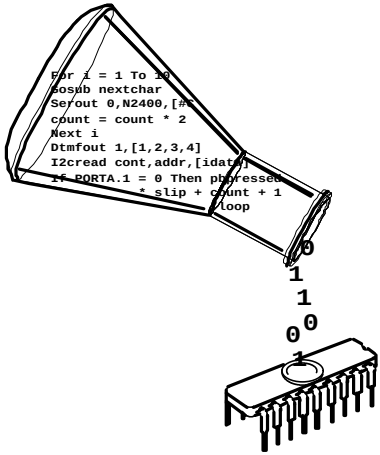
PicBasic Pro Compiler



microEngineering Labs, Inc.

Copyrights and Trademarks

2/99

PicBasic Pro Compiler

microEngineering Labs, Inc.

TABLE OF CONTENTS

1. Introduction

The PicBasic Pro Compiler (or PBP) is our next-generation programming language that makes it even quicker and easier for you to program Microchip Technology's powerful PICmicro microcontrollers. The English-like BASIC language is much easier to read and write than the quirky Microchip 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. One of these was to add a real **`IF..THEN..ELSE..ENDIF`** instead of the **`IF..THEN(GOTO)`** of the Stamps. These differences are spelled out later in this manual.

PBP defaults to create files that run on a PIC16F84-04/P 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. Many PICmicros other than the 16F84, as well as oscillators of frequencies other than 4Mhz, may be used with the PicBasic Pro Compiler.

1.1. The PICmicros

The PicBasic Pro Compiler produces code that may be programmed into a wide variety of PICmicro microcontrollers having from 8 to 68 pins and various on-chip features including A/D converters, hardware timers and serial ports.

There are some PICmicros that will not work with the PicBasic Pro Compiler, notably the PIC16C5x series including the PIC16C54 and PIC16C58. These PICmicros are based on the older 12-bit core rather than the more current 14-bit core. The PicBasic Pro Compiler requires some of the features only available with the 14-bit core, the foremost of which being the 8-level stack.

There are many, many PICmicros, some pin-compatible with the '5x series, that may be used with the PicBasic Pro Compiler. Currently, the

list includes the PIC16C554, 556, 558, 61, 62(AB), 620(A), 621(A), 622(A), 63(A), 64(A), 65(AB), 66, 67, 71, 710, 711, 715, 72(A), 73(AB), 74(AB), 76, 77, 773, 774, 84, 923, 924, the PIC16F83, 84, 873, 874, 876, 877, the PIC12C671 and 672 and the PIC14C000, with Microchip adding more at a rapid rate. For direct replacement of a PIC16C54 or 58, the PIC16C554, 558, 620 and 622 work well with the compiler and are very nearly the same price.*

For general purpose PICmicro development using the PicBasic Pro Compiler, the PIC16F84 (or PIC16C84 if the 'F84 is not available) is the current PICmicro of choice. This 18-pin microcontroller uses flash (or EEPROM) technology to allow rapid erasing and reprogramming to speed program debugging. With the click of the mouse in the programming software, the PIC16F84 can be instantly erased and then reprogrammed again and again. Other PICmicros in the 12C67x, 16C55x, 16C6xx, 16C7xx and 16C9xx 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.

The PIC16F84 (and 'C84) also contains 64 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 PICmicro's code space whether the power is on or off.)

By using the 'F84 for initial program testing, the debugging process may be sped along. Once the main routines of a program are operating satisfactorily, a PICmicro with more capabilities or expanded features of the compiler may be utilized.

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

*Selling price is dictated by Microchip Technology Inc. and its distributors.

1.2. About This Manual

This manual cannot be a full treatise on the BASIC language. It describes the PicBasic Pro 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 and there are additional example programs on the disk that can help get you started.

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 - all the inner workings of the compiler.

2. Getting Started

2.1. Software Installation

The PicBasic Pro files are compressed into a self-extracting file on the diskette. They must be uncompressed before use. To uncompress the files, create a subdirectory on your hard drive called PBP or another name of your choosing by typing:

```
md PBP
```

at the DOS prompt. Change to the directory:

```
cd PBP
```

Assuming the distribution diskette is in drive a:, uncompress the files into the PBP subdirectory:

```
a:\pbpxxx -d
```

Don't forget the -d option on the end of the command. This ensures that the proper subdirectories within PBP are created.

Make sure that FILES and BUFFERS are set to at least 50 in your CONFIG.SYS file. Depending on how many FILES and BUFFERS are already in use by your system, allocating an even larger number may be necessary.

2.2. Your First Program

For operation of the PicBasic Pro Compiler you'll need a text editor or word processor for creation of your program source file, some sort of PICmicro programmer such as our EPIC Plus Pocket PICmicro 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 create the BASIC source file for the program using your favorite text editor or word processor. If you don't have a favorite, DOS EDIT (included with MS-DOS) or Windows NOTEPAD (included with Windows

and Windows95/98) may be substituted. The source file name should end with (but isn't required to) the extension .BAS.

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 PICmicro in the real world. You may type it in or you can simply grab it from the SAMPLES subdirectory included on the PicBasic Pro Compiler distribution diskette. 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

loop: 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 loop         ' Go back to loop 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 entering PBP 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:

```
PBP 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 its assembler to complete the task. If all goes well, the final PICmicro 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 to write the entire 100,000 line monolith all at once and then try to debug it from end to end.

If you don't tell it otherwise, the PicBasic Pro Compiler defaults to creating code for the PIC16F84. To compile code for PICmicros other than the 'F84, 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 PIC16C74, compile it using the command:

```
PBP -p16c74 blink
```

2.3. Program That PICmicro

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

The PicBasic Pro Compiler generates standard 8-bit Merged Intel HEX (.HEX) files that may be used with any PICmicro programmer including our EPIC Plus Pocket PICmicro Programmer. PICmicros cannot be programmed with BASIC Stamp programming cables.

The following is an example of how a PICmicro might be programmed using our EPIC Programmer with the DOS programming software. If Windows95/98/NT is available, using the Windows version of EPIC is recommended.

Make sure there are no PICmicros installed in the EPIC Programmer programming socket or any attached adapters.

Hook the EPIC Programmer 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 EPIC Programmer (or attach 2 fresh 9-volt batteries to the programmer and connect the "Batt ON" jumper).

The LED on the EPIC Programmer may be on or off at this point. Do not insert a PICmicro into the programming socket when the LED is on or before the programming software has been started.

Enter:

```
EPIC
```

at the DOS command prompt to start the programming software. The EPIC software should be run from a pure DOS session or from a full screen DOS session under Windows or OS/2. (Running under Windows is discouraged. Windows (all varieties) alters the system timing and plays with the port when you're not looking, which may cause programming errors.)

The EPIC software will take a look around to find where the EPIC Programmer is attached and get it ready to program a PICmicro. If the EPIC Programmer is not found, check all of the above connections and verify that there is not a PICmicro or any adapter connected to the programmer. Typing:

```
EPIC /?
```

at the DOS command prompt will display a list of available options for the EPIC software.

Once the programming screen is displayed, use the mouse to click on Open file or press **Alt-O** on your keyboard. Use the mouse (or keyboard) to select `BLINK.HEX` or any other file you would like to program into the PICmicro from the dialog box.

The file will load and you should see a list of numbers in the window at the left. This is your program in PICmicro code. At the right of the screen there is a display of the configuration information that will be programmed into the PICmicro. Verify that it is correct before proceeding.

In general, the Oscillator should be set to XT 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) PICmicro. You may not be able to erase a windowed PICmicro that has been code protected.

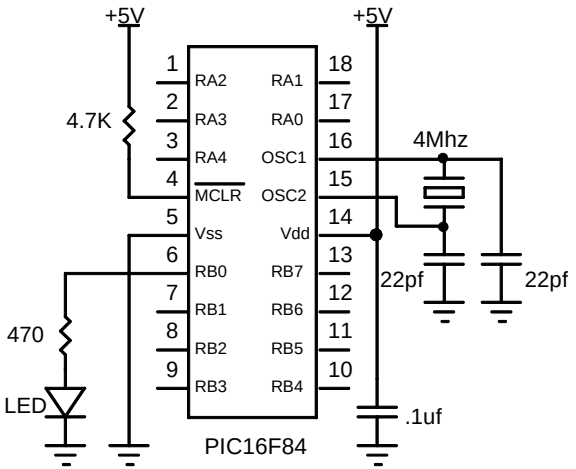
When it all looks marvelous, it is time to insert a PICmicro into the programming socket and click on Program or press **Alt-P** on the keyboard. The PICmicro will first be checked to make sure it is blank and then your code will be programmed into it. If the PICmicro is not

blank and it is a 16F84 or 16C84, you can simply choose to program over it without erasing first.

Once the programming is complete and the LED is off, 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 PICmicro 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 breadboard and plug in the PICmicro you just programmed. Our line of **PICProto** prototyping boards is perfect for this kind of thing.

Connect a power supply. Your PICmicro 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 PICmicro.

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

2.5. I've Got Troubles

The most common problems with getting PICmicros running involve making sure the few external components are of the appropriate value and properly connected to the PICmicro. 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 PICmicro will work but usually it won't. The PICmicro has an on-chip power-on-reset circuit so in general just an external pull-up resistor is adequate. But in some cases the PICmicro may not power up properly and an external circuit may be necessary. See the Microchip PICmicro 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 2 22pf (picofarad) ceramic disk capacitors is a good start for most PICmicros. 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 PICmicro itself consumes very little power, the power supply must be filtered fairly well. If the PICmicro 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 PICmicro 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 PICmicro to lose its mind.

Check the PICmicro data sheets. Some devices have features that can interfere with expected pin operations. The PIC16C62x parts (the 16C620, 621 and 622) are a good example of this. These PICmicros have analog comparators on PORTA. 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. Any PICmicro with analog inputs, such as the PIC16C7xx series devices, will come up in analog mode. You must set them to digital if that is how you intend to use them:

```
ADCON1 = 7
```

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 collector output rather then 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.

All of the PICmicro 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 PICmicro data sheets to become familiar with the idiosyncrasies of a particular part.

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.

2.6. 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 PICmicro 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.

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:”. Even a label like “loop:” is more descriptive (though only slightly). 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

Finally, 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. **GOSUB**s 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 the following command format:

PBP *Options Filename*

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, though no spaces may occur within an *Option*.

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

```
PBP -p16c71 -ampasm blink
```

will cause the file BLINK.BAS to be compiled using MPASM as the assembler and targeted for a PIC16C71 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 is performed without error. 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
H(?)	Display Help screen
I	Use a different Include path
L	Use a different Library file
O	Pass Option to assembler
P	Specify target Processor
S	Skip execution of assembler when done
V	Verbose mode

3.2.1. Option -A

PBP has the capability to use either PM, which is included with PBP, or Microchip's MPASM as its assembler. To specify MPASM (which must be acquired from Microchip), use **-ampasm** on the command line:

```
PBP -ampasm filename
```

If no assembler is specified on the command line, PM is used. See the section on assembly language programming for more information.

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.

```
PBP -c filename
```

3.2.3. 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.

3.2.4. Option -I

The **-I** option lets you select the include path for files used by PicBasic Pro.

3.2.5. 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.

```
PBP -lpbpps2 filename
```

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

3.2.6. Option -O

The **-O** option causes the letters following it to be passed to the assembler on its command line as options. Some pertinent PM options are listed in the following table:

PM Option	Description
OD	Generates Listing, Symbol Table, and Map File
OL	Generates Listing only

```
PBP -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.

The PICmicro Macro Assembler's manual on disk contains more information on the assembler and its options.

3.2.7. 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 PIC16C74, the command line should look something like the following:

```
PBP -p16c74 filename
```

3.2.8. 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 an executable image. 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** option are effectively overridden.

```
PBP -s filename
```

3.2.9. Option -V

The **-V** option turns on PBP’s verbose mode which presents more information during program compilation.

```
PBP -v 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 32.

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 bits, bytes or words. 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 identifier, excluding keywords, as described above. *Size* is **BIT**, **BYTE** or **WORD**. Optional *Modifiers* add additional control over how the variable is created. Some examples of creating variable are:

```
dog   var   byte
cat   var   bit
w0    var   word
```

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 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 additional temporary variables for use in 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 bit0 of dog
```

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
BIT9 or 9	Create alias to bit 9 of word
BIT10 or 10	Create alias to bit 10 of word
BIT11 or 11	Create alias to bit 11 of word
BIT12 or 12	Create alias to bit 12 of word
BIT13 or 13	Create alias to bit 13 of word
BIT14 or 14	Create alias to bit 14 of word
BIT15 or 15	Create alias to bit 15 of word
BYTE0 or LOWBYTE	Create alias to low byte of word
BYTE1 or HIGHBYTE	Create alias to high byte of word

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**. *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 allocated in memory, there are size limits for each type:

Size	Maximum Number of elements
BIT	128
BYTE	64
WORD	32

See the section on memory allocation for more information.

4.6. Constants

Named constants may be created in a similar manner to variables. It can be more convenient to use a constant name instead of 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
```

4.7. Symbols

SYMBOL provides yet another method for aliasing variables and constants. **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
```

4.8. Numeric Constants

PBP allows numeric constants to be defined in the three bases: decimal, binary and hexadecimal. Binary values are defined 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).

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

4.9. String Constants

PBP doesn't provide string handling capabilities, 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).

```
"Hello" ‘ String (Short for "H","e","l","l","o")
```

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

4.10. Pins

Pins may be accessed in a number of different ways. The best 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 should 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
```

For compatibility with the BASIC Stamp, pins used in PicBasic Pro Compiler commands may also be referred to by number, 0 - 15. These pins are physically mapped onto different PICmicro hardware ports dependent on how many pins the microcontroller has.

No. PICmicro Pins	0 - 7	8 - 15
8-pin	GPIO*	GPIO*
18-pin	PORTB	PORTA*
28-pin (except 14C000)	PORTB	PORTC
28-pin (14C000)	PORTC	PORTD
40-pin	PORTB	PORTC

*GPIO and PORTA do not have 8 I/O pins.

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 PICmicro. Depending on the particular PICmicro, 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 PORTC.0 for a PIC14C000) .

Pins may be referenced by number (0 - 15), name (e.g. **`Pin0`**, if one of the bsdefs.bas files are 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 latter 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 file “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**, **B0-B25**, **W0**-**W12** and most of the other BS2 pin and variable names.

When a PICmicro 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 pin an output. Setting a TRIS bit to 1 makes its 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 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.

4.11. Comments

A PBP comment starts with either the **REM** keyword or the single quote (**’**). 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** (although **REM** itself is not valid).

4.12. 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.

4.13. 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]
```

4.14. 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 where 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"
```

4.15. 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. See the appropriate sections of the manual for specific information on these definitions.

```
DEFINE BUTTON_PAUSE 10          'Button debounce
                                delay in ms

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 HSER_BAUD  2400          'Set baud rate
DEFINE HSER_RCSTA 90h           'Set rcv reg
DEFINE HSER_TXSTA 20h           'Set transmit reg
```

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

DEFINE I2C_INTERNAL 1         'Use for internal
                              EEPROM on 16CExxx
                              and 12CExxx
DEFINE I2C_SLOW   1           'Use for >8mHz OSC
                              with standard
                              speed devices

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_BITS   4           'LCD bus size 4 or
                              8
DEFINE LCD_LINES  2           'number lines on
                              LCD

DEFINE OSC 4                  '3 4 8 10 12 16 20

DEFINE OSCCAL_1K  1           'Set OSCCAL for
                              PIC12C671
DEFINE OSCCAL_2K  1           'Set OSCCAL for
                              PIC12C672
```

4.16. Math Operators

Unlike the BASIC Stamp, the PicBasic Pro Compiler performs all math 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 are unsigned and performed with 16-bit precision. The operators supported are:

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
COS	Cosine
DCD	2n Decode
DIG	Digit
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.

4.16.1. Multiplication

PBP performs 16x16 multiplication. The '*****' operator returns the lower 16 bits of the 32-bit result. This is the typical multiplication found in most programming languages. The '******' operator returns the upper 16 bits of the 32-bit result. These two operators can be used in conjunction to perform 16x16 multiplication that produces 32-bit results.

```
W1 = W0 * 1000     ' Multiply value in W0 by 1000
                   and place the result in W1
W2 = W0 ** 1000    ' Multiply W0 by 1000 and
                   place the high order 16 bits
                   (which may be 0) in W2
```

The '***/**' operator returns the middle 16 bits of the 32-bit result.

```
W3 = W1 */ W0      ' Multiply W1 by W0 and place
                   the middle 16 bits in W3
```

4.16.2. Division

PBP performs 16x16 division. The '**/**' operator returns the 16-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
```

4.16.3. Shift

The '**<<**' and '**>>**' operators shift a value left or right, respectively, 1 to 15 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)
```

4.16.4. 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.

```
B1 = ABS B0
```

4.16.5. 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
```

4.16.6. DCD

DCD returns the decoded value of a bit number. It changes a bit number (0 - 15) 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
```

4.16.7. DIG

DIG returns the value of a decimal digit. Simply tell it the digit number (0 - 4 with 0 being the rightmost digit) you would like the value of, and voila.

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

4.16.8. 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)
```

4.16.9. NCD

NCD returns the priority encoded bit number (1 - 16) 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
```

4.16.10. REV

REV reverses the order of the lowest bits in a value. The number of bits to be reversed is from 1 to 16.

```
B0 = %10101100 REV 4    ' Sets B0 to %10100011
```

4.16.11. 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
```

4.16.12. 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 no larger than the actual result.

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

4.16.13. 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     ' Reverse state of bit 0
                        of B0
```

4.17. Comparison Operators

Comparison operators are used in **IF..THEN** statements to compare one expression with another. The operators supported are:

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

```
If i > 10 Then loop
```

4.18. 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. The operators supported are:

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

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

Be sure to use parenthesis to tell PBP the exact order you want the operations to be performed.

5. PicBasic Pro Statement Reference

@	Insert one line of assembly language code.
ASM..ENDASM	Insert assembly language code section.
BRANCH	Computed **GOTO** (equiv. 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.
COUNT	Count number of pulses on a pin.
DATA	Define initial contents of on-chip EEPROM.
DEBUG	Asynchronous serial output to fixed pin and baud.
DISABLE	Disable **ON INTERRUPT** processing.
DTMFOUT	Produce touch-tones on a pin.
EEPROM	Define initial contents of on-chip EEPROM.
ENABLE	Enable **ON INTERRUPT** processing.
END	Stop execution and enter low power mode.
FOR..NEXT	Repeatedly execute statements.
FREQOUT	Produce up to 2 frequencies on a pin.
GOSUB	Call BASIC subroutine at specified label.
GOTO	Continue execution at specified label.
HIGH	Make pin output high.
HSERIN	Hardware asynchronous serial input.
HSEROUT	Hardware asynchronous serial output.
I2CREAD	Read bytes from I^2C device.
I2CWRITE	Write bytes to I^2C device.
IF..THEN..ELSE..ENDIF	Conditionally execute statements.
INPUT	Make pin an input.
{**LET**}	Assign result of an expression to a variable.
LCDOUT	Display characters on LCD.
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 INTERRUPT	Execute BASIC subroutine on an interrupt.
OUTPUT	Make pin an output.
PAUSE	Delay (1mSec resolution).
PAUSEUS	Delay (1uSec resolution).

PEEK	Read byte from register.
POKE	Write byte to register.
POT	Read potentiometer on specified pin.
PULSIN	Measure pulse width on a pin.
PULSOUT	Generate pulse to 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.
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.
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.
WHILE..WEND	Execute statements while condition is true.
WRITE	Write byte to on-chip EEPROM.
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 PBP program. You can use this shortcut to freely 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. 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 freely mix assembly language code with PicBasic Pro statements.

The maximum size for an assembler text section is 8K. This is the maximum size for the actual source, including comments, not the generated code. If the text block is larger than this, 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.

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
```

5.3. BRANCH

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

BRANCH causes the program to jump to a different location based on a variable index. This is similar to On..Goto in other BASICs.

Index selects one of a list of labels. 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 labels, no action is taken and execution continues with the statement following the **BRANCH**. Up to 256 *Label*s may be used in a **BRANCH**.

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.

```
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)
```

5.4. 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 differences are that it can jump to a *Label* that is in a different code page than the **BRANCHL** instruction and that 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 (2K or less of ROM), using **BRANCH** instead of **BRANCHL** will minimize memory usage.

Index selects one of a list of labels. 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 labels, no action is taken and execution continues with the statement following the **BRANCHL**. Up to 128 *Label*s may be used in a **BRANCHL**.

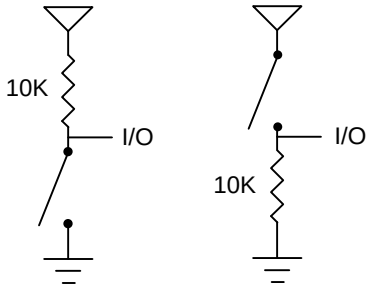
```
BRANCHL B4,[dog,cat,fish]
' Same as:
'  If B4=0 Then dog (goto dog)
'  If B4=1 Then cat (goto cat)
'  If B4=2 Then fish (goto fish)
```

5.5. BUTTON

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

Read *Pin* and optionally performs 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.

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
```

5.6. 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. Using **CALL**, a *Label* in an assembly language section can be accessed that is otherwise inaccessible to PBP.

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

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

5.7. 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 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 set in the program to an appropriate initial state rather than using **CLEAR**.

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

5.8. 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 **DEFINEd** 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]
```

5.9. DATA

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 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 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 PIC16F84 and PIC16C84. 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.

```
        ‘ Store 10, 20 and 30 starting at location 5
        DATA @5,10,20,30

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

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

5.10. DEBUG

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

Send one or more *Item*s to 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** also supports the same data modifiers as **SEROUT2**. Refer to the section on **SEROUT2** for this information.

DEBUG is one of several built-in asynchronous serial functions. It is the smallest 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
```

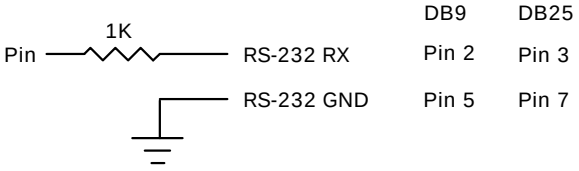
DEBUG assumes a 4Mhz oscillator when generating its bit timing. To maintain the proper baud rate timing with other oscillator values, be sure to **DEFINE** the OSC setting to any different oscillator value.

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 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 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 PICmicro, most applications don't require level converters. Rather, inverted TTL (DEBUG_MODE = 1) can 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
```

5.11. DISABLE

DISABLE

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** is encountered.

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

5.12. DTMFOUT

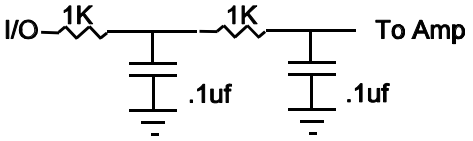
DTMFOUT *Pin*,{*Onms*,*Offms*,}**[***Tone*{,*Tone*...}**]**

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 get rid of some of the harmonics that are generated:



DTMFOUT works best with a 20Mhz oscillator. It can also work with a 10Mhz 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 **DTMFOUT** to generate a frequency that is a ratio of the actual oscillator used and 20Mhz which will not be very useful for sending touch tones.

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

5.13. 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 PIC16F84 and PIC16C84. 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.

```
' Store 10, 20 and 30 starting at location 5
EEPROM 5,[10,20,30]
```

5.14. ENABLE

ENABLE

ENABLE interrupt processing that was previously **DISABLEd** following this instruction.

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

5.15. 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
```

5.16. 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-sized 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
```

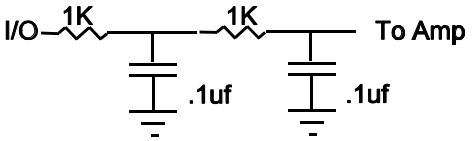
5.17. FREQOUT

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 get rid of some of the harmonics that are generated:



FREQOUT works best with a 20Mhz oscillator. It can also work with a 10Mhz 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.

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

5.18. GOSUB

GOSUB *Label*

Jump to the subroutine at *Label* saving its return address on the stack. Unlike **GOTO**, when a **RETURN** statement is reached, execution resumes with the statement following the 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 call another subroutine. Such subroutine nesting should be restricted to no more than four levels deep.

```
      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
```

5.19. 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
```

5.20. 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
```

5.21. 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
```

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.

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
```

The parity setting, along with all of the other HSER **DEFINE**s, affect both **HERIN** 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 defines.

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**.

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

```
HSERIN [B0, dec W1]
```

5.22. 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 **DEFINEs**:

```
' 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
```

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 **DEFINEs**:

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

' Use only if odd parity desired
DEFINE HSER_ODD   1
```

The parity setting, along with all of the other HSER **DEFINEs**, affect both **HERIN** 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**.

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

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

5.23. 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.

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.

If a word-sized *Var* is specified, 2 bytes are read and stored into the *Var* high byte first, followed by the low byte. This order is different than the way variables are normally stored, low byte first.

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 12CExxx and 16CExxx 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
```

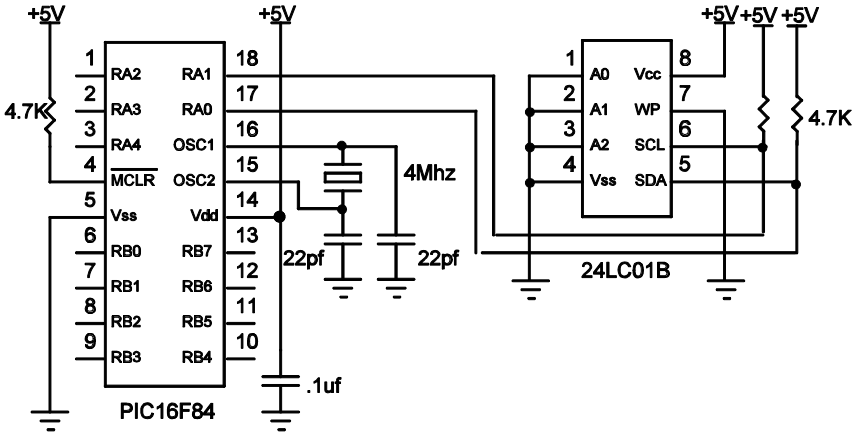
See the Microchip data sheets for these devices for more information.

The timing of the I^2C instructions is set so than 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
```

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.




```
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.

5.24. 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).

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.

When writing to a serial EEPROM it is necessary to wait 10ms (device dependent) 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-sized *Value* is specified, 2 bytes are sent, high byte first, followed by the low byte. This order is different than the way variables are normally stored, low byte first.

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 12CExxx and 16CExxx 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
```

See the Microchip data sheets for these devices for more information.

The timing of the I²C instructions is set so than 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
```

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
```

5.25. IF..THEN

```
IF Comp {AND/OR Comp...} THEN Label
IF 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. All comparisons are unsigned since PBP only supports unsigned types.

Be sure to use parenthesis to specify the order the operations should be tested in. 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**. Another statement may not be placed after the **THEN**, it must be a label.

```
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 loop
```

In the second form, **IF..THEN** can conditionally execute a group of *Statements* following the **THEN**. The *Statements* must be followed by an **ELSE** or **ENDIF** to complete the structure.

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

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

5.26. 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 whole TRIS register at once:

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

5.27. {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
```

5.28. 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** can also use any of the modifiers used with **SEROUT2**. See the section on **SEROUT2** for this information.

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

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 (beginning of first line)
$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, $C0	Move cursor to beginning of second line

Note that there is a command to move the cursor to the beginning of the second line of a 2-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 $0 and the second line starts at $40. The command:

```
LCDOUT $FE, $C0
```

sets the display to start writing characters at the beginning of the second 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.

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 B0, #B1
```

The LCD may be connected to the PICmicro 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, it 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 should be tied to ground as the **LCDOUT** command is write only.

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

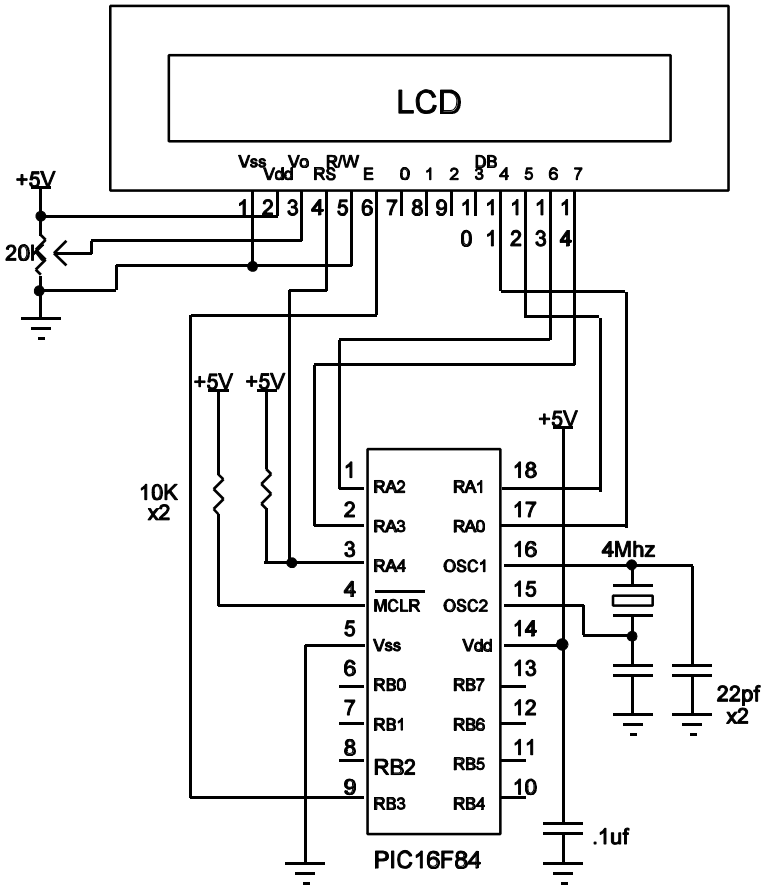
It is also preset to initialize the LCD to a 2 line display.

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   PORTB
‘ Set starting Data bit (0 or 4) if 4-bit bus
DEFINE LCD_DBIT   4
‘ Set LCD Register Select port
DEFINE LCD_RSREG  PORTB
‘ Set LCD Register Select bit
DEFINE LCD_RSBIT  1
‘ Set LCD Enable port
DEFINE LCD_EREG   PORTB
‘ Set LCD Enable bit
DEFINE LCD_EBIT   0
‘ Set LCD bus size (4 or 8 bits)
DEFINE LCD_BITS   4
‘ Set number of lines on LCD
DEFINE LCD_LINES  2
```

This setup will tell PBP a 2-line LCD is connected in 4-bit mode with the data bus on the top 4 bits of PORTB, Register Select on PORTB.1, and Enable on PORTB.0.

The following schematic shows one way to connect an LCD to a PICmicro:



5.29. 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, no action is taken and *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.

```
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
```

5.30. 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, no action is taken and *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 *Value* greater than the *Search* parameter by using "**>**" as the *Test* parameter. If *Test* is left out, "**=**" is assumed.

The *Value* list can be a mixture of 16-bit 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.

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,<[10,100,1000],B0
```

5.31. 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, no action is taken and *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.

```
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
```

5.32. 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, no action is taken and *Var* remains unchanged.

The *Value* list can be a mixture of 16-bit 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.

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
```

5.33. 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
```

5.34. NAP

NAP *Period*

Places the microcontroller into low power mode for short periods of time. During this **NAP**, power consumption is reduced to minimum. The listed periods are only approximate because the timing is derived from the Watchdog Timer which is R/C driven and can vary greatly from chip to chip and over temperature. Since **NAP** uses the Watchdog Timer, its timing is independent of the oscillator frequency.

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
```

5.35. 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. 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**.

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 an instruction after every statement to check whether or not an interrupt has occurred. **DISABLE** turns off the addition of this instruction. **ENABLE** turns it back on again. Usually the additional instruction 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
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
```

5.36. 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 whole TRIS register at once:

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

5.37. PAUSE

PAUSE *Period*

Pause the program for *Period* milliseconds. *Period* is 16-bits, so delays can be up to 65,535 milliseconds (a little over a minute). 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
```

5.38. 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. It has the same accuracy as the system clock.

PAUSEUS takes a minimum number of cycles to operate. Because of this and depending on the frequency of the oscillator, delays of less than a minimum number of microseconds are not possible using **PAUSEUS**. To obtain precise delays less than this, use an assembly language routine in an **ASM..ENDASM** construct. The table below lists the minimum number of microseconds obtainable at a particular oscillator frequency.

OSC	Minimum delay
3 (3.58)	20us
4	24us
8	12us
10	8us
12	7us
16	5us
20	3us

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
```

5.39. PEEK

PEEK *Address,Var*

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

PEEK and **POKE** allow direct access to all of the registers on a PICmicro including PORTA, PORTB, PORTC, PORTD, PORTE and their associated data direction (TRIS) registers. **PEEK** and **POKE** operate on all of the bits, ie. the entire byte, of the particular register at once. When you **POKE** data to PORTA, the entire port is updated, not just one individual bit.

```
PEEK PORTA,B0       ' Get current PORTA pin states
                    to variable B0
```

The PicBasic Pro Compiler can directly access the registers and bits without the need for **PEEK** and **POKE**. It is recommended this direct access be used instead of **PEEK** and **POKE**.

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

5.40. POKE

POKE *Address*,*Value*

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

```
POKE $85,0  ' Write 0 to register hexadecimal 85
            (Sets PORTA to all outputs)
```

The PicBasic Pro Compiler can directly access the registers and bits without the need for **PEEK** and **POKE**. It is recommended this direct access be used instead of **PEEK** and **POKE**.

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

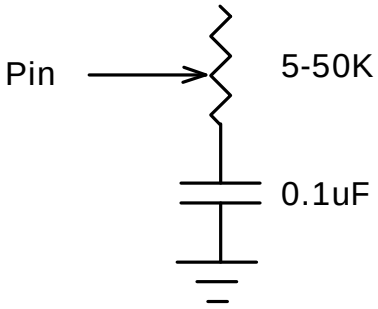
5.41. POT

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 255. Under these conditions, *Var* will produce an appropriate value for *Scale*. (NOTE: This is the same type of process performed by the **Alt-P** option of the BS1 environment).




```
POT 3,255,B0              ' Read potentiometer on
                          Pin3 to determine scale
Serout 0,N2400,[#B0]      ' Send pot value
                          serially out Pin0
```

5.42. 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. If an 8-bit variable is used, only the LSB of the 16-bit measurement is used. *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.

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

5.43. 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
```

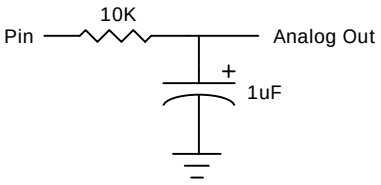
5.44. PWM

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* would be about 5ms long. If a 20Mhz oscillator is used, each *Cycle* would be about 1ms in length. Defining an OSC value has no effect on **PWM**. The *Cycle* time always changes with the actual oscillator speed.

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
```

5.45. 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    ' Get a random number to W4
```

5.46. RCTIME

RCTIME *Pin,State,Var*

RCTIME measures the time a *Pin* stays in a particular *State*. It is basically half a **PULSIN**. *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.

```
Low PORTB.3              ' Discharge cap to start
Pause 10                 ' Discharge for 10ms
RCTIME PORTB.3,0,W0      ' Read potentiometer on
                         Pin3
```

5.47. READ

READ *Address,Var*

Read the on-chip EEPROM at the specified *Address* and stores the result in *Var*. This instruction may only be used with a PICmicro that has an on-chip EEPROM data area such as the PIC16F84 or PIC16C84.

```
READ 5,B2    ' Put the value at EEPROM location 5
             into B2
```

5.48. 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 the *Label* instead of where it was when it was interrupted. In this case, any other return addresses on the stack will no longer be accessible.

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
```

5.49. 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
```

5.50. 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
```

5.51. 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 units of 1 millisecond.

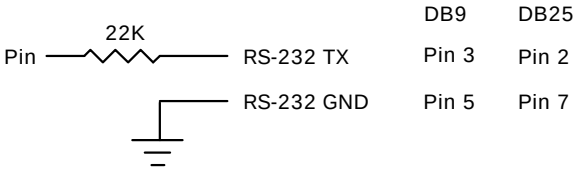
The list of data items to be received may be preceded by one or more qualifiers enclosed within brackets. **SERIN** must receive these bytes in

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.

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 PICmicro allow most applications to run without level converters. Rather, inverted input (**N300**..**N9600**) can be used is conjunction with a current limiting resistor.




```
SERIN 1,N2400,["A"],B0  ' Wait until the
                        character "A" is
                        received serially on
                        Pin1 and put next
                        character into B0
```

5.52. 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

Bit 13 selects even 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).

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).

If *ParityLabel* is included, this label will be jumped to if a character with bad parity is received. It should only be used if even 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 units of 1 millisecond.

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..16}	Receive binary digits
DEC{1..5}	Receive decimal digits
HEX{1..4}	Receive 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.

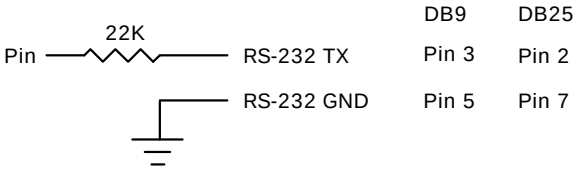
2) A variable preceded by **DEC** will receive the ASCII representation of its decimal value. For example, if **DEC B0** is specified and “123" is received, B0 will be set to 123.
3) A variable preceded by **HEX** will receive the ASCII representation of its hexadecimal value. For example, if **HEX B0** is specified and “FE" is received, B0 will be set to 254.
4) **SKIP** followed by a count 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 string length is determined by the count 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.

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 PICmicro, 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).




```
‘ 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]
```

5.53. 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	

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. For example, if W0 = 123, then #W0 (or #123) will send '1', '2', '3'.

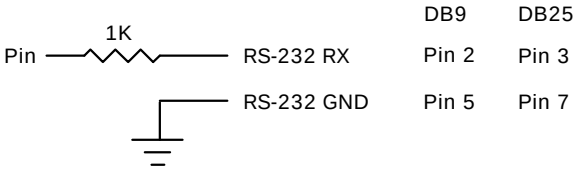
SEROUT assumes a 4Mhz oscillator when generating its bit timing. To maintain the proper baud rate timing with other oscillator values, be sure to **DEFINE** the OSC setting to the new oscillator value.

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 PICmicro, 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
```

5.54. SEROUT2

SEROUT2 *DataPin*{*FlowPin*},*Mode*,{*Pace*,}
{*Timeout*,*Label*,}**[***Item...***]**

Send one or more *Item*s to *DataPin* in standard asynchronous 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 the *FlowPin* does not change to the enabled state within a certain amount of time. *Timeout* is specified in units of 1 millisecond.

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

Bit 13 selects even 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).

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.

Some examples of *Mode* are: *Mode* = 84 (9600 baud, no parity, true, always driven), *Mode* = 16780 (2400 baud, no parity, inverted, driven), *Mode* = 60657 (300 baud, even parity, inverted, open).

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..16}	Send binary digits
{I}{S}DEC{1..5}	Send decimal digits
{I}{S}HEX{1..4}	Send hexadecimal digits
REP c\n	Send character c repeated n times
STR ArrayVar{\n}	Send string of n characters

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 **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 will repeat the character, count time. 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 string length is determined by the count or when a 0 character 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 are unsigned. 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 "-

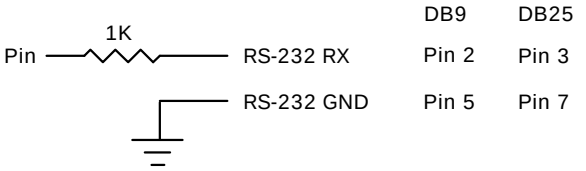
1" since the high order bit is sent. So with a little trickery, the unsigned math of PBP can yield signed results.

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 of 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.

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 PICmicro, 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).




```
' 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]
```

5.55. SHIFTIN

SHIFTIN *DataPin*,*ClockPin*,*Mode*,[*Var*{*Bits*}...]

Clock *ClockPin*, synchronously shift in bits on *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).

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 *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.

Mode	*Mode* No.	Operation
MSBPRE	**0**	Shift data in highest bit first, Read data before sending clock
LSBPRE	**1**	Shift data in lowest bit first, Read data before sending clock
MSBPOST	**2**	Shift data in highest bit first, Read data after sending clock
LSBPOST	**3**	Shift data in lowest bit first, Read data after sending clock

```
SHIFTIN 0,1,MSBPRE,[B0]
```

5.56. 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 *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.

Mode	*Mode* No.	Operation
LSBFIRST	**0**	Shift data out lowest bit first
MSBFIRST	**1**	Shift data out highest bit first

```
SHIFTOUT 0,1,MSBFIRST,[B0]
SHIFTOUT PORTA.1, PORTA.2, 1,[wordvar\4]
```

5.57. SLEEP

SLEEP *Period*

Place microcontroller into low power mode for *Period* seconds. *Period* is 16-bits, so delays can be up to 65,535 seconds (just over 18 hours).

SLEEP uses the Watchdog Timer so it is independent of the actual oscillator frequency. The granularity is about 2.3 seconds and may vary based on device specifics and temperature. This variance is unlike the BASIC Stamp. The change was necessitated because when the PICmicro 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
```

5.58. SOUND

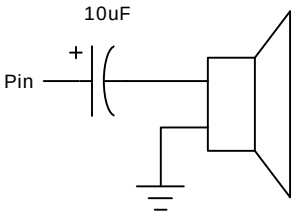
SOUND *Pin*,**[***Note*,*Duration*{,*Note*,*Duration...*}**]**

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 PICmicro, 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
```

5.59. 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
```

5.60. 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 and word 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       ‘ New way
```

5.61. 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
```

5.62. 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
```

5.63. WRITE

WRITE *Address,Value*

Write *Value* to the on-chip EEPROM at the specified *Address*. This instruction may only be used with a PICmicro that has an on-chip EEPROM data area such as the PIC16F84 or PIC16C84.

WRITE is used to set the values of the on-chip EEPROM at runtime. To set the values of the on-chip EEPROM at programming-time, use the **DATA** or **EEPROM** statement.

Each **WRITE** is self-timed and takes about 10 milliseconds to execute on a PICmicro.

```
WRITE 5,B0  ' Send value of B0 to EEPROM
            location 5
```

5.64. 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 the automatically made an input to receive data from the X-10 interface. *ZeroPin* is automatically made an input to received 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.

XOUT below lists the functions as well as the wiring information.

```
housekey    var   word

      ‘ Get X-10 data
loop: XIN PORTA.2,PORTA.0,[housekey]

      ‘ Display X-10 data on LCD
      Lcdout $fe,1,“House=”,#housekey.byte1,
      “Key=”,#housekey.byte0

      Goto loop   ‘ Do it forever


      ‘ Check for X-10 data, go to nodata if none
      XIN PORTA.2,PORTA.0,1,nodata,[housekey]
```

5.65. 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 the automatically made an output to send data to the X-10 interface. *ZeroPin* is automatically made an input to received 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

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]

loop: 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 loop
```

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 exactly what is going on.

6.1. Target Specific Headers

Three target 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 configuration parameters. For the PIC16F84, 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 `P16F84.INC`.

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 PICmicros, 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 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 PICmicro register names.

6.3. PBP 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 library that are used in the .ASM file. If everything is error free up to this point, it spawns 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 PICmicro 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 PICmicro 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 rather than the current top speed of 9600 baud.

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 other technique is to use a different oscillator frequency and tell PBP of your intentions. This is done through the use of a **DEFINE**. **DEFINE**, as demonstrated in the **LCDOUT** command in a previous section, 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 to assume an 8Mhz oscillator will be used instead. The acceptable oscillator definitions are:

OSC Value	Oscillator Used
3	3.58Mhz
4	4Mhz
8	8Mhz
10	10Mhz
12	12Mhz
16	16Mhz
20	20Mhz

Telling PBP the oscillator frequency allows it to compensate and produce the correct timing for **COUNT**, **DEBUG**, **DTMFOUT**, **FREQOUT**, **HSERIN**, **HSEROUT**, **I2CREAD**, **I2CWRITE**, **LCDOUT**, **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. 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.

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. Devices with internal oscillators default to INTRC.

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 progam, 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 PICmicro's RAM registers. The first available RAM location is $0C for the PIC16F84 and some of the smaller PICmicros, and $20 for the PIC16C74 and other larger PICmicros. Refer to the Microchip PICmicro 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 word arrays first (if any), followed by byte and bit arrays. Then space is allocated for words, bytes and finally individual bits. Bits are packed into bytes as possible. This order makes the best use of available RAM.

Arrays must fit into a single bank. They may not cross a bank boundary. This effectively limits the length of an individual array. See the previous section on arrays for these limits.

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.

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.

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 or the names of the PICmicro registers.

The pseudo-ops, variable types and commands keywords are listed in their appropriate sections. The names of the PICmicro registers are defined in the file PIC14EXT.BAS. 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.

PICmicros have a segmented code space. PICmicro instructions 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. 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 for any PICmicro code that crosses a 2K boundary or for any forward references on PICmicros with more than 2K 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 on the other side of a 2K boundary. If the PICmicro has 2K or less of program space, **BRANCH** should be used as it takes up less space than **BRANCHL**. If the microcontroller has more than 2K 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.

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 PicBasic Pro, 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 PICmicro Macro Assembler, and MPASM, Microchip's assembler. PM is included with the compiler while MPASM must be obtained directly from Microchip via their web site or is included with their PICmicro programmers.

There are benefits and drawbacks to using each assembler. PM is handy because it is included as part of PBP. It is also much 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 PICmicro Macro Assembler, see the `PM.TXT` file on disk.

MPASM, on the other hand, has the capability of creating a `.COD` file. This file 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, simply copy all of the MPASM files into their own subdirectory, perhaps named MPASM. This subdirectory must also be in the DOS PATH.

MPASM may be used with PBP in two ways. If the command line option "-ampasm" is used, MPASM will be launched following compilation to complete the process. MPASM will display its own screen with its progress.

```
PBP -ampasm filename
```

Alternatively, the command line option "-amp" will launch MPASM in quiet mode and only display any errors. However, the launcher consumes additional memory that is therefore not available to MPASM.

```
PBP -amp filename
```

For maximum memory availability to MPASM, the command line option "-ampasm" should be used or the Windows version of MPASM should be used.

In any case, MPASM is not included with PBP and must be obtained from Microchip.

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 PM, the PICmicro Macro 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.

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 be reduce problems. If you should have a name collision, the compiler 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 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 2K (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 2K, 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

PICmicro 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 Jump. 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 PICmicro 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 to begin with.

The PicBasic Pro library routines can use up to 4 stack locations themselves. The remaining 4 are reserved for **CALL**s and nested BASIC **GOSUB**s. You must make sure that your **GOSUB**s are only nested

3 deep at most with no **CALLs** 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 the PIE register.

Refer to the Microchip PICmicro 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 in conjunction with 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 PICmicro. 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 after 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 every time the GIE bit is checked.

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.

One final note about interrupts in BASIC: If the program uses the form:

```
loop: Goto loop
```

and expects to be interrupted, it isn't going to happen. Keep in mind the interrupt flag is checked after each instruction. There really isn't a place for the check after a **GOTO**. It immediately jumps to the loop with no interrupt check. Some other statement must be placed in the loop for the interrupt check to happen.

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 16C74(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.

The best place to put the assembly language interrupt handler is probably at the very beginning of your PicBasic Pro program. This should ensure that it is in the first 2K to minimize boundary issues. 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 the PICmicro 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
```

You must restore these registers at the end of your assembler interrupt handler. If the PICmicro has 2K or less of code space, 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
```

Label is the beginning of your interrupt routine. PBP will place a jump to this *Label* at location 4 in the PICmicro.

```
' 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
      swapf STATUS, W
      clrf  STATUS
      movwf ssave
      movf  PCLATH, W
      movwf psave
```

```
; 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

loop: Goto loop   ‘ 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 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 PICmicro (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 PICmicro 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 PICmicro, 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 PICmicro 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 will define the standard BS1 or BS2 variables. However, it is recommended that you assign your own variables with meaningful names rather than using either of 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 change nibble variable types to bytes.

10.7. Math Operators

Mathematical expressions now 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.

Parenthesis should be used to group operations into the order in which they are to be performed. In this manner, there will be no doubt about the order of the operations.

The following table list the operators in hierarchal order:

Highest Precedence
( )
NOT
~
-
SQR ABS DCD NCD COS SIN
*

Highest Precedence
**
*/
/
//
+
-
<<
>>
MIN
MAX
DIG
REV
&
^
|
&/
/|
^/
&& AND
^^ XOR
|| OR
Lowest Precedence

10.8. [ ] 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.

10.9. DATA, EEPROM, READ and WRITE

The BASIC Stamp allows EEPROM not used for program storage to store non-volatile data. Since PBP programs execute directly from the PICmicro's ROM space, EEPROM storage must be implemented in some other manner.

The PIC16F84 (the default target for PBP programs), PIC16F83 and PIC16C84 have 64 bytes of on-chip EEPROM. PBP programs may use this for EEPROM operations and supports the Stamp's **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.

10.10. 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.

Question marks (?) in **DEBUG** statements are ignored. The modifier ASC? should not be used.

10.11. 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 PICmicros 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 still available, subroutines may still be nested up to four levels deep and the number of **GOSUB**s is limited only by the PICmicro's code space.

10.12. 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.

10.13. IF..THEN

The BASIC Stamps and 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**.

10.14. 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.

10.15. 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.

10.16. SLEEP

PBP's **SLEEP** command is based solely on the Watchdog Timer. It is no longer calibrated using the system clock oscillator. This change was necessitated by the effect Watchdog Timer resets have on the PICmicro.

Whenever the PICmicro was reset during **SLEEP** calibration, it altered the states of some of the internal registers. For smaller PICmicros with few registers, these registers could be saved before and restored after calibration resets. However, since PBP may be used on many different PICmicros with many registers that are altered upon reset, this save and restore proved to be too unwieldy.

Therefore it has been decided to run **SLEEP** in an uncalibrated mode based strictly upon the accuracy of the Watchdog Timer. This ensures the stability of the PICmicro 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

Summary of Microchip Assembly Instruction Set

ADDLW	k
ADDWF	f,d
ANDLW	k
ANDWF	f,d
BCF	f,b
BSF	f,b
BTFSC	f,b
BTFSS	f,b
CALL	k
CLRF	f
CLRW	
CLRWDT	
COMF	f,d
DECF	f,d
DECFSZ	f,d
GOTO	k
INCF	f,d
INCFSZ	f,d
IORLW	k
IORWF	f,d
MOVF	f,d
MOVLW k	
MOVWF f	
NOP	
RETFIE	
RETLW	k
RETURN	
RLF	f,d
RRF	f,d
SLEEP	
SUBLW	k
SUBWF	f,d
SWAPF	f,d
XORLW	k
XORWF	f,d

b - bit address
d - destination; 0 = w, 1 = f
f - register file address
k - literal constant

Appendix B

Contact Information

Technical support and sales may be reached at:

microEngineering Labs, Inc.
Box 7532
Colorado Springs CO 80933-7532
(719) 520-5323
(719) 520-1867 fax
http://www.melabs.com
email:support@melabs.com

PICmicro data sheets, CD-ROMs and literature may be obtained from:

Microchip Technology Inc.
2355 W. Chandler Blvd.
Chandler AZ 85224-6199
(602) 786-7200
(602) 899-9210 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 disk 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 disk 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.