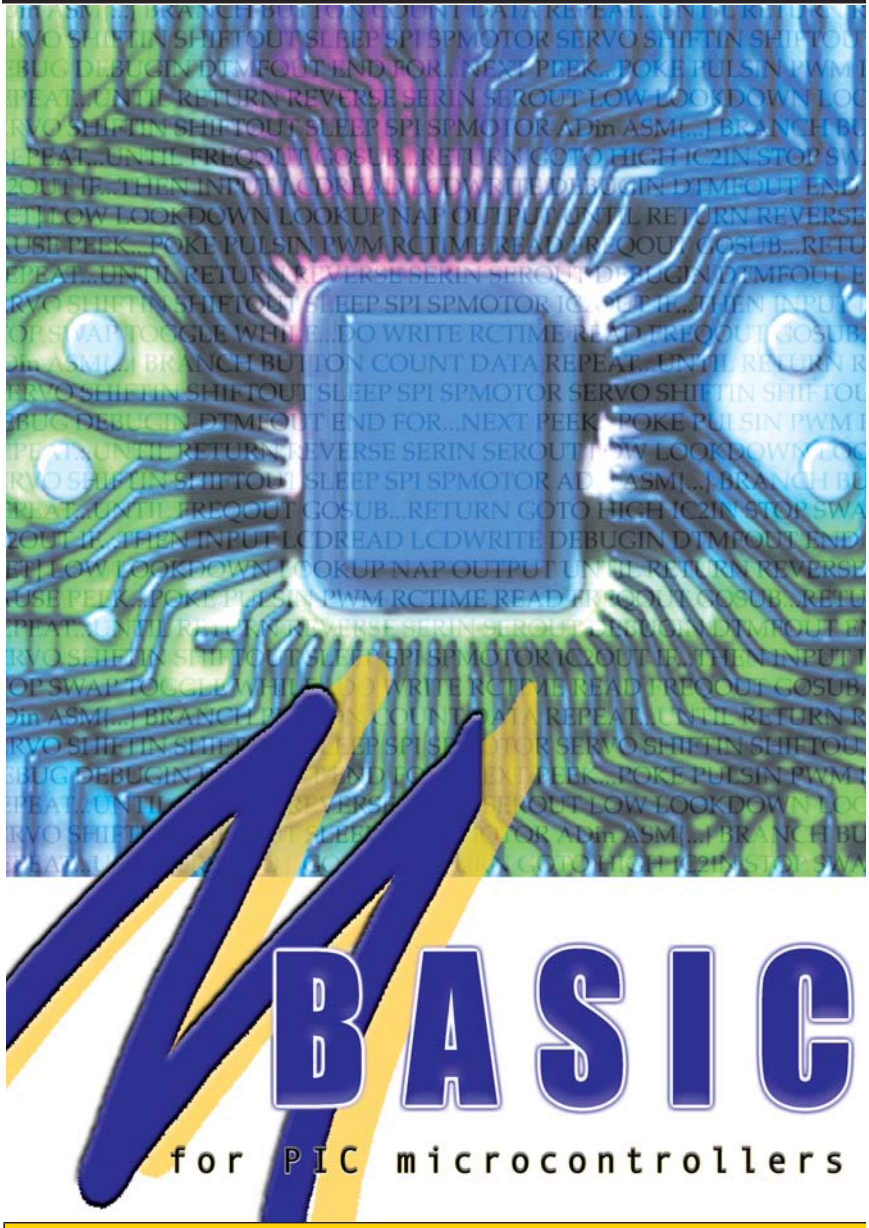
BASICMICRO

Microcontrollers made easy

User's Guide



BASIC

for PIC microcontrollers

Learn to Program PIC Mircrontrollers in easy to use Basic

Revision 5.2

Warranty

Basic Micro warranties its products against defects in material and workmanship for a period of 90 days. If a defect is discovered, Basic Micro will at our discretion repair, replace, or refund the purchase price of the product in question. Contact us at support@basicmicro.com
No returns will be accepted without the proper authorization.

Copyrights and Trademarks

Disclaimer

Basic Micro cannot be held responsible for any incidental, or consequential damages resulting from use of products manufactured or sold by Basic Micro or its distributors. No products from Basic Micro should be used in any medical devices and/or medical situations. No product should be used in a life support situation.

Contacts

Email: sales@basicmicro.com
Tech support: support@basicmicro.com
Web: http://www.basicmicro.com

Discussion List

A web based discussion board is maintained at
http://www.basicmicro.com

Updates

In our continuing effort to provide the best and most innovative products, software updates are made available by contacting us at
support@basicmicro.com

Table of Contents

BASIC

for PIC microcontrollers

Contents

Introduction

BASIC

for PIC microcontrollers

Introduction

Welcome to the world of microcontrollers. We would like to Thank you for your purchase.

What is MBasic ?

MBasic is an advanced programming language modeled after BASIC, creating a cost effective and flexible way to program PIC Microcontrollers. MBasic maintains the simplicity of Basic, but offers more power. MBasic includes a super set of the BS2 instruction set, allowing you to easily port your current BS2 applications with little effort.

This Manual

This manual in general applies to MBasic and minimal hardware. It is not in the scope of this manual to cover any Chip specific or Peripherals features of PICmicros. Further information can be found in the Data Sheets available from Microchip.com for all PICmicros.

This manual will explain the MBasic programming language in depth . The main purpose of this manual is to teach the general syntax. Which will give you, the end user, a good understanding of how to effectively use MBasic.

We will continue to update and improve this manual. All updates will be made available for download from our web site at http://www.basicmicro.com.

On-line Discussion Forums

We maintain discussion forums at http://www.basicmicro.com in order to help you to connect with a wide range of related information and users. The discussion forums are free and will allow you to find information and help fast.

Updates

MBasic updates will be available to new and current customers. To receive email notifications of updates, join the discussion forums at http://www.basicmicro.com.

Technical Support

Technical support is provided via e-mail and the discussion forums at www.basicmicro.com. When technical support is required please send e-mail to **support@basicmicro.com** . In order to assure a proper response please include a copy of the program you are having problems with, the hardware you are using, MBasic revision number, prototyping board and so on. By including this information with your e-mail, you can help us answer your questions quickly. Additional technical support is often provided by several experienced users of the discussion forums at http://www.basicmicro.com

Basic PICmicro Setup

BASIC
for PIC microcontrollers

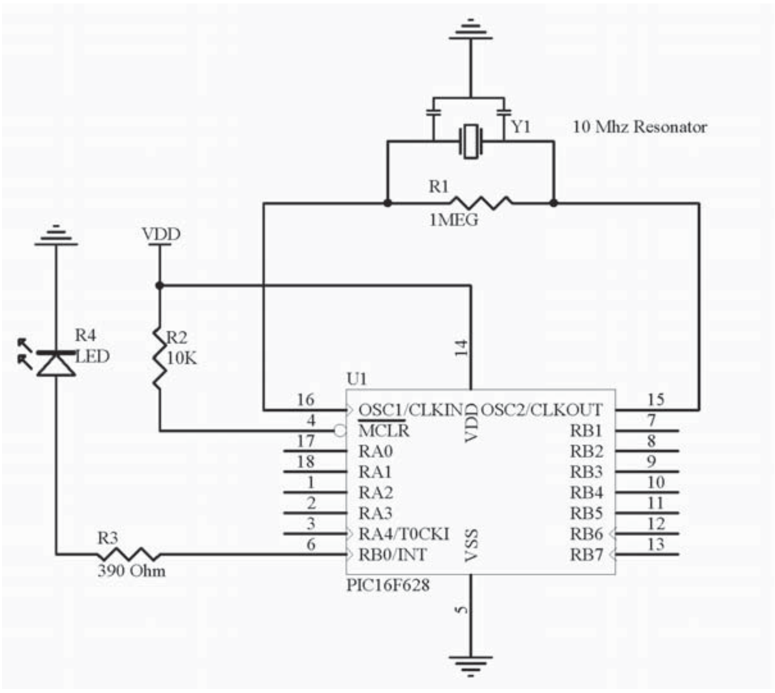
Which PICmicro should I use ?

Doing a quick search on the internet will turn up a ton of links and projects with sample code using the 16F84. The 16F84 has 1K of code space and 68 bytes of RAM at a cost around $5.00 in small quantity. Microchip came out with a 100% pin compatible chip that is a replacement for the 16F84. This PICmicro is a 16F628 which has 2K of code space and 224 bytes of RAM. The 16F628 also has many hardware features and it can be purchased for about $2.50 in small quantity. Any sample code for the 16F84 will work with the 16F628. With the 16F628 you get more code space, more ram for a smaller price.

The second chip of choice, the one this manual and most of the sample code is written for is the 16F876. This is a 28 pin, 8K program space, 384 bytes of RAM PICmicro.

Basic PICmicro Schematic

The schematic shown is the most basic circuit for setting up a PICmicro. The basic circuit consist of a resonator with built in caps, 10K resistor, power and ground. The PICmicro requires +5Volts.



What Next ?

You can build the circuit shown with the 16F628 or 16F876. You should download the data sheet from Microchip.com on the chip you plan on using. The best approach to experimenting with PICmicros is a development platform. This will save endless hours with wiring troubles and faulty circuits. There are several “Getting Started” packages available from the Basic Micro web site. This isn’t a sales pitch this is years of experience with PICmicros. Just about 90% of the problems you will run into is bad hardware. Using something like the 2840 Solderless Development board and ISP-PRO will dramatically reduce the amount of time you spend trouble shooting hardware. Plus save time, the PICmicro can be programmed in circuit. So for every typo, syntax error or program mistake, you won’t need to remove the PICmicro from its circuit. There will be plenty of those. Now with all that said lets get started.

Getting Started

BASIC

for PIC microcontrollers

Software Installation

MBasic software will run on 95, 98, ME, NT, 2000, and XP. DOS is not supported.

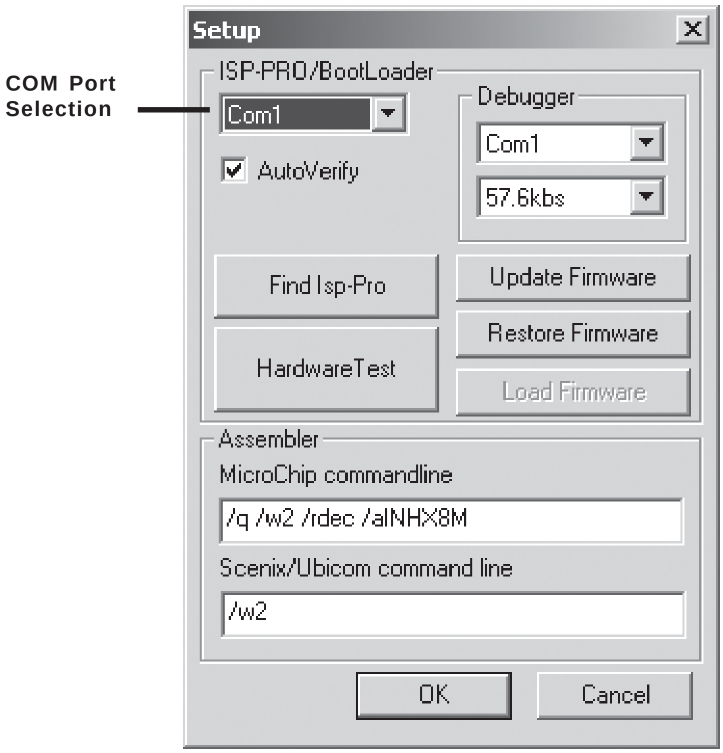
If the software is downloaded, double click the .exe application it will automatically unzip. Then double click the setup.exe. Follow the on screen prompts (It is not recommended to change the default directories until you fully understand the software). Once the installation is complete restart your computer.

If the software is installed from the CD-ROM; insert the CD-ROM into your computer. If auto-run is enabled an installer menu will appear, select your software and the installation process will begin automatically. Restart your computer after the installation is complete.

Note: Explore the CD-ROM after installation, it contains sample code, data sheets and more.

Configuring MBasic

If you are using the ISP-PRO programmer the COM port will need to be configured. Open the System Setup Menu under Tools (Tools-> System Setup) Choose the COM port the ISP-PRO is attached to.



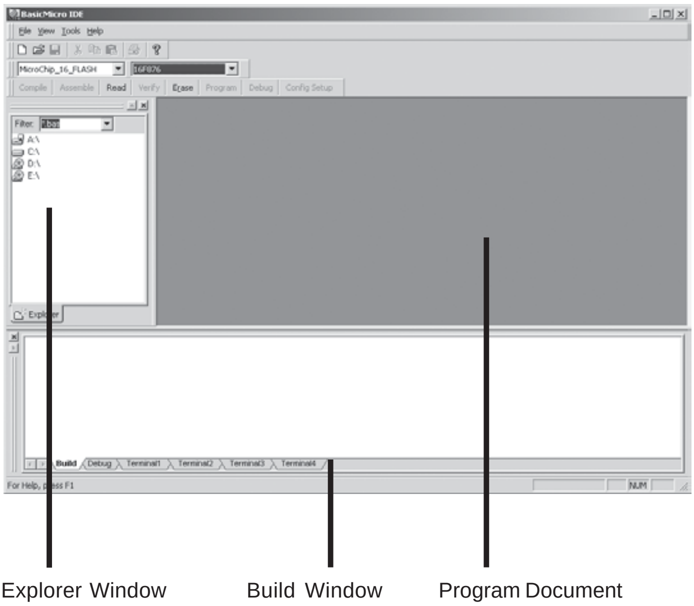
ISP-PRO/BootLoader: Com1
✓ AutoVerify
Debugger: Com1, 57.6kbs
MicroChip commandline: /q /w2 /rdec /aINHX8M
Scenix/Ubicom command line: /w2

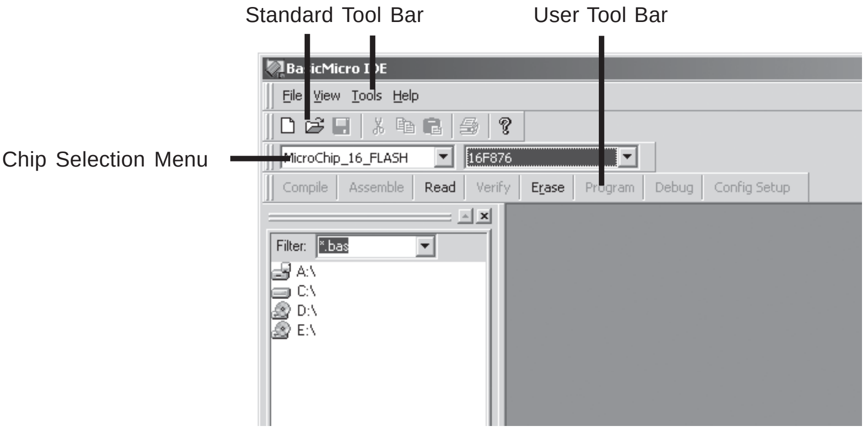
What is an IDE ?

IDE stands for Integrated Development Environment. The IDE associated with MBasic software is used to perform all the tasks associated with using PICmicros, such as Writing code, Compiling, and Programming.

Getting Familiar with the IDE

Take a few minutes to read through this section and familiarize yourself with the IDE. Understanding all the features of the IDE will make it easier to use MBasic more efficiently.





File Explorer Window
This window displays all the files on your computer. It is not auto updating, occasionally you may need to right click on it and select refresh.

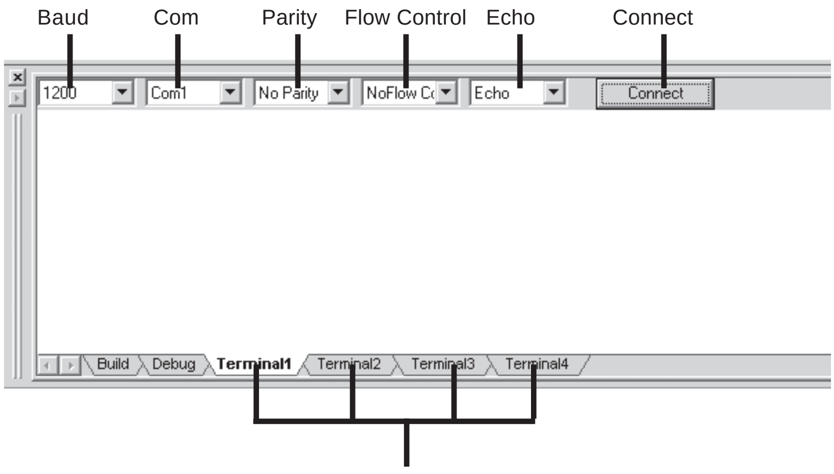
Build Window
The Build Window is used to display compiling information and progress of the program being executed. Available program memory, ram and errors will be listed here after compiling is complete. If an error is listed simply double click on it and the IDE will automatically highlight the problem line of the program.

Program Document
This is a multi document interface. You can have several programs open at once. The IDE will only compile and deal with the current document that is in focus.

User Tool Bar
This menu contains all the tasks that can be performed such as, Compile, Program and Debug.

Standard Tool Bar
These are small Icons representing different shortcuts from the menu bar. Dragging the cursor over each ICON will display a small message that explains the function of that ICON.

Chip Selection Menu
This menu contains all the supported chip names. When programming or creating a new file, the correct chip you are using must be selected using this menu.



Terminal Window Selection

Terminal Window Selection

The IDE comes with built-in RS-232 serial communication terminals much like Hyper Term. You can have up to 4 terminal windows running at once. The number of terminal windows connected to COM ports are also limited by the number of COM ports your computer is equipped with.

Control Characters

The terminal window supports several control characters for cursor positioning. The following chart is a list of all the control characters supported. The terminal window only supports the ASCII values. No symbols are pre-defined.

Name	ASCII Value	Description
Clear Screen	0	Clear the screen and move cursor to the home position
Home	1	Move cursor to the upper left corner of the screen
Move to X,Y	2	Move cursor to location specified by X,Y (i.e. 2,X,Y theses values can not be greater than 255)
Cursor Left	3	Move cursor one character left.
Cursor Right	4	Move cursor one character right.
Cursor Up	5	Move cursor one character up.
Cursor Down	6	Move cursor one character down.
Bell	7	Beep PC speaker.
Backspace	8	Delete one character to the left.
Tab	9	Tab to next column.
Line Feed	10	Move cursor down one line.
Clear Right	11	Clear entire line to the right.
Clear Down	12	Clear screen contents below cursor.
Carriage Return	13	Move cursor to first position of next line.

Terminal Window Functions

Baud
The Baud menu sets the communication speed of the terminal window. Baud rates from 300 to 460800 bits per second are possible.

Com
The Com menu is used to select the COM port the terminal window will use. Only the COM ports available on your computer will be shown here.

Parity
The Parity menu is used to select if the parity is used or not being used. Most setups will require No Parity.

Flow Control
The FlowControl menu is used to select software flow control. If Flow Control is selected, the CTS and RTS hardware lines are used to control the flow of data.

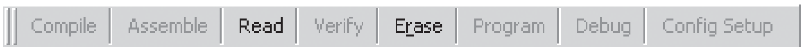
Echo
The Echo menu is used to display the sent character automatically. When this menu is set to No Echo and a character is typed in the output screen, nothing will be displayed, only data received will be shown. If Echo is selected any character typed will be displayed.

Connect
Clicking the “Connect” button connects the serial port that the terminal window is set to use. Once the serial port is open the connect button will display “Disconnect” . As long as the serial port is open it will be unaccessible by other programs. Once you are finished close the connection.

User Tool Bar

Main programming functions are available as buttons on the tool bar.

Compile Assemble Read Verify Erase Program Debug ConfigSetup



Compile
The compile function will compile the current program and check for syntax errors.

Assemble
Will pass an assembly file to the assembler.

Read
This function will only work with the available Boot Loader or ISP-PRO and a PICmicro conntected. Read will read the contents of the PIC and display the file in HEX format to the screen.

Verify
The verify function will read in from the PICmicro and match the current open file to verify they are the same.

Erase
This function will only work with the available Boot Loader or ISP-PRO and a PICmicro conntected. Erase will clear or set all the programming locations on the PICmicro to 3fff. Which is considered blank.

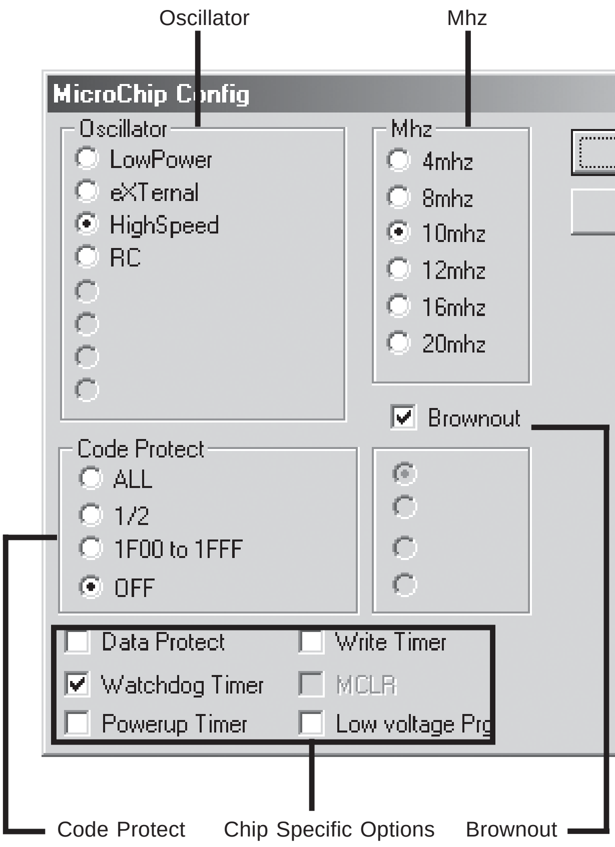
Program
The program button will compile, then program the target device. The program function is one click.

Debug
The debug function will perform the same task as the Program option, but will add special code the ICD and set the IDE to Debug mode. Debug requires the ISP-PRO programmer our supported boot loader device.

Config Setup
The config setup menu is used to set the configuration settings on the selected PICmicro. Most problems when programming are caused due to improper configuration settings.

Config Setup Menu

The config setup menu will list all the specific options for a selected device. Any time before programming a target device the config setup menu should be checked to ensure all the options are set correctly for the selected device.



MicroChip Config

Oscillator: ( ) LowPower ( ) eXTernal [✓ HighSpeed] ( ) RC

Mhz: ( ) 4mhz ( ) 8mhz [✓ 10mhz] ( ) 12mhz ( ) 16mhz ( ) 20mhz

[✓ Brownout]

Code Protect: ( ) ALL ( ) 1/2 ( ) 1F00 to 1FFF [✓ OFF]

[ ] Data Protect [ ] Write Timer
[✓ Watchdog Timer] [ ] MCLR
[ ] Powerup Timer [ ] Low voltage Prg

OK Cancel

Oscillator
The oscillator section will display all the different options available for a selected device. These options will vary from device to device. Some of the common options are as follows:

LowPower	=	Low power mode.
eXTernal	=	External oscillator.
HighSpeed	=	Set when oscillator is 10mhz and above.
RC	=	Sets internal RC circuit for oscillator.

The High Speed option should be selected during the design phase. Afterwards other options can be used. To understand all the options available refer to the specific target device data sheet.

Mhz
The mhz option is a list of available frequency the selected device can operate within. When setting this option check the device for its maximum range. Some parts can run up to 4mhz while others can run up to 20mhz. There is no way for the software to detect which speed part is in use, so this must be set according to the device being used.

Code Protect
The code protect section will display various options based on the device selected. Some of the more common ones are as follows:

ALL	=	Code Protect all of the device.
1/2	=	Code Protect the first half. 8K = 4K protected.
1F00 to 1FFF	=	Code Protect location 1F00 to 1FFF.
OFF	=	No Code Protect.

On some windowed UV erasable parts setting the code protect will ruin them for further use. Once the code protect is set it can not be erased. On flash parts this is not the case, they can be erase once code protected.

Chip Specific Options
The chip specific options will vary from device to device. The more common options are as follows:

Data Protect	=	Code Protect the on Board EEPROM.
Watch Dog Timer	=	Watch Dog timer on / off.
Powerup Timer	=	Delayed program execution on / off.
Write Timer	=	Write Timer on / off.
MCLR	=	Set MCLR pin on / off.
Low Voltage Prg	=	LVP on / off Set this off always.

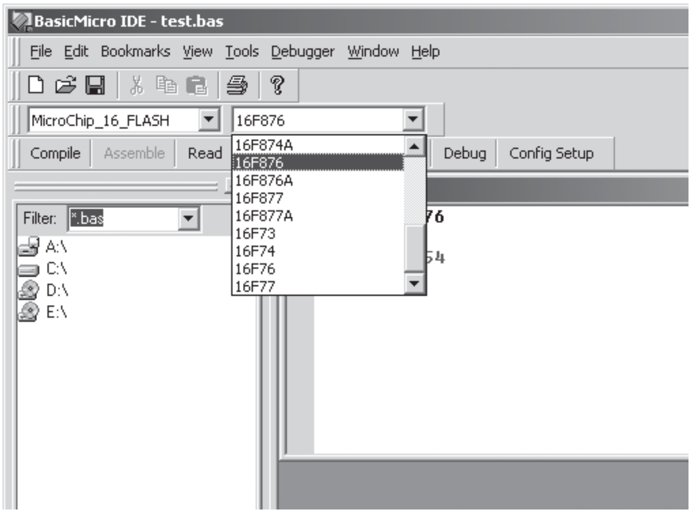
Brownout
The brownout option sets the brownout circuit on or off. The brownout detect is an internal circuit on some devices. This circuit detects voltage levels. When the voltage falls below a certain threshold this circuit will reset the device and continue to reset the device until voltage levels return to normal. This is used to prevent erratic device operation.

First Program

A 16F876 is assumed for this first example. If a different chip is used replace the reference to 16F876 to your chip type. Most of the configuration settings will be the same.

1. Select File ->New. A file type dialog will appear. Select Mbasic file.

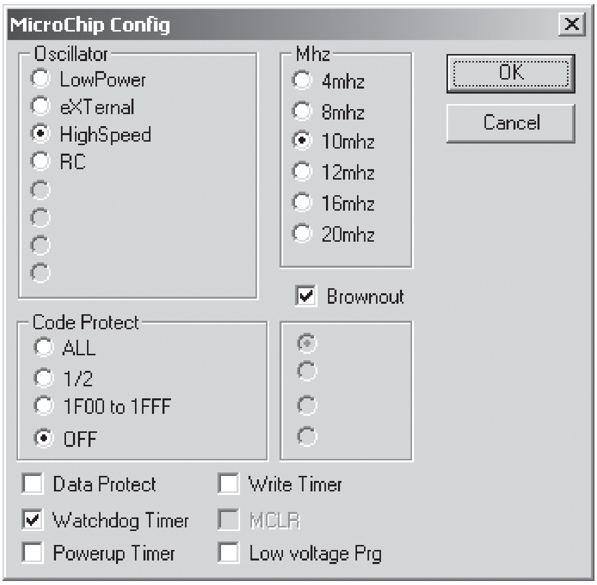
2. Set the Chip Selection Menu to 16F876 as shown below.



BasicMicro IDE - test.bas
File Edit Bookmarks View Tools Debugger Window Help
MicroChip_16_FLASH
16F876
Compile Assemble Read Debug Config Setup
16F874A
16F876
16F876A
16F877
16F877A
16F73
16F74
16F76
16F77
Filter: *.bas
A:\
C:\
D:\
E:\

3. Click the “Config Setup” button (or menu) select “High Speed” for the oscillator type. If you are using a Basic Micro development board the factory supplied oscillator is a 10mhz. In the config menu select “10” for the Mhz setting. If your are using some other oscillator speed set the “Mhz” setting to the speed in which you are using. Make sure “Low Voltage Programming” is off (or unchecked). The remaining configuration settings are defaults and will work in most situations. Once you have completed the above steps, close the config setup menu. The config menu and header of your MBasic file should look as shown (Provided you are using 16F876 and default oscillator).

```
CPU = 16F876
MHZ = 10
CONFIG 16254
```


MicroChip Config

Oscillator: LowPower, eXTernal, [✓ HighSpeed], RC

Mhz: 4mhz, 8mhz, [✓ 10mhz], 12mhz, 16mhz, 20mhz

[✓ Brownout]

Code Protect: ALL, 1/2, 1F00 to 1FFF, [✓ OFF]

☐ Data Protect ☐ Write Timer

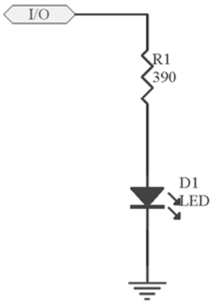
☑ Watchdog Timer ☐ MCLR

☐ Powerup Timer ☐ Low voltage Prg

4. Next type in the following program:

```
Main
    High B0
    Pause 200
    Low B0
    Pause 200
    End
```

5. Once you have entered the above program choose File -> Save As. Wire up an LED as shown in the circuit below to pin B0.



6. Ensure the ISP-PRO is connected, power is on and attached to the target device. Next click the “Program” button. This will compile your program and download it to the PICmicro. If the PICmicro was programmed successfully no error messages should appear. If you do receive an error message

recheck your connections and ensure the correct COM port was selected for the ISP-PRO.

If you have connected the LED correctly and successfully downloaded the program you should have a blinking LED ! Congratulations you have successfully programmed your first PICmicro !

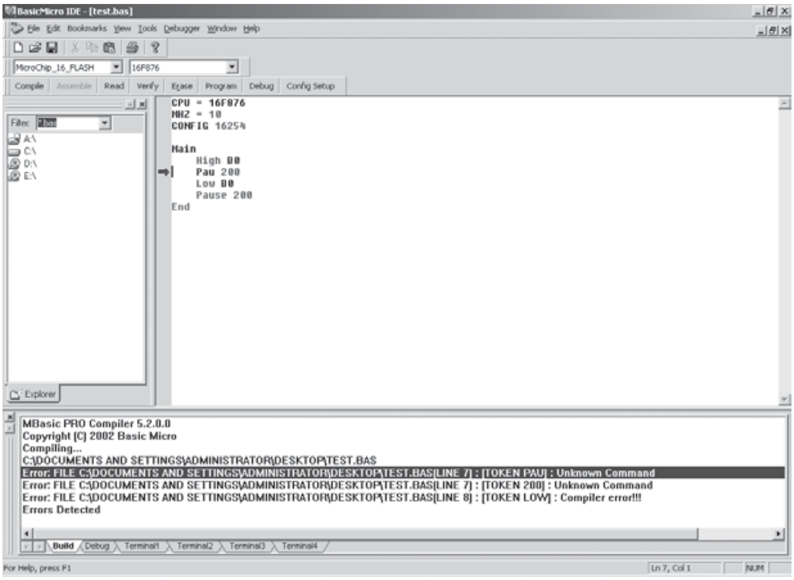
Error Reporting

In next example program we are going to intentionally produce an error. This next example will demonstrate the error reporting abilities of MBasic and how to use it. Make sure to include the intentional misspelling of the Pause command leaving off the "se".

Type in the following program:

```
	CPU = 16F876
	MHZ = 10
	CONFIG 16254

Main
		High B0
		Pau 200
		Low B0
		Pause 200
	End
```

Once you have entered the example program, click "Compile". MBasic will report a few errors in the build window as shown.


BasicMicro IDE - [test.bas]
File Edit Bookmarks View Tools Debugger Window Help
MicroChip_16_FLASH
16F876
Compile Assemble Read Verify Erase Program Debug Config Setup
Filter:
A:\
C:\
D:\
E:\
Explorer
Build Debug Terminal1 Terminal2 Terminal3 Terminal4
For Help, press F1
Ln 7, Col 1
NUM

```
CPU = 16F876
MHZ = 10
CONFIG 16254

Main
    High B0
    Pau 200
    Low B0
    Pause 200
End
```

```
MBasic PRO Compiler 5.2.0.0
Copyright (C) 2002 Basic Micro
Compiling...
C:\DOCUMENTS AND SETTINGS\ADMINISTRATOR\DESKTOP\TEST.BAS
Error: FILE C:\DOCUMENTS AND SETTINGS\ADMINISTRATOR\DESKTOP\TEST.BAS(LINE 7) : [TOKEN PAU] : Unknown Command
Error: FILE C:\DOCUMENTS AND SETTINGS\ADMINISTRATOR\DESKTOP\TEST.BAS(LINE 7) : [TOKEN 200] : Unknown Command
Error: FILE C:\DOCUMENTS AND SETTINGS\ADMINISTRATOR\DESKTOP\TEST.BAS(LINE 8) : [TOKEN LOW] : Compiler error!!!
Errors Detected
```

By clicking on the actual error in the build window the cursor and a blue arrow will indicate at what line the error occurred.

In most cases more than one error will be reported, even when there is only one error. This is due to the first error causing MBasic to treat the following values as garbage. Any time you get multiple errors start from the top down.

In many cases the error shown may have actually occurred on a line before the actual line indicated. This is because MBasic may interpret the first error as a label and the next statement did not make sense to the compiler. The error reporting is a generalization of the actual error. In some cases it may take some time to find the actual error.

Conclusion

In most cases programming PICmicros can be that easy. However there are many factors to each PICmicro that may make the learning process difficult to a beginner. You must in several cases refer to the data sheet for the specific device you are using. MBasic is not an end all stop for programming PICmicros. There are several hardware specific issues you may have to learn, in order to use any given PICmicro.

In Circuit Debugger

BASIC

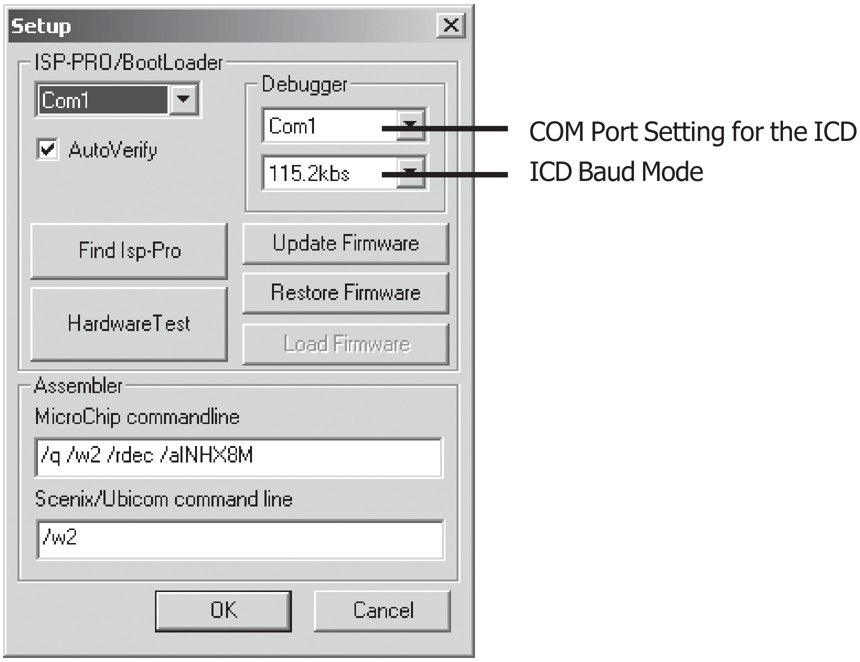
for PIC microcontrollers

ICD Setup

There are a few options that must be set before you begin to use the ICD. The first option is what COM port you wish to use. This is only valid if your are using a Boot Loader chip or have more than one ISP-PRO. Otherwise the COM port will need to stay the same as the COM port selected for the attached ISP-PRO. The next option is the speed at which the ICD will run at. The maximum value for most computers will be 115K baud. When using the ICD, if you get erratic operations attempt to lower your baud speed.

USB To Serial Adapter

Since USB to Serial adapters can handle higher speeds, you can set the ICD to a faster speed. The ISP-PRO must be attached with a USB to Serial adapter.



Setup

ISP-PRO/BootLoader: Com1

✓ AutoVerify

Debugger: Com1 — COM Port Setting for the ICD

115.2kbs — ICD Baud Mode

Assembler

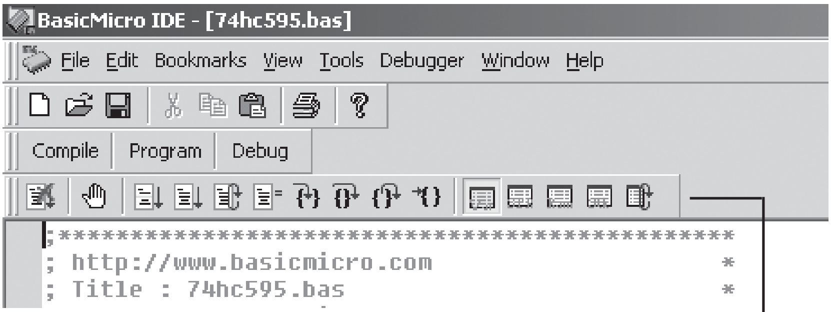
MicroChip commandline: /q /w2 /rdec /aINHX8M

Scenix/Ubicom command line: /w2

What is an ICD ?

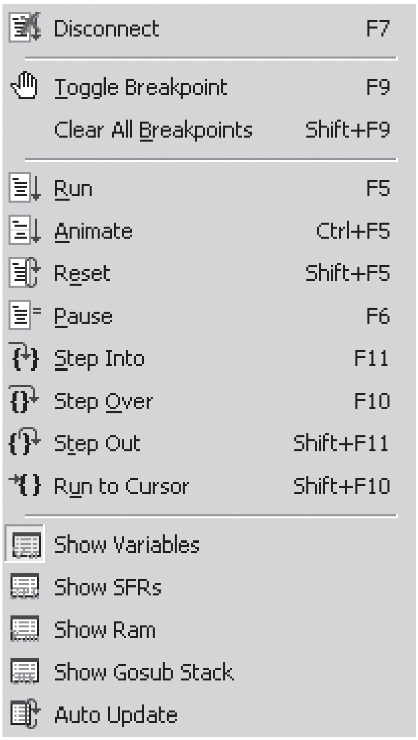
ICD stands for In Circuit Debugger. The ICD provides an easy way to DEBUG your code on-the-fly. With MBasic's ICD you can watch your code run line-by-line live as each instruction is executed by the PICmicro MCU. In the past ICDs have been complicated and could only be used by the most experienced users. With MBasic's ICD this is no longer the case, any user can easily learn to use the ICD built into MBasic.

Getting Familiar with the ICD Controls



BasicMicro IDE - [74hc595.bas]
File Edit Bookmarks View Tools Debugger Window Help
Compile Program Debug

```
;*************************************************
; http://www.basicmicro.com                      *
; Title  : 74hc595.bas                           *
```



Disconnect	F7
Toggle Breakpoint	F9
Clear All Breakpoints	Shift+F9
Run	F5
Animate	Ctrl+F5
Reset	Shift+F5
Pause	F6
Step Into	F11
Step Over	F10
Step Out	Shift+F11
Run to Cursor	Shift+F10
Show Variables	
Show SFRs	
Show Ram	
Show Gosub Stack	
Auto Update	

Debug Tool Bar

Debug Menu

Connect
The connect button is used to establish communications between the ICD / Watch Window and the target device. This is done automatically after programming using the “Debug” button. A green bar will highlight the first line of code indicating it has successfully connected. Once connection has been established the button will change to “Disconnect”

Toggle Break Point
If a break point is not set on the line with the cursor, the toggle break point will set a break point on that line. If a break point is set it will clear it.

Animate
The animate button will animate the displayed program line-by-line as it is executed on the target device. This is done with a high lighted green line across the screen. This mark indicates the current line of code being executed. The marker will flow with the program as it is executed.

Run
The run button will start program execution on the target once the connection is established.

Reset
Reset is used to restart the program currently running on the target device. Reset does not clear out any previously stored values in ram, which means any previous values stored in variables will still be there. To clear all the RAM locations when your program starts running make sure to use a CLEAR command in the beginning of the program. This will set all variables to zero’s each time Reset is used when running the ICD.

Pause
The pause button will pause program execution, to resume the program, click RUN or ANIMATE.

Step Into
The step button allows you to step through the current running program line-by-line

Step Over
This allows the program to step over a routine, mainly a gosub and or for..next loop.

Step Out
Step out will allow you to step out of a gosub routine. This allows you to skip any gosub in your program and go to the next line after the routine.

Run To Cursor
Clicking on any part of the displayed program will produce a blinking cursor. Using the “Run to Cursor” function will allow the program to run until the cursor is reached.

Show Variables
The variable button if clicked, will open a small new window that displays all the current variables used in the current program and the current values in HEX, Decimal, Binary and floating point. (Note: AutoUpdate must be enabled for these values to be updated.)

Show SFRs
SFRs stands for Special Function Registers. These are the registers built into the PICmicro MCU. Some examples are the GIE, ADCON and so on. If the SFRs button is clicked a small window will display all the current SFRs used in the current program and their status.

Show Ram
The ram button if clicked, will open a small window that displays all the ram values in the Atom.

Show Gosub Stack
Displays the gosub stack. This indicates where a program is in a gosub routine. Useful if a program has many levels of gosubs. (Note: For advanced Users.)

Set Auto Update
The auto update feature if checked will automatically update the SFR, Variable, RAM and Stack screens when the program is in animate mode.

Using the ICD

To best illustrate the ICD, we will setup and run a simple program. Once you have completed this exercise you should be able to use the ICD with any program.

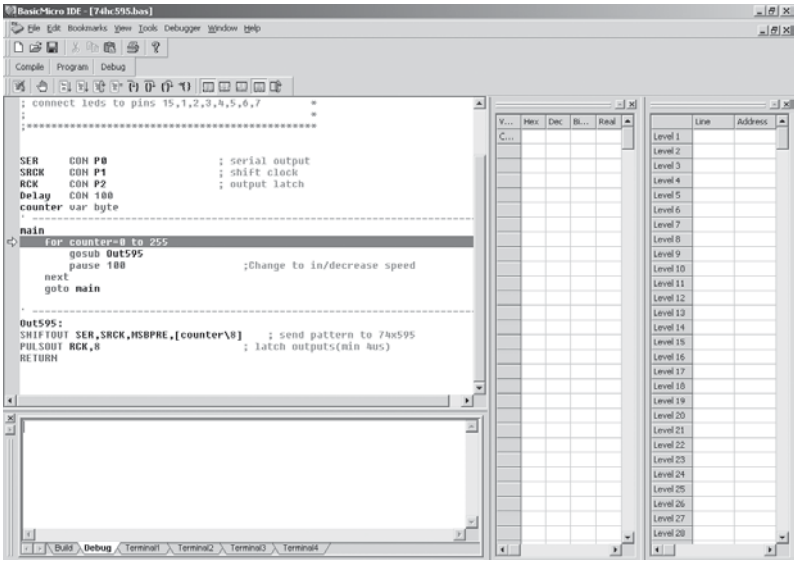
Important Note

When using the ICD, the running program has an added delay anywhere from .5ms to 500ms per command. The user must take this into account when any timing critical commands or program functions are being debugged.

Exercise

Start the IDE, create a new file. Save the file as icd.bas. It will be referred to as such in the remainder of this section (Refer to *Getting Started* section for basic setup)

```
CPU = 16F876
MHZ = 10
CONFIG 16254

Temp    var     Byte
Temp1   Var     Word

Temp1 = 0
Temp = 0

Main
    For temp = 1 to 20
        Temp1 = Temp1 + 10
        Debug [DEC Temp1,13]
        If Temp1 = 50 then skip
    Next
Goto Main

Skip
    Debug [“OK”,13]
    Temp1 = 60
Goto Main
```

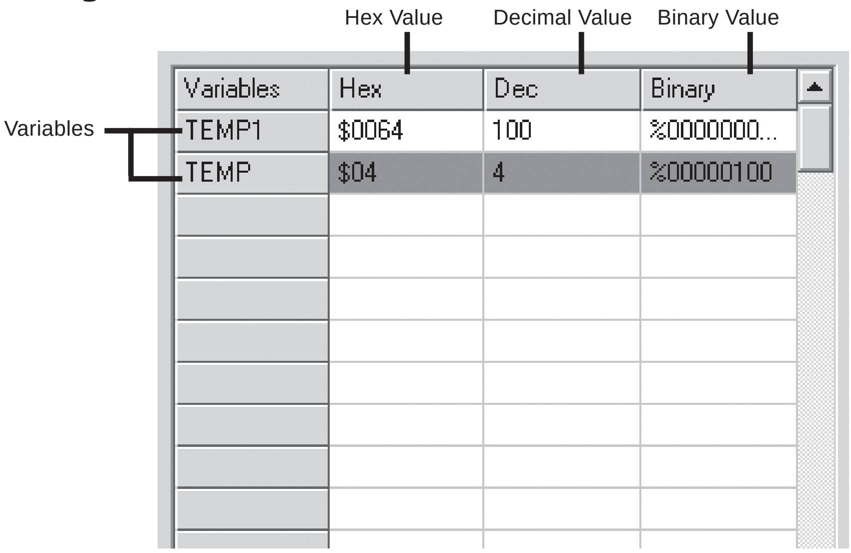

BasicMicro IDE - [74hc595.bas]
File Edit Bookmarks View Tools Debugger Window Help
Compile Program Debug
V... Hex Dec Bi... Real
Line Address
Level 1
Level 28
Build Debug Terminal1 Terminal2 Terminal3 Terminal4

```
; connect leds to pins 15,1,2,3,4,5,6,7       *
;                                             *
;**********************************************


SER     CON P0                  ; serial output
SRCK    CON P1                  ; shift clock
RCK     CON P2                  ; output latch
Delay   CON 100
counter var byte
' ---------------------------------------------------------------------
main
    for counter=0 to 255
        gosub Out595
        pause 100                    ;Change to in/decrease speed
    next
    goto main

' ---------------------------------------------------------------------
Out595:
SHIFTOUT SER,SRCK,MSBPRE,[counter\8]    ; send pattern to 74x595
PULSOUT RCK,8                        ; latch outputs(min 4us)
RETURN
```

Once you have entered the program, save it. Make sure the target device is connected and power is applied. Next, click the “Debug” button on the tool bar.

The program will compile, then a progress bar will appear. This indicates the target device is being programmed. Once this is finished the IDE will change to debug mode. If the program did not compile, check your program for syntax errors.

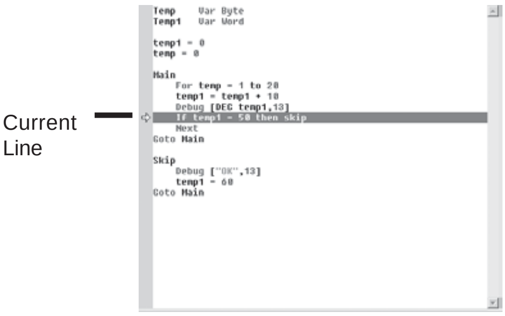
Debug Mode



Variables	Hex	Dec	Binary
TEMP1	$0064	100	%0000000...
TEMP	$04	4	%00000100

After the IDE is in debug mode, a green line will appear at the beginning of your program. This is indicates a connection has been established. Next, click the Auto update button. Then click the Variable button, another window will appear. This is the variable watch window. Click the AutoUpdate button to enable updating of the status windows. Now click Animate.

Variable Watch Window



```
Temp    Var Byte
Temp1   Var Word

temp1 = 0
temp = 0

Main
    For temp = 1 to 20
    temp1 = temp1 + 10
    Debug [DEC temp1,13]
    If temp1 = 50 then skip
    Next
Goto Main

Skip
    Debug ["OK",13]
    temp1 = 60
Goto Main
```

Once the program is running the variable watch window will update the status of each variable used in the program. It will display the values in Hex, Dec, Binary and Real.

Shown is the actual program running, The ICD will show a small yellow arrow and a green bar, indicating where the program is at during execution.

Watch the flow of the program. If the program was entered correctly, after the variable, Temp1, equals 50, the program should jump down to the Skip label. The text “OK” should appear in the watch window. The program should then return to the label Main.

Congratulations you have successfully mastered the ICD!

Important Notes

When running debug, there is a block of code that is added to your program. This code is added to allow the IDE to gather information about the program being executing such as variable values, what is the current program line and so on. This information is provided to the IDE after each line is executed. Because of this, Debug is much slower than running a program normally. Timing sensitive programs will be affected.

When running in debug mode a PC is required to run the target device. Once you have finished using debug reprogram the target device with the “Program” button.

Trouble Shooting

Q. Error: ISP-PRO not connected

A. Check power is applied and the serial cable is connected. Check to ensure the correct comm port has been selected.

Q. A connection has been established, but nothing is happening ?

A. Make sure Auto Update has been checked and after the connection is made, click “Animate”. Check your syntax.

Q. My program runs and then stops part way through the execution for no apparent reason ?

A. Commands such as SERIN, SEROUT, SHIFTIN, SHIFTOUT or any command that affects pins and ports on the target device will cause this problem if they are used to modify the I/O pins the ICD is using to talk to the target device.

Q. After I'm done debugging and I disconnect thePC the PICmicro dies ?

A. Once you are finished debugging, you must reprogram the PICmicro with the normal program button.

Q. During debug the PICmicro is real slow ?

A. When the PICmicro is programmed using debug, extra code is added. During debug the PIC's entire RAM contents is dumped after every instruction.

The Basic’s

BASIC

for PIC microcontrollers

Bits, Bytes, Words, Longs

Throughout this manual and when dealing with programming languages in general, bits, bytes and words will be referred to often. The following is a quick break down:

Type	Bit Size	Range
Bit	1	1 or 0
Nib	4	0 to 15
Byte	8	0 to 255
SByte	8	-127 to +128
Word	16	0 to 65535
SWord	16	-32767 to +32768
Long	32	-2147483647 to +2147483648

MSBs and LSBs

Another often referred to term is LSB or MSB. These terms mean LSB = Least significant bit, MSB = Most significant bit. A byte value of %11110000, the MSB is %1 and the LSB is %0. MSB is the first bit on the left. LSB is the first bit on the right.

RAM, EEPROM and Program Memory

RAM is random access memory, this type of memory is used to store variable values, and system values. RAM is also used to store the return location of GOSUB statements. The more RAM available, the more variables or variable arrays can be used. The 16F876 has about 300 bytes of user RAM available on average.

EEPROM is on chip data storage. This is commonly used to store values that will remain even when the PICmicro is powered down. The EEPROM stores data in byte size. It can store 256 bytes depending on what PICmicro is used. (Refer to the device data sheet)

Program Memory is the actual memory where your program will reside. The size of available program memory will limit the size of your program. The more complicated the program will be, the more memory you will want.

Built-in Hardware

Built-in Hardware refers to additional hardware that is independent of the main microcontroller. Examples of Built-in Hardware are Analog To Digital converters, Pulse Width Modulators, UARTS, Timers and so on. Built-in hardware adds pseudo multi tasking abilities to the PICmicro, because in most cases it can be setup in your program and left to run while the program does other things.

Hexadecimal 101

The hexadecimal number system, also known as "hex" and "base 16", uses 16 characters as its numbers. We normally use the characters 0123456789; this is the "decimal" or "base 10" system.

Computers don’t understand decimal numbers; they "know" only two states; on/off, yes/no, high/low, 1/0. The smallest unit of information they can store is a "bit", short for "binary digit". One bit can store one state; 1 or 0 (On or Off).

Computers count higher than 1 the same way we do; they group the digits to make larger numbers. In base 10, 9+1=10. In base 2 (also known as "binary"), 1+1=10. (That’s "one zero", not "ten".)

Binary numbers get long in a hurry. For example, the decimal number 201 is 11001001 in binary. However, if we group them in groups of four bits (1100 1001) then each group can have 16 possible combinations.

So, how do we represent 16 combinations with only 10 numbers available? We turn to the alphabet to make up the difference. We need 6 more characters, so we use A-F. Using the example above, 1100 binary=12 decimal="C" hex; 1001 binary=9 decimal=9 hex.

Since we are using the same characters in the various bases, we need some way to tell them apart. Is 1101 in binary, decimal, or hex? Binary is usually represented by adding a "b" to the end or “%” to the beginning; i.e. 1101b, %1101. Decimal is either a "d" or is not specified.

Hex is designated in several ways. The following all mean the same thing: $C9 and 0xC9.

Most people use the "$" form. And yes, $FB00 is the same as $0000FB00.

Also, note that hex digits mostly come in pairs. Two digits together are known as a "byte", and are treated as a single unit.

ASCII

MBasic will automatically convert quoted text into its respective ASCII values. This is useful when you are trying to work with the literal value of characters.
Example:

```
    SEROUT 1,I9600,[“A”]
```

The above code example will transmit the ASCII value of the character in quotes.

Programming Practices

As with all programming you may rewrite a program several times before finishing it. In the process you may not remember what you were attempting to do with your program during one of these rewrites. This will lead to spending unnecessary time trying to figure out what you were attempting to do. The best way around this is to use comments as you go along. This may seem tedious but after you have written a program and have not touched it in several weeks or just days, you will begin to appreciate the value of comments. Using comments in your program will help you to pick up where you left off. When using comments have them tell you something useful that can be easily understood. An example of this is:

```
time var byte
time = 0
for time = 0 to 10 step 1
high B0
pause time
low B0
next
end
```

The above program works but what does it do ? Take a look below.

```
time var byte                    ; Variable definition for timer
LED con B0                       ; Constant pointer for Pin 0

time = 0                         ; Initialize timer variable

for time = 0 to 10 step 1        ; Loop to count through timer

    high LED                     ; Set LED on
    pause time                   ; Delay for current timer value
    low LED                      ; Set LED off
next                             ; Loop
end
```

As you can see, commenting a program is the difference between night and day. By looking at the second example it was clear that it was a small program to blink an LED. By using comments, variable names and constants your program will be easier to follow in general. So if later in the program you decide to do more with the LED you could simply use the constant LED instead of B0. This may not be an issue in a short program, but can be very helpful in larger programs. Imagine trying to keep track of what ten different pins do, by just using 0,2,7,12.

Other coding techniques such as labeling sections of code using names that describe the purpose of the code can greatly increase the ease of programming.

```
;Program to flash an LED at different rates
    timer           var byte;
    bigtimer        var word
    LED             con B0              ; Alias PIN B0 to LED

    Start:
        gosub SlowFlasher
        gosub FastFlasher
        goto ENDPROG

    SlowFlasher:
        for timer = 0 to 100            ; Flash 100 times
        Toggle LED                      ; if LED on turn off, If LED off turn on
        pause 1000                      ; Pause 1 sec
        next
    return

    FastFlasher:
        for timer = 0 to 10000          ‘ Flash 10000 times
        Toggle LED                      ‘ if LED on turn off, If LED off turn on
        pause 10                        ‘ pause 10 ms
        next
    return

    ENDPROG:
    end
```

Note: Comments can begin with a semi colon or a single quote.

Optimizing

MBasic software can only optimize a user's program in the sense of emphasizing on code reuse and avoiding redundancy. There are many ways to write a program to accomplish a task. An example of this is:

```
Temp var Byte

Main
    Serout B0, I9600,["Hello World"]
    Serin B1,I9600,[Temp]

    If Temp = "O" Then Fish
    Serout B0, I9600,["Hello World"]

Fish
    Serin B1,I9600,[Temp]
    If Temp = "K" Then Main
Goto Fish
```

The above program sends some serial data out and waits for a specific response. The coding style will generate larger code size and slower programs. A more efficient way to write the above program is shown below:

```
Temp var Byte

Main
    Gosub DoLoop
    Gosub OutLoop

    If Temp = "O" Then Gosub OutLoop
    If Temp = "K" Then Gosub DoLoop
    Goto Main

DoLoop
    Serout B0, I9600,["Hello World"]
Return

OutLoop
    Serin B1,I9600,[Temp]
Return
```

The optimizing is in the Gosub statements, if this program was to get larger you could call the DoLoop and OutLoop labels again and again. Opposed to inserting the same SERIN / SEROUT statements all over the program.

Line Labels

In order to access different sections of code you must use line labels. Unlike the original Basic language, MBasic does not use line numbers. As an example:

```
    Loop: goto Loop                  ;This line repeats infinitely
```

The above goto statement GOTO calls a line label LOOP, which is in front of the GOTO statement. The above line will repeat infinitely. Line labels can not be duplicated or used as variable names once defined as a label.

Variables

Variables are used to store temporary information in the program. They are created using the VAR keyword. Variables can be BITs, NIBBLEs, BYTEs, WORDs and LONGs. Before you can use a variable it must be defined.

Variable name:	Variable:	Size:
Temp	Var	Byte

The above states; Temp will contain a byte (8 bits)

Variable names must start with a letter. They can contain letters, numbers and special characters. However they can not be the same name as MBaic commands, or labels used in a program. The same variable name can not be defined twice with the Var statement. MBasic does not distinguish between upper and lower case, so the name TVAR is equivalent to TVar. The maximum character length can be up to 1024 characters.

Some examples of defined variables:

```
DOG         Var         Bit         ;0 to 1
POST        Var         Nib         ;0 to 15
LOG         Var         Byte        ;0 to 255
STICK       Var         Word        ;0 to 65535
TREE        Var         Long        ;0 to 4,294,967,295
```

Some Tips on assigning sizes to Variables:
1. When assigning sizes to variables, keep in mind what the variable is being used for. Such as, storing the literal value of the letter A, which will require a Byte (0 to 255).
2. A variable should be the smallest size to hold the largest value that will be stored. If a variable will hold the High / Low transition of an input pin (1 or 0), use a bit.

3. If you are storing character sized values with the SERIN command you would use a byte sized variable. If a variable exceeds its maximum size, the excess bits will be truncated. If you were to load the binary value %11110000 into a NIB sized variable the %1111 part would be lost.

Arrays

As your programs begin to perform more complex tasks, there will be times when you want a variable to hold many values. An Array is a structure that can store multiple values of the same type. For example, you can create an array that can hold five different values of the same type as shown below:

```
Temp     var        Word(5)
```

The number 4 in parenthesis shows the variable temp has 5 cells. Once the array has been defined, each cell can be accessed by its number:

```
Temp(0) = 10
Temp(1) = 25
Temp(2) = 45
Temp(3) = 55
Temp(4) = 65
```

The above will assign the value of 10 to the first cell in the 5 cell array, 25 to the second cell and so on. Using arrays can simplify your program as shown below:

```
Temp            Var       Byte(5)             ;variable temp now has 5 cells
Count           Var       Byte

For Count = 0 to 4                  ;increment each cell
Temp(Count) = Count +2              ;the value is cell number + Count + 2
Next
```

The above code example will load each array, 0 to 4 (5 cells) with the array number + 2. Which if done manually would equal the following:

```
Temp(0) = 2
Temp(1) = 3
Temp(2) = 4
Temp(3) = 5
Temp(4) = 6
```

Tables

Tables are made up of constant values which can be byte, word or long size. One example of tables would be if you had a program that sent data to a LCD screen. If your program was setup to send data to the LCD in a menu format. The menu format was made up of many different text messages that would appear conditionally. Instead of copying the text strings over and over where ever they are need in the program you could use a table as shown:

```
Splash ByteTable "Hello"
FirstMenu ByteTable "Enter An Option"

    LCDWRITE B7\B5\B6, OUTA, [str Splash\5]
    LCDWRITE B7\B5\B6, OUTA, [str firstmenu\15]
```

Since there is five characters in our first Byte table we can use the STR modifier with a count of 5 to send out all the data in the table. In our second table there is 15 characters so we use a count of 15 to output all the data. The Table format is shown below:

```
    Label TableType Data, Data, ......Data
```

Label is the name of the table used to call or access the table. TableType is the size of the table data. Tables can be ByteTable, WordTable, LongTable or FloatTable. Data is the constant values or constant expression stored in the table.

Aliases

Aliases are alternate names for defined variables. As an example:

```
    DOG  Var   Byte    ;DOG is assigned as an 8 bit variable (Byte)
    CAT  Var   DOG     ;CAT now points to the variable DOG
```

In the above example if DOG were equal to 10, any time the variable CAT was accessed it would equal 10 since it points to the same RAM location. Aliases are a good idea when you want to use a temporary variable with a name that suits its function.

Variable Modifiers

Variable modifiers are used to access only parts of aliases. An example would be if you wrote the word value of %0111111100000001 but only required access to the high byte of the word. You would then use a modifier with the alias as shown below:

```
Dog   Var    Word
Cat    Var    Dog.HighByte
```

If the binary value of %0111111100000001 was written to Dog, by modifying the alias Cat with the HighByte modifier you would only see %01111111 when Cat was accessed. If the LowByte modifier was used you would then see %00000001.

Another example of modifiers and aliases:

```
MyTimer1 var Byte
MyTimer2 var MyTimer1.LowNib

MyTimer1 = %11110000          ;Mytimer1 now equals 240
Mytimer2 =   %1001            ;MyTimer1 now equals 249
```

In the above examples by using MyTimer1 with MyTimer2 you can easily effect the value of the aliased variable. In the case of MyTimer2 you are able to modify only the low nibble of the aliased byte. The following is a break down of a Word and Byte sized value:

```
                         HighByte       LowByte
            Word Value   %1100110101000101
```

HighByte is the first 8 bits of a word from left to right and Lowbyte being the opposite.

```
                         HighNib   LowNib
          Byte Value       %11001101
```

HighNib is the first 4 bits of a byte from left to right and LowNib being the last 4 bits.

The table below shows all the different modifiers that can be used:

Modifier	Create alias to
LOWBIT	bit 0 of variable
BIT0	bit 0 of variable
BIT1	bit 1 of variable
BIT2	bit 2 of variable
BIT3	bit 3 of variable
BIT4	bit 4 of variable
BIT5	bit 5 of variable
BIT6	bit 6 of variable
BIT7	bit 7 of variable
BIT8	bit 8 of variable
BIT9	bit 9 of variable
BIT10	bit 10 of variable
BIT11	bit 11 of variable
BIT12	bit 12 of variable
BIT13	bit 13 of variable
BIT14	bit 14 of variable
BIT15	bit 15 of variable
BIT16	bit 16 of variable
BIT17	bit 17 of variable
BIT18	bit 18 of variable
BIT19	bit 19 of variable
BIT20	bit 20 of variable
BIT21	bit 21 of variable
BIT22	bit 22 of variable
BIT23	bit 23 of variable
BIT24	bit 24 of variable
BIT25	bit 25 of variable
BIT26	bit 26 of variable
BIT27	bit 27 of variable
BIT28	bit 28 of variable
BIT29	bit 29 of variable
BIT30	bit 30 of variable
BIT31	bit 31 of variable
HIGHBIT	Always the highest bit
LOWNIB	nibble low of variable

NIB0	nibble 0 of variable
NIB1	nibble 1 of variable
NIB2	nibble 2 of variable
NIB3	nibble 3 of variable
NIB4	nibble 4 of variable
NIB5	nibble 5 of variable
NIB6	nibble 6 of variable
NIB7	nibble 7 of variable
HIGHNIB	nibble high of variable
LOWBYTE	byte low of variable
BYTE0	byte 0 of variable
BYTE1	byte 1 of variable
BYTE2	byte 2 of variable
BYTE3	byte 3 of variable
HIGHBYTE	byte high of variable
LOWWORD	word low of variable
WORD0	First 16 bytes
WORD1	Last 16 bytes
HIGHWORD	word high of variable

Command Modifiers

Command modifiers can be used to modify data in a command directly. Modifiers can be used with any commands that show {Modifier} in their syntax. An example would be if you used the SERIN command and stored a value into a variable called temp. The values stored in temp would reflect the ASCII value of the characters received. To modify the value stored in the variable temp an example would be:

```
SERIN 0, i9600, [DEC TEMP]
```

If the characters 123 were received then the value held in the variable temp would be 123 (One Hundred and Twenty Three). If HEX was used as the modifier in place of DEC the value would be $123 or BIN the value would be %1111011. There are Input and Output modifiers as shown below:

I/O Modifiers

dec	Decimal Value
hex	Hexadecimal Value
bin	Binary Value
str	Input or Output Array Variables

Signed I/O Modifiers

sdec	Decimal Value
shex	Hexadecimal Value
sbin	Binary Value

Indicated I/O Modifiers

ihex	Hexadecimal Value
ibin	Binary Value

Combination I/O Modifiers

ishex	Hexadecimal Value
isbin	Binary Value

Output Only Modifiers

rep	Output character *n* times
real	Output Floating point numbers

Input Only Modifiers

waitstr	Compares count characters to variable array waits until equal or until optional eol character is found in variable array
wait	Wait *n* string of characters
skip	Skip *n* input

HEX - DEC - BIN
Converts a value to decimal, binary or hexadecimal.

SERIN 0,i9600, [DEC TEMP]

SDEC - SHEX - SBIN
Converts a value to decimal, binary or hexadecimal then signs the value with “-” for negative or nothing for positive.

SERIN 0,i9600, [SDEC TEMP]

IHEX - IBIN
Converts a value to binary or hexadecimal then assigns an indicator “%” or “$”.

SEROUT 1,i9600, [IBIN TEMP] ;sends out “%11100110”

ISHEX - ISBIN
Converts a value to binary or hexadecimal. Assigns an indicator “%” or “$”. Then adds a sign “-” for negative numbers.

SEROUT 1,i9600, [ISBIN TEMP] ;sends out “%-11100110”

REP
Repeats character *n* times.

SEROUT 1,i9600, [REP TEMP\10] ;sends the value in temp 10 times

STR
str var\count{\eol char} - Store values in array(var) until count characters received or EOL (End of line) character received. The EOL will override the count value if both EOL and count values are specified.

SERIN 1,i9600, [STR TEMP\10\”E”] ;stores values in an array until “E” is received or 10 charcaters have been received.

WAITSTR
waitstr var\count{\eol char} - Compares *n* characters to variable array, waiting until equal or until EOL (End of line) is found. The EOL will override the count value if both EOL and count values are specified. If no EOL is specified, there is no default end of line character and the count will complete.

SERIN 1,i9600, [WAITSTR TEMP\10\”E”] ;compares values in an array until “E” is received or 10 characters have been received.

WAIT
wait (constant list) - Constant list can be "text" or 1,2,3,4,5(comma delimited). The wait modifier will wait for the specific characters.

SERIN 1,i9600, [WAIT 1,”a”,3,”c”] ;waits until 1, “a”, 3, “c” has been received.

SKIP
skip count - Skip n characters received.

SERIN 1,i9600, [SKIP\10 TEMP] ;skips 10 characters before loading anything into the variable temp.

Constants

Constants are fixed values and once declared their values do not change in a program. When creating a program it can be beneficial to use constants for certain values that don't change. Such as the following:

```
Meter       CON   1        ;Meter is a constant = 1

Centimeter  CON   100      ;Centimeter is a constant = 100

Millimeter  CON   1000     ;Millimeter is a constant = 1000
```

This way you can keep your program readable. Another good use of the constant is when you have values that are based on other values:

```
Meter       CON   1                ;Meter is a constant = 1

Centimeter  CON   Meter * 100      ;Centimeter is a constant = 100

Millimeter  CON   Centimeter * 10  ;Millimeter is a constant = 1000
```

In the above example "centimeter" and "millimeter" values were derived from the constant "meter". There are a 100 centimeters in a meter and a 1000 millimeters in a meter.

Pin names are also considered to be constants so they can be used in the following way:

```
RedLed      Con A0
GreenLed    Con 0

Main
      High RedLed
      High GreenLed
Goto Main
```

RedLed and GreenLed are now constants that point to the pin A0. When writing complex programs it may be beneficial to use constant labels as shown above. One thing to consider is constants are always true so they should not be used in comparison expressions like below:

```
Tpin Con Bo

Input Tpin
If Tpin = 0 Then SomeLabel
```

The previous If...Then statement will always be true since constants can not be modified. The correct way to perform a true / false statement on a constant pin would be as shown below:

```
Tpin Var PortB.bit0

    Input Tpin
    If Tpin = 0 Then SomeLabel
```

Tpin can no longer be a constant since we are checking the state of a pin. Since constant do not change, Tpin will need to be a variable pointing to the pin bit. Using a variable will allow the value stored in Tpin to change from 1 to 0.

Pins

Pins are a description of an I/O pin on a microcontroller. Ports are defined as a group of pins. Usually 8 pins make up a port (Byte). Ports that are short of a full eight pins are still accessed the same way as if there were all eight pins when reading or witting to the whole port. The non existent pins bits will simply be ignored.

```
    INPUT B1          ; Sets Pin 1 on port B as an Input
```

Pin names (Depending on MCU type):

A0 to A5
B0 to B 7
C0 to C7
D0 to D7
E0 to E 7

Pin names for Stamp compatibility:

A0 - A5	=	0-5
B0 - B 7	=	8-15
C0 - C7	=	16-23
D0 - D7	=	24-31
E0 - E 7	=	32-39

The above stamp compatibility pins only apply to a PICmicro controller.

Pins can be called by either name A0 or 0, B0 or 8 and so on. This was added to simplify compatibility with converting a stamp program and add flexibility to your programs.

As discussed earlier pin names are constant. Since a constant can not change, using direct pin names will not always work as might be expected. As an example:

```
If A1 = 1 Then Main
```

The above statement will always be true since A1 is a constant. Shown below is the correct way to access the pin condition:

```
If PortA.Bit1 = 1 Then Main
```

The above statement PortA.Bit1 is a pointer to pin A1. Since we are now looking directly at A1, we can now check its condition for a true / false statement.

Ports can be access all at once. To do this you would treat the port name as a variable.

```
PORTA = $ff                   ;set all pins in PortA High
```

The above example is a little misleading. You have set all the PINS high; however without setting the pins to be outputs you haven't done anything yet. There are three ways to set port status. You can use the input/output commands which will set individual pins or you can treat the PORTs I/O address as a variable using either the ports TRIS variable or the BS2 style DIRS command.

```
TRISA = $00                   ;set all pins in PortA to outputs
DIRL = $00                    ;sets all pins on PortA to outputs
```

There is one TRIS variable for each PORT. Note; you have to set the individual bits OFF to make them OUTPUTS.

```
TRISB = $F0                   ;Set PINS 0-3 to be outputs.  Set PINS
                              4-7 to be INPUTS (hex value)
```

Or with binary values:

```
TRISB = %11110000;Set PINS 0-3 to be outputs.  Set PINS
                  4-7 to be INPUTS (binary value)
```

The BS2 direction commands work the same as the TRIS variables except they are predefined and only allow access to the first two ports (A & B).

Important Note

PORT and TRIS variables. Because they are true variables you can use them just like any other variable. Using a PORT that has some pins set to input and some pins set to output can lead to problems when debugging.

Example

The example program below illustrates using an entire port in a loop to illuminate a row of LEDs:

```
LEDpin VAR byte
counter VAR byte

for counter = 8 to 15
    ; Counter specifies the pin to be activated (B0 to B7)

    LEDpin = counter        ;starts at PORTB PIN 8
    HIGH LEDpin             ;LED on
    pause 10                ;wait 10 milliseconds
    LOW LEDpin              ;LED off
    pause 10                ;wait 10 milliseconds
next                        ;repeat loop
end
```

In the example you will notice that NO pin names have been used. This is because each pin also has a number associated with it for Basic Stamp compatibility.

Pin Variables

Sometimes it may be easier to deal with individual bits, bytes or nibbles of a port. MBasic has special names to make it easier to deal with small pieces of each register. Below is a table of port names that are Basic Stamp compatible. To determine which name is associated with what pin refer to the *Pins* section of this manual.

Name:	Byte Name:	Nibble Names:	Bit Names:	
INS	INL, INH	INA, INB	IN0 – IN7	
		INC, IND	IN8 – IN15	;Input pins
OUTS	OUTL, OUTH	OUTA, OUTB	OUT0 – OUT7	
		OUTC, OUTD	OUT8 – OUT15	;Output pins
DIRS	DIRL, DIRH	DIRA, DIRB	DIR0 – DIR7	
		DIRC, DIRD	DIR8 – DIR15	;I/O direction

The previous chart is for Basic Stamp compatibility only. Below is the correct port names for accessing pieces of a port in Mbasic.

MBasic does not differentiate between input or output when accessing port pins. You would simply use the port name in your command. The command would either access the port as an output or input depending on the command. An example would be as shown:

```
PortB = %1111111
```

The above statment sets all the pins on port B high. The next statement sets only B0 to B3 high:

```
PortB.nib0 = %1111
```

The following table shows the MBasic port names.

Name:	Byte Name:	Nibble Names:	Bit Names:
PORT	PORTA, PORTB PORTC, PORTD PORTE	NIB0, NIB1	BIT0 – BIT7

MBasic Specific’s

MBASIC
for PIC microcontrollers

Stack

The below option is for advance users only and therefore doesn’t appear in the header file. The STACK setting is set as word size and defaults to a size of 20 words. This mean that the compiler uses word sizes instead of byte sizes for the stack. Rarely will the stack need to be adjusted. Setting the stack to something else other than the default is not recommend unless by an experienced user. The maximum stack size available for any supported chip can be determined by the amount of RAM available in the first bank of memory for the chip being used.

```
    STACK = 20
```

Oscillator Settings

All of the time sensitive instructions, use constant values, that do not change regardless of the processor speed. When compiling a program, MBasic will automatically adjust the timings of speed sensitive commands. This feature will allow a program to be re-compiled for different speeds without modification. If a program is run on a faster chip than it was compiled for, all timings will be off unless the program has been re-compiled for that speed.

Internal RC Calibration

To calibrate the IntRC of a chip that has internal RC, (12C671) use the option register. Refer to the device data sheet for settings. Below is an example of the IntRC calibration register:

```
    OSCCAL = %10000000
```

Memory

There are no issue with MBasic and using a PICmicro MCU with more than 2K. MBasic was actually designed to become more efficient beyond 1K. The latest trend with new micro controllers has been larger programming space averaging 8K and above. MBasic was designed to take advantage of this. Since some of the more common PICmicro MCUs available are 16F87x which range from 4K to 8K. All bank and page switching is done automatically. This allows hassle free programming.

Compiling

When MBasic compiles a program it first takes the program and breaks it down into pieces. It then takes these pieces and inserts the necessary assembly code. Once this task is complete the compiler will generate an .ASM file. This .ASM file is then passed to MPASM which then takes the .ASM file and converts it to the final .HEX file. This resulting .HEX file is then used to program the PICmicro MCU.

Reserved Symbols

When creating a BASIC program using MBasic there are some special considerations to take into account. The first, MBasic uses an underscore '_' to define some commands internally. So the use of an underscore first to define a variable or label is not allowed in a BASIC program (i.e. : _cat or _label). Command names such as SERIN cannot be defined as a variable or a label. Please refer to the IDE help file for a list of reserved words.

Device Specific Issues

Read the data sheets on each microcontroller you are using. Some devices have features that can interfere with normal pin operations if not setup correctly. The best example is the 16F87x. The 87x PORT A defaults to analog which will cause erratic behavior if the port is used digitally, like blinking LEDs. In most cases MBasic will automatically set pins to digital.

MBasic and Assembly

MBASIC
for PIC microcontrollers

What is Assembly ?

Assembly language is the main programming language used to program microcontrollers. MBasic actually translates your Basic program into an assembly file which is then passed to an assembler to produce the resulting hex file, used for programming the microcontroller. An example assembly program is shown below:

```
	movlw		10
	movwf		Temp
	movlw		50
asmloop
	addwf		LongTemp
	skpnc
	incf		LongTemp+1,F
	decfsz		Temp
	goto		asmloop
```

Mixing Assembly and MBasic

Assembly language routines can be added at any time and anywhere in a MBasic program. They are passed as is, from the program and inserted directly into the compiled assembly file. There are several advantages to combining assembly language with MBasic. For the beginner using small snippets can aid in learning and understanding microcontrollers better. For the professional MBasic can be used to create PAUSE, IF...THEN, SERIN, SEROUT routines in BASIC, quicker and easier than in assembly.

In-line Assembly

MBasic allow single assembly command insertion or blocks of assembly in a Basic program. An example is shown below:

```
	temp var byte

		clrf	temp
	main
		High B0
		Pause 500
		if temp < 20 then skip
		Low B0
skip
	 incf	temp,f
	 goto main

end
```

In line assembly routines can easily be called by a BASIC program by using the following method:

```
Cat
      ASM
      {           movlw   10
                  movwf   temp
      }
Return
```

To call the assembly routine simply use a GOSUB command. To return to where the GOSUB command made the call use a RETURN command after the enclosing bracket for the ASM command. The label must be in place before the ASM command.

Assembly and Variables

Reading and writing variables that are defined in BASIC using assembly can been done with MBasic.A variable can be accessed from assembly that is defined as a byte in Basic. Accessing Nib, bit, or word size variables gets a little more complicated. It is outside the scope of this manual to describe this process.

Math

Numerical Types

Numbers can be defined in two different ways. Binary numbers are defined using only 0 and 1. Hexadecimal uses characters ‘0’ to ‘F’. Binary and hexadecimal numbers must have an indicator.

1234 or d’1234’	: Standard Decimal number
$1F2A or 0x1F2A	: Hexadecimal notation
%1001 or b’1001’	: Binary notation

The character $(string) indicates Hexadecimal and the percentage sign % indicates binary data. These special characters must be used in order to let the Atom know what numerical types they are.

Adding and Subtracting

```
temp VAR byte
temp = 100          ; byte variables can NOT exceed 255 in value
temp = temp - 99    ; Valid since Value of temp is still positive
temp = 0
temp = temp -127    ; Here is where it gets interesting
```

As you can see in the above example the last value of temp should be negative. When dealing with negative numbers in unsigned variable types you will actually be storing the maximum value that variable can hold - the absolute value of the number (i.e.: instead of -127 you are storing 128). In order for MBasic to use negative numbers correctly (without help from the user), you should use signed variable types (i.e.: SByte,SWord or Long).

BITs	can’t be negative
NIBs	can’t be negative
SBytes	-127 to +128
SWords	-32767 to +32768
LONGs	-2147483647 to +2147483648

Multiplication

Mbasic can perform 32x32 bit multiplication (i.e. 4,294,967,295 x 4,294,967,295 max.). If you use the maximum values only the low 32 bits will be returned. That is why a second multiplication command is used. The ‘**’ command allows you to get the high 32bits of data from the multiplication. So as the example below shows, you must use two commands to get the full value from a large multiplication:

```
        lowmult         VAR long
        highmult        VAR long
        middilemult VAR long

        lowmult = 4,294,967,295 * 4,294,967,295
                ; returns the LOW 32 bits of * value = 5119617025

        highmult = 4,294,967,295 * 4,294,967,295
                ; returns the HIGH 32 bits of * value = 1844674406

        middlemult = 4,294,967,295 */ 4,294,967,295
                ; returns the MIDDLE word of * value = 7440651196
```

Using the */ function is equivalent to multiplying by the HIGH 16bit number + (LOW 16bit number / 65536). This allows simple fractional multiplication without using the floating point math system (which is significantly slower than integer math)

```
example1:
temp var long     ;temp needs to equal 1000 * .05
```

;to calculate the fraction .05 for use with the following MULM('*/') function use this formula. fraction = 0.05 * 65536 = 3276.8 ~= 3276

```
temp = 1000 */ 3276        ;temp will equal 49

example2:
temp var long     ;temp needs to equal 1000 * 3.145

temp = 1000 */ 206110    ;temp = 3000 + 144 = 3144
```

You should note that some round off error will occur.

Division

Mbasic can perform 32x32 bit division (i.e. 4,294,967,295 / 4,294,967,295 max.). The division command has two separate commands associated with it. The first command '/ ' will return the quotient value. Which is the actual result from the divide. The second command "//", is used to return the remainder if there is one :

```
    divval VAR word
    divrem VAR word
    divval = 65000 / 101        ; Returns the Quotient value = 643
    divrem = 65000 // 101       ; Returns the remainder value = 57
```

Integer Math in general

Mbasic uses standard algebraic syntax. In Basic Stamp math: 2+2 * 5 / 10 would equal 2. In Mbasic 2+2*5/10 = 3. This is because each math operator has a precedence. The multiply and divide operators have equal precedence. In the above calculation 2*5 will be calculated first (equaling 10), then the divide by 10 (equals 1), then the addition of 2 (equaling 3). To ensure the above equation works as you expect you need to use parenthesis (i.e.: ((2+2)*2) /2 would equal the Basic Stamp syntax.

Operator Precedence:

Order:	Operation:
1st	Not, ABS, SIN, COS, - (NEG), DCD, NCD, SQR, Random, FNEG, INT, FLOAT, DEC2BCD, BCD2DEC
2nd	<, <=, =, >=, >, <>
3rd	And, Or, Xor
4th	Rev, Dig
5th	<<, >>
6th	MAX, MIN
7th	&, |, ^, !&, !|, !^
8th	*, **, */, /, //, FMUL, FDIV
9th	+, -, FADD, FSUB

General Math Functions

The following is a list of special math functions MBasic can perform.

UNARY Commands

variable = COMMAND value

ABS	Absolute value
SIN	sine of value(0-255)
COS	cosine of value(0-255)
DCD	2 to the nth power(n = value)
NCD	smallest power of 2 that is GREATER than value.
SQR	square root of value.
DEC2BCD	Integer to packed BCD format
BCD2DEC	Packed BCD to integer.

BINARY Commands

variable = value1 COMMAND value2

MAX	smaller of the two values.
MIN	larger of the two values.
DIG	digit of value 1 at value2 position.
REV	reverses value2 bits of value1 starting with LSB

Bitwise Operators

Bitwise operators are commands that directly effect the bits of a value.

Bitwise operators

variable = value1 COMMAND value2

{!}& (AND)	AND bits of value1 with {Inverted}value2 variable = %01010101 & %00001111 ;variable equals %0101
{!}| (OR)	;value1 OR value2 variable = %11110000 | %00001111 ;variable = %11111111
{!}^ (XOR)	;value1 XOR value2 variable = %10101010 ^ %11110000 ;variable equals %01011010
>> (SHIFTRIGHT)	;value1 bits shifted right value2 times variable = %11110000 >> 4 ;variable equals %1111
<< (SHIFTLEFT)	;value1 shifted left value2 times variable = %00001111 << 4 ; variable equals %11110000

Comparison Operators

Comparison operators are used when comparing two or more values. Examples are the IF...THEN and LOOKDOWN commands.

Compare Op.	Description
=	Equal
<>	Not Equal
<	LessThan
>	GreaterThan
<=	LessThan Equal
>=	GreaterThan or Equal

Logical Operators

Logical operators are slightly different in use than comparison operators. When using an IF...THEN statement that contains more than one comparison you must combine them using a logical operator. The example below illustrates this:

```
If (Variable < 100) AND (Variable > 10) Then Label
```

As you can see from the example if BOTH are true then the program jumps to the label.

Logical Op.	Description
AND	Logical AND
OR	Logical OR
XOR	Logical Exclusive OR
NOT..AND	Logical NAND
NOT...OR	Logical NOR
NOT..XOR	Logical NXOR

Floating Point Math

Mbasic can perform 32x32 bit IEEE 754 Compliant Floating Point calculations. 32 bit floating point math is CPU intensive since the PICmicro is only an 8 bit processor. Some special commands are required to handle floating point math. No command can actually use floating point math in its variable or expression. You will need to do some conversion with the floating point value to send it (except when using the REAL modifier) using SEROUT or receive it using SERIN.

INT	= Convert floating point number to integer number
FLOAT	= Convert integer number to floating point number
FNEG	= negates floating point numbers
FADD	= Adds two floating point numbers
FSUB	= Subtracts two floating point numbers
FMUL	= Multiplies two floating point numbers
FDIV	= Divides two floating point numbers
FLOATTABLE	= stores constant floating point data in flash

Floating Point Format

The floating point math Mbasic uses differs some what from the normal IEEE 754 floating point standard. The differences are minor. The next section gives a good indicator on how they differ.

Floating Point Example Program

```
temp var long
temp2 var long

Fconst Con 1.12345
        ;Stores the Microchip format floating point number 1.12345
Iconst Con INT 1.12345
        ; Truncates the value 1.12345 to 1

Main
temp = 0                ;Stores the integer value 0 in temp
temp2 = float 0         ;Stores the floating point value 0.0 in temp2
```

The first part of the program stores the integer number 0 in Temp and the floating-point number 0.0 in Temp2.

```
temp = 1.12345
```

We then store the floating point number 1.12345 in Temp.

```
temp = fconst
```

```
        temp2 = iconst
```

Fconst is a floating point constant that was defined at the begging of the program. So Temp is equal to 1.12345. We also defined iconst which is an integer constant.

```
        temp = float 1
        of
        temp = 1.0
```

We now load the floating point value of 1 into Temp. As shown above, if a constant is to be used in a floating point calculation it must be represented as a floating point number. The second case above the number is automatically understood to be a floating point value by the compiler because of the .0 extension. In the first example the float command must be used because there is no way for the compiler to tell whether the constant should be an integer or a floating point number

```
        temp = temp fmul float 2
        or
        temp = temp fmul 2.0
```

Next we multiply the value in Temp by a floating point number 2. The result of Temp is now 2.0 which is a floating point number.

```
        temp = float 567
```

Above we load Temp with FP (Floating Point) number 567.0. We then divide Temp with 5.0 which is a FP number.

```
        temp = temp fdiv float 5
        or
        temp = temp fdiv 5.0
```

Temp now equals 113.4.

```
        temp = float 123
        or
        TEMP = FLOAT 123.0
```

Above we load Temp with FP (Floating Point) number 123.0. We then add Temp with 123.5 which is a FP number.

```
        temp = temp fadd float 123.5
```

```
or
temp = temp fadd 123.5
```

Temp now equals 246.5.

```
temp = float 123
temp = temp fsub float 123.5
or
temp = 123.0
temp = temp fsub 123.5
```

Temp now is equal to 0.5. Since Temp is now equal to 0.5 we negate Temp from its self below and we end up with -0.5. Floating point math also will handle negative numbers.

```
temp = fneg temp
```

Important Notes

All variables that will contain a floating point number must be 32 bits (Longs).

Syntax

BASIC

for PIC microcontrollers

ADin

ADIN pin,clk,adsetup{,var}
Sets up the Hardware A/D converter, and stores the value in an optional var.

> **Pin** can be a constant or variable. The pin specifies which analog pin to use with the ADIN command some microcontrollers only have 2, while others have 8 or more (refer to the device data sheet for details)
>
> **Clk** can be a constant or variable. The Clk option sets the sampling time for the A/D conversion. There are only 4 choices for Clk 0, 1, 2, 3 (0-2 are based on internal cycles) CLK option 3 is based on an internal RC divisor (refer to the device data sheet)
>
> **Adsetup** can be a constant or variable. Adsetup is used to setup the ADCON1 register (refer to the device data sheet for details on the ADCON1 register)
>
> **Var** can only be a variable. Var specifies where the conversion result will be stored.

Explanation

Analog to Digital conversion allows your program to receive an analog signal and convert it into digital values. The AD hardware can read 0 to 5 volts. The ADIN command can be used to start the A/D conversion and wait for the result or it can be used to multi task by starting the hardware A/D conversion and processing the result later in the program.

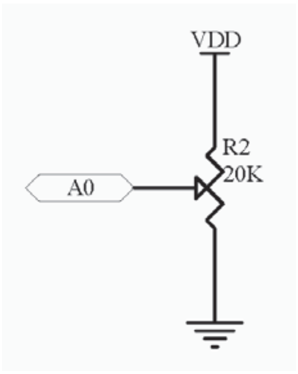
The program below converts an analog input on RA0. The following example will output the decimal value of the conversion to the debug window.

```
temp var word      ;temp is now a word sized variable

AN0  con 0    ;this sets the pin AN0 / RA0,

CLK  con 2     ;CLK options are 0, 1, 2, 3 (0-2 are based on int cycles)
                ;CLK option 2 is based on 1/16 of the external osc speed
                ;the CLK slow does down the sampling time.

ADSETUP con %10001110   ;sets up the ADCON1 register
                          ;AN0 / RA0 is now analog

main
 ADIN AN0,CLK,ADSETUP,temp
                      ;loads the variable temp with the sample
```

```
debug[dec temp,13]          ;out puts the value in temp
goto main
```

Schematic

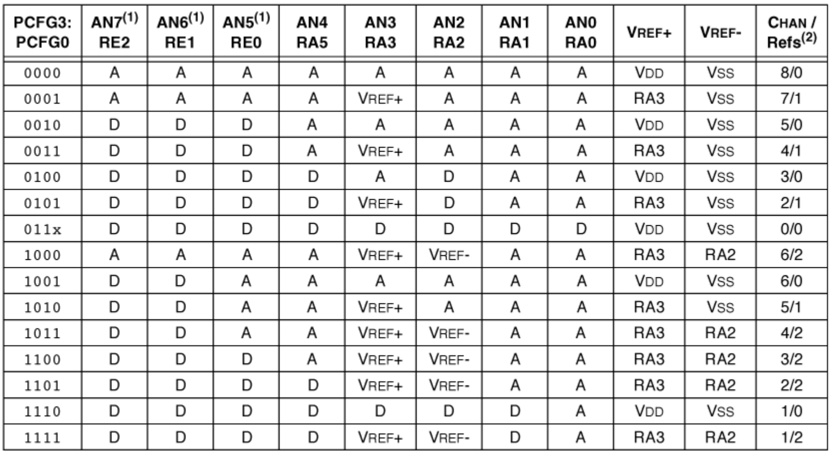
The below schematic shows the configuration for the above code to work properly. The 20K pot can be moved from RA0 / AN0 to another analog pin but the ADSETUP value will need to be changed along with the specified pin.



Adsetup

ADCON1 sets the A/D converter pin options. Which allows you to set some or all of the A/D pins to analog, with several other options as shown in Table2-3. The last four bits which are 1110 in the example program shownset pin RA0 / AN0 to analog and the remainder pins to digital. The first bit which is 1 in the example program is the left and right justify. 1 = Right Justified, 0 = Left Justified.

ADCON1 REGISTER SETTINGS: (Table 2-3)



PCFG3: PCFG0	AN7[1] RE2	AN6[1] RE1	AN5[1] RE0	AN4 RA5	AN3 RA3	AN2 RA2	AN1 RA1	AN0 RA0	VREF+	VREF-	CHAN / Refs[2]
0000	A	A	A	A	A	A	A	A	VDD	VSS	8/0
0001	A	A	A	A	VREF+	A	A	A	RA3	VSS	7/1
0010	D	D	D	A	A	A	A	A	VDD	VSS	5/0
0011	D	D	D	A	VREF+	A	A	A	RA3	VSS	4/1
0100	D	D	D	D	A	D	A	A	VDD	VSS	3/0
0101	D	D	D	D	VREF+	D	A	A	RA3	VSS	2/1
011x	D	D	D	D	D	D	D	D	VDD	VSS	0/0
1000	A	A	A	A	VREF+	VREF-	A	A	RA3	RA2	6/2
1001	D	D	A	A	A	A	A	A	VDD	VSS	6/0
1010	D	D	A	A	VREF+	A	A	A	RA3	VSS	5/1
1011	D	D	A	A	VREF+	VREF-	A	A	RA3	RA2	4/2
1100	D	D	D	A	VREF+	VREF-	A	A	RA3	RA2	3/2
1101	D	D	D	D	VREF+	VREF-	A	A	RA3	RA2	2/2
1110	D	D	D	D	D	D	D	A	VDD	VSS	1/0
1111	D	D	D	D	VREF+	VREF-	D	A	RA3	RA2	1/2

A = Analog input, D = Digital I/O

Note   Not all channels are available on all chips with A/D. Refer to the device data sheet for details.

ASM {...}

ASM...{Assembly commands}
In-line Assembly

Explanation

ASM {..} instructions allow the use of assembly in MBasic. The command, when used, will tell the compiler that the code in between the two brackets is in assembly and it is to be passed directly to the assembler. These commands can be used as freely as needed.

Example

```
    Temp VAR byte
LongTemp VAR word

main

ASM
{
    clrf        LONGTEMP & 0x7F
    movlw       10                      ;load W with 10
    movwf       TEMP & 0x7F             ;store 10 in Temp
    movlw       50
asmloop
    addwf       LONGTEMP & 0x7F,f       ; Add 25 to LongTemp
    skpnc                               ; If Low byte of longtemp carries
    incf  (LONGTEMP+1) & 0x7F,F         ; Increment Long Temp+1
    decfsz      TEMP & 0x7F,f           ; Loop 10 times
    goto        asmloop
}
    debug [DEC longtemp,13]             ;longtemp will increment each
loop
    goto main
```

Important Notes

All variables used in the in-line assembly must be either words or bytes. Also note that words are denoted by the variable name and the variable name+1 (ie:Longtemp and Longtemp+1)

It is not in the scope of this manual to teach Assembly language. The ASM command is strictly for advanced users that completely understand assembly language. No support is offered for assembly language programming.

Branch

BRANCH index, [Label1,...LabelN]
Go to the Label specified by index.

> **Index** is a variable or constant that specifies the address to branch to.
>
> **Label1,...LabelN** are labels that specify where to branch to.

Explanation

The Branch command allows the program to jump to different locations based on a variable index. The BRANCH instruction can be useful to simplify something like the following:

```
IF temp = 0 THEN dog            ;temp =0; go to label dog
IF temp = 1 THEN cat            ;temp =1: go to label cat
IF temp = 2 THEN mouse          ;temp =2: go to label mouse
```

BRANCH can be used to organize the above into a single statement:

```
BRANCH temp, [dog, cat, mouse]
```

The above works the same as the previous IF...THEN example. If the value is greater than 3 the BRANCH does nothing and the program continues to the next line.

Example

In the following example we have 5 players 0-4. We will display each player in turn. As each player is displayed our variable Player is incremented. The value in Player is index to the constant list used in the Branch command.

```
Player var byte          'Used to hold the current player number
Player = 0               'Start out as first player

'The main loop
loop:
   debug ["#",dec player," "]
   Branch Player,[Mike,Joe,Fred,Bill,Jane]

       'Fall through because there is no player #5
      debug ["Don't know who this is.  Lets start over.",10,13,10,13]
      player = 0 'Reset to first player
      goto loop  'Now go do it again

'These are the branch routines
Mike:
        debug ["Mike's Turn ",13]
        Player = Player  + 1 'Get next player
goto loop

Joe:
        debug ["Joe's Turn ",13]
        Player = Player  + 1 'Get next player
goto loop

Fred:
        debug ["Fred's Turn ",13]
        Player = Player  + 1 'Get next player
goto loop

Bill:
        debug ["Bill's Turn ",13]
        Player = Player  + 1 'Get next player
goto loop

Jane:
        debug ["Jane's Turn ",13]
        Player = Player  + 1 'Get next player
goto loop
```

This program shows how the BRANCH command can be used. First we assign our variable a value of 0. Then we define our labels. Since the first position is considered 0 and the variable index equals 0 the branch command will cause the program to jump to the first label in the brackets [Mike]

Button

BUTTON pin, downstate, delay, rate, bytevariable, targetstate, address
Debounce, button input, perform auto-repeat, and branch to address if button is in target state. Button circuits may be active-low or active-high.

Pin is a variable/constant that specifies the I/O pin to use. This pin will be made an input.

Downstate is a variable/constant (0 or 1) that specifies which logical state occurs when the button is pressed.

Delay is a variable/constant (0–255) that specifies how long the button must be pressed before auto-repeat starts. The delay is measured in cycles of the Button routine. Delay has two special settings: 0 and 255. If Delay is 0, Button performs no debounce or auto-repeat. If Delay is 255, Button performs debounce, but no auto-repeat.

Rate is a variable/constant (0–255) that specifies the number of cycles between auto repeats. The rate is expressed in cycles of the Button routine.

Bytevariable is the work space for Button.

Targetstate is a variable/constant (0 or 1) that specifies which state the button should be in for a branch to occur.
(0=not pressed, 1=pressed)

Address is a label that specifies where to branch if the button is in the target state.

Explanation

When a button is pressed or a switch flipped, the contacts make or break a connection. During this a 1 to 20-ms burst of noise occurs as the contacts scrape and bounce against each other. The BUTTON command has a feature that prevents this noise from being interpreted as more than one switch action. This is called debounce.

The BUTTON command also will react to a button press the way a computer keyboard does. When a key is pressed, a character will appear on the screen. If the key is held down, there is a small delay, then a rapid stream of characters appear on the screen. The BUTTON command has an auto-repeat function that can be set up to act the same way.

BUTTON was designed to be used inside a program loop. Each time cycle through the loop, the BUTTON command checks the state of the specified pin. When it first detects the DownState it debounces the switch. Then with

TargetState, it either branches to address based on what the TargetState is set to (TargetState = 1 then jump or TargetState = 0 then don't).

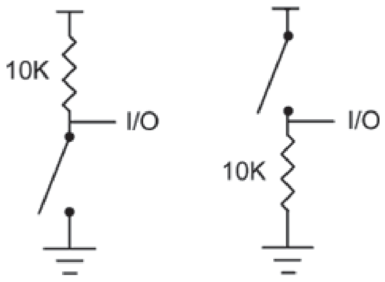
If the connected switch stays in a down state, BUTTON will count the number of program loops that execute. When this count equals Delay, BUTTON will once again trigger the action specified by TargetState and address. After which, if the switch remains in DownState, BUTTON waits the Rate number of cycles between actions. There is a work space variable which is used by BUTTON to keep track of how many cycles have occurred since the pin switched to TargetState or since the last auto-repeat. BUTTON will not stop program execution. In order for the BUTTON commands delay and auto repeat functions to work properly it must be executed from within a program loop.

Example

The following program is an auto repeat example of the button command. This demonstration requires a button wired to pin 1. Using state 1 as shown in the schematic. Place a LED between pin 2 and VSS. When you click the button you get a single led blink. Hold the button down and it auto repeats.

```
'The button command needs a work variable.
work var byte
work =0



Input B1        'Button
output B2       'LED
low B2          'Make sure our display pin is low to begin with

Loop:
    Button B1,1,20,2,work,1,press
    pause 20            'We need some delay or its too fast.
goto loop

'This is where you do your button work
press:
    high B2
    pause 10
    low B2
goto loop
```

Schematic



State 0 **State 1**

Clear

CLEAR
Clear user RAM.

Explanation

The Clear command will clear (Set to 0's) all of the user ram. User ram is set aside space for all the variables a user program will use. If the Atom is reset using RES / ATN, any values left in RAM will remain. To clear out the ram and set all variables to a value of 0 use the Clear command. It is a good idea to use the Clear command to create a known state after each reset. It can also be used any where in the program to set all variables to zero.

Example

```
    Temp        Var Byte
    Temp1       Var Byte

Main
    Temp = Temp + 1
    Temp1 = Temp1 + 2
Goto Main
```

If the above program ran through 20 complete loops Temp would equal 20 and Temp1 would equal 40. If the PICmicro was then reset using ATN / RES the ram space for Temp and Temp1 would still equal 20 and 40. The only way to clear this ram space is to completely power down the PICmicro, or set Temp = 0 and Temp1 = 0 at the beginning of the program, or use the Clear command. For a large program with several variables it would be easier to use the Clear command.

```
    Temp        Var Byte
    Temp1       Var Byte

    Clear       ;clears all variables (RAM)

Main
    Temp = Temp + 1
    Temp1 = Temp1 + 2
Goto Main
```

The above program will now set Temp and Temp1 to zero each time the PICmicro is reset or the program is ran.

Count

COUNT pin, period, variable

Count the number of cycles (0-1-0) on the specified pin during period number of milliseconds and store that number in variable (Minimum of 4us pulse width).

Pin is a variable/constant that specifies the I/O pin to use. This pin will be placed into input mode by writing a 0 to the corresponding bit of the TRIS register.

Period is a variable/constant (1 to 4294967296) specifying the time in milliseconds during which to count.

Variable is a variable (usually a word) in which the count will be stored.

Explanation

COUNT checks the state of PIN in a tight loop and will count the low to high transitions. COUNT is ideal for figuring out frequency of certain waves or timings based on an incoming signal. The PICmicro can COUNT signals with a minimum pulse width of 4µs.

Example

The program demonstrates the count command.

```
    counter var word       'Use this to hold the count
    Input B0               'Set the pin to input

    Loop:
    count B0,1000,counter
        ' If duration is 1 second the we will get the freq in hertz.

    debug ["Frequency: ",dec counter,10,13] 'Lets display it
 goto loop
```

The above program will count the low to high transitions on PIN B0 for 1 second and then store the results in the variable *counter.* If the state of pin B0 changes 100 times in one second the value stored in *counter* would equal 100.

Data (EEPROM)

DATA {@address,} value, {@address,} value

Data command will pre-load the eeprom on the PICmicro during programming. The Data command is not a run time command.

> **Address** is optional and specifies the first address to start at. Multiple address values can be used after each value is written. The Data command will start writing at the first location of the eeprom by default. Address must be a number.
>
> **Value** are constant values that will be written to the PICmicro's on board eeprom. These values must be constants since the values are placed during programming.

Explanation

The DATA command can be used for an easy method to access the built in EEPROM of the PICmicro. When using a DATA command, all the values specified will be placed in the EEPROM starting at location 0 by default unless an address is specified. To write or read data during run time from the internal EEPROM, refer to the READ / WRITE or READDM / WRITEDM commands.

Example

```
DATA 10,20,"Hello"
```

Stores the values 10,20 and the ASCII values for
"Hello" , on the internal EEPROM starting at location 0.

```
DATA @5, 10,20,"Hello"
```

Stores the values 10,20 and the ASCII values for
"Hello" , on the internal EEPROM starting at location 5.

```
DATA @5, "Hello", @10, "Good Bye"
```

Stores the ASCII values for
"Hello" , on the internal EEPROM starting at location 5 and the ASCII values for "Good Bye"

This next example shows how to pre-load values into the PICmicro's on board EEPROM and use the READ command to read them out.

```
'The data statement defaults to address 0.
Data 12,26,3,32

'Variables used to hold the data
day1 var byte
day2 var byte
day3 var byte
day4 var byte

clear 'Set all variables to 0

'Now read in the data
read 0,day1
read 1,day2
read 2,day3
read 3,day4


'Just display it forever
main:
	debug ["Day1 default=",dec day1,10,13]
	debug ["Day2 default=",dec day2,10,13]
	debug ["Day3 default=",dec day3,10,13]
	debug ["Day4 default=",dec day4,10,13]
	debug [10,13]
goto main
```

*(See read, write commands)

Debug

DEBUG [{Options} item, {{Options} item}]
Sends values of specified variables or constants to the debug watch window.

> **Options** are DEC, HEX, BIN or REAL. These modifiers will convert Item to DEC = Decimal, HEX = Hexadecimal, BIN = Binary digits or REAL = Value.
>
> **Item** can be a constant or variable. There is no limit to the amount of items used other than program memory.

Explanation

The DEBUG command will send any values stored in a given variable or constant to the debug watch window. The DEBUG command is also linked with the IDE's in circuit debugger (Refer to the *In Circuit Debugger* section of this manual). Variables used by themselves are automatically truncated to character size.

Example

```
dog VAR byte

dog = 14

DEBUG [DEC dog,13]
;output the decimal 14 to the debug window with a carriage return
```

The above sample will output the decimal number 14 to the debug window. The ,13 is the decimal value of a carriage return. This is not necessary, but it makes the data coming into the debug window easier to read. If a carriage return is not used the data will simply scroll across the screen line by line. If HEX was used in place of DEC, the hexadecimal value of the number 14 would have been sent E.

```
DEBUG ["TEMP"]
```

The above line of code would send the word TEMP to the debug window.

Using Debug

Once you have placed DEBUG statements in your program you will need to use the Debug button on the IDE to program the PICmicro. If you use the normal program mode the statements will be ignored. This allows the DEBUG statements to be placed and then later ignored without removing them and affecting your program. (Refer to the *In Circuit Debugger* section of this manual)

Important Note

When using the ICD, the running program has an added delay anywhere from .5ms to 500ms per command. Any timing critical commands or program functions will be off.

Once you have placed DEBUG statements in your program you will need to use the DEBUG button on the IDE to program the PICmicro. If you use the normal program mode the statements will be ignored.

SEE ALSO

DEBUGIN
In Circuit Debugger (ICD)

Debugin

DEBUGIN [(Options) item]
Receives byte values from the IDE DEBUG window and stores them in a specified variable on the Atom.

> **Options** are DEC, HEX, BIN or REAL. These modifiers will convert Item to DEC = Decimal, HEX = Hexadecimal, BIN = Binary digits or REAL = Value.
>
> **Item** can only be a byte variable. Stores the received byte value in the specified variable.

Explanation

The DEBUGIN command allows you to send data to your program on-the-fly from the IDE DEBUG Watch Window and stores them in a specified variable. This can be useful for adjusting a program on the fly.

Example

```
dog VAR byte

DEBUG  DOG
;stores byte value in DOG received from IDE DEBUG window.
```

The above line will simply take the value received and store in the variable DOG.

This next example demonstrates the Debugin command used with the Debug command.

```
'Holds age for use with the debugin command.
age var byte
age=0



'Our main loop
main:
	debug ["Enter your age _"]  'Output
	debugin [dec age]            'Input.  Load value into age variable

'Now display the result
	debug ["Wow you are ",dec age," years old",10,13,10,13]
goto main
```

Using Debugin

Once your program is ready for testing simply click the DEBUG button for programming. After programming is complete the Debug Watch Window will come into focus at the bottom of your screen. See figure 2-

1. Once the window is in focus and programming is complete you will need to connect (Refer to the *In Circuit Debugger* section of this manual). To send data to the running program, click in the watch window, a cursor should appear. You can now type your data in this screen and it will automatically be sent to the running program.

Important Notes

If the PROGRAM button is used instead of the DEBUG button the DEBUG and DEBUGIN statements will be ignored and not affect the running program.

SEE ALSO:
DEBUG

Do...While

Do statements While some expression is true
Repeat a group of commands while expression is true

> **Expression** is any combination of variables, constants, math and logic operators

Explanation

Execute a group of commands while some expression is true. DO...WHILE will run at least once. True can be any other value than 0.

Example

The below program will execute until Temp is equal to 0:

```
temp VAR byte
      temp = 100
Do
temp = temp -1         ;program will count down from 100 to 0

While temp <> 0        ;keep going until temp is equal to 0
```

The above program will count down from 100 to 0. The symbol <> means not equal so as long as the variable temp is not equal to 0 (a true condition) the program will continue to decrement the variable temp until it equals 0 (a false condition).

The example requires a button tied to pin B1 with pull down resistor tied to VSS and other side of the button tied to VDD.

```
  input B1
  output B2    'led tied between vss and pinB2

  'The main loop
  Main
                  'Check to see if the button is down
        if portb.bit1 = 1 then
        high B2 : pause 300 : low B2      'Blink the LED

                  'Now just waste some time until button is released
                  'As long as B1 = 1 we will stay in this loop
  do
  while B1 = 1

        endif
goto main
```

DTMFout

DTMFOUT pin,{ontime,offtime,}[,tone...]
Generate dual-tone, multifrequency tones for DTMF (i.e., telephone "touch" tones).

Pin is a variable/constant that specifies the I/O pin to use. This pin will be set to an output during tone generation. After tone generation is complete, the pin is left as an input.

Ontime is an optional entry; a variable or constant (0 to 65535) specifying a duration of the tone in milliseconds. If ontime is not specified, DTMFout defaults to 200 ms on.

Offtime is an optional entry; a variable or constant (0 to 65535) specifying the length of silent pause after a tone (or between tones, if multiple tones are specified). If offtime is not specified, DTMFout defaults to 50 ms off.

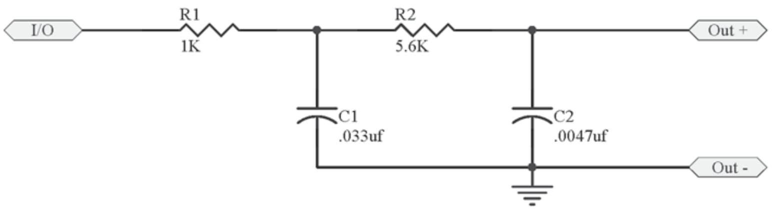
Tone is a variable or constant specifying the DTMF tone to send. Tones 0 through 11 correspond to the standard layout of the telephone keypad, while 12 through 15 are the fourth-column tones used by phone test equipment and in ham-radio applications.

0 to 9 Digits 0 - 9
10 = *
11 = #
12—15 Fourth column tones A through D

Explanation

DTMF tones are used to dial the phone or to remotely control certain radio equipment. The PICmicro is a digital controller. DTMF tones are analog waveforms, consisting of two sine waves at different frequencies. The DTMFout command creates and mixes two sine waves mathematically, then using the resulting stream of numbers to control the duty cycle of a pulse-width modulation (PWM) routine. Presence of high-frequency noise on the input. The output generated by the DTMFout command will need to be filtered as shown in the schematic. This filter circuit is only a starting point. Different conditions will require different types of filters.

Schematic



Example

```
        DTMFOUT B0,[1,7,3,4,4,2,5,1,7,4,4]
                                        ;Using default delays
        or

        DTMFOUT B0,500,200,[1,7,3,4,4,2,5,1,7,4,4]
                                        ;Using custom delays
```

The above code with an RC filter as shown above will generate tones directly, to dial a number through a phone from an attached speaker.

SEE ALSO

FREQOUT
DTMFOUT2
SOUND
SOUND2

DTMFout2

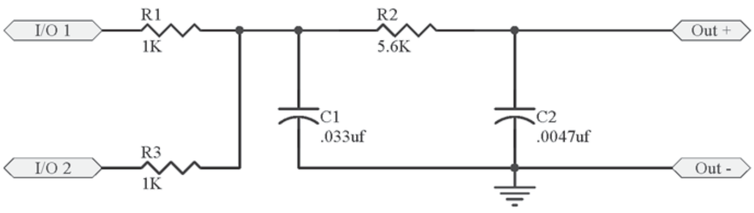
DTMFOUT2 pin1 \ pin2,{ontime,offtime,}[,Tone...]
Uses two pins to generate dual-tone, producing a cleaner signal (i.e., telephone "touch" tones).

> **Pin1 / Pin2** is a variable/constant that specifies the I/O pins to use. These pins will be set as outputs, during tone generation. After tone generation is complete, the pins are set to inputs.
>
> **Ontime** is an optional entry; a variable or constant (0 to 65535) specifying a duration of the tone in milliseconds. If ontime is not specified, DTMFout defaults to 200 ms on.
>
> **Offtime** is an optional entry; a variable or constant (0 to 65535) specifying the length of silent pause after a tone (or between tones, if multiple tones are specified). If offtime is not specified, DTMFout defaults to 50 ms off.
>
> **Tone** is a variable or constant specifying the DTMF tone to send. Tones 0 through 11 correspond to the standard layout of the telephone keypad, while 12 through 15 are the fourth-column tones used by phone test equipment and in ham-radio applications.
>
> 0 to 9 Digits 0 - 9
> 10 = *
> 11 = #
> 12—15 Fourth column tones A through D

Explanation

The DTMFOUT2 command follows the same basic format as DTMFOUT (refer to DTMFOUT), except it generates the multifrequency tones on two pins. These two pins can be tied together using a 390 ohms resistor. The tones generated by DTMFOUT2 are cleaner and of a higher quality. The DTMFOUT2 command is ideal in a noisy environment or where higher quality tones are desired.

Schematic



Example

```
DTMFOUT2 B0\B1,[1,7,3,4,4,2,5,1,7,4,4]
                                    ;Using default delays
or

DTMFOUT2 B0\B1,500,200,[1,7,3,4,4,2,5,1,7,4,4]
                                    ;Using custom delays
```

The above code with an RC filter (Schematic) will generate tones directly, to dial a number through a phone from an attached speaker.

SEE ALSO

FREQOUT
DTMFOUT
SOUND
SOUND2
SOUND8

End

END
Ends the program.

Explanation

END will stop the program after execution and enter low power mode indefinitely. All I/O lines will remain at their last know state.

For...Next

FOR variable = startVal to endVal {STEP stepVal} ... NEXT

Create a repeating loop that executes the program lines between FOR and NEXT, increment or decrement the variable according to stepVal, until the value of the variable passes the endVal.

> **Variable** is a bit, nibble, byte, word or long variable used as a counter.
>
> **StartVal** is a variable or constant that specifies the initial value of the variable.
>
> **EndVal** is a variable or constant that specifies the end value of the variable. When the value of the variable passes endVal execution stops and the program goes to the instruction after Next.
>
> **StepVal** is an optional variable or constant by which the variable increases or decreases with each trip through the FOR/NEXT loop. Negative values for StepVal will decrement and Positive values will increment.

```
For counter = 20 to 1 step -1 ; this will decrement -1
For counter = 20 to 1   ; this will over flow to 0 and count to 1

For counter = 1 to 20 step 1   ; this will increment +1
For counter = 1 to 20    ; this will increment +1 (assumed)
```

Explanation

The FOR...NEXT loop will allow your program to execute a series of instructions for a specific number of repetitions. By default, the counter variable is incremented by 1 each time through the loop. It will continue to loop until the result of the counter is outside of the range set by StartValue and EndValue.

Example

The following program counts down from 1 to 20 incrementing the variable *counter* by one, each time the program loops :

```
counter VAR byte

FOR counter = 1 to 20
      Debug [DEC counter, 13]
NEXT
END
```

The example program will execute a tight loop until the variable *counter* is equal to 20.

The following program is another For...Next loop demonstration. In the following program we set the Step value. The Step value allows you to specify which direction and what value to increment by every time through the loop. If a Step value of -2 is used the For..Next loop will count down by a value of two each time. So if the start value is 10 and the end value is 0 the loop will run 5 times.

```
                'Just some work variables
    x var byte
    y var word

                'It all starts here
    main
                'Count from 1 to 5
        debug ["1 to 5 : Display 1 2 3 4 5",13]
        for x = 1 to 5
        debug [dec x," "]
    next
        debug [13,13]

                'Count down from 5 to 0
        debug ["5 to 0 step -1 : Display 5 4 3 2 1 0",13]
        for x = 5 to 0 step -1
        debug [dec x," "]
    next
        debug [13,13]

                'Count from 10 to 50 by tens
        debug ["10 to 50 step 10 : Display 10 20 30 40 50",13]
        for y = 10 to 50 step 10
        debug [dec y," "]
    next
        debug [13,13]

                'Count from 100 to 500 by hundreds
    debug ["100 to 500 step 100 : Display 100 200 300 400 500 ",13]
        for y = 100 to 500 step 100
        debug [dec y," "]
     next
        debug [13,13]

                'Count down from 5000 to 1000 by thousands
 debug ["5000 to 1000 step -1000 : Display 5000 4000 3000 2000 1000 ",13]
	for y = 5000 to 1000 step -1000
		debug [dec y," "]
	next

		debug [13,13]

					'Some exceptions
					'Nothing in this loop will be executed
		for x = 1 to 0
		debug [dec y," "]
	next

		debug [13,13]

					'Wait 5 seconds then go back and do it all again
		pause 5000
	goto main
	end
```

Important Notes

A problem can occur when EndValue is set higher than what the counter variable can hold. The example below uses a byte-sized variable (Times), but the EndValue is set to a number greater than what can fit into a byte variable:

```
	Times VAR byte				;Counter variable
	FOR Times = 0 TO 500			;Each loop will add 1
	DEBUG [DEC Times]			;Show Times in debug window
	NEXT
```

Times is a byte variable. A byte sized variable can only hold a range from 0 to 255. The EndValue has been set to 500, which is greater than 255. This code will loop indefinitely. This is because when Times is 255 and the FOR…NEXT loop adds 1, Times becomes 0 (bytes will rollover after 255). The result of Times is 0, which is then compared against the range of 0 to 255 which is within the range, so the FOR…NEXT loop continues.

Freqout

FREQOUT pin, duration, freq1{,freq2}
Generates one or two tones for a specified duration.

Pin is a variable/constant that specifies the I/O pin to use. This pin will be put into output mode during generation of tones and left in that state after the instruction is completed.

Duration is a variable/constant specifying the length in milliseconds (1 to 65535) of the tone(s).

Freq1 is a variable/constant specifying frequency in hertz (Hz, 0 to 32767) of the first tone.

Freq2 is a variable/constant specifying frequency (0 to 32767 Hz) of the optional second tone

Explanation

FREQOUT generates one or two sine waves using a pulse-width modulation algorithm. The FREQOUT command can be used to play tones through a speaker or audio amplifier. FREQOUT can also be used to play simple songs. (See LOOKUP command)

Example

```
FREQOUT B2,1000,2500
```

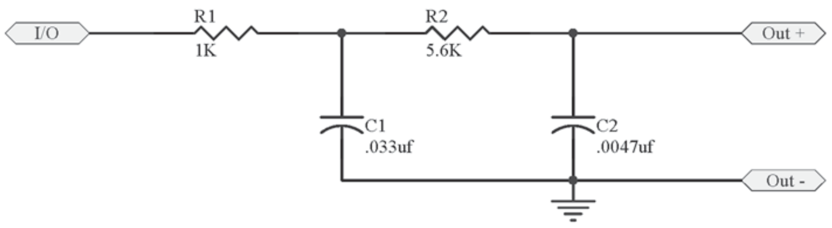
This instruction generates a 2500-Hz tone for 1 second (1000 ms) through pin 2. To play two frequencies:

```
FREQOUT B2,1000,2500,3000
```

The frequencies mix together to form a chord. To generate a silent pause, specify frequency value(s) of 0.

Below is a simple filter schematic for use with the FREQOUT command. The filter is set from 0 to 20Khz this should be good for most songs.

Schematic



Gosub...Return

GOSUB Label
Store the label after GOSUB, then go to the point in the program specified by Label.

Label specifies the section of the program to jump to.

Explanation

GOSUB is a close relative of the GOTO command. The GOSUB command tells the program to go execute code at the beginning of the specified label. Unlike GOTO, GOSUB stores the location of the next line of code immediately following itself, when the program encounters a RETURN instruction in the subroutine, it then tells the program to return to the stored location.

When GOSUBs are used, a RETURN statement is necessary (at the end of the subroutine) to take the program back to the instruction after the most recent GOSUB.

Example

```
Main
GOSUB Fish          ;Executes subroutine named fish
Goto Main

cat:                ;gosub will skip this section
    HIGH B1
    PAUSE 10
    LOW B1

Fish:
    HIGH B0
    PAUSE 10
    LOW B0
RETURN
END
```

Important Notes

The only limit is available RAM, to how many GOSUBs are allowed per program, or how many deep. In other words, a subroutine that is the destination of the first GOSUB can contain a GOSUB to another subroutine and so on. This is limited to four deep on the Basic Stamp. Since the stack size is adjustable this is not the case with the MBasic.

Goto

GOTO Label

Go to the point in the program specified by Label. (Label is a label that specifies where to go.)

Label specifies the section of the program to jump to.

Explanation

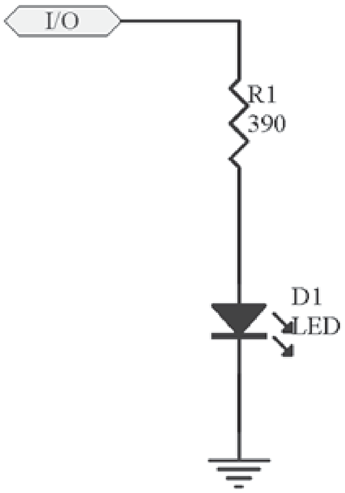
The GOTO command makes a program jump to a specific label and execute the code that starts at that location. BASIC programs are read from left to right / top to bottom, just like in the English language. The GOTO command forces the program to jump to another section of code. One use for GOTO statements is to create endless loops.

Example

```
Main
    High B0
    Pause 200
    Low B0
    Pause 200
    Goto Main        ;goto the label main and start execution
```

The above program will loop indefinitely toggling B0 from low (0 Volts) to high (5 Volts). With the below circuit connected to B0, the previous code would indefinitely blink the LED.

Schematic



High

HIGH pin
Makes the specified pin an output and sets it to high
(+5 volts is High)

Pin is a variable/constant that specifies the I/O pin to use.

Explanation

The HIGH command is used to set the designated pin to an output and to +5 volts. This allows your program to easily turn on an LED or other such devices.

Example

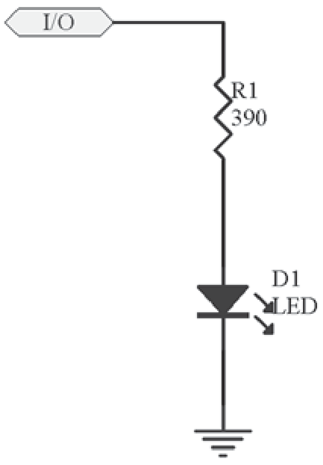
```
HIGH B0                  ;Makes pin 0 high
```

You can also assign a variable to a pin as shown below. This can help make writing a program much easier when trying to remember what pins are connected to what.

```
led con B0

HIGH led     ;Makes pin 0 high
```

The schematic below illustrates how to connect an LED for use with the HIGH command.

Schematic



I2Cin

I2CIN DataPin, ClockPin,{ErrLabel,}Control,{Address,} [{mods}Var,...VarN]
Receives data from an I2C device (EEPROM, External A/D Converter)

DataPin is a variable or constant that specifies the I/O pin to use for SDA.

ClockPin is a variable or constant that specifies the pin that the MASTER I2C device will use to clock the bus signal. (SCL)

ErrLabel is a label that the program will jump to if the I2CIN com mand fails (i.e.: device is not connected).

Control is a variable or constant that specifies the I2C device's control byte. The control byte consist of two parts The first four bits are the device type (EEproms use %1010). The next three bits are the device ID. If the address lines of the serial EEPROM (i.e. : A0, A1, A2) are grounded then the next three bits of the control byte must be zero.(ie: %1010000). The last bit is a flag used to deter mine the addressing format, 1 =16bit addressing, 0 = 8bit addressing.

Address is an optional variable or constant that specifies the starting address to read from.

Mods are command modifiers which can be used to modify the variable directly.

Var is a variable or constant that specifies the data being sent from the current bus master.

VarN is a list of variables and/or constants that specifies the data being sent from the current bus master.

What is I2C ?

I2C protocol is a form of synchronous serial communication. It only requires two I/O pins which can be shared between multiple I2C devices. I2C can communicate with peripheral chips such as EEPROM, A/D, D/A and RTC. Two lines may be used, a data clocking line and a data. Data and clocking (SCL) lines may be bussed, and appropriate chip can be selected.

Explanation

The I2CIN command allows your program to receive data from an I2C device. An example of the I2CIN command:

Read from current address:

```
I2CIN B1,B0,%10100001,[temp]
    ;stores the value of the last known address into temp
```

Read from a specific address:

```
I2CIN B1,B0,%10100001,0,[temp]
    ;stores the value of address 0 into temp
```

Read a sequence of bytes:

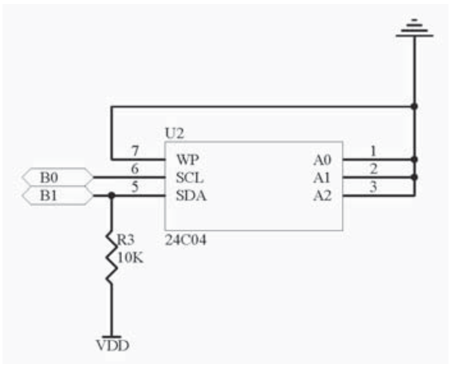
```
I2CIN B1,B0,%10100001,0,[temp,temp2]
    ;load temp with address 0 load temp2 with address 1
```

The I2CIN command will first transmit data and then receive data. This is done to first establish some information such as an ID, read or write and then an address in order to tell the I2C device what information it wants to receive. The schematic shows the proper wiring for the example to work.

The examples illustrates how to read from a I2C serial EEPROM. Pin B1 is the SDA pin and pin B0 is the clock pin (SCL). The next part of the I2C command is the control byte. The first part of the control byte for I2C serial EEPROMs is %1010, this is the start byte. The next part is 0001 because the address lines of the I2C are grounded (A0,A1,A2). The last part of the control byte sets the addressing mode, WORD size or not. Refer to the device data sheet for what addressing mode is required.

So the last bit in the control byte needs to be 1, for word addressing. The data sheet for the I2C devices will list if the device uses WORD size addressing or not. The above explanations refer to the 24C04 I2C serial EEPROM from Microchip. (Refer to I2COUT)

Schematic



Example Program

The following program will write and read a value to an I2C serial EEPROM. Note the last bit of the control byte is 1, this is because the I2C EEPROM is a 24C01 which does not use WORD size addressing. If an 24C64 were used the last bit would be a 0 for WORD size addressing. The SCL/SDA lines are pin B0/B1

```
CAT VAR BYTE
DOG VAR BYTE

CAT = 2                                     ;sets CAT = 2
DOG = 0                                     ;clears DOG = 0

Main:
    I2COUT B1,B0,%10100001,0,[CAT]
            ;writes value of 2 to address 0 of the serial eeprom

    PAUSE 1000
    I2CIN B1,B0,%10100001,0,[DOG]
            ;read value of address 0 and stores it in DOG

    PAUSE 1000
    IF DOG = 2 THEN LOOP
            ;make sure value was read correctly

Goto Main

Loop:              ;if the LED blinks the write/read was successful
    Debug [“I2C Write Was Successful”]
End
```

I2Cout

I2COUT DataPin, ClockPin,{ErrLabel,}Control,{Address,} [{mods} Var,...VarN]
Sends data to an I2C device (EEPROM, External A/D Converter)

DataPin is a variable or constant that specifies the I/O pin to use. (SDA)

ClockPin is a variable or constant that specifies the pin that the MASTER I2C device will use to clock the bus signals. (SCL)

ErrLabel is a label that the program will jump to if the I2CIN com mand fails, timeout occurs or a device is not connected.

Control is a variable or constant that specifies the I2C device's control byte. The control byte consist of two parts The first four bits are the device type (EEproms use %1010). The next three bits are the device ID. If the address lines of the serial EEPROM (i.e. : A0, A1, A2) are grounded then the next three bits of the control byte must be zero.(ie: %1010000). Thelast bit is a flag used to determine the addressing format, 1 =16bit addressing, 0 = 8bit addressing.

Address is an optional variable or constant that specifies the starting address to write the data to.

Mods are command modifiers which can be used to modify the variable directly.

Var is a variable or constant that specifies the data being sent from the current bus master.

VarN is a list of variables and/or constants that specifies the data being sent from the current bus master. This allows for multiple variables to be written to the I2C device by automatically incrementing the last given address.

What is I2C ?

I2C protocol is a form of synchronous serial communication. It only requires two I/O pins which can be shared between multiple I2C devices. I2C can communicate with peripheral chips such as EEPROM, A/D, D/A and RTC. Two lines may be used, a data clocking line and a data line. Data and clocking (SCL) lines may be bussed, and appropriate chip can be se- lected.

Explanation

The I2COUT command allows your program to write data to an I2C device. To write a byte to a 24C64 I2C serial EEPROM with Pin B1 of the PICmicro connected to SDA of the I2C device and Pin B0 connected to SCL use the following code snippet:

```
    I2COUT B1,B0,%10100001,0,[$ff]
        ;last bit of the control byte is 1 for 16bit addressing
```

In order to use an I2C serial EEPROM that does NOT use 16bit addressing (24C01) you must set the last bit of the Control byte to 0.

```
    I2COUT B1,B0,%10100000,0,[$ff]
        ;last bit is set to 0 for 8bit addressing
```

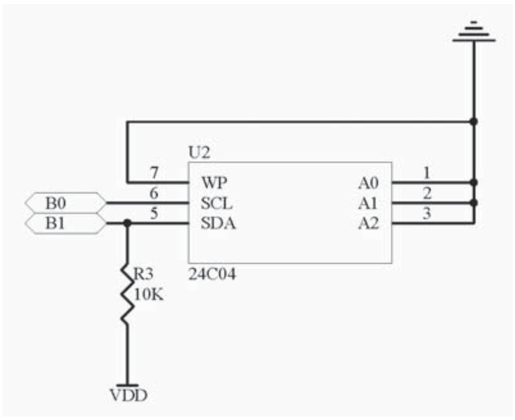
The I2COUT command is used in the same fashion as the I2CIN command. Everything that applies to I2CIN will be the same for I2COUT.

Example Program

Word addressing is required with the schematic below since the 24C04 is word addressing.

```
    I2COUT B1,B0,%10100001,0,[“H”]
        ;last bit of the control byte is 1 for WORD addressing
```

Schematic



This next I2c EEPROM program demonstrates the I2Cin and I2cout commands. This program will record the state of pin A0 over a period of time and play it back on pin A1. It was designed for use with the 512 byte EEPROM 24C04

This program makes use of the serial port to display program status. Use one of the IDE terminal windows to display the results.

```
input A0            'Sensor or button to monitor
output A1           'Led for sensor play back

      address var word
      logdata var byte
      cbit var byte                   'control bit
      cbit =%10100000                 'Set control bits starting state

      Low A1                          'turn led off

'----------------------------------------------------------------------
' Record section
'----------------------------------------------------------------------
      Debug  ["Now recording Pin A0",10,13,10,13]

      for address = 0 to 50

            logdata = porta.bit0
            porta.bit1 = porta.bit0
            Debug ["Write to address ",dec address, |
                                                                      "
value=",dec logdata,10,13]

                  'Write the state of pinA0 to address
            cbit.bit1 = address.bit8
            cbit.bit2 = address.bit9
                  'Build the control bit for address
            I2cout B1,B0, outerror,cbit,address, [logdata]
            pause  5
      next

'-----------------------------------------------------------------------
' Playback section
'-----------------------------------------------------------------------
      playback:
            Debug  ["Now playing back Pin 1",10,13,10,13]

      for address = 0 to 50
```

```
                    'Read the recorded state of pinA0 from address

                    cbit.bit1 = address.bit8:cbit.bit2=address.bit9
                    'Build the control bit for address
          I2cin B1,B0, inerror,cbit,address, [logdata]

          Debug ["Read from address ",dec address, |
          " value=",dec logdata,10,13]

          porta.bit1 = logdata
          pause 15
          'A bit of a delay because we can read faster than we can write
     next

     goto playback  'keep displaying play back until shutdown

'-----------------------------------------------------------------
' Error section
'-----------------------------------------------------------------
                    'If we get any errors then just shut down
     outerror:
          Debug  ["We had a write error",10,13,10,13]
     end

     inerror:
          Debug  ["We had a read error",10,13,10,13]
     end
```

SEE ALSO

I2CIN

If...Then...Elseif...Else...Endif

IF *Compare* THEN {Gosub} *Label*
Compare, if true(not 0) jump to label or:

IF *Compare* THEN	(If condition is true then jump to label)
Statements...	(If not true goto elseif)
ELSEIF *Compare*	(If true execute statements)
Statements...	(If not true goto else)
Else	(If nothing is true then execute statements)
Statements...	
Endif	(terminate after else statements ran)

The IF...THEN...ELSEIF...ELSE...ENDIF evaluates one or more conditions and, if true, jumps to a label. If false then skip next function

Condition is a statement, such as "x = 7" that can be evaluated as true or false.

Gosub is optional. Choosing GOSUB allows you to return to the next line of your program after running a subroutine. The default is to jump to a label.

Label is a label that specifies where to go in the event that the condition is true.

Explanation

The If...Then command a decision maker of sorts. There are two ways in which If...Then can be used. The first, test a condition and, if that condition is true, goto or gosub to a point in the program specified by an address label. The condition that IF...THEN tests is written as a mixture of comparison and logic operators. The comparison operators are:

= equal	< less than
<> not equal	>= greater than or equal to
> greater than	<= less than or equal to

IF...THEN is essentially a true or false goto. If the statement is true then the program will goto the given label. If the condition is false the program will continue onto the next line.

The second use of the If...Then can conditionally execute a group of statements following the THEN. The statements must be followed by Elseif or Else with an Endif.

Example

This example illustrates how to use IF...THEN the first way described. Multiple IF...THEN statements can be used together as such:

```
    IF Cat < 10 THEN Dog
        ;If the variable Cat is less than 10 jump to the label Dog
```

You can use the IF...THEN statement to execute a group of conditions:

```
    IF lip = 0 AND Cat < 10 THEN Dog
        ;Both conditions must be true in order to jump to Dog
```

You can use the GOSUB option to simplify your program:

```
    hat = 1
    IF hat = 1 THEN GOSUB Tip         ;this program will loop forever
    STOP

    Tip:
    RETURN    ;send program back to the next line after the gosub
```

The second use of the IF...THEN statement is multiple conditionally executed statements. This method allows several statements to be executed based on TRUE conditions, with a closing statement execute if all is FALSE.

```
    Hat var Byte

    Hat = 1

    IF hat=0 THEN       ;condition is false since Hat equals 1
        High B1         ;skipped since condition was false
    ELSEIF Hat=0        ;condition is false since Hat equals 1
        High B2         ;skipped since condition was false
    Else                ;conditionally executes if all conditions were false
High B3                 ;executes and pin 0 is set high
Endif                   ;closes the IF...THEN statement
```

Multiple conditions can be compared as shown below:

```
    Hat var Byte
    Hat = 1
    IF hat=0 THEN       ;condition is false since Hat equals 1
        High B1         ;skipped since condition was false
    ELSEIF Hat=0                  ;condition is false since Hat equals 1
        High B2                   ;skipped since condition was false
    ELSEIF Porta.bit0=0           ;is pin A0 true (low) ?
        High B3                   ;skip if false
    ELSEIF Porta.bit1=0           ;is pin A1 true (low) ?
        High B4                   ;skip if false
    ELSEIF Porta.bit2=0           ;is pin A2 true (low) ?
        High B5                   ;skip if false
    ELSEIF Porta.bit3=0           ;is pin A3 true (low) ?
        High B6                   ;skip if false
    Else            ;conditionally executes if all conditions were false
High B7             ;executes and pin 10 is set high
Endif               ;closes the IF...THEN statement
```

The above program will only work if pin A0, A1, A2, A3 are set to inputs first. The program will check to see if pins A0 through A3 are set low (VSS). If all pins are not set low then execute ELSE and set pin B7 high, if any pins are set low, then execute next statement, finish comparisons, skip ELSE since one condition was true and exit. The ELSE is always used if you want your program to do something if all comparisons are false. If you want the program to just exit if all conditions are false then omit ELSE once the group of commands are finished running the program will exit on the ENDIF and continue on to the next line of code. The ELSEIF is unlimited, you can use as many needed. The only limit to this is code size.

Important Notes

A few conditions must be considered when using the IF...THEN statement. Constants can not be used with the IF...THEN statement. This is because constants only have one condition and the statement will always be TRUE or FALSE. This also applies to PIN names since these are constants to. The below line will always return a FALSE state:

```
    IF B0 = 1 THEN Main
```

The above statement has no way of knowing that you mean the value of pin 0. It assumes this is a constant. The correct way to check the condition of a PIN in the IF...THEN statement would be to access the pin and port directly as shown below:

```
    IF portb.bit0 = 1 THEN Main
```

The above statement will now look directly at the condition of pin B0. This statement will now change based on the condition of pin B0 (High or Low).

The next example of the If...Then...Else command is a little more useful when testing logic and conditional executing code. To get a real good idea of how the If...Then...Else works run this program with the ICD and use the step or animate option to follow the code.

```
y var word

y = 1

        debug [10,13]
main:
        debug [dec y," "] 'Display the current count

                                        'Test for 5
        if y=5 then gosub five

                                        'Test for 21
        if y = 21 then
        debug [10,13,"Have 21",10,13]
endif

        'Here we will test to see how close we are to being done.
if y = 50 then
        debug [10,13,"All Done",10,13]
        debug ["However we will do it all again",10,13,10,13]
        y=1           'We are done reset the counter
        goto main
elseif y = 45
        debug [10,13,"5 more to go",10,13]
        y=y+1    'At this point lets increment the counter
        goto main
else
        y=y+1    'At this point lets increment the counter
        goto main
endif

                'A little subroutine test
five:
        debug [10,13,"Have 5",10,13]
return
```

Input

INPUT pin
Makes the specified pin an input

Pin is a variable or constant that specifies the I/O pin to use.

Explanation

There are several ways to make a pin an input. When a program begins, all of the pins should be inputs. Input instructions PULSIN, SERIN will automatically change the specified pin to input and leave it in that state.

Example

```
INPUT B0 ;Make pin B0 an input
INPUT B1 ;Make pin B1 an input
```

You can also assign a variable to a pin as shown below for the INPUT command:

```
        myswitch VAR portb.bit0
                ;myswitch is now an alias of pin B0

    MAIN:
        IF myswitch = 0 THEN MAIN
                ;check if pin B0 is low

    IF myswitch = 1 THEN RUNIT
                ;check if switch is on if so goto label RUNIT

    GOTO MAIN
                ;will continue to loop until pin 0 is high

    RUNIT:
        HIGH B1         ;pin B1 is set high turn on a LED
        PAUSE 500       ;a delay so we can see the LED on
        LOW B1          ;Turns the LED off
    GOTO MAIN           ;goes back to main loop and starts over again
    END
```

Lcdread

LCDREAD RegSel\Clk\RdWrPin, Nib, Address, [{mods} Var]
Reads the RAM on a LCD module using the Hitachi 44780 controller or equivalent.

RegSel can be a constant or variable specifying the pin for the R/S line of the LCD.

Clk can be a constant or variable specifying the pin for the E (Enable) line of the LCD.

RdWrPin can be a constant or variable specifying the pin for the R/W (Read / Write) line of the LCD.

Nib can be a constant or variable specifying the pins (Total of Four) for the data lines of the LCD. The LCD data port is arranged in 8 bits. Only 4 bits are required 4 to 7 of the LCD. So if Portb.nib0 is selected then LCD bit 4 is connected to pin B0 on the PICmicro and LCD bit 5 is connected to pin B1 on the PICmicro and so on.

Address can be a constant or variable that specifies the address location of RAM you are trying to read. Address from 0 to 127 return the current character in the display memory. Address 128 and above return Character RAM values.

Mods are command modifiers which can be used to modify the variable directly.

Var is the variable where the value returned will be stored.

Explanation

Accessing the on board RAM of a LCD module is very straight forward. If the LCDWRITE command is used as shown in the example below. The character in address 0 will be "H', address 1 will be 'e' and so on.

Example

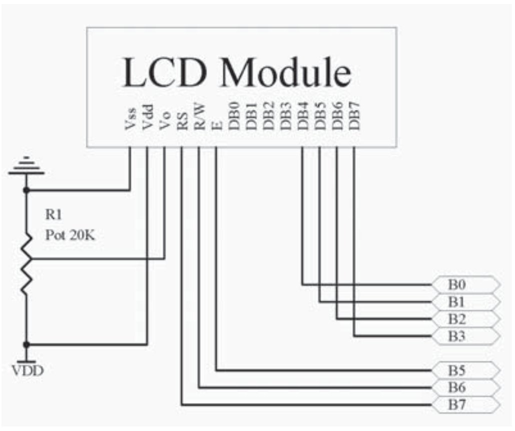
```
    pause 500              ;Allows the LCD to initialize

    LCDWRITE B4\B5\B6, portb.nib0, [INITLCD1, INITLCD2, TWOLINE,
CLEAR, HOME, SCR,"Hello"]      ;prints the word 'Hello' to the screen
```

If the above code example is executed, the on-board LCD RAM will have the word 'Hello' loaded into it. To read what was printed on the LCD screen, use the LCDREAD command as shown:

```
    character VAR byte
```

```
LCDREAD B4\B5\B6, portb.nib0, 0, [character]          ;Character  = “H”
LCDREAD B4\B5\B6, portb.nib0, 1, [character]          ;Character  = “e”
LCDREAD B4\B5\B6, portb.nib0, 2, [character]          ;Character  = “l”
LCDREAD B4\B5\B6, portb.nib0, 3, [character]          ;Character  = “l”
LCDREAD B4\B5\B6, portb.nib0, 4, [character]          ;Character  = “o”
```

Schematic



Important Notes

When using the LCDREAD command the R/W pin must be connected to a pin. The R/W pin is not necessary when the LCDWRITE command is used wihtout the LCDREAD command.

SEE ALSO

LCDWRITE

Lcdwrite

LCDWRITE RegSel \ Clk {\RdWrPin}, Nib, [{mods} Exp]
Sends Text to an LCD module using an Hitachi 44780 controller or equivalent.

> **RegSel** is optional for LCDWRITE and can be a constant or variable specifying the pin for the R/S line of the LCD.
>
> **Clk** can be a constant or variable specifying the pin for the E (Enable) line of the LCD.
>
> **RdWrPin** can be a constant or variable specifying the pin for the R/W (Read / Write) line of the LCD.
>
> **Nib** can be a constant or variable specifying the pins (Total of Four) for the data lines of the LCD. The LCD data port is arranged in 8 bits. Only 4 bits are required 4 to 7 of the LCD. So if Portb.nib0 is selected then LCD bit 4 is connected to pin B0 on the PICmicro and LCD bit 5 is connected to pin B1 on the PICmicro and so on.
>
> **Mods** are command modifiers which can be used to modify the variable directly.
>
> **Exp** can be a constant or variable that is the data to be written.

Explanation

When using the LCDWRITE command you will need to first initialize the LCD screen. This can be done by adding the below line to your program:

```
	pause 500		;Allows the LCD to initialize

	LCDWRITE B4\B5\B6, portb.nib0, [INITLCD1, INITLCD2, TWOLINE,
CLEAR, HOME, SCR]
		;after the first initialization no more inits. will need to be sent.
```

The first line of code is the PAUSE statement. This is used to give the LCD enough time to power up internally. A value of 500 is often used.   This pause is not necessary on all LCD displays.

Example

The code below will print the character string “Hello” then “World” to the LCD display.

```
	Temp Var Byte

	Pause 500		;Allows the LCD to initialize
```

```
    LCDWRITE B4\B5\B6, portb.nib0, [INITLCD1, INITLCD2, TWOLINE,
CLEAR, HOME, SCR]
        ;after the first initialization no more inits. will need to be sent.

    Main
        Temp = 30
        LCDWRITE B4\B5\B6, portb.nib0, [“Hello”]
        Pause 200
        LCDWRITE B4\B5\B6, portb.nib0, [CLEAR, HOME, “World”]
        Pause 200
        LCDWRITE B4\B5\B6, portb.nib0, [DEC TEMP]
                        ;prints the decimal numbers 30
        Pause 200
        LCDWRITE B4\B5\B6, portb.nib0, [iHEX TEMP]
                        ;prints the Hex value of 30 with indicator ($1E)
    Goto Main
```

There are several control commands that can be used with LCDWRITE such as CLEAR and HOME. Each additional control command used with LCDWRITE must be separated with a “,” (comma) inside of the brackets “[...]”. Below is a chart of all the available control commands for use with LCDWRITE.

LcdWrite Comand Table

Command Name		Description
$133	INITLCD1	Initialize LCD display
$132	INITLCD2	Initialize LCD display
$101	CLEAR	Clear Display
$102	HOME	Return Home
$104	INCCUR	Auto Increment Cursor(default)
$105	INCSCR	Auto Increment Display
$106	DECCUR	Auto Decrement Cursor
$107	DECSCR	Auto Decrement Display
$108	OFF	Display,Cursor, and Blink off
$10C	SCR	Display on,†Cursor and Blink off
$10D	SCRBLK	Display and Blink on, Cursor off
$10E	SCRCUR	Display and Cursor on, Blink off
$10F	SCRCURBLK	Display, Cursor, and Blink on
$110	CURLEFT	Move Cursor left
$114	CURRIGHT	Move cursor right
$118	SCRLEFT	Move Display left
$11C	SCRRIGHT	Move Display right
$120	ONELINE	Set display for 1 line LCDs
$128	TWOLINE	Set display for 2 line LCDs
$140	CGRAM | address	Set CGRAM address for R/W
$180	SCRRAM | address	Set Display ram address for R/W

Cursor Position and Screen Ram

Screen ram is where the characters are stored before being printed to the screen. To print a given character to the screen the internal cursor must be positioned at the given location. This is normally handled automatically. If you want to print to other places on the screen instead of a left to right fashion you must first position the internal cursor at the correct screen ram location. If you wanted to print to the second line, first character of a 16x2 LCD then you would use the following:

```
    LCDWRITE B4\B5\B6, portb.nib0, [scrram+$40]
```

The above command tells the LCD to point the cursor at location 40h, which is the location of line two, first character. The "+" symbol is require since your adding 40h to the current location of the screen ram. You can use the SCRRAM control command to move the cursor anywhere on or off the LCD screen. Since all HD44780 type controllers have the same amount of ram regardless of the LCD size the internal ram is mapped out as shown below:

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
40	41	42	43	44	45	46	47	48	49	4a	4b	4c	4d	4e	4f

All values shown above are in HEX. The main controller (HD447800) is setup up for a maximum of 128 bytes of screen ram. A 2x16 display would only use 32 bytes so the remainder would be off screen ram. Which can be accessed by shifting the display left after writing to an off screen ram location.

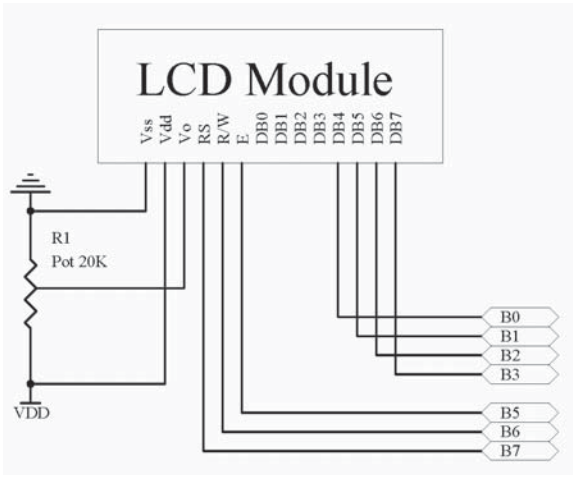
Off screen ram would be as follows; 10h to 3fh would be off screen for the first line, then 40h for the first location of the second line on screen. Then 50h to 7fh for off screen on the second line for a two line LCD. To print in off screen ram, move the pointer to location 10h. Then send the characters to be printed. If you send the message "Hello" to location 10h to 15h this will be in off screen ram. To move this message on to the screen do the following:

```
    LCDWRITE B4\B5\B6, portb.nib0, [SCRLEFT]
```

This will shift the display once to the left. So the first letter of the message will appear "H". To move the remainder of the message onto the screen you must shift it left 4 more times by sending the above 4 times.

Below is a schematic of a LCD with an Hitachi 44780 type controller for use with LCDWRITE and using the default system settings:

Schematic



Let

LET Var = expression
Assign a value to a variable

> **Var** is the labeled variable
>
> **Value** can be a constant or another variable or the result of an expression.
>
> **Expression** can be any (legal) combination of math operators.

Explanation

LET is an optional command; as an example Temp=2 is the same as LET Temp=2. The LET command is often used to make programming code more human readable.

Example

The statement [LET] will assign values to variables using operators as shown below:

```
LET Cat = 1 * Dog + Fish

OUTPUT B0        ;sets pin B0 to an output
LET B0 = 0       ;makes pin B0 low
```

Important Note

The LET command can also use modifiers.

Low

LOW pin
Make the specified pin output low

> **Pin** is a variable or constant that specifies the I/O pin to use.

Explanation

The LOW command will make the specified pin Low (0 Volts), which will also make the specified pin an output. Pin can be a constant (0, A0, 2, A2) or a variable that contains a pointer that directly corresponds to an available pin on any given port.

Example

```
HIGH A0                 ;Makes pin A0 high
```

You can also assign a variable to a pin as shown below. This can help make writing a program much easier when trying to remember what pins are connected to what.

```
led VAR byte

led = PortB.bit0        ; makes the variable led point to pin B0.
LOW led                 ;Makes pin 0 Low (0 Volts)
```

Lookdown

LOOKDOWN value,{comparisonOp,}[value0,
value1,...valueN],resultVariable

Compare a value to a list of values according to the relationship specified by the comparison operator. Store the index number of the first value that makes the comparison true into resultVariable. If no value in the list makes the comparison true, resultVariable is unaffected.

Value is a variable or constant to be compared to the values in the list.

ComparisonOp is optional and maybe one of the following:

= equal	< less than
<> not equal	>= greater than or equal to
> greater than	<= less than or equal to

If no comparison operator is specified, Then MBasic defaults to equal (=).

Value0, value1... make up a list of values (constants or variables) up to 16 bits in size.

ResultVariable is a variable in which the index number will be stored if a true comparison is found.

Explanation

LOOKDOWN works very much like the index of a book. You can search for specific topic and return the page number. LOOKDOWN searches values in a list, and stores the item number of the first match in a variable. In other words, Lookdown compares the user value to a values in a list, the first comparison that is true, will return the value of the index position the match was found. The index list starts at 0 not 1. This is because 0 is always treated as the true first number when dealing with computers and microcontrollers.

Example

```
value VAR byte
result VAR nib
value = 2
LOOKDOWN value,[10,150,180,2001000,25,2],result
```

In the above code example we have specified a value of 2 without a comparison operator. So the default comparison will be equals. Lookdown will search through the list searching for the first true comparison. The first true comparison is 2 which is in the 6th position, so the value returned, in result will be 6.

Lookdown can also index constant strings as shown below. Any spaces are also counted as positions.

```
	value var byte
	result var byte
Main
	value = "R"
	lookdown value,["See Spot Run"],result
	Debug [dec result]
Goto main
```

The result would be 9. This is because R is in the 9th position.

Lookup

LOOKUP index, [value0, value1,...valueN], resultVariable

Look up the value specified by the index and store it in a variable. If the index exceeds the highest index value of the items in the list, variable is unaffected. A maximum of 256 values can be included in the list.

> **Index** is the item number (constant or variable) of the value to be retrieved from the list of values.
>
> **Value0, value1**... make up a list of values (constants or variables) up to 16 bits in size.
>
> **ResultVariable** is a variable in which the retrieved value will be stored.

Explanation

Lookup could be considered the opposite of Lookdown. Lookup retrieves an item from a list based on the item's position (index) in the list.

Example

```
index VAR nib
result VAR byte
index = 3
result = 255

LOOKUP index,[26,177,13,1,0,17,99],result
```

The above example will go to position 3 in the list, starting from left to right. The list starts at 0 not 1. So the returned value stored in result would be 1.

The following program uses the LOOKUP command to play “Mary had a little lamb” using the FREQOUT command.

```
AH con 440*2            ;assigns a value for the note and octave
AS con 466*2            ;the note is 466*2 = A sharp second octave
BH con 494*2
CH con 523*2
CS con 554*2
DH con 587*2
DS con 622*2
EH con 659*2
FH con 698*2
FS con 740*2
GH con 784*2
GS con 831*2
```

```
;above we assign our notes as constants with values for the FEQOUT
command

temp var byte
temp2 var word

main
for temp = 0 to 33
        lookup temp,[CS,BH,AH,BH,CS,0,CS,0,CS,BH,0,BH,0,BH,0,CS,|
EH, 0,EH,CS,BH,AH,BH,CS,0,CS,0,CS,BH,0,BH,CS,BH,AH],temp2
        ; the above statement must be formatted to work correctly
        ; the “|” or symbol is used to indicate a line break, when
        ‘ a command is too long to fit on one line.
         if temp2 = 0 then finish
        freqout B0,500,temp2
finish:
next
pause 5000

goto main
```

Nap

NAP period
Enter sleep mode for a short period. Power consumption is reduced to about 50 µA assuming no loads are being driven. All pins are output low.

Period is a variable or constant that determines the duration of the reduced power nap. The duration is (2^period) * 18 ms. (Read as "2 raised to the power of ‘period’, times 18 ms.") Period can range from 0 to 7, resulting in the following nap lengths:

Period	2 period	Length of Nap
0 -	1	18ms
1 -	2	36ms
2 -	4	72ms
3 -	8	144ms
4 -	16	288ms
5 -	32	576ms
6 -	64	1152ms (1.152 seconds)
7 -	128	2304ms (2.304 seconds)

Explanation

Nap uses the same shutdown and start-up mechanism as Sleep, with one difference. During Sleep, variations are automatically compensated for. As a result, longer Sleep intervals are accurate to approximately ±1 percent. Nap intervals are directly controlled by the watchdog timer without compensation. Variations in temperature, supply voltage, and manufacturing tolerance of the PICmicro you are using can cause the actual timing to vary.

One use for NAP is in a battery-powered application where some small amounts of time are spent doing nothing. This would apply to a program that loops endlessly, performing some task, and pausing for approximately 300 ms each time through the loop. You could replace the PAUSE 300 with NAP 4, as long as the timing of the pause was not critical. The NAP 4 would pause your program for about 288ms, plus put the microcontroller in low-power mode, which would extend battery life. During wake up the I/O pins would be interrupted briefly. One method around this would be to use weak pullup / pulldown resistors such as 10K or 100K.

Example

```
High B0      ;turns an LED on connected to pin B0
NAP 4        ;Low power mode for about 288ms
Low B0       ;Turns LED off
```

Output

OUTPUT pin
Makes the specified pin an output

> **Pin** is a variable or constant that specifies the I/O pin to use.

Explanation

There are several ways to make a pin an output. Commands that rely on output pins (SEROUT) automatically change the specified pin to output. The OUTPUT command allows your program to directly affect the state of the specified pin.

Example

```
    OUTPUT B0            ;Pin B0 is now an output (+5 volts)
```

Using a variable to set the pins state:

```
    time VAR byte
    time = PortB.bit0

    OUTPUT time          ;Pin B0 is now an output
```

OWIN

OWIN Pin,Mode,{NCLabel,} [{Mods} Var]
Protocol used to communicate to 1-wire devices.

Pin is a variable or constant that specifies the I/O pin to use for the One wire command.

Mode is a variable/constant/expression indicating the mode of data transfer. Mode controls placement of reset pulses, detection of presence pulses, byte / bit input and normal / high speed. The proper value for Mode will depend on the 1-wire device used. Consult the device data sheet to determine the correct Mode. See chart below:

Mode	Setting
0	No Reset, Byte mode, Low speed
1	Reset before data, Byte mode, Low speed
2	Reset after data, Byte mode, Low speed
3	Reset before and after data, Byte mode, Low speed
4	No Reset, Bit mode, Low speed
5	Reset before data, Bit mode, Low speed

NCLabel is a label the program can jump to if a connection failure occurs with the OWIN command (ie. No chip present).

Pin is a variable or constant that specifies the I/O pin to use for the One wire command.

Mods are command modifiers which can be used to modify the variable directly.

Var is the variable or variable array where the value(s) returned will be stored.

Explanation

1-wire protocol is a form of asynchronous serial communication developed by Dallas Semiconductor. It requires only one I/O pin. This one pin can be shared between multiple 1-wire devices. The OWIN command allows the PICmicro to receive data from 1-wire devices.

The 1-Wire protocol synchronizes the slave devices to the master. The master initiates and controls all activities on the 1-Wire bus. 1-Wire uses CMOS/TTL logic levels. A resistor connects the data line of the 1-Wire bus to the 5V supply of the bus master.

Hardware Configuration

The 1-Wire bus has only a single line I/O. It is important that each device on the bus be able to drive this single I/O line at the appropriate time. Each device attached to the 1-Wire bus must have an open drain connection or 3-state outputs. The DS2401 is an open drain part. The bus master requires a pullup resistor at the master end of the bus, with the bus master circuit equivalent to the one shown in for the schematic.
The value of the pullup resistor should be 4.7K for short line lengths. A multi 1-wire bus consists of a 1-Wire bus with multiple slaves attached.

Important Note

Wiring of the Dallas part is important. It may seem simple but allot of issues arise from incorrect wiring. Use the code example below to check your wiring. The code below is a part ID check. You can also use this code to identify any devices you may have on the bus.

```
Temp            VAR Byte
Counter VAR Byte

Main
    OWout B0,1,main,[$cc]

    For Counter = 0 to 7
        OWin B0,0,[temp]
        Debug [hex temp," "]
    Next
        Debug [13]

Goto Main
```

The above code will print the part ID in the Debug window. To run the program use the ICD. Also you will notice a NCLabel is used “main”. This label will keep the program in a loop until something is received from the part connected.

Example

```
Temp Var Byte
Counter Var Byte


Main
        Owout B0,1,main,[$33]
        ;Sense Presence and call 1-Wire RomCheck routine

    For Counter = 0 to 7  ;Read 8 byte Code from DS2401
        Owin B0,0,[temp]
        Debug [hex temp," "]        ;If code does not match recheck
```

```
	Next
		Debug [13]
	Goto Main
```

The example works with the DS2401 which is a device that guarantees a unique id per device. These parts are ideal for security applications. The first part of the example Owout B0,1,main,[$33] is a rom check routine (See Rom Functions). Every 1-wire device has a rom id. By using the rom check routine you can ensure there is a 1-wire device attached. This will only work with one device attached. The next part of the code a loop to get all 8 unique ID bytes from the DS2401. This key will be different for every DS2401 part.

Rom Functions

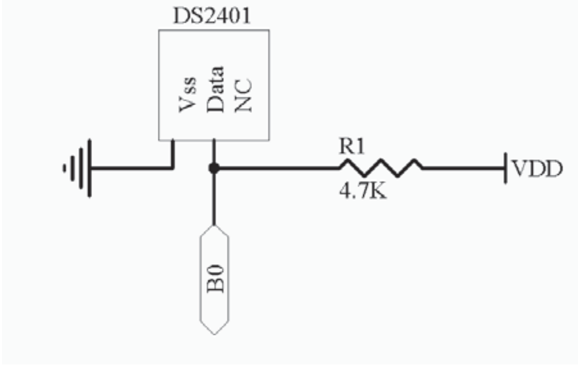
Read ROM $33 or $0F
Only works if a single 1-wire device is on the line.

Skip ROM $CC
Address a 1-wire device without its 64-bit ID. Only works if a single 1-wire device is on the line.

Search ROM $F0
Read the 64-bit IDs of all 1-wire devices on the line. A process of elimination can be used to distinguish each unique device.

Match ROM $55
Match Rom, followed by a 64-bit ID, will allow addressing of a unique device on a multiple device line.

Schematic



OWOUT

OWOUT Pin,Mode,{NCLabel,} [{Mods} Exp]
Protocol used to communicate to 1-wire devices.

Pin is a variable or constant that specifies the I/O pin used for the One wire command.

Mode is a variable/constant/expression indicating the mode of data transfer. Mode controls placement of reset pulses, detection of presence pulses, byte / bit input and normal / high speed. The proper value for Mode will depend on the 1-wire device used. Consult the device data sheet to determine the correct Mode. See chart below:

Mode	Setting
0	No Reset, Byte mode, Low speed
1	Reset before data, Byte mode, Low speed
2	Reset after data, Byte mode, Low speed
3	Reset before and after data, Byte mode, Low speed
4	No Reset, Bit mode, Low speed
5	Reset before data, Bit mode, Low speed

NCLabel is a label the program can jump to if a connection failure occurs with the OWOUT command (ie. No chip present).

Mods are command modifiers which can be used to modify the variable directly.

Exp is a variable, variable array, constant or expression containing the data to be sent.

Explanation

1-wire protocol is a form of asynchronous serial communication developed by Dallas Semiconductor. It requires only one I/O pin. This one pin can be shared between multiple 1-wire devices. The OWOUT command allows the PICmicro to send data to 1-wire devices. The OWout and OWin are tightly integrated. In most cases you will need both to talk to any one wire part.

The 1-wire protocol has a very specific format. Every transaction will consists of four parts:

1. Initialization.
2. The ROM Function Command.
3. Memory Function Command.
4. Data

The ROM Function and Memory Function will always be 1 byte. This byte is sent LSB first. The Initialization will consists of a reset pulse which is generated by the master device. Which is followed by a presence pulse generated by any attached slave devices. The next operation is the reset pulse which is controlled by the lowest two bits of the mode (Refer to Modes).

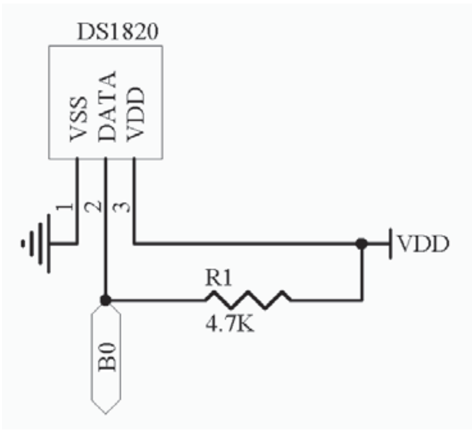
Following this Initialization is the ROM Function. The ROM Function is used to address a specific 1-wire device. When more than one device is attached each will need to be address individually using the Match ROM command.

Example

The following example will read the temperature from a DS1820 one wire device. The result is displayed in Celsius with the decimal point intact. This is accomplished by using a long variable and the float command.

```
temp            var word
temp1           var word
Convert         var long
counter         var byte

main
    Owout B0,1,main,[$cc,$44]

Wait
    owin B0,0,[temp]
    if temp = 0 then wait
        Owout B0,1,main,[$cc,$be]
        owin B0,0,[temp.byte0,temp.byte1]
        Convert = float temp fdiv 2.0
    Debug ["Temperature = ",real convert," C",13]
goto main
```

Schematic



Pause

PAUSE milliseconds
Pause the program (do nothing) for the specified number of milliseconds.

> **Milliseconds** is a variable or constant specifying the length of the pause in ms. Pauses may be up to a 32 bit number

Explanation

The PAUSE command delays the execution of the program for the specified number of milliseconds.

Example

```
flash:
      lOW B0
      PAUSE 100
      HIGH B0
      PAUSE 100
GOTO flash
```

This code causes pin B0 to go low for 100 ms, then high for 100 ms. The delays produced by PAUSE are as accurate as the ceramic resonator time base, ±1 percent. So a statement of PAUSE 1000 would be 1 second

Pauseclk

PAUSECLK cycles
Pause the program (do nothing) for the specified number clock cycles divided by 4.

> **Cycles** is a variable or constant specifying the length of the pause. Pauses may be up to a 32 Bit number.

Explanation

The PAUSECLK command delays the execution of the program for the specified number of clock cycles divided by 4. Since each oscillator has a variances of 0.05% you will need to determine your own timings. This is only necessary if precision timing is required.

Example

```
Flash:
      Low B0
      PAUSECLK 100
      High B0
      PAUSECLK 100
Goto Flash
```

The above code will delay or pause for 100 internal clock cycles divided by 4.

Pauseus

PAUSEUS microseconds
Pause the program (do nothing) for the specified number of micro seconds.

> **Microseconds** is a variable or constant specifying the length of the pause in µs. Pauses may be up to 65535 µs.

Explanation

The PAUSE command delays the execution of the program for the specified number of microseconds.

Example

```
Flash:
    Low B0
    PAUSEUS 100
    High B0
    PAUSEUS 100
Goto Flash
```

The above code causes pin 0 to go low for 100 µs, then high for 100 µs.

Important Notes

The delays produced by PAUSEUS may vary depending on how it is used. If the value for Pauseus is a constant its accuracy will be greater, opposed to using a variable. This is due to the time it takes to load the value and execute the Pauseus command. The minium time for Pauseus is 1µs.

PEEK...POKE

```
PEEK address, variable
POKE address, expression
```

Read/Write specified RAM location

Address is a variable or constant which denotes a memory location.

Variable is a variable name and is where the results will be stored.

Expression is any combination of variables, constants and math operations.

Explanation

PEEK and POKE are considered advance commands and should only be used by experienced users. The explanation of these commands are kept short intentionally. Use of the PEEK command allows a specific address to be read and store its value in the assigned *variable .* The PEEK and POKE commands allow direct access to all of the registers, including all of the PORT states and their data direction registers (TRIS).

Example

```
store VAR byte

PEEK B0,store
    :get current register status of B0 and store it in the variable store

POKE B0,store
    ;write the value of store (PIN B0)
```

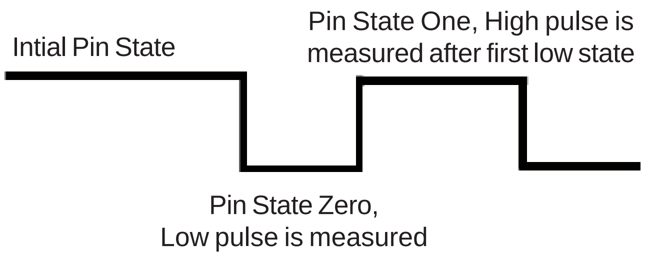
Pulsin

PULSIN pin, state, {TimeoutLabel,TimeoutMultiple,} Var
Measure the width of a pulse.

> **Pin** is a variable or constant that specifies the I/O pin to use. This pin will be placed into input mode during pulse measurement and left in that state after the instruction finishes.
>
> **State** is a variable or constant (0 or 1) that specifies whether the pulse to be measured begins with a 0-to-1 transition (1) or a 1-to-0 transition (0).
>
> **TimeoutLabel** is an optional label that specifies where to go if a time out occurs. The default time out value is 65,535 microseconds.
>
> **TimeoutMultiple** is a variable or constant that specifies the amount of time to wait before timing out. The time out value is multiplied by 65,535 microseconds which is the default value. If a value of 10 is used the command would wait 655,350 microseconds.
>
> **Var** is a variable in which the pulse duration will be stored.

Explanation

PULSIN will measure the pulse width on a specified pin. If the state is zero, the width of a low pulse is measured. If the state is one, the width of a high pulse is measured. The measured width is then placed in Var. If the pulse edge never happens or the width pulse is too great to measure Var will default to 0. The pulse is measured in 16 bits, if a one byte Var is used only the lower 16 bits are used. Pulsin will timeout after 65,535 microseconds if the optional timeout label is not used.


Intial Pin State
Pin State One, High pulse is measured after first low state
Pin State Zero, Low pulse is measured

PULSIN will return the pulse width in µs.

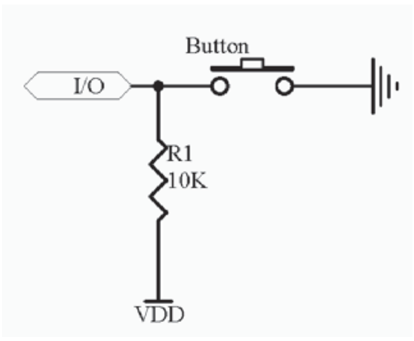
Example

```
PULSIN B0,0,Cat
      ;Measure low pulse on P0 store it in the variable Cat
```

The following program uses Pulsin to measure the time a button is pressed as shown in the schematic. The value returned is in Pulsin units. A 10K pullup resistor is used to keep the line high. Pulsin will wait until the line changes state. Since the state setting is 0 Pulsin will wait until the line state is low (0). Then begin the measurement of time while the pin state remains low. Once the line is released and set high again Pulsin will stop counting and return a value to Var. The command will reset every 655,350 micro seconds since we used the timeout label.

```
temp var word

      Main:                                       'this program will loop forever
            PULSIN B0,0,main,10,temp
                        'Measure negative pulse will time out at 655,350us.
            IF temp = 0 then Main                 'If 0, try again.
            serout B1, i9600, [dec temp]          'display result.
      goto Main                  'repeat
      end
```

Schematic



Pulsout

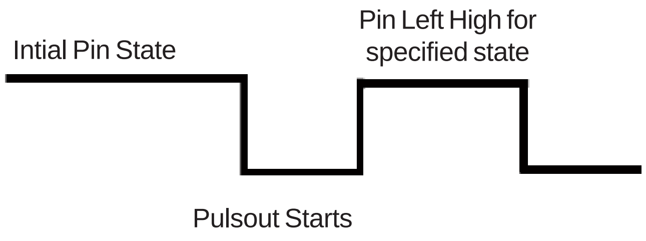
PULSOUT pin, time
Output a pulse.

> **Pin** is a variable or constant that specifies the I/O pin to use. This pin will be placed into output mode immediately before the pulse and left in that state after the instruction finishes.
>
> **Time** is a variable or constant (0-65535) that specifies the duration of the pulse in µs.

Explanation

PULSOUT will generate a pulse on the specified pin for the given period. The pulse is generated by toggling the pin state twice. The initial state of the pin will determine the polarity of the pulse. The pin specified to generate the pulse is automatically made an output.

PULSOUT will generate a pulse with a period in 1 µs increments. The minimum pulse width is 4 µs. You can not go below this value.



Example

```
PULSOUT B0,1000
    ;Generate a pulse for 1 millisecond long to B0
```

The small program below illustrates one use of the PULSOUT command:

```
Low   B4                        ;Preset Pin to LOW

main                            ;this program will loop forever
    pulsout B4,25000            ;Pulse pin High for 25 of a millisecond
    pause 1000                  ;Wait 1 second until repeating
goto main
end
```

This program will simply generate a infinite pulse on pin B4. The PULSOUT command has many uses. One of which can be to blink a LED for a certain duration.

Pwm

PWM pin, duty, cycles
Convert a digital value to analog output via pulse-width modulation.

> **Pin** is a variable or constant that specifies the I/O pin to use. This pin will be placed into output mode during pulse generation then switched to input mode when the instruction finishes.
>
> **Duty** is a variable or constant (0-255) that specifies the analog output level (0 to +5V).
>
> **Cycles** is a variable or constant (0-65535) specifying an approximate number of milliseconds of PWM output.

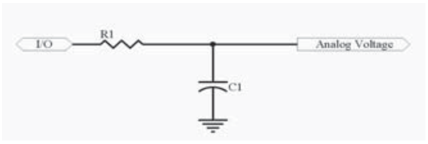
Explanation

Pulse-width modulation (PWM) allows a digital device to generate an analog voltage. To do this a pin is set to Output high (5 Volts) or Output low (close to 0 Volts). If you switch the pin rapidly between high and low at a given interval so that it was high half the time and low the other half. The average voltage over time will be halfway between 0 and 5V which would be 2.5V. This is the idea of PWM; to produce an analog voltage by outputting a stream of digital 1‘s and 0‘s in a particular proportion. The proportion of 1‘s to 0‘s in PWM is called the duty cycle. The duty cycle controls the analog voltage in a very direct way; the higher the duty cycle the higher the voltage.

The duty cycle can range from 0 (0%) to 255 (100%). Duty is literally the proportion of 1s to 0s output by the PWM instruction. In order to convert PWM into an analog voltage we have to filter out the pulses and store the average voltage. A resistor and capacitor combination is typically used to do this shown in Figure 1-1.

Schematic

Figure 1-1



The capacitor will hold the voltage set by PWM after the instruction has completed. How long it holds the voltage will depend on how much current is drawn, by external circuitry, and internal leakage of the capacitor. In order to hold the voltage the program must periodically repeat the PWM instruction to give the capacitor a refresh charge. The PWM command lets you set the charging time in terms of PWM cycles.

After outputting PWM pulses, the pin is left in input mode. When a pin is set to input mode, the pin's driver is effectively disconnected. If it was not, the steady output state of the pin would change the voltage on the capacitor and undo the analog voltage.

Example

The PWM function is dependent on the oscillator used. The PICmicro can generate a clock about 1ms in length. The specified pin will be made an output prior to the pulse generation and will revert to an input once the pulse stops.

```
    PWM B0,127,1000
        ; 50% duty cycle PWM signal on B0 for 1 second
```

The above code snippet would give and output voltage of about 2.5 volts for about 1 second. You can put the PWM in a subroutine and call it when ever it is needed.

Important Notes

The term PWM or Pulse Width Modulation is loosely applied to the action of the PWM command. PWM is mostly done by splitting a fixed period of time into an on time (1) and an off time (0). Predictability you can calculate the exact frequency of the pulses and their widths are then controlled by the duty cycle. The PWM command does not work in this way. It outputs a rapid sequence of on (1) / off (0) pulses in which the overall proportion over the course of a full PWM cycle of approximately a millisecond is equal to the duty cycle. The advantage here is to quickly zero in on the desired output voltage. In doing so it can not produce the clean, orderly pulses that might be expected. This method is also used to generate pseudo-sine wave tones with the DTMFOUT and FREQOUT commands.

Random

Variable = RANDOM expression (Seed Value)
Generate a pseudo random number. Random is a Math function.

> **Variable** is a variable that will store the results of the random command. This can be a Byte or Word sized variable depending on the expected results.
>
> **Expression** is any combination of variables, constants, math and logic operators.

Explanation

RANDOM generates pseudorandom numbers ranging from 0 to 65535. The term pseudorandom is used because the RANDOM function will generate numbers that appear random, but are actually generated by a logic operation that uses the initial seed value to “tap” into a sequence of 65535 numbers. If the same “seed” value is used repeatedly, then the same sequence of numbers is generated.

Example

```
Temp var word

Main
    Temp = Random 25        ;25 is the seed value
    Debug [DEC Temp]        ;displays the results of temp
    pause 200               ;delay to see the values displayed
    Goto Main               ;repeat loop forever
```

The above program will generate pseudorandom numbers. Since the same seed value is always used the numbers will have a repeating pattern. To generate better pseudorandom numbers you would want to change the “seed” value on the fly:

```
Temp var word
Temp = 25                   ;start value

Main
    Temp = Random Temp      ;seed value is now based on results
    Debug [DEC Temp,13]     ;displays the results of temp
    pause 200               ;delay to see the values displayed
    Goto Main               ;repeat loop forever
```

The “seed” value is now changing each time through the loop. This will eventually produce repeating numbers if left running long enough. The method shown will produce the best random numbers possible.

RCtime

RCTIME pin, state, {TimeoutLabel,TimeoutMultiple,}, resultVariable
Count time while pin remains in state—usually to measure the charge or discharge time of a resistor and capacitor circuit. (RC)

> **Pin** is a variable or constant that specifies the I/O pin to use. This pin will be placed into input mode and left in that state when the instruction finishes.
>
> **State** is a variable or constant (1 or 0) that will end the RCTIME period.
>
> **TimeoutLabel** is an optional label that specifies where to go if a time out occurs. The default time out value is 65,535 microseconds.
>
> **TimeoutMultiple** is a variable or constant that specifies the amount of time to wait before timing out. The time out value is multiplied by 65,535 microseconds which is the default value. If a value of 10 is used the command would wait 655,350 microseconds.
>
> **ResultVariable** is a variable in which the time measurement (0 to 65535 in µs units) will be stored.

Explanation

RCTIME can be used to measure the charge / discharge time of a resistor and capacitor circuit (RC). RCTIME can also be used as a fast stopwatch for recording events of very short duration. This allows measuring resistance or capacitance using R or C sensors (i.e. thermistors or capacitive humidity sensors); or respond to user input through a potentiometer. (Typically 5k to 50k pot.) The resolution of RCTIME is in 2µs increments.

How it works

When RCTIME starts to execute, it will start a counter. It stops this counter as soon as the specified pin is no longer in the start State (0 or 1). If pin remains in a certain state longer than 65535 timing cycles, RCTIME returns 0.

In the example below if the pin never changes State a value of 0 is returned:

```
    LOW B0                      ;Start discharge cap
    PAUSE 10                    ;Discharge for 10ms
    RCTIME B0, 1,Cat            ;Read potentiometer on B0
```

Figure 1-1 shows a suitable RC circuit for use with the RCTIME command.

The circuit shown as State 1 is preferred. This is because the logic threshold is approximately 1.5 volts. Which means that the voltage seen by a pin will start at 5V then fall to 1.5V before RCTIME stops. With the circuit shown for State 0, the voltage will start at 0V and rise to 1.5V, before RCTIME stops. There is more of a voltage variation in the circuit shown for STATE 1 therefore giving a better resolution for the same basic circuit configuration.

Before RCTIME begins to execute, the capacitor will need to be put into the state specified in the RCTIME command. A capacitor is charged when there is a voltage difference between its two plates. When both plates are at +5Volts, the cap is considered to be discharged. This makes the input high.

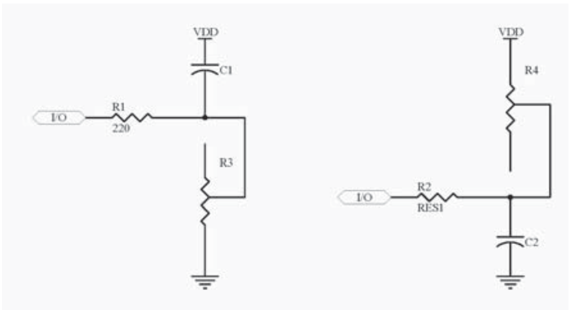
Example

The following program shows the standard use of the RCTIME command measuring an RC charge / discharge time. Use the circuit shown for RCTIME State 1. The appropriate values should be R3 = 10 k pot. and C1 = 0.1 µf. Connect the circuit to Port B Pin 0 and run the program. Follow the instruction for using DEBUG. Adjust the pot and watch the value shown in the Watch Window change.

```
Temp var Word                 ;Temp will hold the result

Main
    High B0                   ;Discharging the cap
    Pause 1                   ;1 ms pause
    RCTIME B0, 1, Temp        ;Measuring RC charge time
    DEBUG [DEC Temp,13]       ;Display value
Goto Main
```

Schematic

Figure 1-1



For use with state 1 For use with state 0

Read

READ location, variable
Read EEPROM location and store value in variable.

> **Location** is a variable or constant that specifies the EEPROM address to read from.
>
> **Variable** holds the byte value read from the EEPROM.

Explanation

The READ statement will allow a program to read the on-chip EEPROM at the specified address and store the result in the given variable.

```
    READ 0,Cat
    ;Read the value at EEPROM location 0 and store it in variable Cat
```

Example

The following method shows how the DATA command can be used to write data to the internal EEPROM and the READ command to read it back.

```
DOG     var byte
CAT     var byte

DATA "Hello","World"
    ;Set the initial value stored in eeprom at programming time

Main
    FOR CAT = 0 to 10 STEP 1  ;increments address 0 to 10
    READ  CAT,DOG      ;reads current address, stores result in DOG
    Debug [HEX dog," ",dog,13]   ;Display results
NEXT
```

The above program will write the ASCII values of Hello World to the internal EEPROM starting at address 0. Since “Hello World” will take 10 address locations, a “for...next“ loop can be used to easily retrieve the data from the internal EEPROM. The READ command is then used to load the values into the variable DOG. As each value is loaded into DOG it is then sent out using the DEBUG command.

SEE ALSO

WRITE
DATA

ReadDM

readdm address,[{mod}var,...,{mod}varN]
Read EEPROM location and store value in variable or variables.

> **Address** is a value pointing to a location in the eeprom from which to start.
>
> **Mod** is any appropriate input or output modifier
>
> **Var** is any variable type

Explanation

The ReadDM is similar to the Read command. The ReadDM statement is setup for sequential reads at a starting point from the internal eeprom. These reads will start at a specified address and store the result in a given variable or variables, such as an array.

```
    ReadDM 0,[str string\11]
```

The above example will begin reading at address 0 from the internal eeprom. Since we specified the STR modifier and 11 bytes, the ReadDM command will continue to read from address 0 to 10 and store the results in each location of the array.

Example

The following program illustrates the usefulness of the ReadDM and WriteDM command. We begin with an Array to store our returned data.

```
string var byte(12)
temp var byte

    main
        writedm 0,["Hello world"]
        readdm 0,[str string\11]
mainlp
    for temp = 0 to 11
    Debug [string(temp),13]
    Next
goto mainlp
```

The above program will write the entire string “Hello World” to the internal eeprom. Since the string “Hello World” consist of 11 bytes the first 11 bytes of the eeprom will be written to. Then the ReadDM command is used to read the 11 bytes back. By using the STR modifier we tell the ReadDM command we want to read 11 bytes from the internal eeprom and store them in our 12 byte variable array. Next we use the SEROUT command to display the results to a terminal window.

Important Notes

There are only 256 internal eeprom locations on a PIC16F876. If the ReadDM or WriteDM command begins in a valid location and continues outside of the 256 locations it will wrap around to the beginning location.

SEE ALSO

WRITEDM
STR Modifier

ReadPM

Readpm address,[{mod}var,...,{mod}varN]
Read Flash Program memory location and store value in variable or variables. Requires 16F87x part.

> **Address** is a value pointing to a location in the flash program memory from which to start. Requires 16F87x part.
>
> **Mod** is any appropriate input or output modifier
>
> **Var** is any variable type

Explanation

The ReadPM command is similar to the ReadDM command except it reads from the internal program memory of a 16F87x PICmicro. The ReadPM statement is setup for sequential reads at a starting point from the internal program memory. These reads will start at a specified address and store the result in a given variable or variables, such as an array.

```
    ReadPM 0x400,[str string\11]
```

The above example will begin reading at address 400h from the internal program memory. Since we specified the STR modifier and 11 bytes, the ReadPM command will continue to read from address 400h to 410h and store the results in each location of the array.

Example

The following program will write “Hello World” using the WritePM command starting at address 400h. Then the ReadPM command is used to read the written values to program memory and print the result to the debug window.

We begin with an Array to store our returned data.

```
string var byte(12)
temp var byte

    main
        writepm 0x400,["Hello world"]
        readpm 0x400,[str string\11]
mainlp
    for temp = 0 to 11
    Debug [string(temp)]
    Next
    Debug [13]
goto mainlp
```

The example program will write the entire string “Hello World” to the internal program memory. Since the string “Hello World” consist of 11 characters, 11 bytes will be written to the internal program memory starting at location 400h. If you run the program once and read back the programmed hex file from the PICmicro at location 400h will be the values for “Hello World”.

By using the STR modifier we tell the ReadPM command we want to read 11 bytes from the internal eeprom and store them in our 12 byte variable array. Next we use the Debug command to display the results to the debug window.

Important Notes

The READPM and WRITEPM commands will only work with 16F87x parts. You can not write or read actual program space used for the running program. You can write and read from address locations that do not contain program data. If the ReadPM command is used to read locations of actual running program space, erratic results will occur.

SEE ALSO

WRITEPM
STR Modifier

Repeat...Until

REPEAT......UNTIL expression

> **Expression** is any combination of variables, constants, math and logic operators

Explanation

Repeat a group of commands until some expression is true. True being any other value than 0.

Example

In the following example, the instructions between the REPEAT...UNTIL commands will execute until the variable Temp is equal to 100. With each run through the loop UNTIL will check the status of Temp.

```
Temp var byte
clear

Main
REPEAT                          ; repeat the following commands.
        temp = temp + 1
        Debug [Dec Temp,13]
    UNTIL temp  = 25            ; repeating until temp is equal to 25
End
```

Since we are looking for something to be TRUE at the UNTIL statement we will need to use an expression. The next example shows how to check the status of a pin:

```
Temp var Byte
Clear

    Input B0                    ;sets B0 to an input
Main
REPEAT                          ; repeat the following commands.
    Debug [“False”]
    UNTIL PortB.bit0 = 1        ; repeating until B0 is High (+5Volts)
    Debug [“True”]
End
```

Reverse

REVERSE pin
Reverse the data direction of the specified pin.

> **Pin** is a variable or constant that specifies the I/O pin to use. This pin will be placed into the opposite of its current input/output (I/O) mode.

Explanation

Reverse is a convenient way to switch the I/O direction of a pin. If a pin is set as an input, the REVERSE command, will change it to an output; REVERSE is an easy way to use Bi-State devices like Bi-Color LEDs or two create a Bi-State.

Example

```
OUTPUT B0                 ; Makes P0 an output
REVERSE  B0               ;Changes P0 to an input
```

A good use for REVERSE would be to quickly change a pin state to an input to effectively “remove it from the circuit”.

The term “input” really has two meanings: (1) Setting a pin to input making it possible to detect the state of that pin. (2) Setting a pin to input also disconnects the output driver, which would have little effect on external circuitry.

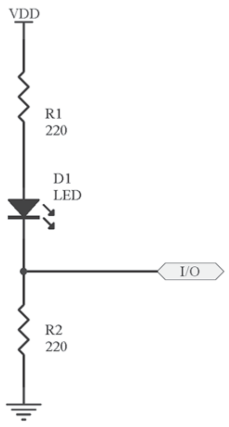
The following program with the circuit shown in figure 1-1 illustrates this second effect. The LED will have two levels of brightness:

```
Low B0              ;P0 is low, sets output driver.

Main
    Pause 300       ;300ms pause
    REVERSE  B0     ;Reverse B0 I/O direction to input
Goto Main           ;Do forever
```

The above program sets B0 as an output by telling the program that B0 is set to Low (0 Volts). This will cause the circuit shown in figure 1-1 to illuminate brightly. By executing the REVERSE command, B0 is now an input and will effect the circuit causing the LED to illuminate dimly. This is because the total resistance 220 + 220 = 440 ohms when B0 is an input. When B0 is set as an output and low, this bypasses the second resistor to ground and the ground is now coming from B0, so the total resistance is now 220 ohms.

Schematic

Figure 1-1



Serdetect

Serdetect pin,mode,var
Detect incoming baud rate. Used for auto detecting baud rates.

Pin is a variable or constant that specifies the I/O pin that will be used to receive the sync character.

Mode is the settings for Bits 13,14 and 15. Bit 13 ($2000 hex) is a flag that controls the number of data bits and parity (0=8 bits and no parity, 1=7 bits and even parity). Bit 14 ($4000 hex) controls polarity (0=noninverted, 1=inverted). Bit 15 ($8000 hex) is not used by SERIN. Constants from the below table can be used for the Mode:

IMODE	= Inverted
NMODE	= Non Inverted
IEMODE	= Inverted, Even Parity
NEMODE	= Non Inverted, Even Parity
IOMODE	= Inverted, Open Drain
NOMODE	= Non Inverted, Open Drain
IEOMODE	= Inverted, Even Parity, Open Drain
NEOMODE	= Non Inverted, Even Parity, Open Drain

Var is a word sized variable that will hold the calculated baudmode value which can be used by serin and serout.

Explanation

Serdect is used to auto detect an incoming baud rate. This is ideal for applications or products that can be used at multiple baud rates and be software switched. Serdetect can take the place of hard wired jumpers or switches for changing baud rates.

In order for serdetect to calculate the bitrate a character must be received. For inverted mode the binary value of the character to send for calculating bitrate would be %01XXXXXX. This is because Serdetect calculates one bit width. For inverted modes Seretect would calculate the first bit which as the start bit, thus the need for %01XXXXXX. For non inverted modes the first character bit would be used for calculation %101XXXXX. The values in Hex for inverted mode would be $55 and a value for non inverted would be $AA.

The serdetect command counts the period between the first 1's bit and the next 1's bit. The time is than converted to the bit rate and IOR'd with the mode. In the case of inverted mode, the first 1's bit is the start bit of the character and the second 1's bit is the second LSB of the character. In the case of non inverted mode the first 1's bit is the LSB of the Character and the second 1's bit is the 3rd LSB of the character.

Example

```
SerBaud Var Word    ;Word sized variable for Serdetect results

Main
      SerDetect B0, IMODE, SerBaud
                        ;waits until a character is received $55

      Serout B1, SerBaud, [“Locked in”]
                        ;Transmit using the results from Serdetect
Goto Main
```

The program above will wait until it receives a sync byte. Once serdetect has received the sync byte with the results it will send back out “Locked In” using the baud mode and rate calculated by serdetect.

The next program demonstrates the Serdetect, Serin, and Serout commands. Once the program is loaded to the PICmicro and connected to your PC. Use one of the IDE terminal windows and set the baud to 2400 or 9600. Any baud up to 57600 can be used. To start the progam send a upper case “AA” or “55”

Then program will then build a menu which will allow you select options.

```
      bval var word                   'Used to hold the auto detect baud rate.
      name var byte(30)
      cmd var byte

      clear

      dodetect:
            serdetect B0,IMODE,bval           'Lets wait for sync byte
            serout B1,bval, ["We have your baud rate. ",10,13]

'--------------------------------------------------------------------
'Display the menu
'--------------------------------------------------------------------
      domenu:
            'Menu system start here.
            serout B1,bval,[10,13]
            serout B1,bval,["1: Put PICmicro to Sleep",10,13]
            serout B1,bval,["2: Detect Baud Rate",10,13]
            serout B1,bval,["3: Enter Name",10,13]
            serout B1,bval,["4: Display Name",10,13]
      serout B1,bval,["5: Shut the PICmicro down",10,13]
      serout B1,bval,[10,13]
      serout B1,bval,["Option "]
```

```
          'Get the menu item here
     serin B0,bval,[dec1 cmd]

     branch cmd,[bad,dosleep,dodetect,getname,dispname,doend]
          'Falls through or jumps here on 0

     bad:
          serout B1,bval,[10,13,"Invalid Command",10,13]
     goto domenu

     dosleep:
          serout B1,bval,[10,13,"Night Night..",10,13]
          sleep 10                    'Put the PICmicro to sleep for 10 seconds.
     goto domenu

     doend:
          serout B1,bval,[10,13,"Night Night..",10,13]
     end                              'Shut the PICmicro down (sleep forever)

'-----------------------------------------------------------------
'Get the users name
'-----------------------------------------------------------------
     getname:
          'Ask for name
          serout B1,bval,["What is your name? "]

          'Read in users name here.  Maximum of 30 characters or CR
          serin B0,bval,[str name\30\13]

          'Display name until 30 characteres or end of data
          serout B1,bval,["Hello ",str name \30\0]
          serout B1,bval,[10,13]
     goto domenu

'-------------------------------------------------------------
'Display the users name
'-------------------------------------------------------------
     dispname:
          'Display name
          serout B1,bval,["Hello ",str name \30\0]
          serout B1,bval,[10,13]
     goto domenu
```

Serin

SERIN rpin{\fpin},baudmode,{plabel,}{timeout,tlabel,}[inputData]
Receive asynchronous (e.g., RS-232) data.

Rpin is a variable or constant that specifies the I/O pin through which the serial data will be received.

Fpin is an optional variable or constant that specifies the I/O pin to be used for flow control. This pin will switch to output mode and remain in that state after the end of the instruction. (Caution Fpin requires the 'slash')

Baud mode is a 16-bit variable or constant that specifies serial timing and configuration. The lower 13 bits are interpreted as the bit period. Bit 13 ($2000 hex) is a flag that controls the number of data bits and parity (0=8 bits and no parity, 1=7 bits and even parity). Bit 14 ($4000 hex) controls polarity (0=noninverted, 1=inverted). Bit 15 ($8000 hex) is not used by SERIN.

Plabel is an optional label indicating where the program should go in the event of a parity error. This argument may only be provided if baud mode indicates 7 bits, and even parity, otherwise the label is ignored.

Timeout is an optional variable/constant (0–65535) that tells SERIN how long, in milliseconds, to wait for incoming data. If data does not arrive in time, the program will jump to the address specified by Tlabel.

Tlabel is an optional label which must be provided along with Timeout, indicating where the program should go in the event that data does not arrive within the period specified by Timeout.

InputData is a list of variables and modifiers that tells SERIN what to do with incoming data. SERIN can store data in a variable or array; interpret numeric text (decimal, binary, or hex), and store the corresponding value in a variable; wait for a fixed or variable sequence of bytes; or ignore a specified number of bytes. These actions can be combined in any order in the inputData list.

SERIN Modes

Inverted?	*Parity?*	*Baud Rate*	*Constant*
No	No	300	N300
Yes	No	300	I300
No	Yes	300	NE300
Yes	Yes	300	IE300
No	No	1200	N1200
Yes	No	1200	I1200
No	Yes	1200	NE1200
Yes	Yes	1200	IE1200
No	No	...	*N...
Yes	No	...	*I...
No	Yes	...	*NE...
Yes	Yes	...	*IE...
No	No	57600	N57600
Yes	No	57600	I57600
No	Yes	57600	NE57600
Yes	Yes	57600	IE57600

* 1200, 2400, 4800, 9600, 14400, 19200, 28800, 33600, 38400, 57600

Improtant Note

There are several modifiers for use with the SERIN / SEROUT commands. Refer to Command Modifier section of this manual.

Explanation

One of the most used forms of communication between electronic devices is serial communication. The two most used types of serial communication are asynchronous and synchronous. The SERIN and SEROUT commands use an asynchronous method to receive and send serial data. The term asynchronous means “no clock.” Data is transmitted and received without the use of a separate “clock” wire. The PC’s serial ports (COM ports, RS-232) use asynchronous serial communication.

Important Notes

Normal logic, 5 volts is a logic 1 and 0 volts is logic 0, RS-232 uses -12 volts for logic 1 and +12 volts for logic 0. This allows communication over long wire lengths without amplification. Most circuits for RS-232 use a line driver / receiver (MAX232). The MAX232, (1) converts the ±12 volt swing of RS-232 to TTL-compatible logic levels (0 to +5 Volts). It then inverts the logic, so 5 volts = logic 1 and 0 volts = logic 0.

Making the Connection

There are a few ways to connect a PICmicro to a serial port of a computer or any other RS-232 port. (1) Using a MAX 232 as shown in figure 1-1. (2) Using a 22k current limiting resistor as shown in figure 1-2, which would be non-inverted (N9600). The best method would be using the MAX232. Using the 22K resistor method only protects the PICmicro. Some PC’s / RS-232 ports will not correctly receive data and require a ±12 volt swing (Refer to SEROUT). When receiving data through a MAX232 circuit you will need to invert the data using the “i” modifiers with the baudrate:

```
SERIN B0, i9600, [temp]          ;receive byte inverted
```

Schematic

Figure 1-1

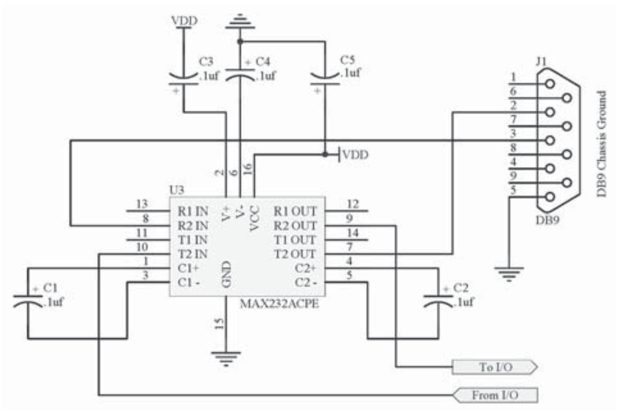
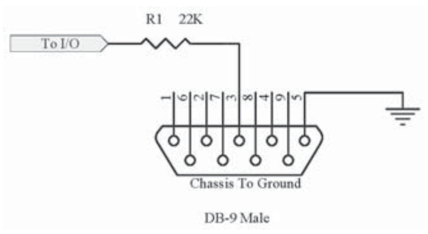


Figure 1-2



Function:	DB-9 Pin Number:
Data Carrier Detect (DCD)	1
Receive Data (RD)	2
Transmit Data (TD)	3
Data Terminal Ready (DTR)	4
Signal Ground (SG)	5
Data Set Ready (DSR)	6
Request to Send (RTS)	7
Clear to Send (CTS)	8

Some extra connections that may need to be made are DTR-DSR-DCD tied together and RTS-CTS tied together. This is only necessary if you are using software or hardware that expects hardware handshaking.

Important Notes

Asynchronous serial communication is based on precise timing. Sender and receiver must be set for identical timing, expressed in bits per second (bps), referred to as the baud rate. The SERIN command requires a few values (Pin, Mode and Baud) that tell it the speed of the incoming serial data, the bit period, number of data and parity bits, and polarity. SERIN is able to listen to data from 300 to 115k. The maximum recommend for 100% reliable transmission is 57600 Baud.

Example

```
hello       VAR     byte
string      VAR     byte(10)
goodbye     VAR     byte

SERIN B0,N9600,[hello]
    ;one byte at 9600 baud and store in variable hello

SERIN B0\B1,N4800,[Sbin hello]
    ;one Signed byte from serial at 4800 baud

SERIN B0,E14400,parityerror,[hello,STR string\9,IShex goodbye]
    ;receive one byte from serial at 14400 with parity.
```

```
          ;Receive one string of 9 characters
          ;long and receive one Signed indicated (i.e.: $)
          ;hexadecimal value and store in goodbye

parityerror:
          ;if parity error occurs in the serin command jump to this label

          Debug [“error”]
Goto parityerror
```

The previous program shows different ways the SERIN command can be used.

Serin Modifiers

The SERIN command has several modifiers for formatting incoming data and adding functionality to the SERIN command (Refer to Command Modifiers). One of the SERIN modifiers can be configured to wait for specified data before it retrieves any additional data. For example, a device attached to the PICMicro is known to send many different sequences of data, but the only data you require appears right after some unique characters such as “A12”. You can modify the SERIN command to “Wait” for these characters to be received before loading any values into a variable as shown:

```
Temp var Byte

SERIN B0, i9600, [WAIT(“A12”), temp]
```

The above program will wait until “A”, “1”, “2” is received in the order shown, then load the next value received value into the variable “Temp”. The data received by SERIN is case sensitive. If “a” was received first it would have been ignored. Also SERIN is looking at the ASCII values of “A”, “1”, “2”.

Once SERIN is executed the program halts until the data expected is received. If no serial data is received the program will halt indefinitely. In order to limit the amount of time SERIN waits, you can use the Timeout option as shown:

```
SERIN B0, i9600, 1000, Timeerror, [WAIT(“A12”), temp]

Timeerror:
          Debug [“Error”]           ;display message in watch window
Goto Timeerror                      ;loop forever
```

The above program will only wait 1000ms to receive the expected incoming data until it will time out and jump to the specified label “Timeerror”.

Important Notes

Some limitations of SERIN are sending or receiving data, the program can’t execute other instructions. When executing other instructions, the program can’t send or receive data.

There is no serial buffer as on a PC. So watching data follow is important. One way to address this is to use flow control. This will set an additional pin on the Atom for flow control. Using the circuit in figure 1-1 ties this additional pin to Clear to Send (CTS), which is pin 8 on the DB-9 Male connector. This will prevent the PC from sending data until SERIN is ready.

```
Low B1
SERIN B0\B1, i9600,[WAIT(“A12”), temp]

Done:
      Debug [“Got it”]
Goto Done
```

The above program will set the pin B1, when the SERIN command is executed. This tells the PC “OK I’m ready”.

SEE ALSO

SEROUT

Serout

SEROUT tpin,baudmode,{pace,}[outputData]
SEROUT tpin\fpin,baudmode,{timeout,tlabel,}[outputData]
Transmit asynchronous (e.g., RS-232) data.

Tpin is a variable or constant that specifies the I/O pin through which the serial data will be sent.

Baudmode is a 16-bit variable or constant that specifies serial timing and configuration. The lower 13 bits are interpreted as the bit period. Bit 13 ($2000 hex) is a flag that controls the number of data bits and parity (0=8 bits and no parity, 1=7 bits and even parity). Bit 14 ($4000 hex) controls the bit polarity (0=noninverted, 1=inverted). Bit 15 ($8000 hex) determines whether the pin is driven to both states (0/1) or to one state and open in the other (0=both driven, 1=open).

Pace is an optional variable or constant (0–65535) that tells SEROUT how long in milliseconds it should pause between transmitting bytes.

OutputData is a list of variables, constants and modifiers that tells SEROUT how to format outgoing data. SEROUT can transmit individual or repeating bytes; convert values into decimal, hex or binary text representations; or transmit strings of bytes from variable arrays.

Fpin is an optional variable or constant that specifies the I/O pin to be used for flow control (byte-by-byte handshaking). This pin will switch to input mode and remain in that state after the instruction is completed. (Caution Fpin requires the 'slash')

Timeout is an optional variable/constant (0–65535) used in conjunction with fpin flow control. Timeout tells Serout how long in milliseconds to wait for fpin permission to send. If permission does not arrive in time, the program will continue at tlabel.

Tlabel is an optional label used with Fpin flow control and timeout. Tlabel indicates where the program should go in the event that permission to transmit data is not granted within the period specified by the Timeout command.

SEROUT Modes

Driven?	*Inverted?*	*Parity?*	*Baud Rate*	*Constant*
Yes	No	No	300	N300
Yes	Yes	No	300	I300
Yes	No	Yes	300	NE300
Yes	Yes	Yes	300	IE300
No	No	No	300	NO300
No	Yes	No	300	IO300
No	No	Yes	300	NEO300
No	Yes	Yes	300	IEO300
Yes	No	No	...	N...
Yes	Yes	No	...	I...
Yes	No	Yes	...	NE...
Yes	Yes	Yes	...	IE...
No	No	No	...	NO...
No	Yes	No	...	IO...
No	No	Yes	...	NEO...
No	Yes	Yes	...	IEO...

* 1200, 2400, 4800, 9600, 14400, 19200, 28800, 33600, 38400, 57600

Improtant Note

There are several modifiers for use with the SERIN / SEROUT commands. Refer to Command Modifier section of this manual.

Table 2-2 lists the predefined Baudmode constants available in MBasic. As you can see from the table there are several different baudmodes for each actual baud rate. The following describes each baudmode modifier:

N	Normal (not inverted) signal
I	Inverted signal
E	Even Parity(otherwise no parity)
O	Open drain(otherwise both high and low are driven)

Table 2-3 lists the command modifiers for the Output data

Explanation

One of the most used forms of communication between electronic devices is serial communication. The two types of serial communication are asynchronous and synchronous. The SERIN and SEROUT commands use an asynchronous method to receive and send serial data. The term asynchronous means "no clock." Data is transmitted and received without the use of a separate "clock" wire. The PC's serial ports (COM ports, RS-232) use asynchronous serial communication.

Important Notes

Normal logic, 5 volts is a logic 1 and 0 volts is logic 0, RS-232 uses -12 volts for logic 1 and +12 volts for logic 0. This allows communication over long wire lengths without amplification. Most circuits for RS-232 use a line driver / receiver (MAX232). The MAX232, (1) converts the ±12 volt swing of RS-232 to TTL-compatible logic levels (0 to +5 Volts). It then inverts the logic, so 5 volts = logic 1 and 0 volts = logic 0.

Making the Connection

There are a few ways to connect a PICMicro to a serial port of a computer or any other RS-232 port. (1) Using a MAX 232 as shown in figure 1-1. (2) Using a direct connection as shown in figure 1-2 which is non-inverted (N9600).The best method would be using the MAX232. Some PC's require a ±12 volt swing, so the direct connection may not work (Refer to SERIN). When transmitting data through a MAX232 circuit you will need to invert the data using the "i" modifiers with the baudrate:

```
SEROUT B0, i9600, ["Test"]          ;transmit "Test" inverted
```

Schematic

Figure 1-1

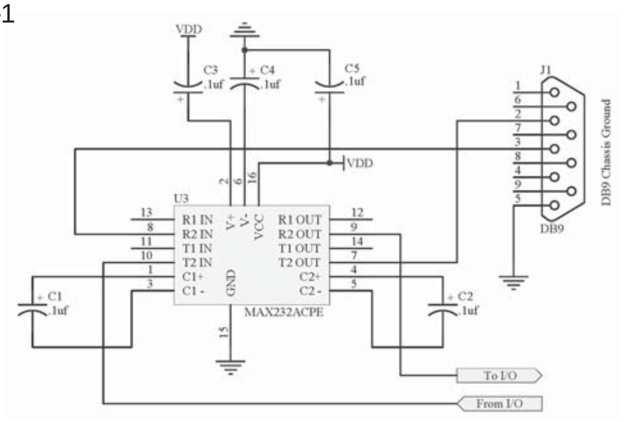
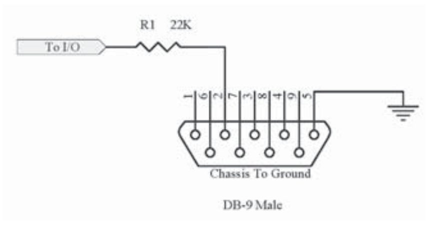


Figure 1-2



Function:	DB-9 Pin Number:
Data Carrier Detect (DCD)	1
Receive Data (RD)	2
Transmit Data (TD)	3
Data Terminal Ready (DTR)	4
Signal Ground (SG)	5
Data Set Ready (DSR)	6
Request to Send (RTS)	7
Clear to Send (CTS)	8

Some extra connections that may need to be made are DTR-DSR-DCD tied together and RTS-CTS tied together. This is only necessary if you are using software or hardware that expects hardware handshaking.

Important Notes

Asynchronous serial communication is based on precise timing. Sender and receiver must be set for identical timing, expressed in bits per second (bps), referred to as the baudrate. The SEROUT command requires a few values (Pin, Mode and Baud) that tell it the speed to transmit the serial data, the bit period, number of data and parity bits, and polarity. SEROUT is able to send data from 300 to 115k. The maximum recommend for 100% reliable transmission is 57600 Baud.

Example

```
hello VAR byte
string VAR byte(10)
goodbye VAR byte

SEROUT B0,i9600,[“Hello”]
      ;send “hello”, through the serial port at 9600 baud inverted
      ;for use with a MAX232 or SIN / SOUT

SEROUT B0,N4800,[Sbin hello]
      ;send one Signed byte from hello, through serial at 4800baud
      ;non-inverted, for use with a direct connection

SEROUT B0,IE14400,100,[hello,STR string\9,IShex goodbye]
                  ;send one byte from serial at 14400 with parity.
                  ;Send string of bytes 9 characters
                  ;long and one Signed indicated (i.e.: $)
                  ;hexadecimal value from goodbye
                  ;Pace the bytes 100ms apart
                  ;inverted, for use with a MAX232
```

The above code snippets illustrates different ways SEROUT can be modified using modifiers.

Serout Modifiers

The SEROUT command can use several formatting modifiers (Refer to Command Modifiers). An example would be the decimal modifier (DEC), which will translate the value of “8” and “2” into the ASCII codes for the characters “8” and “2” and then transmit them:

```
SEROUT B0, i9600, [ DEC 82 ]
```

The SEROUT command will send quoted text exactly as it appears in quotes:

```
SEROUT B0, i9600, [ “Result = “, DEC 10]
```

This will display “Result = 10”.

SEROUT can be configured to pause between transmitted bytes, called pacing:

```
    SEROUT B0, i9600, 500, [ “Slow” ]
```

SEROUT will transmit the word “Slow” with a 500ms delay between each character. The Pace feature is used to support devices that require more than one stop bit. SEROUT would normally send data as fast as it can! A stop bit is simply a resting state in the driven line. Using the Pace option will add multiple stop bits. Since some devices may require 2 or more stop bits. The Pace modifier can also be used for devices that can only process one byte at a time, using the Pace modifier would give such a device time to process the byte then return to wait for another byte.

The string (STR) modifier can be useful for transmitting a string of characters from a byte array variable. A string of characters is a set of characters that are arranged in a certain order. The characters “1234” could be stored in a string with the “1”, “2” then “3” and “4”. A byte array is similar to a string in that it contains data that is arranged in a certain order. Each of the elements in an array is the same size. See the “Defining Arrays” section in this manual for more information on arrays. An example that transmits 4 bytes from a byte array is shown below:

```
    TempString var byte(4)        ;create a 4-byte array
    TempString(0) = “T”           ;set the piece 1 of the array
    TempString(1) = “E”           ;set the piece 2 of the array
    TempString(2) = “S”           ;set the piece 3 of the array
    TempString(3) = “T”           ;set the piece 4 of the array


SEROUT B0, i9600, [ STR TempString\4 ]
                                  ;send 4-byte string
```

The use of the optional \n argument is to specify how many characters are to be sent. Otherwise it would continue until it found a byte equal to 0. Since the last byte of 0 in the array is not 0 it would go on forever. Another useful modifier is Repeat (Rep\n):

```
    SEROUT B0, i9600, [ Rep Temp\10 ]
                                  ;send temp ten times
```

The repeat modifier will repeatedly send a byte value \n times. The number of times its repeated is specified by \n.

SEE ALSO

SERIN

Servo

SERVO pin, rotation{, repeat}

> **Pin** is the pin controlling the servo
>
> **Rotation** is a variable / constant that specifies the position you want the servo to rotate. A value from -1200 to + 1200 is used with 0 being center. The value -1200 being a rotation to the farthest position in a direction and +1200 being the farthest rotation in the opposite direction. The maximum +1200 and minimum -1200 will vary based on the servo being used. Take caution not to exceed these values.
>
> **Repeat** (optional)Specifies the number of internal cycles the command runs(defaults to 30).

Explanation

Various servo manufacturers use different color coding for their servos. Below is a color cross reference chart for some popular servos.

Manufacturer	*+5 Volts*	*GND (Vss)*	I / O
Airtronics	Red	Black	Brown
Futaba J	Red	Black	White
KO Propo	Red	Black	Blue
Kyosho/Pulsar	Red	Black	Yellow
Japan Radio(JR)	Red	Brown	Orange

The SERVO command works by simply pulsing the I/0 pin used very similar to PWM. By adjusting the width and duration you can control the SERVO clockwise, counter clockwise and speed. The SERVO command automatically handles the calculations for you.

Example

```
Main
      Servo B0, -1200 ;rotate servo negative
      Pause 200       ;give servo time to complete rotation
      Servo B0, 1200  ;rotate servo positive
Goto Main                         ;repeat forever.
```

In the example the Servo will rotate all the way to one direction then all the way to the opposite direction. There is no exact scale for what value will move the servo to what position. This is due to the fact that servo motors may vary. Use the program below with your servo connected to determine which value corresponds to what position:

```
Temp var Word
      Temp = 0

      Main
            temp = temp +10
            Servo B0, temp              ;rotate servo positive
            Debug [DEC Temp, 13]
            Pause 200                   ;time to complete motion
      Next
      Goto Main                         ;repeat forever.
```

The above example will print the value of Temp to the debug watch window. To work out the opposite direction change the values of temp to negative numbers.

Shiftin

SHIFTIN Dpin,Cpin,Mode,[result{\bits}{,result{\bits}...}]
Shift data in from a synchronous-serial device.

Dpin is a variable or constant that specifies the I/O pin that will be connected to the synchronous-serial device's data output. This pin's I/O direction will be changed to input and will remain in that state after the instruction is completed.

Cpin is a variable or constant that specifies the I/O pin that will be connected to the synchronous-serial device's clock input. This pin's I/O direction will be changed to output.

Mode is a value (0—3) or 4 predefined symbols that tells SHIFTIN the order in which data bits are to be arranged and the relationship of clock pulses to valid data. Here are the symbols, values, and their meanings:

MSBPRE	**0**	Data msb-first; sample bits before clock pulse
LSBPRE	**1**	Data lsb-first; sample bits before clock pulse
MSBPOST	**2**	Data msb-first; sample bits after clock pulse
LSBPOST	**3**	Data lsb-first; sample bits after clock pulse

(Msb is most-significant bit; the highest or left most bit of a nibble, byte, or word. Lsb is the least-significant bit; the lowest or right most bit of a nibble, byte, or word.)

Result is a bit, nibble, byte, or word variable in which incoming data bits will be stored.

Bits is an optional entry specifying how many bits (1—16) are to be input by SHIFTIN. If no bits entry is given, SHIFTIN defaults to 16 bits.

Explanation

SHIFTIN provides an easy method of receiving data from synchronous-serial devices. Synchronous serial differs from asynchronous serial (i.e. SERIN and SEROUT) in that the timing of data bits is specified in relationship to pulses on a clock line. Data bits may be valid after the rising or falling edge of the clock line. This kind of serial protocol is commonly used by controller peripherals like ADCs, DACs, clocks, memory devices, etc. Trade names for synchronous-serial protocols include SPI and Microwire.

The SHIFTIN command will first set the clock pin to output and low. The data pin is then switched to input mode. SHIFTIN will now either generate a clock pulse then read the data pin (POSTmode) or read the data pin and generate a clock pulse (PRE mode). SHIFTIN will continue to generate clock pulses and read the data pin for as many data bits required. Using SHIFTIN with a particular device is nothing more than matching the mode and number of bits to the device's protocol. Manufacturers will generally provide a timing diagram to illustrate the clock and data.

The bit argument on SHIFTIN defaults to 16 unlike the BS2. If no bit argument is used 16 bits will be acquired. For example:

```
SHIFTIN B0, B1, LSBPRE,[dog \8]
```

Using the example above if %10001111 was received the first bit would become the least significant bit. MSB and LSB work like reading English. From left to right. When looking at a value MSB starts from the left to LSB ending on the right. So above %1000 is the MSB of our 8 bit value. Once the entire 8 bits have been received in our variable DOG it would read as follows %11110001.

Example

This next example show how to setup and read the time and data from a Dallas 1302.

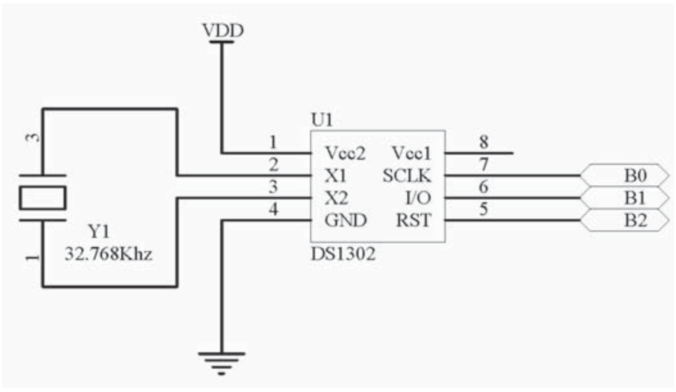
```
RTCCmd        VAR BYTE
Clk           CON B0
Dta           CON B1
Reset         CON B2
Temp          VAR BYTE
Temp1         VAR BYTE
timedata      var byte(6)
oldseconds    var byte
Seconds       VAR timedata(0)
Minutes       VAR timedata(1)
Hours         VAR timedata(2)
Date          VAR timedata(3)
Month         VAR timedata(4)
Year          VAR timedata(5)

;Define registers of the DS1302

SecReg        CON %00000
MinReg        CON %00001
HrsReg        CON %00010
DateReg       CON %00011
MonReg        CON %00100
YrReg         CON %00110
CtrlReg       CON %00111
BrstReg       CON %11111
```

```
Preset bytetable $00,$00,$12,$20,$03,$02

; Clear Write Protect bit in control register

Temp = $10
RTCCmd = CtrlReg
      GOSUB PreSetData

;Preset time registers using a byte table preset

for temp1 = 0 to 5
      Temp = Preset(temp1)
      RTCCmd = temp1
      GOSUB PreSetData
NEXT

Temp = $80
RTCCmd = CtrlReg
      GOSUB PreSetData

; read loop. Only writes to the screen if the seconds have changed
; The “|” is used at the end of a line that is to long to fit on the screen
; the location of the “|” may change depending where the breaks are
; in the command on your screen

Loop:
      GOSUB  ReadData
      if oldseconds <> seconds then
            serout B7,i38400,[0,"Time: ",dec2 (BCD2BIN Hours)|
            \2,":",DEC2 (BCD2BIN Minutes)\2,":",DEC2 (BCD2BIN |
            Seconds)\2,13,"Date: ",DEC2 (BCD2BIN Month)\2,"/",DEC2 |
            (BCD2BIN Date)\2,"/",DEC2 (BCD2BIN Year)\2]
      endif
GOTO  Loop

; The PreSetData loop is used to preset the values from the Preset
; bytetable so the data and time can be set. Change the values in the
; table to be current

PreSetData:
      'Write to DS1302 RTC
      HIGH Reset
      SHIFTOUT Dta, Clk, LSBFIRST, [%0\1,RTCCmd\5,%10\2,Temp\8]
      LOW  Reset
RETURN

ReadData:
      HIGH Reset
```

```
    SHIFTOUT DTA, Clk, LSBFIRST, [%111111\6,%10\2]
    oldseconds = seconds
    SHIFTIN DTA, Clk, LSBPRE,
[Seconds\8,Minutes\8,Hours\8,Date\8,Month\8,Year\8,Year\8]
    LOW Reset
RETURN
```

The above program will read out the time and date, displaying the results in a terminal window. This program can be customized. The Preset bytetable contains all the data the sets the values the DS1302 will start from. Change these values to start with the current data and time. The starts from the left and read Seconds, Minutes, Hours, Date, Month, and Year.

Schematic



SEE ALSO

SHIFTOUT

Shiftout

SHIFTOUT Dpin,Cpin,Mode,[data{\bits}{,data{\bits}...}]
Shift data out to a synchronous-serial device.

Dpin is a variable or constant that specifies the I/O pin that will be connected to the synchronous-serial device's data input. This pin's I/O direction will be changed to output and will remain in that state after the instruction is completed.

Cpin is a variable or constant that specifies the I/O pin that will be connected to the synchronous-serial device's clock input. This pin's I/O direction will be changed to output and will remain in that state after the instruction is completed.

Mode is a value (0 or 1) or a predefined symbol that tells SHIFTOUT the order in which data bits are to be arranged. Here are the symbols, values, and their meanings:

Symbol	Value	Meaning:
MSBPRE	**0**	Data msb-first; sample bits before clock pulse
LSBPRE	**1**	Data lsb-first; sample bits before clock pulse
MSBPOST	**2**	Data msb-first; sample bits after clock pulse
LSBPOST	**3**	Data lsb-first; sample bits after clock pulse

Backwards Compatibility:

LSBFIRST	**1**	Data shifted out lsb-first.
MSBFIRST	**0**	Data shifted out msb-first.

(Msb is most-significant bit; the highest or left most bit of a nibble, byte, or word. Lsb is the least-significant bit; the lowest or right most bit of a nibble, byte, or word.)

Data is a variable or constant containing the data to be sent.

Bits is an optional entry specifying how many bits (1—16) are to be output. If no bits entry is given, SHIFTOUT defaults to 16 bits.

Explanation

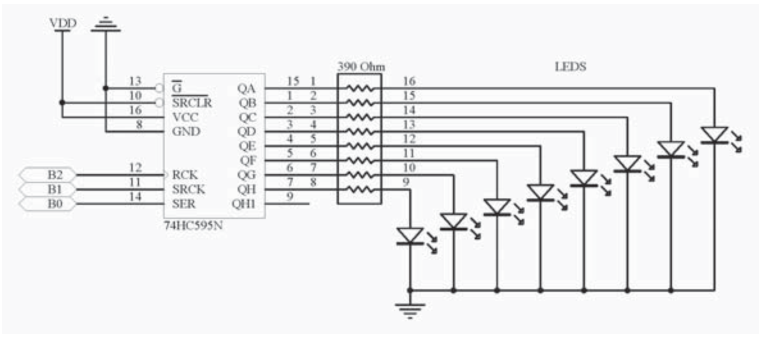
SHIFTOUT provides an easy method of transferring data to synchronous-serial devices. Synchronous serial differs from asynchronous serial (like SERIN and SEROUT) in that the timing of data bits is specified in relationship to pulses on a clock line. Data bits may be valid after the rising or falling edge of the clock line. This kind of serial protocol is commonly used by controller peripherals like ADCs, DACs, clocks, memory devices, etc. Trade names for synchronous-serial protocols include SPI and Microwire.

Example

The below example program will send out the values 0 through 255 to the 74HC595 which is a serial to parallel register. Use the schematic below to make the proper connections.

```
SI          CON   B0          ; serial output
SCK         CON   B1          ; shift clock
RCK         CON   B2          ; output latch
Delay       CON   100
Counter VAR Byte

main
    for counter=0 to 255
        gosub Out595
        pause 100           ;Change to in/decrease speed
    next
goto main


Out595:
    Shiftout SI,SCK,MSBPRE,[counter\8]
                            ; send pattern to 74x595
    Pulsout RCK,8           ; latch outputs(min 8us)
Return
```

Schematic



SEE ALSO

SHIFTIN

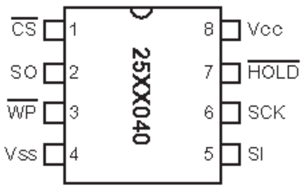
Example of SPI communications

SPI communication is accomplished via the SHIFTIN and SHIFTOUT commands. Following below is a brief tutorial on using SPI devices.

SPI is a serial communication protocol. CS, SCK, SI and SO need to be connected to I/O pins on the PICmicro. In this example we will refer to the Microchip 25C040 shown below.

PIN DESCRIPTION:

CS	- Chip Select Input
SO	- Serial Data Output
WP	- Write Protect Pin
VSS	- Ground
SI	- Serial Data Input
SCK	- Serial Clock Input
HOLD	- Hold Input
VCC	- Supply Voltage



Important Note

In some uses of SHIFTIN / SHIFTOUT may not work correctly with Debug.This is a timing related issue and will be addressed in a later release.

Example

The following example will detail writing and reading data to a 25c040 SPI eeprom.

```
temp var byte
cntr var word
cntr = 0
SCK con B1
SO con B3
SI con B2
CS con B0

low SCK            ;sets chip in a known state
high CS

main
    low CS
    shiftout SI,SCK,MSBPRE,[0x06\8] ; sets 25c040 for a write
    high CS
    low CS
    shiftout SI,SCK,MSBPRE,[0x02\8]
    shiftout SI,SCK,MSBPRE,[cntr\8] ; sends address location for write
shiftout SI,SCK,MSBPRE,[cntr\8]
            ; using the value of address as the value to be writen
high CS
pause 5
low CS
shiftout SI,SCK,MSBPRE,[0x03\8,cntr\8] ; sets 25c040 for a read
shiftin SO,SCK,MSBPOST,[temp\8] ; reads and stores results in temp
high CS
serout c0,i9600,["Hello",dec temp,13]
           ; prints value stored in temp to terminal window
cntr = cntr + 1
goto main
```

Sleep

SLEEP seconds
Put the PICmicro into low-power sleep mode for a specified number of seconds.

> **Seconds** is a variable or constant (1-65535) that specifies the duration of sleep in seconds.

Explanation

SLEEP will place the PICmicro into a low power mode for the specified period of seconds. Period is 16 bits, so delays of up to 65,535 seconds (a little over 18 hours) would be the limit. SLEEP uses the Watchdog Timer so it is independent of the oscillator frequency. Since SLEEP relies on the watchdog timer there is a short wake up period every 1.5 seconds. The I/O pins remain in there last known state.

The SLEEP command will not affect the internal registers, allowing a program to continue executing upon waking up from the SLEEP period.

To achieve the lowest possible current draw during sleep you should set all pins to outputs and low.

Example

In the example below we can easily create a long pause when blinking an LED. Commonly the SLEEP command is used to reserve battery power.

```
Again
    HIGH B0             ; LED on.
    PAUSE 1000          ; Wait 1 second.
    LOW B0              ; LED off.
    SLEEP 60            ; Sleep for 1 minute.
Goto Again
```

The above example will blink an LED then put the PICmicro into a one minute sleep cycle.

Sound

Sound pin,[duration1\note1,...durationN\noteN]
Generate specific note from one pin.

> **Pin** is a variable/constant that specifies the I/O pin to use. This pin will be set to an output during tone generation and left in that state after the instruction is completed.
>
> **Duration** is a variable/constant specifying the length in milliseconds (1 to 65535) of the tone(s).
>
> **Note** is a variable/constant specifying frequency in hertz (Hz, 0 to 32767) of the first tone.

Explanation

The sound command generates a pulse at the specified frequency. The sound command can be used to play tones through a speaker or audio amplifier. Sound can also be used to play simple songs.

Example

```
SOUND B1,[1000 \ 2500]
```

This instruction generates a 2500-Hz tone for 1 second (1000 ms) through B1. To play two notes:

```
SOUND B1,[1000 \ 2500, 2000 \ 2500]
```

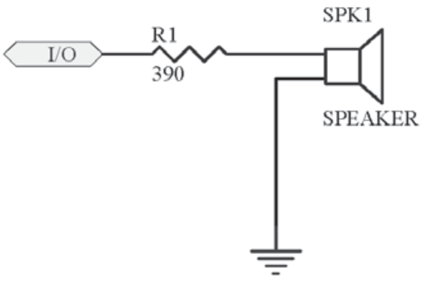
The above code will play two notes first one being 2500Hz for 1 second and the second 2500Hz for 2 seconds.

The notes generated are clean without filtering. Refer to the schematic

The next program demonstrates the Sound command and should give you a better idea of what values to ue to generate certain tones.

```
'Variables used to hold the frequency and duration of the sound command

freq var word
dur var word

    main:
        Freq = 2500
        Dur = 1000
                    ;Now play it
        sound B0,[dur\freq]
    goto main
```

Schematic



Sound2

Sound2 pin1\pin2,[duration1\note1\note2_1,...durationN\noteN\note2_N]
Generate specific notes one on each of the two defined pins.

> **Pin1 \ Pin2** is a variable/constant that specifies the I/O pins to use. This pin will be set to an output during tone generation and left in that state after the instruction is completed. The two specified pins can be tied together as shown in the schematic
>
> **Duration** is a variable/constant specifying the length in milliseconds (1 to 65535) of the tone(s).
>
> **Note** is a variable/constant specifying frequency in hertz (Hz, 0 to 16000) of the tones.

Explanation

Sound2 generates two pulses at the specified frequency one on each pin specified. The sound2 command can be used to play tones through a speaker or audio amplifier. Sound2 can also be used to play more complicated songs. By generating two frequencies on separate pins, a more defined sound can be produced.

Example

```
SOUND2 B1 \ B2,[1000 \ 2500 \ 3500]
```

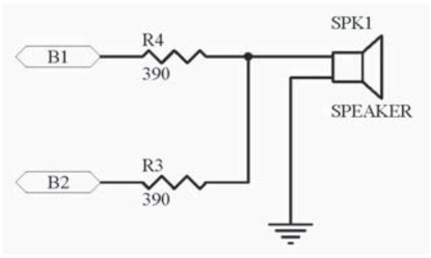
The above code generates a 2500Hz tone and a 3500Hz tone for 1 second (1000 ms). The first note specified is played from the first pin specified and the second note from the second pin specified. To play multiple notes:

```
SOUND2 B1 \ B2,[1000 \ 2500 \ 3500, 2000 \ 2500 \ 3500]
```

The above code will play two sets of notes 2500Hz and 3500Hz for 1 second and the second two notes, 2500Hz and 3500Hz for 2 seconds.

The notes generated are clean without filtering. The pins should be summed together to produce multiple melody songs. With the sound2 command more complex music can be played by the PICmicro. Refer to the schematic.

Schematic



Spmotor

SPMOTOR pin, delay, step

Pin can be a variable or constant. Pin specifies the first pin out of 4 control pins required. If B0 was used, the control pins would then be B0, B1, B2, B3.

Delay can be a variable or constant and is a value from 0 to 65365 in milliseconds. Delay controls the speed at which the stepper motor will rotate. The delay will also vary from stepper motor to stepper motor.

Step can be a variable or constant and is the number of steps and the direction. The direction is determined by the value of Step. Positive values being clockwise and negative numbers being counter clockwise. The step value can be a range from -32682 to +32682

Explanation

Stepper motors are precision motors which have an absolute amount of travel per step. This is ideal in situation where precise positioning is necessary. Stepper motors are commonly found in XY positioning tables. Steppers motors can be purchased from several sources. Chances are you may have a few laying around. They are commonly salvaged from old disk drives and laser printers.

There are two types of stepper motors. Unipolar and Bipolar. Unipolar means one pole. This is usually a common ground between 4 coils. Unipolar stepper motors are easier controlled with minimal circuitry. Bipolar motors indicate two poles. Bipolar motors require additional circuity in order to drive them. The SPMOTOR command does not support Bipolar motors. In most cases you can easily distinguish between then two types. Unipolar stepper motors have 5 wires. Bipolar motors usually have 4.

The use of the SPMOTOR command requires a simple circuit using a darlington array (ULN2803A) to sink the load from the stepper motor. Some small low power stepper motors can be driven from the microcontroller directly. However this is not recommended. Other circuits can be used to sink the load from the stepper motor. The ULN2803A is the most commonly used.

Example

The following program will rotate a stepper motor 1000 steps clockwise and counter clockwise.

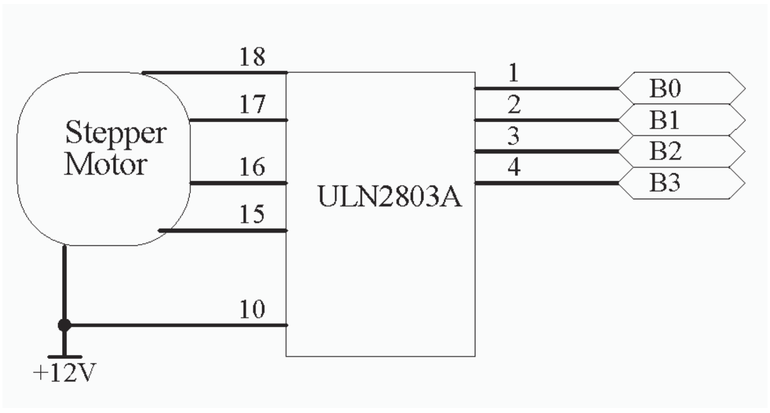
```
Main
      SPMOTOR B0, 10, 1000
      SPMOTOR B0, 10, -1000
      Goto Main
```

The SPMOTOR command can be used to drive more than one motor. To do this you would use the SPMOTOR command more than once with new pins defined.

Important Note

The SPMOTOR command will not work correctly when used with Debug. This is due to the serial data the PICmicro must send to keep in sync with the IDE.

Schematic



Stop

STOP
Stops program execution.

Explanation

STOP prevents the program from executing any further instructions until it is reset. The STOP command is similar to END but it does not put the Atom into low-power mode. The Atom draws just as much current as if it were actively running program instructions. The only way to start the program again is to reset the chip.

```
STOP            ;Stops program
```

Example

Stop can be used to quickly exit a loop based on a certain condition as shown in the example below:

```
myswitch var portb.bit0         ;myswitch is now an alias of B0

MAIN:
    IF myswitch = 0 THEN MAIN
                    ;check if PORTB PIN 0 is low if so it loops

    IF myswitch = 1 THEN STOPIT
                    ;check if switch is on if so goto label RUNIT
GOTO MAIN
                    ;will continue to loop until P0 is high

STOPIT:
    STOP
                    ;stops program execution
```

The above code example is a software ON/OFF switch, when power to the rest of the circuit needs to remain on but the PICmicro must be stopped.

Swap

SWAP variable,variable

Variable is the value to be swapped

Explanation

Swap any two variable's values with each other. If Dog = 10 and Temp = 0 then by using the swap command Dog will now equal 0 and Temp will equal 10.

Example

The below program will run once. The If..Then does a comparison, since temp = 0 it continues. Then the Swap command swaps the values in dog with temp. The If..Then will now be true since temp is now equal to 10.

```
Dog Var Byte
Temp Var Byte

Dog = 10     ;Dog equals 2
Temp = 0     ;Temp equals 0

Main
    if temp = 10 then ExitLoop
    Swap Dog, Temp          ;Temp now equals 10
Goto Main

ExitLoop
    end
```

Toggle

TOGGLE pin
Invert the state of a pin.

Pin is a variable or constant that specifies the I/O pin to use.

Explanation

TOGGLE inverts the state of an I/O pin, changing 0 to 1 and 1 to 0. The pin is automatically made an output.

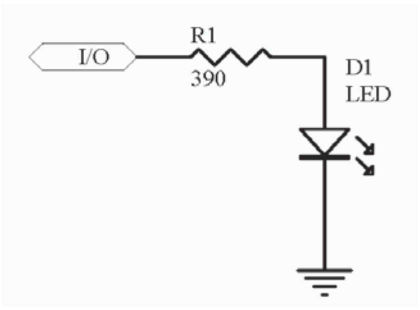
Example

The following example will blink an LED connected to the specified I/O pin.

```
Low B0          ;set initial pin state

    Main
    Toggle B0   ;switch pin state
    Pause 200
Goto Main
```

The above program will toggle B0 each loop. The first run will toggle the pin from low to high. The second time through the loop will then toggle the pin from high to low.

Schematic



While...Wend

While expression is true do the following

> **Expression** is any combination of variables, constants, math and logic operators

Explanation

Repeat a group of commands while some expression is true. True being any other value than 0. Each time the group of commands are executed the WEND checks to see if the expression is still true.

Example

In the following example, the instructions between the WHILE...WEND commands will execute until the variable Temp is less than 20. With each run through the loop WEND will check the status of Temp.

```
    Temp var Byte
Check var PortB.bit0

Clear

    While temp < 20        ; repeat the following commands.
        if check = 1 then JumpOut
        Temp = Temp +1
    Wend          ; repeating until temp is greater than 20

JumpOut
    End
```

The next program demonstrates the While Wend command by blinking an LED attached to pin B2 of the PICmicro. The LED will blink a total of 100 times with a 500ms pause between each blink.

```
    output B2 'Place an led on pin 2

    counter var byte
    counter = 0

    while counter < 100
        Toggle B2 'Change the state of pin B2
        counter = counter +1
        pause 500
    wend
```

Write

WRITE address,byte
Write a byte of data to the EEPROM.

> **Address** is a variable or constant specifying the EEPROM address to write to.
>
> **Byte** is a data byte to be written into EEPROM.

Explanation

EEPROM differs from RAM, the memory in which variables are stored, in several respects:

1- Writing to EEPROM takes more time than storing a value in a variable.

2- The EEPROM can accept a finite number of write cycles per byte before it wears out, which is usually around 10 million write cycles and an unlimited number of Reads. If a program frequently writes to the same EEPROM location, it makes sense to estimate how long it might take to exceed 10 million writes. For example, at one write per second (86,400 writes/day) it would take nearly 116 days of continuous operation to exceed 10 million.

WRITE is used to set values of on-chip EEPROM during run time. To set values during programming use the DATA commands.

Each write will take about 10ms to execute. Keep this in mind when using the WRITE command.

```
WRITE 6,F   ;Sends the byte F to location 6 on the EEPROM
```

Example

The below program writes the ASCII value of H to address 0 of the internal EEPROM on the PICmicro, then reads address 0 and if the value equals H turns on a LED connected to B4.

```
DOG         var       byte
write       0,"H"     ;writes the ASCII value of H to address 0
read        0,dog     ;reads address 0 stores the value in ledon
if dog =  "H"  then ledon
                                ;if ledon = the ASCII value H
                                ;turn the LED on
stop
ledon:      High B4 ;turns the LED on connected to B4
end
```

The following program will demonstrates the Data, Read, and Write commands

```
	'Day 1-4
	Data @0, 12,26,3,32

	'Day 5-8
	Data @4,22,12,31,45


	'Define some variable to use for loading the eeprom data
	day1 var byte
	day2 var byte
	day3 var byte
	day4 var byte
	day5 var byte
	day6 var byte
	day7 var byte
	day8 var byte

	clear		'Make sure we clear the variables

'-------------------------------------------------------
' Step 1
' Lets load the eeprom data
'-------------------------------------------------------
	read 0,day1
	read 1,day2
	read 2,day3
	read 3,day4
	read 4,day5
	read 5,day6
	read 6,day7
	read 7,day8

	'Display the default data
	debug ["Day1 default=",dec day1,10,13]
	debug ["Day2 default=",dec day2,10,13]
	debug ["Day3 default=",dec day3,10,13]
	debug ["Day4 default=",dec day4,10,13]
	debug ["Day5 default=",dec day5,10,13]
	debug ["Day6 default=",dec day6,10,13]
	debug ["Day7 default=",dec day7,10,13]
	debug ["Day8 default=",dec day8,10,13]
	debug [10,13]
```

```
'----------------------------------------------------------
' Step 2
' Ok now lets overwrite day3 and day8 defaults.
' Remember the address is 0 based
'----------------------------------------------------------
        write 2,day3+1 'Add one
        write 7,day8+2 'Add two

'----------------------------------------------------------
' Step 3
' Lets load the eeprom data again
'----------------------------------------------------------
        read 0,day1
        read 1,day2
        read 2,day3
        read 3,day4
        read 4,day5
        read 5,day6
        read 6,day7
        read 7,day8

'And display it one more time
        debug ["Day1 default=",dec day1,10,13]
        debug ["Day2 default=",dec day2,10,13]
        debug ["Day3 default=",dec day3,10,13]
        debug ["Day4 default=",dec day4,10,13]
        debug ["Day5 default=",dec day5,10,13]
        debug ["Day6 default=",dec day6,10,13]
        debug ["Day7 default=",dec day7,10,13]
        debug ["Day8 default=",dec day8,10,13]

'--------------------------------------------------------
'  All done.  Put the PICmicro to sleep.  Note the values and
'  restart the chip.

        end
```

WriteDM

writedm address,[{mod}expression,...,{mod}expressionN]
Write EEPROM locations and store value in variable.

> **Address** is a value pointing to a location in the eeprom from which to start.
>
> **Mod** is any appropriate input or output modifier
>
> **Var** is any variable type
>
> **Expression** is any legitimate math expression

Explanation

The WriteDM is similar to the Write command. The WriteDM statement is setup for sequential writes at a starting address in the internal eeprom. These writes will start at a specified address and will sequentially continue until the end of the data or eeprom range is reached.

```
    WriteDM 0,[“Hello world”]
```

The above example will begin writing at address 0 to the internal eeprom. The string “Hello World” contains 11 bytes (Space included) since the starting address is 0 the last written address will be 10.

Example

The following program illustrates the usefulness of the ReadDM and WriteDM command. We begin with an Array to store our returned data.

```
string var byte(12)

    main
        writedm 0,[“Hello world”]
        readdm 0,[str string\11]

mainlp
    serout B0,i9600,[str string\11,13]
goto mainlp
```

The above program will write the entire string “Hello World” to the internal eeprom. Since the string “Hello World” consist of 11 bytes the first 11 bytes of the eeprom will be written to. Then the ReadDM command is used to read the 11 bytes back. By using the STR modifier we tell the ReadDM command we want to read 11 bytes from the internal eeprom and store them in our 12 byte variable array. Next we use the SEROUT command to display the results to a terminal window.

Important Notes

There are only 256 internal eeprom locations on a 16F876. If the ReadDM or WriteDM command begin in a valid location and continue outside of the 256 locations they will wrap around to the beginning location.

SEE ALSO

ReadDM
STR Modifier

WritePM

Writepm address,[{mod}var,...,{mod}varN]
Write Flash Program memory location. Requires 16F87x part.

> **Address** is a value pointing to a location in the flash program memory from which to start. Requires 16F87x part.
>
> **Mod** is any appropriate input or output modifier
>
> **Var** is any variable type

Explanation

The WritePM command is similar to the WriteDM command except it writes to the internal program memory of a 16F87x PICmicro. The WritePM statement is setup for sequential writes to the internal program memory. These writes will start at a specified address.

```
    writepm 0x400,["Hello world"]
```

The above example will beginwritting at address 400h to the internal program memory.

Example

The following program will write “Hello World” using the WritePM command starting at address 400h. Then the ReadPM command is used to read the written values to program memory and print the result to the debug window.

We begin with an Array to store our returned data.

```
string var byte(12)
temp var byte

    main
        writepm 0x400,["Hello world"]
        readpm 0x400,[str string\11]
mainlp
    for temp = 0 to 11
    Debug [string(temp)]
    Next
    Debug [13]
goto mainlp
```

The example program will write the entire string “Hello World” to the internal program memory. Since the string “Hello World” consist of 11 characters, 11 bytes will be written to the internal program memory starting at location 400h. If you run the program once and read back the programmed hex file from the PICmicro at location 400h will be the values for “Hello World”.

By using the STR modifier we tell the ReadPM command we want to read 11 bytes from the internal eeprom and store them in our 12 byte variable array. Next we use the Debug command to display the results to the debug window.

Important Notes

The READPM and WRITEPM commands will only work with 16F87x parts. You can not write or read actual program space used for the running program. You can write and read from address locations that do not contain program data. If the ReadPM command is used to read locations of actual running program space, erratic results will occur.

SEE ALSO

WRITEPM
STR Modifier

Xin

XIN DataPin\ZeroPin,House,{TimeoutLabel,TimeoutCount,}[{Modifier}Var]
Receive X-10 data and store keycode in a variable.

DataPin is a variable/constant that specifies the I/O pin to use. This pin will be set to an input. The DataPin should be pulled high with a 4.7K resistor.

ZeroPin is a variable/constant that specifies the I/O pin to use. This pin will be set to an input. The ZeroPin should be pulled high with a 4.7K resistor. The ZeroPin is used to detect the zero crossing timings from the X-10 device.

House is a variable/constant used to filter out data with multiple house settings. House is a comparison value to match if the incom ing data matches a particular house code other wise the XIN command will continue waiting. The constant labels for each house code are as follows:

X_A	X_I
X_B	X_J
X_C	X_K
X_D	X_L
X_E	X_M
X_F	X_N
X_G	X_O
X_H	X_P

TimeoutLabel is an optional label used to specify a place to jump to if a time out occurs.

TimeoutCount is an optional value used in conjunction with the TimeoutLabel to specify the amount of time that occurs before jumping to the TimeoutLabel. Timeouts are based on commands from the X-10 module. The value placed for TimeoutCount will wait *N* amount of commands received from the X-10 module before a time out will occur.

Var is a variable/constant used to store the results (KeyCode) of the XIN statement. The incoming data is the keycode which is 5 bits so a byte size variable is needed.

Explanation

XIN allows you receive signals sent through household AC wiring to X-10 modules. XIN requires the use of a special module (TW523) that interfaces to the AC wiring. Other interface modules are available but the TW523 has an input and output connection that allows you to receive and send X-10 data.

The X-10 format is made up of digital codes imposed on a 120 kHz carrier that is transmitted during zero crossings of the AC line. Receiving X-10 commands require the controller to be synchronize to the AC line. To connect to the X-10 interface a four-conductor phone cable is required.

Example

```
    XIN B0\B1, X_A, XError, 100, [Temp]
```

The above code will wait approximately 100 command cycles to find the data corresponding to house code A until it will exit and jump to the label XError. TimoutLabel and TimeoutCount are optional.

```
    XIN B0\B1, X_A, [Temp]
```

The above code will wait indefinitely for data corresponding to house code A. Once the data is received it will be placed in the variable (Temp).

```
Temp Var Byte

Main
    XIN B0\B1, X_A, [Temp]
    DEBUG [BIN TEMP, 13]
Goto Main
```

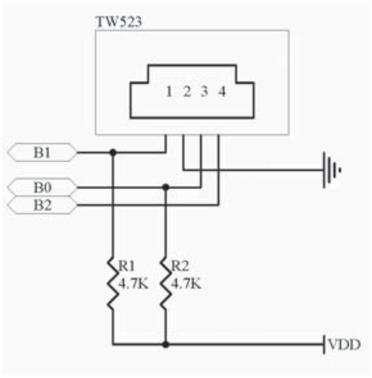
The above code will continue to loop and receive all data for house code A and display it to the debug window.

TW523

The TW523 module is available from many places on the internet. One source is http://www.x10.com Once on the web site do a quick search for TW523. The TW523 is used to remove much of the burden in decoding the X10 data from the AC wiring. The pin outs for the TW523 module are shown below:

Wire No	Connection
1	Zero Crossing
2	Common
3	X-10 Receive data from TW523
4	X-10 Transmit to TW523

Schematic



Important Notes

If the data received is a unit number you can use these constant labels to do a comparison:

Unit	Constant
X_1	% 00110
X_2	% 00111
X_3	% 00100
X_4	% 00101
X_5	% 01000
X_6	% 01001
X_7	% 01010
X_8	% 01011
X_9	% 01110
X_10	% 01111
X_11	% 01100
X_12	% 01101
X_13	% 00000
X_14	% 00001
X_15	% 00010
X_16	% 00011

X10 Issues

Interface modules such as the TW523 and PL513 have limitations to how well they may work in your area. During testing of the XIN and XOUT commands we found the modules would not address some units in the same building. This may be due to noise in the lines, or faulty electrical wiring. The interface units did not work well with filtering power strips either.

When setting up your X10 units, test them close together. The transmitter and receiver should be on the same plug if possible. This will save many hair pulling hours. Once the units demonstrate they work, move the receiver unit to its target location. If it does not work in another locations it is NOT your circuit, provided the receiver worked under the previous mentioned test conditions.

Xout

XOUT DataPin\ZeroPin, House, [{Unit}, {Modifiers} Keycode]
Transmit X-10 House code and Keycode.

DataPin is a variable/constant that specifies the I/O pin to use. This pin will be set to an input. The DataPin should be pulled high with a 4.7K resistor.

ZeroPin is a variable/constant that specifies the I/O pin to use. This pin will be set to an input. The ZeroPin should be pulled high with a 4.7K resistor. The ZeroPin is used to detect the zero crossing timings from the X-10 device.

House is a variable/constant that corresponds to the House Code set on the X-10 module A through P. The constant labels for each house code are as follows:

X_A	X_I
X_B	X_J
X_C	X_K
X_D	X_L
X_E	X_M
X_F	X_N
X_G	X_O
X_H	X_P

Unit is an optional variable/constant that specifies the address of the unit, 1 to 16. Unit codes are also considered KeyCodes.

Unit	Constant
X_1	%00110
X_2	%00111
X_3	%00100
X_4	%00101
X_5	%01000
X_6	%01001
X_7	%01010
X_8	%01011
X_9	%01110
X_10	%01111
X_11	%01100
X_12	%01101
X_13	%00000
X_14	%00001
X_15	%00010
X_16	%00011

KeyCode is a variable/constant that specifies the unit code or function. Multiple KeyCodes can be used in the XOUT command. Only certain commands will work in combination such as DIM codes.

Unit	Constant
X_Units_On	% 10000
X_Lights_On	% 11000
X_On	% 10100
X_Off	% 11100
X_Dim	% 10010
X_Bright	% 11010
X_Lights_Off	% 10110
X_Hail	% 10001
X_Status_On	% 11011
X_Status_Off	% 10111
X_Status_Request	% 11111

Explanation

XOUT is used to control X-10 modules over AC house wiring. XOUT requires the X-10 modules TW523 or PL513.

The XOUT command must first transmit the specified house code and unit code in order to communicate to another X-10 device. The X-10 module with the corresponding code will listen for its house code followed by a KeyCode (ie. dim, on or off)

The X-10 format is made up of digital codes imposed on a 120 kHz carrier that is transmitted during zero crossings of the AC line. Sending X-10 commands requires that the controller is synchronize to the AC line. The X-10 format requires a strict 50 ms timing to transmit an 11-bit code representing the command. XOUT is interfaced to the AC power-line through a device such as a TW523 or PL513. To connect to the X-10 interface a four-conductor phone cable is required.

Example

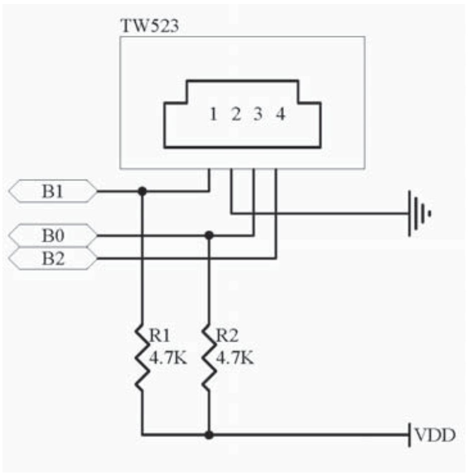
The following code example will blink two X-10 lamp units on and off. The units are set to house code X_A and unit codes X_1, X_2 :

```
Dpin      Con B2  ;Datapin is B2
Zpin      Con B1  ;Zeropin is B1

Main
     XOUT Dpin \ Zpin, X_A, [X_1, X_On, X_2, X_On]  ;turn unit 1 & 2on
     Pause 10000
     XOUT Dpin \ Zpin, X_A, [X_1, X_Off, X_2, X_Off]  ;turn unit 1 & 2on
     Pause 10000
Goto Main         ;repeat
```

The next code example will demonstrate the use of the modifier REP (Repeat) to dim a module *n* times (Refer to command modifiers for more information). To achieve a certain brightness level the dim command may need to be sent more than once as shown below:

```
Dpin    Con B2 ;Datapin is B2
Zpin    Con B1 ;Zeropin is B1

Main
    XOUT Dpin \ Zpin, X_A, [X_1, REP X_Dim \5]
                ;will send the dim command 5 consecutive times
End
```

Schematic



TW523

The TW523 module is available from many places on the internet. One source is http://www.x10.com Once on the web site do a quick search for TW523. The TW523 is used to remove much of the burden in decoding the X10 data from the AC wiring. The pin outs for the TW523 module are shown below:

Wire No	Connection
1	Zero Crossing
2	Common
3	X-10 Receive data from TW523
4	X-10 Transmit to TW523

Hardware Commands

MBASIC
for PIC microcontrollers

Hardware Commands

PICmicros have many hardware features built in. This section explains all the available commands and ways to use the built in hardware features. Some of these commands are considered advanced and should only be used once you have obtained a clear understanding of the MBasic.

Important Note

Not all PICmicros have the same built in hardware. The following commands only work with PICmicros that have such hardware. To determine wether the PICmicro you are using has such hardware refer to its data sheet available from Microchip.com.

The 16F876 and 16F877 have all the built in hardware to use any one of the following commands.

HSERIN..HSEROUT

```
SETHSERIAL mode
HSERIN {label,timeout,}[InputData]
HSEROUT [InputData]
```

Receive and Send Asynchronous RS-232 data.

label,Timeout(optional) is a label that will be jumped to if the hserin buffer has no data available in the buffer and the timeout period has expired. If a timeout is not defined the hserin command will wait indefinitely while the uart receive buffer is empty.

InputData can be a list of variables and or modifiers that tell HSERIN / HSEROUT what to do when sending or receiving data. All the modifiers supported by SERIN and SEROUT are supported by HSERIN and HSEROUT.

Explanation

The 16F876 has a built-in UART. The UART buffer can store a maximum of 94 bytes, 47 bytes input and 47 bytes output. The HSERIN UART is hard wired to pin C7 and HSEROUT is hard wired to C6. The 16F876's UART allows data to be received or sent while the 16F876 is executing another part of your program. Once the UART is setup in software it will run independently. The only consideration to this is the buffer size. If you receive or transmit to much data without accessing it, the buffer will wrap around and begin to overwrite the first location. All data sent or received is inverted. There are no options to change this so a max232 circuit is required when using the 16F876 hardware serial port.

There are several available PICmicros with hardware UARTs. The HSERIN / HSEROUT commands will work with all of them. To determine which PICmicros have UARTs download the respective data sheets.

Example

The 16F876 hardware serial port is straight forward and only requires a one time setup command. Once this command is issued the 16F876s UART is on and will receive data without any further user code.

```
sethserial H9600

temp var byte

main
	hserin [temp]
	hserout [temp]
goto main
```

The previous code snippet will simply transmit what ever was received continuously.

Another example would be to setup the UART and let it run then occasionally check to see if a byte is received as shown below.

```
sethserial H9600

temp var byte
main
	high b0
	pause 200
	low b0
	pause 200
	HSERIN 1000,main,[temp]
	if temp = “A” then loop1
goto main

loop1
	serout B1, i9600,[“Recevied the correct byte”]
goto main
```

The above program can receive a byte at any time regardless of the command that is being executed. Since the incoming data has been buffered. All we need to do is to check this byte and some point.

SetHSerial

The sethserial command is used to setup and turn on the 16F876’s UART. This command is only required to be set once in your program. The following is a list of all the possible bit rates HSERIN / HSEROUT will transmit or receive at:

Baud Modes		High Speed Baud Modes
H1200	H24000	H250000
H2400	H26400	H312500
H4800	H28800	H625000
H7200	H31200	H1250000
H9600	H33600	
H12000	H36000	
H14400	H38400	
H16800	H57600	
H19200	H115200	
H21600		

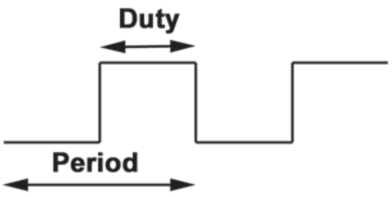
HPWM

Hpwm CCPx, Period, Duty
Generate pulse-width modulation use internal hardware PWM.

> **CCPx** is a variable or constant of 0 or 1 that specifies the PWM hardware to use. The 16F876 has two PWM's available. The first is on pin C2 which is selected by setting a value of 1. The second is on pin C1 and is selected by using a value of 0 for CCPx.
>
> **Period** is a variable or constant from 0 to 16383 that specifies the period of the pulse width in CLK cycles.
>
> **Duty** is a variable or constant from 0 to 16383 that specifies the duty cycle of the pulse width.

Explanation

The 16F876 has two built in Pulse Width Modulators. The actual hardware PWM differs from the software PWM in that it can produce more accurate pulse widths. The PWM hardware can be left running while your program is performing other task. Both PWM's can be ran at the same time. When both are running they will both have the same period but different duties if specified (The Period is the inverse of the frequency). The frequency is how many times from high to low. The period is the total time of one high to low transition. The duty is how long the pulse stays high.



Example

If PWM is used to generate an analog voltage. We can use this to control the brightness of an LED. If you wanted about 50% brightness, we would then need to set PWM to about 2.5 Volts. In order to achieve this we would use the following:

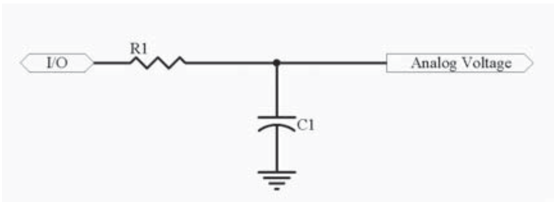
```
    PWM 0,16383, 8191
        ; 50% duty cycle PWM signal using pin 10 (module 0)
```

The above code snippet would give and output voltage of about 2.5 volts using the shown schematic. Once the above command is ran in your program, PWM will continue to output until the 16F876 is shut down or your program stops the PWM hardware.

This next example demonstrates the Hpwm and Count commands. This program can be used to give you a better understanding of what values to use for HWPM. You will need to Make sure you tie pin C2 (Hpwm output) to pin B0 (count input)

```
counter var long        'Used for the count command

hpwm 1,9920,4960        'Set up a 2KHz signal (approx) on pin 9
                        'hpwm 1,8,4 'Set up a 83KHz signal (approx)

Loop:
    count B0,1000,counter
                        'Set up our counter on pin B0 for 1 second

                        'Display the results
    serout B1,i9600,[dec counter," Hz",10,13]
goto loop
```

Schematic



Important Notes

The Duty is a maximum of 10 bits. If period is set to less than 1024 (10bits) then duties accuracy will be reduced. An example would be if period is 256 duty's accuracy will be 8 bits.

If Period is set higher than 1024 then Period will actually be a multiple of 4 (i.e.: 1024,1028,1032 etc...)

If Period is set higher than 4096 then Period will actually be a multiple of 16 (i.e.: 4096,4102,4118 etc...)

SETPULLUPS

SETPULLUPS   mode
Enables or disables internal 10k pullups on pins B0 to B7

Mode is PU_OFF to disable pullups or PU_ON to enable pullups

Explanation

Most PICmicros have internal weak pullup resistor. The internal pullups will only apply to an input state on the pin. The pullups are tied to VDD.

On Reset Commands

On Reset Commands

Reset commands are commands that perform an action in your program based on how the PICmicro was reset. On Reset commands are ideal in situations where power is falling below minimum operating ranges. These can be used for a proper shutdown if external circuitry is attached. The PICmicro can also detect if a reset occurred by the ATN / RES pin.

ONPOR	= Power on reset, jump to label
ONBOR	= Brown out reset, jump to label
ONMOR	= ATN / RES reset, jump to label

The syntax of the on reset commands are:

```
Command Label
```

Command being one of the three on reset commands.

Label being the label to jump to if the condition occurs.

There are three commands that allow your program to perform some action based on resets of the PICmicro. There are three different resets with the PICmicro. The first is the power on reset (ONPOR) which happens when the PICmicro is first powered up or the power was removed and restored. The second is brown out reset (ONBOR) this reset occurs when the VDD level of the PICmicro falls below +4.1 volts this may vary depending on operating conditions. The ONBOR is ideal for battery applications. The third reset that can occur is ATN / RES reset (ONMOR) this reset condition happens by resetting the PICmicro using the MCLR without interrupting VDD.

Example

This example illustrates how to use the ON Reset command to do something in your program when a reset occurs:

```
'Demonstrates the onreset commands

Counter var word

ONPOR startup
ONBOR brownout
ONMOR reset


main:
    counter=counter + 1
```

```
          serout B5,i9600,[".. Program Running :",dec counter,10,13]
          pause 1000
     goto main

     startup:
          clear
          ;When program is powered up we need to reset all variables

          serout B5,i9600,["Program started from powerdown",10,13]
     goto main

     reset:
          serout B5,i9600,["Program started from reset",10,13]
     goto main

     brownout:
          serout B5,i9600,["Program started after brownout",10,13]
     goto main
```

The above program will run normally by executing the main loop. Once a reset occurs and power is restored, the PICmicro will then determined the reset condition and jump to the specified label.

Interrupt Commands

Interrupts

The PICmicro has several interrupt sources. These interrupts can occur from internal or external sources. The interrupts are consider an advanced feature of MBasic. This next section explains each command used for interrupts.

```
Enable {Interrupt Source}
```

Enable interrupt is used to turn on the interrupt system. If no interrupt is given all interrupts setup using ONINTERRUPT are enabled. Enable interrupt can be used to turn on specific interrupts. (Refer to Interrupt Sources)

```
Disable {Interrupt Source}
```

Disable interrupt can be used to turn off specific interrupts or all interrupts at once. If no interrupt is given all interrupts are disabled. (Refer to Interrupt Sources)

```
ONINTERRUPT interrupt source, label
```

OnInterrupt is an operator used to tell your program where to go if a specified interrupt occurs. Interrupt specifies what type of interrupt to act on. Label is used to specify the place to jump to in a program if the interrupt occurs. When an interrupt occurs the label jump is performed as a gosub. The program will only return to its last known place by using the Resume command. An example of the syntax would be as follows:

```
OnInterrupt ExtInt ProgInt
enable ExtInt
Main
	Serout B2, i9600, [“Running”]
Goto Main

disable			;disable all interrupts from here down
ProgInt
	Serout B2, i9600, [“Interrupt Occured”]
Resume
```

The above program would continuously send out “Running”. When an interrupt occurred it would then jump to the label ProgInt. The program would then execute the code. Once the Resume command was encountered it would return to normal program operation and jump back to the next command before the interrupt occurred, in this case Main. (The Resume command works like the Return command in a Gosub routine.). Note that the “disable” command is used in a position that never gets executed. When using disable(or enable) without arguments you can place it on any line(even lines that will not be executed) in order to disable(enable) interrupts from that position down.

Interrupt Sources

EXTINT

External interrupt on B0. There are several option on how the interrupt can be triggered with ExtInt which are list in the SetExtInt section.

RBINT

On change interrupt can occur on B4,B5,B6 or B7. This interrupt will trigger if a pin state changes from low to high (or high to low).

TMR0INT

Tmr0Int interrupt occurs whenever Timer0 overflows (See SetTmr0 command)

ADINT

ADInt interrupt occurs when A/D conversion finishes. Used in conjunction with the ADin command.

RCINT

RCInt interrupt occurs when a byte is received by the Hardware USART (this interrupt is disabled if using HSERIN/HSEROUT).

TXINT

TXInt interrupt occurs when a byte finishes transmitting from the Hardware USART (this interrupt is disabled if using HSERIN/HSEROUT).

SSPINT

(See Synchronous Serial port(coming soon))

CCP1INT

CCPInt interrupt occurs on Capture/Compare/Period match

CCP2INT

CCP2Int interrupt occurs on Capture/Compare/Period match

TMR1INT

Tmr1int interrupt occurs whenever Timer1 overflows (See SetTmr1 command)

TMR2INT

Tmr2Int interrupt occurs whenever Timer2 overflows(See SetTmr2 command)

EEINT

EEInt interrupt occurs when a byte is finished writing to the on board EEprom.

BCLINT

(See Synchronous Serial port (coming soon))

Set Interrupt Source

Several of the interrupt commands have multiple sources that will trigger the interrupt. This next section explains how to set these different interrupt sources.

SETEXTINT

SetExtInt   mode

SetExtInt sets the external interrupt pin to input and sets the state that will cause an interrupt (EXTINT must be enabled)

Mode is the setting that will trigger the actual interrupt. There are two choices available:

EXT_H2L	= Will activate when pin B0 is pulled low (from high)
EXT_L2H	= Will activate when pin B0 is pulled high (from low)

SETTMR0

SETTMR0   mode

SETTMR0 sets Timer0 mode. Timer0 is an internal 8 bit timer module built into the 16F876.

Mode is the setting that will trigger the actual interrupt. There are several options available:

TMR0INT1	Internal timer with 1:1 timing
TMR0INT2	Internal timer with 1:2 timing
TMR0INT4	Internal timer with 1:4 timing
TMR0INT8	Internal timer with 1:8 timing
TMR0INT16	Internal timer with 1:16 timing
TMR0INT32	Internal timer with 1:32 timing
TMR0INT64	Internal timer with 1:64 timing
TMR0INT128	Internal timer with 1:128 timing
TMR0INT256	Internal timer with 1:256 timing
TMR0EXTL1	External counter(Low to High transition on AX3 pin)with 1:1 timing
TMR0EXTL2	External counter(Low to High transition on AX3 pin) with 1:2 timing
TMR0EXTL4	External counter(Low to High transition on AX3 pin) with 1:4 timing
TMR0EXTL8	External counter(Low to High transition on AX3 pin) with 1:8 timing
TMR0EXTL16	External counter(Low to High transition on AX3

	pin) with 1:16 timing
TMR0EXTL32	External counter(Low to High transition on AX3 pin) with 1:32 timing
TMR0EXTL64	External counter(Low to High transition on AX3 pin) with 1:64 timing
TMR0EXTL128	External counter(Low to High transition on AX3 pin) with 1:128 timing
TMR0EXTL256	External counter(Low to High transition on AX3 pin) with 1:256 timing
TMR0EXTH1	External counter(High to Low transition on AX3 pin) with 1:1 timing
TMR0EXTH2	External counter(High to Low transition on AX3 pin) with 1:2 timing
TMR0EXTH4	External counter(High to Low transition on AX3 pin) with 1:4 timing
TMR0EXTH8	External counter(High to Low transition on AX3 pin) with 1:8 timing
TMR0EXTH16	External counter(High to Low transition on AX3 pin) with 1:16 timing
TMR0EXTH32	External counter(High to Low transition on AX3 pin) with 1:32 timing
TMR0EXTH64	External counter(High to Low transition on AX3 pin) with 1:64 timing
TMR0EXTH128	External counter(High to Low transition on AX3 pin) with 1:128 timing
TMR0EXTH256	External counter(High to Low transition on AX3 pin) with 1:256 timing

SETTMR1

```
SETTMR1     mode
```

SetTmr1 sets Timer1 mode. Timer1 is an internal 16 bit timer module built into the 16F876.

Mode is the setting that will trigger the actual interrupt. There are several options available:

TMR1OFF	Disables Timer1(saves power/default on powerup)
TMR1INT1	Internal timer with 1:1 timings
TMR1INT2	Internal timer with 1:2 timings
TMR1INT4	Internal timer with 1:4 timings
TMR1INT8	Internal timer with 1:8 timings
TMR1EXT1	External osc with 1:1 timings
TMR1EXT2	External osc with 1:2 timings
TMR1EXT4	External osc with 1:4 timings
TMR1EXT8	External osc with 1:8 timings
TMR1ASYNC1	External Asynchronous counter with 1:1 timings
TMR1ASYNC2	External Asynchronous counter with 1:2 timings
TMR1ASYNC4	External Asynchronous counter with 1:4 timings
TMR1ASYNC8	External Asynchronous counter with 1:8 timings

Example

This program demonstrates TMR1 by counting using timer interrupt.

```
counter var word

clear           'Clear all variables

SETTMR1 TMR1INT1                    ' Set timer1 mode
ONINTERRUPT TMR1INT,mytimer
                     'Where do we want to go on the timer interrupt
Enable TMR1INT                      'Turn on the interrupt

                'main loop just to display the data
main
    debug ["counter=",dec counter,10,13]
    pause 100
goto main


                'This is where we go on and interrupt.
mytimer:
    counter = counter +1
resume          ' this is how we exit an interrupt
```

SETTMR2

SETTMR2   mode, period

SetTmr2 sets Timer2 mode and reset period. Timer2 is an internal 8 bit timer with an 8 bit period module built into the 16F876.

Mode is the setting that will trigger the actual interrupt. There are several options available.

Period is a reset point. Period will cause the timer reset when timer equals period. Period is a value of 0 to 255

Modes are:

TMR2OFF	Disables Timer2 default on powerup
TMR2PRE1POST1	1:1 prescaler and 1:1 postscaler
TMR2PRE1POST2	1:1 prescaler and 1:2 postscaler
TMR2PRE1POST3	1:1 prescaler and 1:3 postscaler
TMR2PRE1POST4	1:1 prescaler and 1:4 postscaler
TMR2PRE1POST5	1:1 prescaler and 1:5 postscaler
TMR2PRE1POST6	1:1 prescaler and 1:6 postscaler
TMR2PRE1POST7	1:1 prescaler and 1:7 postscaler
TMR2PRE1POST8	1:1 prescaler and 1:8 postscaler
TMR2PRE1POST9	1:1 prescaler and 1:9 postscaler
TMR2PRE1POST10	1:1 prescaler and 1:10 postscaler
TMR2PRE1POST11	1:1 prescaler and 1:11 postscaler
TMR2PRE1POST12	1:1 prescaler and 1:12 postscaler
TMR2PRE1POST13	1:1 prescaler and 1:13 postscaler
TMR2PRE1POST14	1:1 prescaler and 1:14 postscaler
TMR2PRE1POST15	1:1 prescaler and 1:15 postscaler
TMR2PRE1POST16	1:1 prescaler and 1:16 postscaler
TMR2PRE4POST1	1:4 prescaler and 1:1 postscaler
TMR2PRE4POST2	1:4 prescaler and 1:2 postscaler
TMR2PRE4POST3	1:4 prescaler and 1:3 postscaler
TMR2PRE4POST4	1:4 prescaler and 1:4 postscaler
TMR2PRE4POST5	1:4 prescaler and 1:5 postscaler
TMR2PRE4POST6	1:4 prescaler and 1:6 postscaler
TMR2PRE4POST7	1:4 prescaler and 1:7 postscaler
TMR2PRE4POST8	1:4 prescaler and 1:8 postscaler
TMR2PRE4POST9	1:4 prescaler and 1:9 postscaler
TMR2PRE4POST10	1:4 prescaler and 1:10 postscaler
TMR2PRE4POST11	1:4 prescaler and 1:11 postscaler
TMR2PRE4POST12	1:4 prescaler and 1:12 postscaler
TMR2PRE4POST13	1:4 prescaler and 1:13 postscaler
TMR2PRE4POST14	1:4 prescaler and 1:14 postscaler
TMR2PRE4POST15	1:4 prescaler and 1:15 postscaler
TMR2PRE4POST16	1:4 prescaler and 1:16 postscaler
TMR2PRE16POST1	1:16 prescaler and 1:1 postscaler

TMR2PRE16POST2	1:16 prescaler and 1:2 postscaler
TMR2PRE16POST3	1:16 prescaler and 1:3 postscaler
TMR2PRE16POST4	1:16 prescaler and 1:4 postscaler
TMR2PRE16POST5	1:16 prescaler and 1:5 postscaler
TMR2PRE16POST6	1:16 prescaler and 1:6 postscaler
TMR2PRE16POST7	1:16 prescaler and 1:7 postscaler
TMR2PRE16POST8	1:16 prescaler and 1:8 postscaler
TMR2PRE16POST9	1:16 prescaler and 1:9 postscaler
TMR2PRE16POST10	1:16 prescaler and 1:10 postscaler
TMR2PRE16POST11	1:16 prescaler and 1:11 postscaler
TMR2PRE16POST12	1:16 prescaler and 1:12 postscaler
TMR2PRE16POST13	1:16 prescaler and 1:13 postscaler
TMR2PRE16POST14	1:16 prescaler and 1:14 postscaler
TMR2PRE16POST15	1:16 prescaler and 1:15 postscaler
TMR2PRE16POST16	1:16 prescaler and 1:16 postscaler

SETCAPTURE

```
SETCAPTURE    ccppin,mode
```

Ccppin specifies which module to use 0 for CCP1 on pin C2 or 1 for CCP2 on pin C1.

Mode is the setting that will trigger the actual interrupt. There are several options available.

CAPTUREOFF	Disables Capture default on powerup
CAPTURE1H2L	Captures Timer1 value on High to Low transition
CAPTURE1L2H	Captures Timer1 value on Low to High transition
CAPTURE4L2H	Captures Timer1 value on 4th Low to High transition
CAPTURE16L2H	Captures Timer1 value on 16th Low to High transition

GETCAPTURE

```
GETCAPTURE       ccppin,var
```

Ccppin specifies which module to use 0 for CCP1 on pin C2 or 1 for CCP2 on pin C1.

Var is a word sized variable that the results of the 16 bit capture value will be returned to.

SETCOMPARE

SETCOMPARE   ccppin,mode,compare valu**Compare Value** is the value that the comparison must match before the mode's action will occur.

Modes are:

COMPAREOFF	Disables Compare default on powerup
COMPARESETHIGH	Compare sets CCPx pin high on Timer1 comparison match
COMPARESETLOW	Compare sets CCPx pin low on Timer1 comparison match
COMPAREINT	Compare sets Interrupt(CCPxINT) and clears Timer1 on Timer1 comparison match
COMPARESPECIAL	Compare runs Special(Reset Timer1(CCP1) or Reset Timer1 and Activate(if A/D is enabled) A/D conversion(CCP2) on Timer1 comparison match

Example

The following code example demonstrates one way to use the interrupts to generate a real time clock.

```
second  var byte
minute  var byte
hour    var byte
tick    var byte
ftick   var long

second = 0
minute = 36
hour = 11
tick = 0
ftick = 0
tickcnt con float 1.0038677

oninterrupt tmr1int,clock

settmr1 tmr1int4

lcdwrite B4\B5,portb.nib0, [INITLCD1,INITLCD2,CLEAR,HOME,SCR]

enable tmr1int
main
	goto main

Disable
clock
	ftick = ftick fadd tickcnt
	if (int ftick) > 19 then
		ftick = ftick fsub float 19
		second = second + 1
		if second > 59 then
				second = 0
				minute = minute + 1
				if minute > 59 then
						minute = 0
						hour = hour + 1
						if hour > 23 then
								hour = 0
						endif
				endif
		endif
		lcdwrite B4\B5,portb.nib0,[SCRRAM]
		if hour < 10 then
				lcdwrite B4\B5,portb.nib0,[" "]
		endif
		lcdwrite B4\B5,portb.nib0,[dec hour,":"]
		if minute < 10 then
				lcdwrite B4\B5,portb.nib0,["0"]
		endif
		lcdwrite B4\B5,portb.nib0,[dec minute,":"]
		if second < 10 then
				lcdwrite B4\B5,portb.nib0,["0"]
		endif
		lcdwrite B4\B5,portb.nib0,[dec second]
	endif
	resume
```

Example 2

The following code example demonstrates the correct way to enable and disable interrupts. If interrupts are not disabled correctly, the interrupt routines could get call again..and again...etc. This will cause the stack to overflow, causing a reset.

```
w_cnt      var word 'create variable
w_ctover var word 'create variable

w_cnt = 0 'initialize variable
w_ctover = 0 'initialize variable

ONINTERRUPT TMR1INT, mytimer1         'Where to go on interrupt
SETTMR1 TMR1INT8                      '1:8 internal timer interrupt

ENABLE TMR1INT                        'Turn On the Interrupt

Mainloop:
	debug ["Count/OverFlow = ", dec w_cnt," / ",dec w_ctover , 13]
		; show main counter

		if w_cnt >= 65000 then
				; check to see if count is near overflow
		w_ctover = w_ctover + 1 ; increment count overflow
		w_cnt = 0                     ; Reset counter to zero
	endif
	goto mainloop

disable         ; This will disable ALL interrupts from here down.
```

```
                        ; This must always be used before interrupts.
                        ; If you have other code below the interrupt routine you
                        ; must re-enable interrupts.

mytimer1:
    w_cnt = w_cnt + 1               ; increment counter at interrupt
RESUME                              ; return to location before interrupt
```

Important Note

No arguments are used in the example 2 program. Without arguments this command becomes a compile time directive that disables interrupt processing until another enable command is used. This is why it can be on a line that the program will never actually access. The same is true for *enable* without arguments.

BS2 Compatibility

BASIC
for PIC microcontrollers

Differences

Mbasic is a near 99% BS2 compatible. There are however several small difference which are due to some command additions and syntax changes to Mbasic. The Mbasic instruction set is a super set, all BS2 commands are implemented as they would be with PBasic. Included are several other useful commands that make programming much simpler.

Program Storage

Mbasic does not use “Slots” to store programs. The PICmicro uses a flat memory model. All program memory can be filled with one program. There is no need for storing or swapping variables. Any program code in your BS2 program that deals with swapping to the next 2K program slot can be removed. In several cases it may be easier to rewrite the program from scratch since allot of program code on the BS2 is required to deal with the bank swapping.

DATA and EEPROM

The BS2 allows nonvolatile data storage to space that has not been used for program storage. When using a PICmicro only data will be read from and written to the internal EEPROM.

Gosub...Return

The Gosub...Return commands are used for subroutines. The Basic Stamp is limited to the amount of Gosubs it can use. There is no limit to the amount of Gosub commands that can be used with Mbasic. The default stack size for Mbasic is 20. To use more GOSUB commands in your program you will need to increase the stack size. This can be done by adding the following line to your program:

```
    Stack = n
```

N being the stack size such as 20 or 30 and so on. Increasing the stack size will increase the amount of system ram Mbasic will need. Thus reducing the amount of user ram.

Converting Basic Stamp II Program

There are usually only minor revisions you must make to your BS2 program although this may not always be the case. Some issues such as the DIRS values being reversed with Mbasic will need to be addressed when porting programs over. When converting your BS2 program any references to the DEBUG command will need to be removed and replaced with Mbasic Debug command format. Before attempting to convert a program from the BS2 make sure to read the entire manual and become familiar with Mbasic syntax first. Try some simple programs first, before porting any programs over.

DIRS

On the BS2 DIRS = %0000000000000000 sets all the pins to outputs and DIRS = %1111111111111111 sets all the pins to inputs. Mbasic the DIRS is reversed. On the PICmicro DIRS = %1111111111111111 would set all the pins to outputs.

SERIN / SEROUT Timings

The BS2sx and BS2p have a bitrate that is calculated based on a .4us clock. The BS2/2e and Mbasic have a bitrate calculated based on a 1us clock. This means that bitrates calculated for the 2sx will not be correct for the 2/2e or Mbasic. All values should be recalculated using the methods for the BS2 and BS2e. There are several predefined labels for the bit rate such as i9600 and n9600. These predefined handle all the calculations for you.

LCD Commands

The LCD commands between Mbasic and BS2p are completely different. This is because the Mbasic LCD commands were written before the BS2p was available. The Mbasic LCD command allows multiple LCD's to be connected to the PICmicro.

Trouble Shooting

BASIC

for PIC microcontrollers

My Program Won’t Run ?

Often problems occur when the correct syntax is used but the wrong configuration values have been used. Make sure you have selected the correct PICmicro, Mhz setting and config value. LVP should always be off. High Speed Osc should be selected.

During the compiling process the compiler only checks for syntax errors and improper command usage. The compiler has no way to check your actual coding and determine if it will run or not. So check your code for any problems. Sometimes it is a good idea to comment out areas of your program and run little pieces of it. As each piece is checked, see if it works correctly, then move onto the next piece. Keep doing this until you come across any part of your code that does not work as expected.

You can also use the ICD to help debug your code. The ICD is actually the best method when debugging code.

My Program Still Won’t Run ?

Other common problems that may cause your program not to run are parts in backwards or not installed on the correct pins specified in your basic program. Make sure to check all your connections and part orientation if your program does not run.

Error trying to Program the PICmicro ?

If you receive an error when attempting to program the PICmicro go over the following check list:

1. Does the ISP-PRO have power ?
2. Is the PICmicro orientation correct in the circuit ?
3. Is the serial cable a straight through used for programming ?
4. Is the computer your using capable of 115K Baud ?
5. If your not using a development board is your circuit correct ?

Reserved Words

BASIC

for PIC microcontrollers

Reserved Words

There are many reserved words which can not be used as labels, constants or variables. All command names are reserved words. The table below lists all the reserved types and words.

It is not a good idea to use one character for a variable, label or constant. Using one character in the long run will make your code difficult to understand. There is no difference in code size when using one character or two. Numbers are not reserved words, but no variable, label or constant can start with a number otherwise a compiler error will result since the compiler is expecting an expression.

ACKDT
ACKEN
ACKSTAT
ADCON0
ADCON1
ADCS0
ADCS1
ADDCF
ADDDCF
ADDEN
ADDLW
ADDWF
ADFM
ADIN
ADON
ADRESH
ADRESL
AD_LNEG
AD_LON
AD_LPOS
AD_RNEG
AD_RON
AD_RPOS
ANDLW
ANDWF
AX0
AX1
AX2
AX3
B
BANKISEL
BANKSEL
BC
BCF
BDC
BF
BIT
BNC
BNDC
BNZ
BRGH
BSF
BTFSC
BTFSS
BUTTON
BYTE
BYTETABLE
BZ
C
CALL
CAPTURE16L2H",0x07);
CAPTURE1H2L",0x04);
CAPTURE1L2H",0x05);
CAPTURE4L2H",0x06);
CAPTUREOFF",0x00);
CBLOCK
CCP1CON
CCP1M0
CCP1M1
CCP1M2
CCP1M3
CCP1X
CCP1Y
CCP2CON
CCP2M0
CCP2M1
CCP2M2
CCP2M3
CCP2X
CCP2Y
CCPR1H
CCPR1L

CCPR2H
CCPR2L
CGRAM
CHS0
CHS1
CHS2
CKE
CKP
CLEAR
CLEAR
CLRC
CLRDC
CLRF
CLRW
CLRWDT
CLRZ
CODE
COMF
COMPAREINT",0x0A);
COMPAREOFF",0x00);
COMPARESETHIGH",0x08);
COMPARESETLOW",0x09);
COMPARESPECIAL",0x0B);
CONSTANT
COUNT
CREN
CSRC
CURLEFT
CURRIGHT
D
DA
DATA_ADDRESS
DB
DC
DE
DEBUG
DEBUGIN
DECCUR
DECF
DECFSZ
DECSCR
DIR0
DIR1
DIR10
DIR11
DIR12
DIR13
DIR14
DIR15
DIR16
DIR17
DIR18
DIR19
DIR2
DIR20
DIR21
DIR22
DIR23
DIR24
DIR25
DIR26
DIR27
DIR28
DIR29
DIR3
DIR30
DIR31
DIR4
DIR5
DIR6
DIR7
DIR8
DIR9
DIRA
DIRB
DIRC
DIRD
DIRE
DIRF
DIRH
DIRL
DIRM
DIRS
DISABLE
DO
DT
DTMFOUT
DTMFOUT2
DW
D_A
EEADR
EECON1
EECON2
EEDATA
ELSE
ELSEIF

ENABLE
END
ENDC
ENDIF
ENDM
EQU
ERROR
ERRORLEVEL
EXITM
EXPAND
EXTERN
EXT_H2L",0x00);
EXT_L2H",0x40);
FASTLSBPOST", 0x7);
FASTLSBPRE", 0x5);
FASTMSBPOST", 0x6);
FASTMSBPRE", 0x4);
FERR
FILL
FLOATTABLE
FOR
FREQOUT
GCEN
GETCAPTURE
GETWATCHDOG
GLOBAL
GO
GOSUB
GOTO
GOTO
GO_DONE
H115200
H1200
H12000
H1250000
H14400
H16800
H19200
H21600
H2400
H24000
H250000
H26400
H28800
H300
H31200
H312500
H33600
H36000
H38400
H4800
H57600
H600
H625000
H7200
H9600
HIGH
HOME
HPWM
HSERIN
HSEROUT
I115200
I1200
I12000
I14400
I16800
I19200
I21600
I2400
I24000
I26400
I28800
I2CIN
I2COUT
I2C_DATA
I2C_READ
I2C_START
I2C_STOP
I300
I31200
I33600
I36000
I38400
I4800
I57600
I600
I7200
I9600
IBF
IBOV
IDATA
IE115200
IE1200
IE12000

IE14400;
IE16800
IE19200
IE21600
IE2400
IE24000
IE26400
IE28800
IE300
IE31200
IE33600
IE36000
IE38400
IE4800
IE57600
IE600
IE7200
IE9600
IEMODE
IEO115200
IEO1200
IEO120000
IEO14400
IEO16800
IEO19200
IEO21600
IEO2400
IEO24000
IEO26400
IEO28800
IEO300
IEO31200
IEO33600
IEO36000
IEO38400
IEO4800
IEO57600
IEO600
IEO7200
IEO9600
IEOMODE
IF
IFDEF
IFNDEF
IMODE
IN0
IN1
IN10
IN11
IN12
IN13
IN14
IN15
IN16
IN17
IN18
IN19
IN2
IN20
IN21
IN22
IN23
IN24
IN25
IN26
IN27
IN28
IN29
IN3
IN30
IN31
IN4
IN5
IN6
IN7
IN8
IN9
INA
INB
INC
INCCUR
INCF
INCFSZ
INCSCR
IND
INE
INF
INH
INITLCD1
INITLCD2
INL
INM
INPUT
INS

INTEDG
IO115200
IO1200
IO12000
IO14400
IO16800
IO19200
IO21600
IO2400
IO24000
IO26400
IO28800
IO300
IO31200
IO33600
IO36000
IO38400
IO4800
IO57600
IO600
IO7200
IO9600
IOMODE
IORLW
IORWF
IRP
LCALL
LCDREAD
LCDWRITE
LGOTO
LIST
LOCAL
LONG
LONGTABLE
LOOKDOWN
LOOKUP
LOW
LSBFIRST", 0x1);
LSBPOST", 0x3);
LSBPRE", 0x1);
MACRO
MESSG
MOVF
MOVFW
MOVLW
MOVWF
MSBFIRST", 0x0);
MSBPOST", 0x2);
MSBPRE", 0x0);
N115200
N1200
N12000
N14400
N16800
N19200
N21600
N2400
N24000
N26400
N28800
N300
N31200
N33600
N36000
N38400
N4800
N57600
N600
N7200
N9600
NAP
NE115200
NE1200
NE12000
NE14400;
NE16800
NE19200
NE21600
NE2400
NE24000
NE26400
NE28800
NE300
NE31200
NE33600
NE36000
NE38400
NE4800
NE57600
NE600
NE7200
NE9600
NEGF
NEMODE

NEO115200
NEO1200
NEO120000
NEO14400
NEO16800
NEO19200
NEO21600
NEO2400
NEO24000
NEO26400
NEO28800
NEO300
NEO31200
NEO33600
NEO36000
NEO38400
NEO4800
NEO57600
NEO600
NEO7200
NEO9600
NEOMODE
NEXT
NIB
NMODE
NO115200
NO1200
NO12000
NO14400;
NO16800
NO19200
NO21600
NO2400
NO24000
NO26400
NO28800
NO300
NO31200
NO33600
NO36000
NO38400
NO4800
NO57600
NO600
NO7200
NO9600
NOEXPAND
NOLIST
NOMODE
NOP
NOT_A
NOT_ADDRESS
NOT_BO
NOT_BOR
NOT_DONE
NOT_PD
NOT_POR
NOT_RBPU
NOT_RC8
NOT_T1SYNC
NOT_TO
NOT_TX8
NOT_W
NOT_WRITE
OBF
OERR
OFF
ONBOR
ONELINE
ONELINE5X11
ONINTERRUPT
ONMOR
ONPOR
OPTION
OPTION_REG
ORG
OUT0
OUT1
OUT10
OUT11
OUT12
OUT13
OUT14
OUT15
OUT16
OUT17
OUT18
OUT19
OUT2
OUT20
OUT21
OUT22
OUT23
OUT24
OUT25

OUT26
OUT27
OUT28
OUT29
OUT3
OUT30
OUT31
OUT4
OUT5
OUT6
OUT7
OUT8
OUT9
OUTA
OUTB
OUTC
OUTD
OUTE
OUTF
OUTH
OUTL
OUTM
OUTPUT
OUTS
OWIN
OWOUT
P
P0
P1
P10
P11
P12
P13
P14
P15
P16
P17
P18
P19
P2
P20
P21
P22
P23
P24
P25
P26
P27
P28
P29
P3
P30
P31
P4
P5
P6
P7
P8
P9
PAGE
PAGESEL
PAUSE
PAUSECLK
PAUSEUS
PCFG0
PCFG1
PCFG2
PCFG3
PCL
PCON
PEEK
PEN
POKE
PORTA
PORTB
PORTC
PORTD
PORTE
PR2
PROCESSOR
PS0
PS1
PS2
PSA
PSPMODE
PULSIN
PULSOUT
PU_OFF",0x80);
PU_ON",0x00);
PWM
R
RADIX
RC8_9
RC9

RCD8
RCEN
RCREG
RCSTA
RCTIME
RD
READ
READDM
READPM
READ_WRITE
REPEAT
RES
RESETTMR1
RESUME
RETFIE
RETLW
RETURN
RETURN
REVERSE
RLF
RP0
RP1
RRF
RSEN
RX9
RX9D
R_W
S
SBYTE
SCR
SCRBLK
SCRCUR
SCRCURBLK
SCRLEFT
SCRRAM
SCRRIGHT
SEN
SERDETECT
SERIN
SEROUT
SERVO
SET
SETC
SETCAPTURE
SETCOMPARE
SETDC
SETEXTINT
SETHSERIAL
SETPULLUPS
SETTMR0
SETTMR1
SETTMR2
SETZ
SHIFTIN
SHIFTOUT
SKPDC
SKPNC
SKPNDC
SKPNZ
SKPZ
SLEEP
SLEEP
SMP
SOUND
SOUND2
SPACE
SPBRG
SPEN
SPMOTOR
SREN
SSPADD
SSPBUF
SSPCON
SSPCON2
SSPEN
SSPM0
SSPM1
SSPM2
SSPM3
SSPOV
SSPSTAT
STATUS
STEP
STOP
SUBCF
SUBDCF
SUBLW
SUBTITLE
SUBWF
SWAP
SWAPF
SWORD
SYNC
S_IN

```
S_OUT
T0CS
T0SE
T1CKPS0
T1CKPS1
T1CON
T1INSYNC
T1OSCEN
T1SYNC
T2CKPS0
T2CKPS1
T2CON
THEN
TIMEWATCHDOG
TITLE
TMR0
TMR0EXTH1",0x38);
TMR0EXTH128",0x36);
TMR0EXTH16",0x33);
TMR0EXTH2",0x30);
TMR0EXTH256",0x37);
TMR0EXTH32",0x34);
TMR0EXTH4",0x31);
TMR0EXTH64",0x35);
TMR0EXTH8",0x32);
TMR0EXTL1",0x28);
TMR0EXTL128",0x26);
TMR0EXTL16",0x23);
TMR0EXTL2",0x20);
TMR0EXTL256",0x27);
TMR0EXTL32",0x24);
TMR0EXTL4",0x21);
TMR0EXTL64",0x25);
TMR0EXTL8",0x22);
TMR0INT1",0x08);
TMR0INT128",0x06);
TMR0INT16",0x03);
TMR0INT2",0x00);
TMR0INT256",0x07);
TMR0INT32",0x04);
TMR0INT4",0x01);
TMR0INT64",0x05);
TMR0INT8",0x02);
TMR1ASYNC1",0x0B);
TMR1ASYNC2",0x1B);
TMR1ASYNC4",0x2B);
TMR1ASYNC8",0x3B);
TMR1CS
TMR1EXT1",0x07);
TMR1EXT2",0x17);
TMR1EXT4",0x27);
TMR1EXT8",0x37);
TMR1H
TMR1INT1",0x01);
TMR1INT2",0x81);
TMR1INT4",0x21);
TMR1INT8",0x31);
TMR1L
TMR1OFF",0x00);
TMR1ON
TMR2
TMR2OFF",0x00);
TMR2ON
TMR2PRE16POST1",0x07);
TMR2PRE16POST10",0x97);
TMR2PRE16POST11",0xa7);
TMR2PRE16POST12",0xb7);
TMR2PRE16POST13",0xc7);
TMR2PRE16POST14",0xd7);
TMR2PRE16POST15",0xe7);
TMR2PRE16POST16",0xf7);
TMR2PRE16POST2",0x17);
TMR2PRE16POST3",0x27);
TMR2PRE16POST4",0x37);
TMR2PRE16POST5",0x47);
TMR2PRE16POST6",0x57);
TMR2PRE16POST7",0x67);
TMR2PRE16POST8",0x77);
TMR2PRE16POST9",0x87);
TMR2PRE1POST1",0x04);
TMR2PRE1POST10",0x94);
TMR2PRE1POST11",0xa4);
TMR2PRE1POST12",0xb4);
TMR2PRE1POST13",0xc4);
TMR2PRE1POST14",0xd4);
TMR2PRE1POST15",0xe4);
TMR2PRE1POST16",0xf4);
TMR2PRE1POST2",0x14);
TMR2PRE1POST3",0x24);
TMR2PRE1POST4",0x34);
TMR2PRE1POST5",0x44);
TMR2PRE1POST6",0x54);
TMR2PRE1POST7",0x64);
TMR2PRE1POST8",0x74);
```

TMR2PRE1POST9",0x84);
TMR2PRE4POST1",0x05);
TMR2PRE4POST10",0x95);
TMR2PRE4POST11",0xa5);
TMR2PRE4POST12",0xb5);
TMR2PRE4POST13",0xc5);
TMR2PRE4POST14",0xd5);
TMR2PRE4POST15",0xe5);
TMR2PRE4POST16",0xf5);
TMR2PRE4POST2",0x15);
TMR2PRE4POST3",0x25);
TMR2PRE4POST4",0x35);
TMR2PRE4POST5",0x45);
TMR2PRE4POST6",0x55);
TMR2PRE4POST7",0x65);
TMR2PRE4POST8",0x75);
TMR2PRE4POST9",0x85);
TOGGLE
TOUTPS0
TOUTPS1
TOUTPS2
TOUTPS3
TRIS
TRISA
TRISB
TRISC
TRISD
TRISE
TRISE0
TRISE1
TRISE2
TRMT
TSTF
TWOLINE
TX8_9
TX9
TX9D
TXD8
TXEN
TXREG
TXSTA
UA
UDATA
UDATA_ACS
UDATA_OVR
UDATA_SHR
UNTIL
UPPER
VARIABLE
WCOL
WDTPS1",0x08);
WDTPS128",0x0F);
WDTPS16",0x0C);
WDTPS2",0x09);
WDTPS32",0x0D);
WDTPS4",0x0A);
WDTPS64",0x0E);
WDTPS8",0x0B);
WEND
WHILE
WORD
WORDTABLE
WR
WREN
WRERR
WRITE
WRITEDM
WRITEPM
XIN
XORLW
XORWF
XOUT
X_1",0x0C);
X_10",0x1E);
X_11",0x06);
X_12",0x16);
X_13",0x00);
X_14",0x10);
X_15",0x08);
X_16",0x18);
X_2",0x1C);
X_3",0x04);
X_4",0x14);
X_5",0x02);
X_6",0x12);
X_7",0x0A);
X_8",0x1A);
X_9",0x0E);
X_A",0x6);
X_B",0xE);
X_Bright
X_C",0x2);
X_D",0xA);
X_Dim

X_E",0x1);
X_F",0x9);
X_G",0x5);
X_H",0xD);
X_Hail
X_I",0x7);
X_J",0xF);
X_K",0x3);
X_L",0xB);
X_Lights_Off
X_Lights_On
X_M",0x0);
X_N",0x8);
X_O",0x4);
X_Off
X_On
X_P",0xC);
X_Status_Off
X_Status_On
X_Status_Request
X_Units_On
Z

All math functions
Any name starting with a "_"
Any name starting with a number

Appendix – A

BASIC

for PIC microcontrollers

Information Sources

PICmicro Data Sheets

Microchip Technology Inc.
http://www.microchip.com

MBasic Support

Basic Micro Inc.
http://www.basicmicro.com

Appendix – C

BASIC

for PIC microcontrollers

ASCII Chart 0 To 255

Dec	Hex	Oct	Bin	Typed	Char
0	0x0	00	00000000	^@	NUL
1	0x1	01	00000001	^A	SOH
2	0x2	02	00000010	^B	STX
3	0x3	03	00000011	^C	ETX
4	0x4	04	00000100	^D	EOT
5	0x5	05	00000101	^E	ENQ
6	0x6	06	00000110	^F	ACK
7	0x7	07	00000111	^G	BEL
8	0x8	010	00001000	^H	BS
9	0x9	011	00001001	^I	HT
10	0xA	012	00001010	^J	LF
11	0xB	013	00001011	^K	VT
12	0xC	014	00001100	^L	FF
13	0xD	015	00001101	^M	CR
14	0xE	016	00001110	^N	SO
15	0xF	017	00001111	^O	SI
16	0x10	020	00010000	^P	DLE
17	0x11	021	00010001	^Q	DC1
18	0x12	022	00010010	^R	DC2
19	0x13	023	00010011	^S	DC3
20	0x14	024	00010100	^T	DC4
21	0x15	025	00010101	^U	NAK
22	0x16	026	00010110	^V	SYN
23	0x17	027	00010111	^W	ETB
24	0x18	030	00011000	^X	CAN
25	0x19	031	00011001	^Y	EM
26	0x1A	032	00011010	^Z	SUB
27	0x1B	033	00011011	^[	ESC
28	0x1C	034	00011100	^\	FS
29	0x1D	035	00011101	^]	GS
30	0x1E	036	00011110	^^	RS
31	0x1F	037	00011111	^_	US
32	0x20	040	00100000	Space	SPC
33	0x21	041	00100001	!	!
34	0x22	042	00100010	"	"
35	0x23	043	00100011	#	#
36	0x24	044	00100100	$	$
37	0x25	045	00100101	%	%
38	0x26	046	00100110	&	&
39	0x27	047	00100111	'	'
40	0x28	050	00101000	(	(
41	0x29	051	00101001	)	)
42	0x2A	052	00101010	*	*
43	0x2B	053	00101011	+	+
44	0x2C	054	00101100	,	,

Dec	Hex	Oct	Bin	Typed	Char
45	0x2D	055	00101101	-	-
46	0x2E	056	00101110	.	.
47	0x2F	057	00101111	/	/
48	0x30	060	00110000	0	0
49	0x31	061	00110001	1	1
50	0x32	062	00110010	2	2
51	0x33	063	00110011	3	3
52	0x34	064	00110100	4	4
53	0x35	065	00110101	5	5
54	0x36	066	00110110	6	6
55	0x37	067	00110111	7	7
56	0x38	070	00111000	8	8
57	0x39	071	00111001	9	9
58	0x3A	072	00111010	:	:
59	0x3B	073	00111011	;	;
60	0x3C	074	00111100	<	<
61	0x3D	075	00111101	=	=
62	0x3E	076	00111110	>	>
63	0x3F	077	00111111	?	?
64	0x40	0100	01000000	@	@
65	0x41	0101	01000001	A	A
66	0x42	0102	01000010	B	B
67	0x43	0103	01000011	C	C
68	0x44	0104	01000100	D	D
69	0x45	0105	01000101	E	E
70	0x46	0106	01000110	F	F
71	0x47	0107	01000111	G	G
72	0x48	0110	01001000	H	H
73	0x49	0111	01001001	I	I
74	0x4A	0112	01001010	J	J
75	0x4B	0113	01001011	K	K
76	0x4C	0114	01001100	L	L
77	0x4D	0115	01001101	M	M
78	0x4E	0116	01001110	N	N
79	0x4F	0117	01001111	O	O
80	0x50	0120	01010000	P	P
81	0x51	0121	01010001	Q	Q
82	0x52	0122	01010010	R	R
83	0x53	0123	01010011	S	S
84	0x54	0124	01010100	T	T
85	0x55	0125	01010101	U	U
86	0x56	0126	01010110	V	V
87	0x57	0127	01010111	W	W
88	0x58	0130	01011000	X	X
89	0x59	0131	01011001	Y	Y
90	0x5A	0132	01011010	Z	Z

Dec	Hex	Oct	Bin	Typed	Char
91	0x5B	0133	01011011	[	[
92	0x5C	0134	01011100	\	\
93	0x5D	0135	01011101	]	]
94	0x5E	0136	01011110	^	^
95	0x5F	0137	01011111	_	_
96	0x60	0140	01100000	`	`
97	0x61	0141	01100001	a	a
98	0x62	0142	01100010	b	b
99	0x63	0143	01100011	c	c
100	0x64	0144	01100100	d	d
101	0x65	0145	01100101	e	e
102	0x66	0146	01100110	f	f
103	0x67	0147	01100111	g	g
104	0x68	0150	01101000	h	h
105	0x69	0151	01101001	i	i
106	0x6A	0152	01101010	j	j
107	0x6B	0153	01101011	k	k
108	0x6C	0154	01101100	l	l
109	0x6D	0155	01101101	m	m
110	0x6E	0156	01101110	n	n
111	0x6F	0157	01101111	o	o
112	0x70	0160	01110000	p	p
113	0x71	0161	01110001	q	q
114	0x72	0162	01110010	r	r
115	0x73	0163	01110011	s	s
116	0x74	0164	01110100	t	t
117	0x75	0165	01110101	u	u
118	0x76	0166	01110110	v	v
119	0x77	0167	01110111	w	w
120	0x78	0170	01111000	x	x
121	0x79	0171	01111001	y	y
122	0x7A	0172	01111010	z	z
123	0x7B	0173	01111011	{	{
124	0x7C	0174	01111100	|	|
125	0x7D	0175	01111101	}	}
126	0x7E	0176	01111110	~	~
127	0x7F	0177	01111111	Del	Del
128	0x80	0200	10000000	M-^@	M-^@
129	0x81	0201	10000001	M-^A	M-^A
130	0x82	0202	10000010	M-^B	M-^B
131	0x83	0203	10000011	M-^C	M-^C
132	0x84	0204	10000100	M-^D	M-^D
133	0x85	0205	10000101	M-^E	M-^E
134	0x86	0206	10000110	M-^F	M-^F
135	0x87	0207	10000111	M-^G	M-^G
136	0x88	0210	10001000	M-^H	M-^H
137	0x89	0211	10001001	M-^I	M-^I

Dec	Hex	Oct	Bin	Typed	Char
138	0x8A	0212	10001010	M-^J	M-^J
139	0x8B	0213	10001011	M-^K	M-^K
140	0x8C	0214	10001100	M-^L	M-^L
141	0x8D	0215	10001101	M-^M	M-^M
142	0x8E	0216	10001110	M-^N	M-^N
143	0x8F	0217	10001111	M-^O	M-^O
144	0x90	0220	10010000	M-^P	M-^P
145	0x91	0221	10010001	M-^Q	M-^Q
146	0x92	0222	10010010	M-^R	M-^R
147	0x93	0223	10010011	M-^S	M-^S
148	0x94	0224	10010100	M-^T	M-^T
149	0x95	0225	10010101	M-^U	M-^U
150	0x96	0226	10010110	M-^V	M-^V
151	0x97	0227	10010111	M-^W	M-^W
152	0x98	0230	10011000	M-^X	M-^X
153	0x99	0231	10011001	M-^Y	M-^Y
154	0x9A	0232	10011010	M-^Z	M-^Z
155	0x9B	0233	10011011	M-^[	M-^[
156	0x9C	0234	10011100	M-^\	M-^\
157	0x9D	0235	10011101	M-^]	M-^]
158	0x9E	0236	10011110	M-^^	M-^^
159	0x9F	0237	10011111	M-^_	M-^_
160	0xA0	0240	10100000	M-	M-
161	0xA1	0241	10100001	M-!	M-!
162	0xA2	0242	10100010	M-"	M-"
163	0xA3	0243	10100011	M-#	M-#
164	0xA4	0244	10100100	M-$	M-$
165	0xA5	0245	10100101	M-%	M-%
166	0xA6	0246	10100110	M-&	M-&
167	0xA7	0247	10100111	M-'	M-'
168	0xA8	0250	10101000	M-(	M-(
169	0xA9	0251	10101001	M-)	M-)
170	0xAA	0252	10101010	M-*	M-*
171	0xAB	0253	10101011	M-+	M-+
172	0xAC	0254	10101100	M-,	M-,
173	0xAD	0255	10101101	M--	M--
174	0xAE	0256	10101110	M-.	M-.
175	0xAF	0257	10101111	M-/	M-/
176	0xB0	0260	10110000	M-0	M-0
177	0xB1	0261	10110001	M-1	M-1
178	0xB2	0262	10110010	M-2	M-2
179	0xB3	0263	10110011	M-3	M-3
180	0xB4	0264	10110100	M-4	M-4
181	0xB5	0265	10110101	M-5	M-5
182	0xB6	0266	10110110	M-6	M-6
183	0xB7	0267	10110111	M-7	M-7
184	0xB8	0270	10111000	M-8	M-8

Dec	Hex	Oct	Bin	Typed	Char
185	0xB9	0271	10111001	M-9	M-9
186	0xBA	0272	10111010	M-:	M-:
187	0xBB	0273	10111011	M-;	M-;
188	0xBC	0274	10111100	M-<	M-<
189	0xBD	0275	10111101	M-=	M-=
190	0xBE	0276	10111110	M->	M->
191	0xBF	0277	10111111	M-?	M-?
192	0xC0	0300	11000000	M-@	M-@
193	0xC1	0301	11000001	M-A	M-A
194	0xC2	0302	11000010	M-B	M-B
195	0xC3	0303	11000011	M-C	M-C
196	0xC4	0304	11000100	M-D	M-D
197	0xC5	0305	11000101	M-E	M-E
198	0xC6	0306	11000110	M-F	M-F
199	0xC7	0307	11000111	M-G	M-G
200	0xC8	0310	11001000	M-H	M-H
201	0xC9	0311	11001001	M-I	M-I
202	0xCA	0312	11001010	M-J	M-J
203	0xCB	0313	11001011	M-K	M-K
204	0xCC	0314	11001100	M-L	M-L
205	0xCD	0315	11001101	M-M	M-M
206	0xCE	0316	11001110	M-N	M-N
207	0xCF	0317	11001111	M-O	M-O
208	0xD0	0320	11010000	M-P	M-P
209	0xD1	0321	11010001	M-Q	M-Q
210	0xD2	0322	11010010	M-R	M-R
211	0xD3	0323	11010011	M-S	M-S
212	0xD4	0324	11010100	M-T	M-T
213	0xD5	0325	11010101	M-U	M-U
214	0xD6	0326	11010110	M-V	M-V
215	0xD7	0327	11010111	M-W	M-W
216	0xD8	0330	11011000	M-X	M-X
217	0xD9	0331	11011001	M-Y	M-Y
218	0xDA	0332	11011010	M-Z	M-Z
219	0xDB	0333	11011011	M-[	M-[
220	0xDC	0334	11011100	M-\	M-\
221	0xDD	0335	11011101	M-]	M-]
222	0xDE	0336	11011110	M-^	M-^
223	0xDF	0337	11011111	M-_	M-_
224	0xE0	0340	11100000	M-`	M-`
225	0xE1	0341	11100001	M-a	M-a
226	0xE2	0342	11100010	M-b	M-b
227	0xE3	0343	11100011	M-c	M-c
228	0xE4	0344	11100100	M-d	M-d
229	0xE5	0345	11100101	M-e	M-e
230	0xE6	0346	11100110	M-f	M-f
231	0xE7	0347	11100111	M-g	M-g

Dec	Hex	Oct	Bin	Typed	Char
232	0xE8	0350	11101000	M-h	M-h
233	0xE9	0351	11101001	M-i	M-i
234	0xEA	0352	11101010	M-j	M-j
235	0xEB	0353	11101011	M-k	M-k
236	0xEC	0354	11101100	M-l	M-l
237	0xED	0355	11101101	M-m	M-m
238	0xEE	0356	11101110	M-n	M-n
239	0xEF	0357	11101111	M-o	M-o
240	0xF0	0360	11110000	M-p	M-p
241	0xF1	0361	11110001	M-q	M-q
242	0xF2	0362	11110010	M-r	M-r
243	0xF3	0363	11110011	M-s	M-s
244	0xF4	0364	11110100	M-t	M-t
245	0xF5	0365	11110101	M-u	M-u
246	0xF6	0366	11110110	M-v	M-v
247	0xF7	0367	11110111	M-w	M-w
248	0xF8	0370	11111000	M-x	M-x
249	0xF9	0371	11111001	M-y	M-y
250	0xFA	0372	11111010	M-z	M-z
251	0xFB	0373	11111011	M-{	M-{
252	0xFC	0374	11111100	M-|	M-|
253	0xFD	0375	11111101	M-}	M-}
254	0xFE	0376	11111110	M-~	M-~
255	0xFF	0377	11111111	M-^?	M-^?

Index