call it “BASIC Stamp like” rather than BASIC Stamp compatible. PBP has most of the BASIC Stamp I and II instruction set and syntax. However there are some significant differences.

The following sections discuss the implementation details of PBP programs that might present problems. It is hoped that if you do encounter problems, these discussions will help illuminate the differences and possible solutions.

10.1. Execution Speed

The largest potential problem is speed. Without the overhead of reading instructions from the serial EEPROM, many PBP instructions (such as **GOTO** and **GOSUB**) execute hundreds of times faster than their BASIC Stamp equivalents. While in many cases this is a benefit, programs whose timing has been developed empirically may experience problems.

The solution is simple - good programs don't rely on statement timing such as **FOR..NEXT** loops. Whenever possible, a program should use handshaking and other non-temporal synchronization methods. If delays are needed, statements specifically generating delays (**PAUSE**, **PAUSEUS**, **NAP** or **SLEEP**) should be used.

10.2. Digital I/O

Unlike the BASIC Stamp, PBP programs operate directly on the PORT and TRIS registers. While this has speed and RAM/ROM size advantages, there is one potential drawback.

Some of the I/O commands (e.g. **TOGGLE** and **PULSOUT**) perform read-modify-write operations directly on the PORT register. If two such operations are performed too close together and the output is driving an inductive or capacitive load, it is possible the operation will fail.

Suppose, for example, that a speaker is driven though a 10uF cap (just as with the **SOUND** command). Also suppose the pin is initially low and the programmer is attempting to generate a pulse using **TOGGLE** statements. The first command reads the pin's low level and outputs its

complement. The output driver (which is now high) begins to charge the cap. If the second operation is performed too quickly, it still reads the pin's level as low, even though the output driver is high. As such, the second operation will also drive the pin high.

In practice, this is not much of a problem. And those commands designed for these types of interfacing (**SOUND** and **POT**, for example) have built-in protection. This problem is not specific to PBP programs. This is a common problem for PIC MCU (and other microcontroller) programs and is one of the realities of programming hardware directly.

10.3. Low Power Instructions

When the Watchdog Timer time-out wakes a PIC MCU from sleep mode, execution resumes without disturbing the state of the I/O pins. For unknown reasons, when the BASIC Stamp resumes execution after a low power instruction (**NAP** or **SLEEP**), the I/O pins are disturbed for approximately 18 mSec. PBP programs make use of the PIC's I/O coherency. The **NAP** and **SLEEP** instructions do not disturb the I/O pins.

10.4. Missing PC Interface

Since PBP generated programs run directly on a PIC MCU, there is no need for the Stamp's PC interface pins (PCO and PCI). The lack of a PC interface does introduce some differences.

Without the Stamp's IDE running on a PC, there is no place to send debugging information. Debugging can still be accomplished by using one of the serial output instructions like **DEBUG** or **SEROUT** in conjunction with a terminal program running on the PC such as Hyperterm.

Without the PC to wake the PIC MCU from an **END** or **STOP** statement, it remains idle until /MCLR is lowered, an interrupt occurs or power is cycled.

10.5. No Automatic Variables

The PICBASIC PRO Compiler does not automatically create any variables like B0 or W0. They must be defined using **VAR**. Two files are provided: `BS1DEFS.BAS` and `BS2DEFS.BAS` that define the standard BS1 or BS2 variables. However, it is recommended that you assign your own variables with meaningful names rather than using these files.

10.6. No Nibble Variable Types

The BS2's nibble variable type is not implemented in the PICBASIC PRO Compiler. As PBP allows many more variables than a BS2, simply changing nibble variable types to bytes will work in many cases.

10.7. No DIRS

The BASIC Stamp variable names **Dirs**, **Dirh**, **Dirl** and **Dir0**-**Dir15** are not defined and should not be used with the PICBASIC PRO Compiler. TRIS should be used instead, but has the opposite state of **Dirs**.

This **does not** work in PICBASIC PRO:

```
Dir0 = 1     ' Doesn't set pin PORTB.0 to output
```

Do this instead:

```
TRISB.0 = 0 ' Set pin PORTB.0 to output
```

or simply use a command that automatically sets the pin direction.

10.8. No Automatic Zeroing of Variables

The BASIC Stamp sets all the variables and registers to 0 when a program starts. This is not automatically done when a PBP program starts. In general, the variables should be initialized in the program to an appropriate state. Alternatively, **CLEAR** can be used to zero all the variables when a program starts.

10.9. Math Operators

Mathematical expressions in PBP have precedence of operation. This means they are not evaluated in strict left to right order as they are in the BASIC Stamp and original PICBASIC Compiler. This precedence means that multiplication and division are done before adds and subtracts, for example. However, any operators with the same precedence level (on the same line in the table below) will be evaluated from left to right.

Parenthesis should be used to group operations into the order in which they are to be performed. This way there will be no doubt as to order of the operations. The following table lists the operators in hierarchal order:

Highest Precedence
( )
- (unary)
! or NOT, ABS, COS, DCD, DIV32, NCD, SIN, SQR
<<, >>, ATN, DIG, HYP, MAX, MIN, REV
*, /, **, */, //
+, - (binary), ~
= or ==, <> or !=, <, <=, >, >=
&, |, ^, &/, |/, ^/
&& or AND, || or OR, ^^ or XOR, ANDNOT, ORNOT, XORNOT
Lowest Precedence

10.10. [ ] Versus ( )

PBP uses square brackets, **[]**, in statements where parenthesis, **()**, were previously used. This is more in keeping with BASIC Stamp II syntax.

For example, the BS1 and original PICBASIC Compiler **SEROUT** instruction looks something like:

```
Serout 0,T2400,(B0)
```

The PICBASIC PRO Compiler **SEROUT** instruction looks like:

```
Serout 0,T2400,[B0]
```

Any instructions that previously used parenthesis in their syntax should be changed to include square brackets instead.

10.11. ABS

ABS works slightly differently than on the Stamp in that it will take the absolute value of a byte as well as a word and long.

10.12. DATA, EEPROM, READ and WRITE

The BASIC Stamp allows serial EEPROM space not used for program storage to store non-volatile data. Since PBP programs execute directly from the PIC MCU's ROM space, EEPROM storage must be implemented in some other manner.

Many PIC MCUs have between 64 and 1024 bytes of on-chip data EEPROM. This data EEPROM is used by the **DATA**, **EEPROM**, **READ** and **WRITE** commands.

To access off-chip non-volatile data storage, the **I2CREAD** and **I2CWRITE** instructions have been added. These instructions allow 2-wire communications with serial EEPROMs like Microchip Technology's 24LC01B.

READ and **WRITE** will not work on devices with on-chip I2C interfaced serial EEPROM like the PIC12CE67x and PIC16CE62x parts. Use the **I2CREAD** and **I2CWRITE** instructions instead.

10.13. DEBUG

DEBUG in PBP is not a special case of **SEROUT** as it is on the Stamps. It has its own much shorter routine that works with a fixed pin and baud rate. It can be used in the same manner to send debugging information to a terminal program or other serial device.

DEBUG sends serial data out on PORTB, pin 0 at 2400 baud, unless otherwise **DEFINE**d.

Question marks (?) in **DEBUG** statements are ignored. The modifier `ASC?` Is not supported and should not be used.

10.14. FOR..NEXT

The BS2 automatically sets the direction of the **STEP** for a **FOR..NEXT** loop. If the ending value is smaller than the starting value and a **STEP** value is not specified, -1 is assumed. PICBASIC PRO always defaults to 1 if a **STEP** value is not specified. If a **STEP** of -1 is desired to make the loop count backwards, it must be specified:

```
For i = 10 To 1 Step -1
```

10.15. GOSUB and RETURN

Subroutines are implemented via the **GOSUB** and **RETURN** statements. User variable **W6** is used by the BS1 as a four nibble stack. Thus, Stamp programs may have up to 16 **GOSUB**s and subroutines can be nested up to four levels deep.

The PIC MCUs have Call and Return instructions as well as an eight level stack. PBP programs make use of these instructions and may use four levels of this stack, with the other four levels being reserved for library routines. Thus, **W6** is unused, subroutines may still be nested up to four levels deep (12 for 14-bit enhanced core and PIC17 and 27 for PIC18) and the number of **GOSUB**s is limited only by the PIC MCU's code space.

10.16. I2CREAD and I2CWRITE

The **I2CREAD** and **I2CWRITE** commands differ from the original PICBASIC Compiler's **I2CIN** and **I2COUT** commands. The most obvious difference is that the data and clock pin numbers are now specified as part of the command. They are no longer fixed to specific pins.

The other difference is that the control byte format has changed. You no longer set the address size as part of the control byte. Instead, the address size is determined by the type of the address variable. If a byte-sized variable is used, an 8-bit address is sent. If a word-sized variable is used, a 16-bit address is sent.

10.17. IF..THEN

The original PICBASIC compiler only allow a label to be specified after an **IF..THEN**. PICBASIC PRO additionally allows an **IF..THEN..ELSE..ENDIF** construct as well as allowing actual code to be executed as a result of an **IF** or **ELSE**.

10.18. LOOKDOWN and LOOKUP

LOOKDOWN and **LOOKUP** use BS1 syntax. **LOOKDOWN2** and **LOOKUP2** use BS2 syntax. **LOOKDOWN** and **LOOKUP** only support 8-bit constants in the table, not variables as in the BS1. You must use **LOOKDOWN2** or **LOOKUP2** if variables are required in the table.

10.19. MAX and MIN

The **MAX** and **MIN** operator's function have been altered somewhat from the way they work on the Stamp and the original PICBASIC Compiler.

MAX will return the maximum of two values. **MIN** will return the minimum of two values. This corresponds more closely to most other BASICs and does not have the 0 and 65535 limit problems of the Stamp's **MIN** and **MAX** instructions.

In most cases, you need only change **MIN** to **MAX** and **MAX** to **MIN** in your Stamp programs for them to work properly with PBP.

10.20. SERIN and SEROUT

SERIN and **SEROUT** use BS1 syntax. **SERIN2** and **SEROUT2** use BS2 syntax. A BS2 style timeout has been added to the **SERIN** command.

SERIN and **SEROUT** have been altered to run up to 9600 baud from the BS1 limit of 2400 baud. This has been accomplished by replacing the little used rate of 600 baud with 9600 baud. Modes of **T9600**, **N9600**, **OT9600** and **ON9600** may now be used.

600 baud is no longer available and will cause a compilation error if an attempt is made to use it.

10.21. SLEEP

PBP's **SLEEP** command is based solely on the Watchdog Timer. It is not calibrated using the system clock oscillator. This is because of the affect Watchdog Timer resets have on the PIC MCU.

Whenever the PIC MCU was reset during **SLEEP** calibration, it altered the states of some of the internal registers. For smaller PIC MCUs with few registers, these registers could be saved before and restored after calibration resets. However, since PBP may be used on many different PIC MCUs with many registers that are altered upon reset, this save and restore proved to be too unwieldy.

Therefore **SLEEP** runs in an uncalibrated mode based strictly upon the accuracy of the Watchdog Timer. This ensures the stability of the PIC MCU registers and I/O ports. However, since the Watchdog Timer is driven by an internal R/C oscillator, its period can vary significantly based

on temperature and individual chip variations. If greater accuracy is needed, **PAUSE**, which is not a low-power command, should be used.

Appendix A

Serin2/Serout2 Mode Examples

Baud Rate	BIT 15 (Output)	BIT 14 (Conversion)	BIT 13 (Parity)	Mode Number
300	Driven	True	None	3313
300	Driven	True	Even*	11505
300	Driven	Inverted	None	19697
300	Driven	Inverted	Even*	27889
300	Open	True	None	36081
300	Open	True	Even*	44273
300	Open	Inverted	None	52465
300	Open	Inverted	Even*	60657
1200	Driven	True	None	813
1200	Driven	True	Even*	9005
1200	Driven	Inverted	None	17197
1200	Driven	Inverted	Even*	25389
1200	Open	True	None	33581
1200	Open	True	Even*	41773
1200	Open	Inverted	None	49965
1200	Open	Inverted	Even*	58157
2400	Driven	True	None	396
2400	Driven	True	Even*	8588
2400	Driven	Inverted	None	16780
2400	Driven	Inverted	Even*	24972
2400	Open	True	None	33164

Baud Rate	BIT 15 (Output)	BIT 14 (Conversion)	BIT 13 (Parity)	Mode Number
2400	Open	True	Even*	41356
2400	Open	Inverted	None	49548
2400	Open	Inverted	Even*	57740
9600**	Driven	True	None	84
9600**	Driven	True	Even*	8276
9600**	Driven	Inverted	None	16468
9600**	Driven	Inverted	Even*	24660
9600**	Open	True	None	32852
9600**	Open	True	Even*	41044
9600**	Open	Inverted	None	49236
9600**	Open	Inverted	Even*	57428
19200**	Driven	True	None	32
19200**	Driven	True	Even*	8224
19200**	Driven	Inverted	None	16416
19200**	Driven	Inverted	Even*	24608
19200**	Open	True	None	32800
19200**	Open	True	Even*	40992
19200**	Open	Inverted	None	49184
19200**	Open	Inverted	Even*	57376

*For odd parity, add: `DEFINE SER2_ODD 1`.
**Oscillator faster than 4MHz may be required.

Appendix B

Defines

```
DEFINE ADC_BITS 8               'Number of bits in Adcin
                                result
DEFINE ADC_CLOCK 3              'ADC clock source (rc = 3)
DEFINE ADC_SAMPLEUS 50          'ADC sampling time in
                                microseconds
DEFINE BUTTON_PAUSE 10          'Button debounce delay in ms
DEFINE CCP1_REG PORTC           'Hpwm channel 1 pin port
DEFINE CCP1_BIT 2               'Hpwm channel 1 pin bit
DEFINE CCP2_REG PORTC           'Hpwm channel 2 pin port
DEFINE CCP2_BIT 1               'Hpwm channel 2 pin bit
DEFINE CCP3_REG PORTG           'Hpwm channel 3 pin port
DEFINE CCP3_BIT 0               'Hpwm channel 3 pin bit
DEFINE CCP4_REG PORTG           'Hpwm channel 4 pin port
DEFINE CCP4_BIT 3               'Hpwm channel 4 pin bit
DEFINE CCP5_REG PORTG           'Hpwm channel 5 pin port
DEFINE CCP5_BIT 4               'Hpwm channel 5 pin bit
DEFINE CHAR_PACING 1000         'Serout character pacing in us
DEFINE DEBUG_REG PORTB          'Debug pin port
DEFINE DEBUG_BIT 0              'Debug pin bit
DEFINE DEBUG_BAUD 2400          'Debug baud rate
DEFINE DEBUG_MODE 1             'Debug mode: 0 = True, 1 =
                                Inverted
DEFINE DEBUG_PACING 1000        'Debug character pacing in us
DEFINE DEBUGIN_REG PORTB        'Debugin pin port
DEFINE DEBUGIN_BIT 0            'Debugin pin bit
DEFINE DEBUGIN_MODE 1           'Debugin mode: 0 = True, 1 =
                                Inverted
DEFINE HPWM2_TIMER 1            'Hpwm channel 2 timer select
DEFINE HPWM3_TIMER 1            'Hpwm channel 3 timer select
DEFINE HSER_BAUD 2400           'Hser baud rate
DEFINE HSER_SPBRG 25            'Hser SPBRG init
DEFINE HSER_SPBRGH 0            'Hser SPBRGH init
DEFINE HSER_RCSTA 90h           'Hser receive status init
DEFINE HSER_TXSTA 20h           'Hser transmit status init
DEFINE HSER_EVEN 1              'If even parity desired
DEFINE HSER_ODD 1               'If odd parity desired
DEFINE HSER_BITS 9              'Use for 8 bits + parity
DEFINE HSER_CLROERR 1           'Automatically clear Hserin
                                overflow errors
DEFINE HSER_PORT 1              'Hser port to use on devices
                                with more than one
DEFINE HSER2_BAUD 2400          'Hser2 baud rate
DEFINE HSER2_SPBRG 25           'Hser2 SPBRG2 init
DEFINE HSER2_SPBRGH 0           'Hser2 SPBRGH2 init
DEFINE HSER2_RCSTA 90h          'Hser2 receive status init
DEFINE HSER2_TXSTA 20h          'Hser2 transmit status init
DEFINE HSER2_EVEN 1             'If even parity desired
DEFINE HSER2_ODD 1              'If odd parity desired
DEFINE HSER2_BITS 9             'Use for 8 bits + parity
DEFINE HSER2_CLROERR 1          'Automatically clear Hserin
                                overflow errors
```

```
DEFINE I2C_HOLD 1               'Pause I2C transmission while
                                clock held low
DEFINE I2C_INTERNAL 1           'Use for internal EEPROM on
                                PIC16CE and PIC12CE
DEFINE I2C_SCLOUT 1             'Set serial clock bipolar
                                instead of open-collector
DEFINE I2C_SLOW 1               'Use for >8MHz OSC with
                                standard speed devices
DEFINE I2C_SCL PORTA,1          'For 12-bit core only
DEFINE I2C_SDA PORTA,0          'For 12-bit core only
DEFINE INTHAND Label            'Assign assembler interrupt
                                handler label
DEFINE INTLHAND Label           'Assign assembler low priority
                                interrupt handler label for
                                PIC18
DEFINE LCD_DREG PORTA           'LCD data port
DEFINE LCD_DBIT 0               'LCD data starting bit 0 or 4
DEFINE LCD_RSREG PORTA          'LCD register select port
DEFINE LCD_RSBIT 4              'LCD register select bit
DEFINE LCD_EREG PORTB           'LCD enable port
DEFINE LCD_EBIT 3               'LCD enable bit
DEFINE LCD_RWREG PORTE          'LCD read/write port
DEFINE LCD_RWBIT 2              'LCD read/write bit
DEFINE LCD_BITS 4               'LCD bus size 4 or 8
DEFINE LCD_LINES 2              'Number lines on LCD
DEFINE LCD_COMMANDUS 2000       'Command delay time in us
DEFINE LCD_DATAUS 50            'Data delay time in us
DEFINE LOADER_USED 1            'Bootloader is being used
DEFINE NO_CLEAR_STKPTR 1        'See Resume Label
DEFINE NO_CLRWDT 1              'Don't insert CLRWDTs
DEFINE OSC 4                    'Oscillator speed in MHz:
                                3(3.58) 4 8 10 12 16 20 24 25
                                32 33 40 48 64
DEFINE OSCCAL_1K 1              'Set OSCCAL for 1K PIC12
DEFINE OSCCAL_2K 1              'Set OSCCAL for 2K PIC12
DEFINE PULSIN_MAX 65535         'Maximum Pulsin/ Rctime count
DEFINE RESET_ORG 0h             'Change reset address
DEFINE SER2_BITS 8              'Set number of data bits for
                                Serin2 and Serout2
DEFINE SER2_ODD 1               'Set odd parity for Serin2 and
                                Serout2
DEFINE SHIFT_PAUSEUS 50         'Slow down the Shiftin and
                                Shiftout clock
DEFINE USE_LFSR 1               'Use PIC18 LFSR instruction
DEFINE WRITE_INT 1              'Disable/enable global
                                interrupts in Write
DEFINE XINXLAT_OFF 1            'Don't translate Xin commands
                                to BS2 format
DEFINE XOUTXLAT_OFF 1           'Don't translate Xout commands
                                to BS2 format
```

Appendix C

Reserved Words

(See section 7.4. for important information about additional keywords.)

ABS	BIN20	BRANCH	FLAGS	IBIN19
ADCIN	BIN21	BRANCHL	FOR	IBIN20
AND	BIN22	BUTTON	FREQOUT	IBIN21
ANDNOT	BIN23	BYTE	GET	IBIN22
ARRAYREAD	BIN24	BYTE0	GOP	IBIN23
ARRAYWRITE	BIN25	BYTE1	GOSUB	IBIN24
ASM	BIN26	BYTE2	GOTO	IBIN25
ATN	BIN27	BYTE3	HEX	IBIN26
AUXIO	BIN28	CALL	HEX1	IBIN27
BANK0	BIN29	CASE	HEX2	IBIN28
BANK1	BIN30	CLEAR	HEX3	IBIN29
BANK2	BIN31	CLEARWDT	HEX4	IBIN30
BANK3	BIN32	CON	HEX5	IBIN31
BANK4	BIT	COS	HEX6	IBIN32
BANK5	BIT0	COUNT	HEX7	IDEC
BANK6	BIT1	DATA	HEX8	IDEC1
BANK7	BIT2	DCD	HIGH	IDEC2
BANK8	BIT3	DEBUG	HIGHBYTE	IDEC3
BANK9	BIT4	DEBUGIN	HIGHWORD	IDEC4
BANK10	BIT5	DEC	HPWM	IDEC5
BANK11	BIT6	DEC1	HSERIN	IDEC6
BANK12	BIT7	DEC2	HSERIN2	IDEC7
BANK13	BIT8	DEC3	HSEROUT	IDEC8
BANK14	BIT9	DEC4	HSEROUT2	IDEC9
BANK15	BIT10	DEC5	HYP	IDEC10
BANKA	BIT11	DEC6	I2CREAD	IF
BIN	BIT12	DEC7	I2CWRITE	IHEX
BIN1	BIT13	DEC8	IBIN	IHEX1
BIN2	BIT14	DEC9	IBIN1	IHEX2
BIN3	BIT15	DEC10	IBIN2	IHEX3
BIN4	BIT16	DEFINE	IBIN3	IHEX4
BIN5	BIT17	DIG	IBIN4	IHEX5
BIN6	BIT18	DISABLE	IBIN5	IHEX6
BIN7	BIT19	DIV32	IBIN6	IHEX7
BIN8	BIT20	DO	IBIN7	IHEX8
BIN9	BIT21	DTMFOUT	IBIN8	INCLUDE
BIN10	BIT22	EEPROM	IBIN9	INPUT
BIN11	BIT23	ELSE	IBIN10	INTERRUPT
BIN12	BIT24	ELSEIF	IBIN11	IS
BIN13	BIT25	ENABLE	IBIN12	ISBIN
BIN14	BIT26	END	IBIN13	ISBIN1
BIN15	BIT27	ENDASM	IBIN14	ISBIN2
BIN16	BIT28	ENDIF	IBIN15	ISBIN3
BIN17	BIT29	ERASECODE	IBIN16	ISBIN4
BIN18	BIT30	EXIT	IBIN17	ISBIN5
BIN19	BIT31	EXT	IBIN18	ISBIN6

ISBIN7
ISBIN8
ISBIN9
ISBIN10
ISBIN11
ISBIN12
ISBIN13
ISBIN14
ISBIN15
ISBIN16
ISBIN17
ISBIN18
ISBIN19
ISBIN20
ISBIN21
ISBIN22
ISBIN23
ISBIN24
ISBIN25
ISBIN26
ISBIN27
ISBIN28
ISBIN29
ISBIN30
ISBIN31
ISBIN32
ISDEC
ISDEC1
ISDEC2
ISDEC3
ISDEC4
ISDEC5
ISDEC6
ISDEC7
ISDEC8
ISDEC9
ISDEC10
ISHEX
ISHEX1
ISHEX2
ISHEX3
ISHEX4
ISHEX5
ISHEX6
ISHEX7
ISHEX8
LCDIN
LCDOUT
LET
LIBRARY
LONG
LOOKDOWN
LOOKDOWN2
LOOKUP
LOOKUP2
LOOP
LOW
LOWBYTE
LOWWORD
MAX
MIN
MOD
NAP
NCD
NEXT
NOT
OFF
ON
OR
ORNOT
OUTPUT
OWIN
OWOUT
PAUSE
PAUSEUS
PEEK
PEEKCODE
PIN
POKE
POKECODE
POLLIN
POLLMODE
POLLOUT
POLLRUN
POLLWAIT
POT
PULSIN
PULSOUT
PUT
PWM
R0
R1
R2
R3
R4
R5
R6
R7
R8
RANDOM
RB1**
RB2**
RCTIME
READ
READCODE
REM
REP
REPEAT
RESUME
RETURN
REV
REVERSE
RM1
RM2
RR1
RR2
RS1***
RS2***
SBIN
SBIN1
SBIN2
SBIN3
SBIN4
SBIN5
SBIN6
SBIN7
SBIN8
SBIN9
SBIN10
SBIN11
SBIN12
SBIN13
SBIN14
SBIN15
SBIN16
SBIN17
SBIN18
SBIN19
SBIN20
SBIN21
SBIN22
SBIN23
SBIN24
SBIN25
SBIN26
SBIN27
SBIN28
SBIN29
SBIN30
SBIN31
SBIN32
SDEC
SDEC1
SDEC2
SDEC3
SDEC4
SDEC5
SDEC6
SDEC7
SDEC8
SDEC9
SDEC10
SELECT
SERIN
SERIN2
SEROUT
SEROUT2
SHEX
SHEX1
SHEX2
SHEX3
SHEX4
SHEX5
SHEX6
SHEX7
SHEX8
SHIFTIN
SHIFTOUT
SIN
SKIP
SLEEP
SOFT_STACK*
SOFT_STACK
_PTR*
SOUND
SQR
STEP
STOP
STR
SWAP
SYMBOL
SYSTEM
THEN
TO
TOGGLE
UNTIL
USBIN
USBINIT
USBOUT
USBSERVICE
VAR
WAIT
WAITSTR
WEND
WHILE
WORD
WORD0
WORD1
WRITE
WRITECODE
XIN
XOR
XORNOT
XOUT

**PIC17
***PIC18

*12-bit core

Appendix D

ASCII Table

ASCII Control Characters

Decimal	Hex	ASCII Function	Key
0	0	NUL (null)	Ctrl-@
1	1	SOH (start of heading)	Ctrl-A
2	2	STX (start of text)	Ctrl-B
3	3	ETX (end of text)	Ctrl-C
4	4	EOT (end of transmission)	Ctrl-D
5	5	ENQ (enquiry)	Ctrl-E
6	6	ACK (acknowledge)	Ctrl-F
7	7	BEL (bell)	Ctrl-G
8	8	BS (backspace)	Ctrl-H
9	9	HT (horizontal tab)	Ctrl-I
10	A	LF (line feed)	Ctrl-J
11	B	VT (vertical tab)	Ctrl-K
12	C	FF (form feed)	Ctrl-L
13	D	CR (carriage return)	Ctrl-M
14	E	SO (shift out)	Ctrl-N
15	F	SI (shift in)	Ctrl-O
16	10	DLE (data link escape)	Ctrl-P
17	11	DC1 (device control 1)	Ctrl-Q
18	12	DC2 (device control 2)	Ctrl-R
19	13	DC3 (device control 3)	Ctrl-S
20	14	DC4 (device control 4)	Ctrl-T

Decimal	Hex	ASCII Function	Key
21	15	NAK (negative acknowledge)	Ctrl-U
22	16	SYN (synchronous idle)	Ctrl-V
23	17	ETB (end of trans. block)	Ctrl-W
24	18	CAN (cancel)	Ctrl-X
25	19	EM (end of medium)	Ctrl-Y
26	1A	SUB (substitute)	Ctrl-Z
27	1B	ESC (escape)	Ctrl-[
28	1C	FS (file separator)	Ctrl-\
29	1D	GS (group separator)	Ctrl-]
30	1E	RS (record separator)	Ctrl-^
31	1F	US (unit separator)	Ctrl-_

Standard ASCII Character Set

Decimal	Hex	Display/ Key	Decimal	Hex	Display/ Key	Decimal	Hex	Display /Key
32	20	Space	64	40	@	96	60	`
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x

Decimal	Hex	Display/ Key	Decimal	Hex	Display/ Key	Decimal	Hex	Display /Key
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	|
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_	127	7F	DEL

Appendix E

Contact Information

Technical support and sales may be reached at:

microEngineering Labs, Inc.
Box 60039
Colorado Springs CO 80960-0039
(719) 520-5323
(719) 520-1867 fax
http://www.melabs.com
email:support@melabs.com

PIC® MCU data sheets and literature may be obtained from:

Microchip Technology Inc.
2355 W. Chandler Blvd.
Chandler AZ 85224-6199
(480) 792-7200
(480) 792-7277 fax
http://www.microchip.com
email:literature@microchip.com

READ THE FOLLOWING TERMS AND CONDITIONS CAREFULLY BEFORE OPENING THIS PACKAGE.
microEngineering Labs, Inc. ("the Company") is willing to license the enclosed software to the purchaser of the software ("Licensee") only on the condition that Licensee accepts all of the terms and conditions set forth below. By opening this sealed package, Licensee is agreeing to be bound by these terms and conditions.

Disclaimer of Liability

THE COMPANY DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE AND THE IMPLIED WARRANTY OF MERCHANTABILITY. IN NO EVENT SHALL THE COMPANY OR ITS EMPLOYEES, AGENTS, SUPPLIERS OR CONTRACTORS BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR IN CONNECTION WITH LICENSE GRANTED UNDER THIS AGREEMENT, INCLUDING WITHOUT LIMITATION, LOST PROFITS, DOWNTIME, GOODWILL, DAMAGE TO OR REPLACEMENT OF EQUIPMENT OR PROPERTY, OR ANY COSTS FOR RECOVERING, REPROGRAMMING OR REPRODUCING ANY DATA USED WITH THE COMPANY'S PRODUCTS.

Software License

In consideration of Licensee's payment of the license fee, which is part of the price Licensee paid for this product, and Licensee's agreement to abide by the terms and conditions on this page, the Company grants Licensee a nonexclusive right to use and display the copy of the enclosed software on a single computer at a single location. Licensee owns only the enclosed media on which the software is recorded or fixed, and the Company retains all right, title and ownership (including the copyright) to the software recorded on the original media copy and all subsequent copies of the software. Licensee may not network the software or otherwise use it on more than one computer terminal at the same time. Copies may only be made for archival or backup purposes. The enclosed software is licensed only to the Licensee and may not be transferred to anyone else, nor may copies be given to anyone else. Any violation of the terms and conditions of this software license shall result in the immediate termination of the license.