

# **BASIC MICRO STUDIO SYNTAX MANUAL**



---

## Warranty

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

## Copyrights and Trademarks

Copyright© 2009-2011 by Basic Micro, Inc. All rights reserved. "MBasic", "BasicATOM Pro", "The Atom", "BasicATOM", "BasicATOM Nano" and "Basic Micro" are registered trademarks of Basic Micro Inc. Other trademarks mentioned are registered trademarks of their respective holders.

## 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](mailto:sales@basicmicro.com)  
Tech support: [support@basicmicro.com](mailto:support@basicmicro.com)  
Web: <http://www.basicmicro.com>

## Discussion List

A web based discussion board is maintained at <http://forums.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](mailto:support@basicmicro.com) or via our web site.

---

---

<b>Introduction .....</b>	<b>7</b>
What is the BasicATOM? .....	7
Programming Language.....	7
Software .....	7
Resources.....	7
<b>Hardware Introduction .....</b>	<b>9</b>
Module Hardware Comparison.....	9
Chip Hardware Comparison.....	10
Instructions Per Second(IPS) .....	10
Nano 8.....	11
Nano 18 .....	11
Nano 28 .....	12
Nano 40 .....	13
Nano 28X .....	14
Nano 40X .....	15
BasicATOM 24m.....	16
BasicATOM 28m.....	16
BasicATOM 40m.....	17
BasicATOM Pro ONEm .....	18
BasicATOM Pro 24m .....	18
BasicATOM Pro 28m .....	19
BasicATOM Pro 40m .....	19
<b>Quick Start Guide .....</b>	<b>21</b>
First Program.....	26
Downloading The Program.....	26
Blinking LED .....	30
<b>Variables.....</b>	<b>32</b>
Variable Types .....	32
Variable Names.....	33
Variable Modifiers.....	33
Variable Arrays.....	34
<b>Constants .....</b>	<b>37</b>
Constant Tables .....	37
<b>Pins and Ports .....</b>	<b>39</b>
Pin Constants .....	39
Pin Variables.....	39
Pin and Port IN/OUT Variable Names.....	40
Direction Variables .....	41
Input Output .....	41
Pin and Port DIR Variables Names .....	42
<b>Math .....</b>	<b>44</b>
Number Bases .....	44
Math and Operators .....	44
Operators.....	45
Precedence Table .....	46

---

- (Negative)	47
ABS	47
SIN, COS	48
DCD	49
NCD	49
SQR (Square Root)	50
BIN2BCD, BCD2BIN	51
RANDOM	51
- (Subtraction)	52
+ (Addition)	52
* (Multiplication)	52
/ (Division)	53
** (High Multiplication)	53
*/ (Fractional Multiplication)	54
MAX	55
MIN	55
DIG	55
REV	55
<< (Shift Left)	56
>> (Shift Right)	56
& (AND)	57
(OR)	57
^ (Exclusive OR)	58
&/ (AND NOT)	58
/ (OR NOT)	58
^/ (XOR NOT)	59
= (Equal)	59
<> (NOT Equal To)	59
< (Less Than)	59
> (Greater Than)	60
>= (Greater Than or Equal To)	60
AND	60
OR	61
XOR	61
NOT	62
<b>Floating Point Math</b>	<b>63</b>
Floating Point Operators	63
TOINT	63
TOFLOAT	63
FSQRT	64
FSIN	64
FCOS	64
FTAN	64
FASIN	65
FACOS	65
FATAN	65
FLN	65
FEXP	66
Hyperbolic Functions	66
FSINH	66

---

FCOSH .....	66
FTANH .....	67
FATANH.....	67
<b>Modifiers.....</b>	<b>69</b>
DEC .....	71
SDEC .....	72
HEX .....	73
SHEX .....	74
IHEX .....	75
ISHEX .....	76
BIN.....	77
SBIN.....	78
IBIN.....	79
ISBIN.....	80
REP.....	80
REAL.....	81
STR .....	82
SKIP .....	82
WAIT .....	82
WAITSTR .....	82
<b>Command Reference .....</b>	<b>84</b>
ADIN.....	85
ADIN16 .....	87
BRANCH.....	88
BUTTON .....	90
CLEAR.....	93
COUNT.....	94
DEBUG .....	95
DEBUGIN .....	97
DO - WHILE.....	99
DTMFOUT .....	101
DTMFOUT2.....	103
END .....	105
EXCEPTION .....	106
FATAN2 .....	107
FOR...NEXT .....	108
FREQOUT .....	110
GOSUB.....	112
GOTO.....	114
HIGH .....	115
HPWM.....	116
HSERIN.....	118
HSEROUT.....	122
HSERVO .....	123
IF...THEN...ELSEIF...ELSE...ENDIF .....	125
INPUT .....	129
I2COUT.....	130
I2CIN.....	133
LCDINIT.....	136

LCDWRITE .....	138
LCDREAD .....	143
LOOKDOWN .....	145
LOOKUP .....	147
LOW .....	149
NAP .....	150
OWIN.....	151
OWOUT.....	153
OUTPUT .....	155
PAUSE.....	156
PAUSEUS.....	157
PAUSECLK .....	158
PULSIN .....	160
PULSOUT .....	162
PWM.....	164
RCTIME.....	166
READ .....	169
READDM .....	170
REPEAT - UNTIL.....	171
RETURN .....	172
REVERSE.....	174
SERIN .....	175
SEROUT .....	180
SERVO .....	185
SHIFTIN.....	186
SHIFTOUT.....	191
SLEEP .....	194
SOUND.....	195
SOUND2.....	197
STOP .....	198
SWAP.....	199
TOGGLE .....	200
WHILE - WEND.....	201
WRITE .....	203
WRITEDM .....	204
<b>Interrupts.....</b>	<b>206</b>
<b>Basic Stamp Conversion.....</b>	<b>211</b>
<b>Compiler Directives .....</b>	<b>218</b>
<b>Reserved Words .....</b>	<b>223</b>

## Introduction

You have now entered the exciting world of BasicATOMs. No other microcontroller on the market is as easy to learn yet still remains powerful enough to be under the hood of some of the most popular robots to be found on the internet. Some of the top rated Youtube robot videos are powered by a BasicATOM Pro.

### What is the BasicATOM?

The BasicATOM and BasicATOM Pro modules are self contained micro computers. Their power regulation, clock system and communication hardware are all built in. All you need to do is supply power and a connection to your computer. The Nano series is the core of the BasicATOM. Except it requires you to supply the power regulation and communication system. The clocking system is internal to the Nano series.

### Programming Language

MBasic is the programming language which is used to program the BasicATOM series. MBasic is based on a subset of BASIC. In general it is considered to be one of the most widely used language aside from C. The language itself has been around for years. It was the first product created by Microsoft in the early days for one of the first home built programmable computers called the Altair. The BasicATOM Pro does support C but the primary focus of this manual is centered around MBasic.

### Software

Basic Micro Studio is the main piece of software that you will be using to write your code for any of the Basic Micro products such as BasicATOMs or BasicATOM Pro modules. It is commonly referred to as an Integrated Development Environment or IDE for short. The IDE contains 3 main parts. A text editor for writing programs, a compiler to translate your program into something the microcontroller will understand and a loader to download your program to the microcontroller. There are several advance features that will greatly help in creating your program and are documented in this manual.

### Resources

There are several additional resources available when learning MBasic. There are forums available at [Basicmicro.com](http://Basicmicro.com). The forums offer a search function and most answers to your questions can be found. There are additional forums available at [lynxmotion.com](http://lynxmotion.com).





### Hardware Introduction

Lets introduce the hardware. All modules are pin compatible and mostly code compatible. Nanos are not pin compatible with modules but are code compatible. As you move up in the line, the capabilities increase. Some features are not backward compatible. The BasicATOM Nano is the beginning of the scale with BasicATOM in the middle and BasicATOM Pro at the top of the scale in regards to performance and capabilities. Product datasheet are available for download at Basicmicro.com.

### Module Hardware Comparison

Description	Atom24	Atom28	Atom40	ProONE	Pro24	Pro28	Pro40
Temp Range	0 - 70 C	0 - 70 C	0 - 70 C	0 - 70 C	0 - 70 C	0 - 70 C	0 - 70 C
Flash	14K	14K	14K	32K	32K	32K	56K
RAM	0368	368	368	2048	2048	2058	4096
EEPROM	256 Bytes	256 Bytes	256 Bytes	0 Bytes	0 Bytes	4096 Bytes	4096 Bytes
I/O	16 + 4	20	32	8	16	20	32
A/D	9	10	14	4	4	8	8
Clock	20MHz	20Mhz	20Mhz	16Mhz	16Mhz	16Mhz	20Mhz
IPS	33,000	33,000	33,000	100,000+	100,000+	100,000+	125,000+
Floating Point	YES	YES	YES	YES	YES	YES	YES
32 Bit Hardware	NO	NO	NO	YES	YES	YES	YES
32 Bit Software	YES	YES	YES	YES	YES	YES	YES
UARTS	1	1	1	1	1	1	2
PWM Hardware	2	2	2	3	3	3	6

## Chip Hardware Comparison

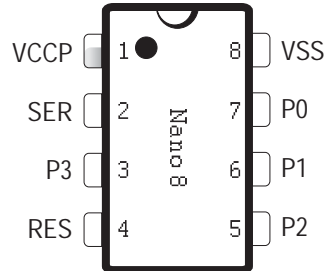
Description	Nano8	Nano18	Nano28	Nano40	Nano28X	Nano40X
Temp Range	0 - 70 C	0 - 70 C	0 - 70 C	0 - 70 C	0 - 70 C	0 - 70 C
Flash	3.5	7K	14K	14K	14K	14K
RAM	128	368	368	368	368	368
EEPROM	0 Bytes	256 Bytes	256 Bytes	256 Bytes	256 Bytes	256 Bytes
I/O	5	15	24	35	22	33
A/D	4	6	11	14	11	14
Clock	8MHZ	8MHz	8MHz	8Mhz	20MHz	20Mhz
IPS	13,000	13,000	13,000	13,000	33,000	33,000
Floating Point	YES	YES	YES	YES	YES	YES
32 Bit Hardware	NO	NO	NO	NO	NO	NO
32 Bit Software	YES	YES	YES	YES	YES	YES
UARTS	0	1	1	1	1	1
PWM Hardware	1	1	2	2	2	2

## Instructions Per Second(IPS)

The IPS specification for the modules is calculated differently on the BasicATOM and BasicATOM Pro modules. The BasicATOM IPS is the absolute maximum basic instructions per second. On the BasicATOM Pro the IPS is the average basic instructions per second of all the basic instructions that do not have inherent delays in them(pause, serin, serout etc are excluded from the calculation since their internal delays are part of their function). Some commands can run over 500,000 IPS and some much slower (i.e. floating point divides).

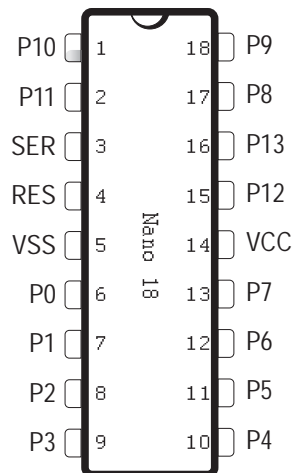
Nano 8

The BasicATOM Nano 8 is a programmable microcontroller. The Nano 8 requires a regulated 5VDC power source. It has 5 general purpose I/O, 3K of program memory, 100 Bytes of RAM and 4 Analog to Digital pins.



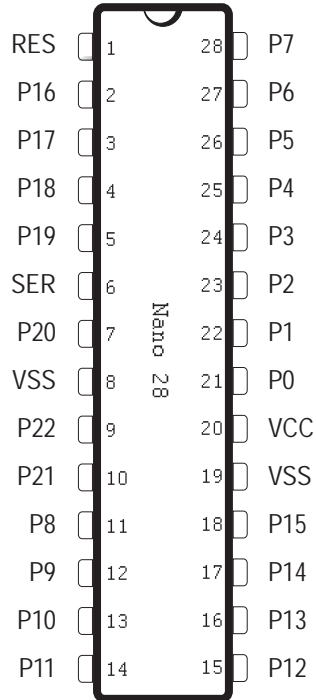
Nano 18

The BasicATOM Nano 18 is a programmable microcontroller. The Nano 28 requires a regulated 5VDC power source. It has 15 general purpose I/O, 7K of program memory, 300 Bytes of RAM, 256bytes of EEPROM and 6 Analog to Digital pins.



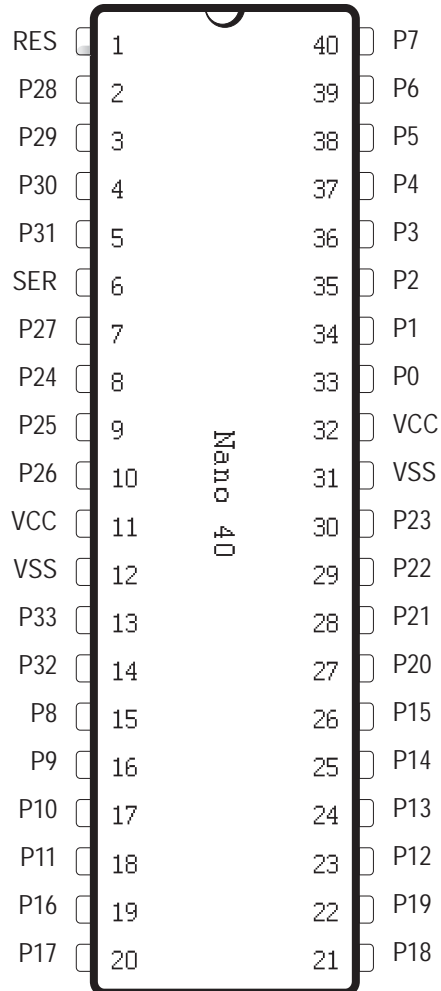
Nano 28

The BasicATOM Nano 28 is a programmable microcontroller. The Nano 28 requires a regulated 5VDC power source. It has 24 general purpose I/O, 14K of program memory, 300 Bytes of RAM, 256bytes of EEPROM and 11 Analog to Digital pins.



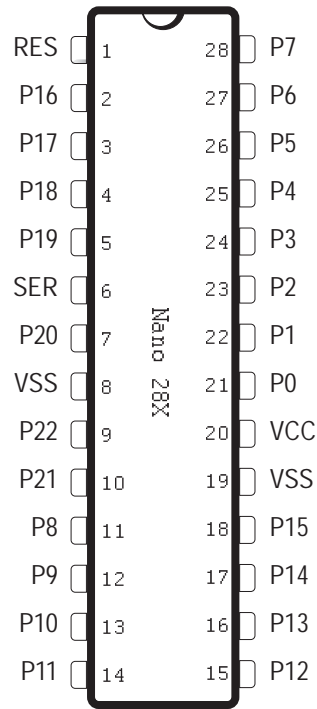
Nano 40

The BasicATOM Nano 40 is a programmable microcontroller. The Nano 40 requires a regulated 5VDC power source. It has 35 general purpose I/O, 14K of program memory, 300 Bytes of RAM, 256bytes of EEPROM and 14 Analog to Digital pins.



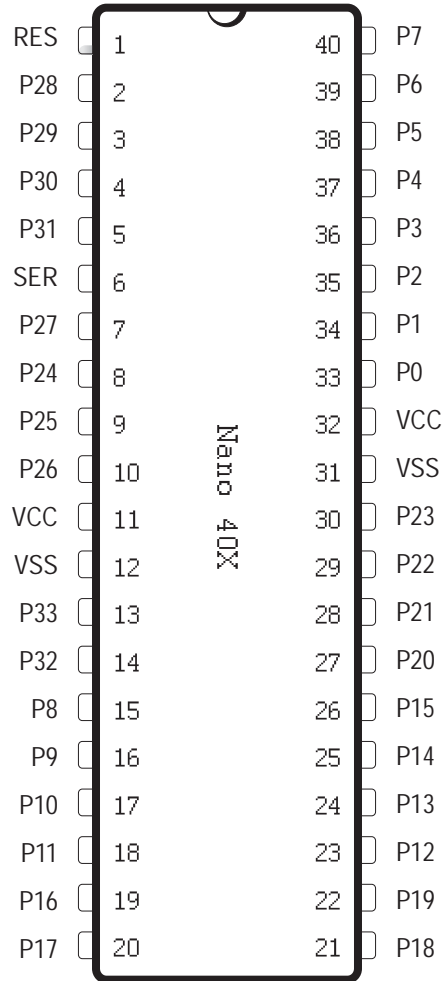
Nano 28X

The BasicATOM Nano 28X is a programmable microcontroller. The Nano 28X requires a regulated 5VDC power source and 20Mhz resonator. It has 22 general purpose I/O, 14K of program memory, 300 Bytes of RAM , 256bytes of EEPROM and 11 Analog to Digital pins.



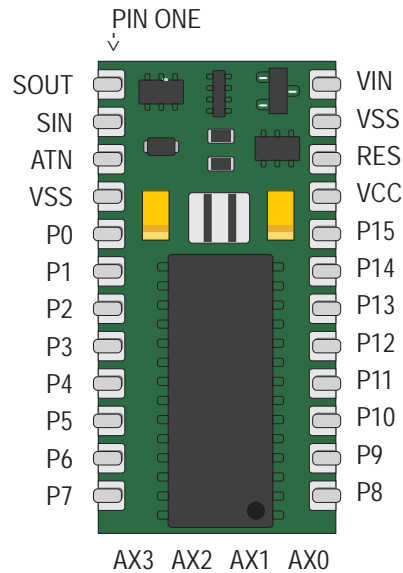
Nano 40X

The BasicATOM Nano 40X is a programmable microcontroller. The Nano 40X requires a regulated 5VDC power source and 20Mhz resonator. It has 33 general purpose I/O, 14K of program memory, 300 Bytes of RAM , 256bytes of EEPROM and 14 Analog to Digital pins.



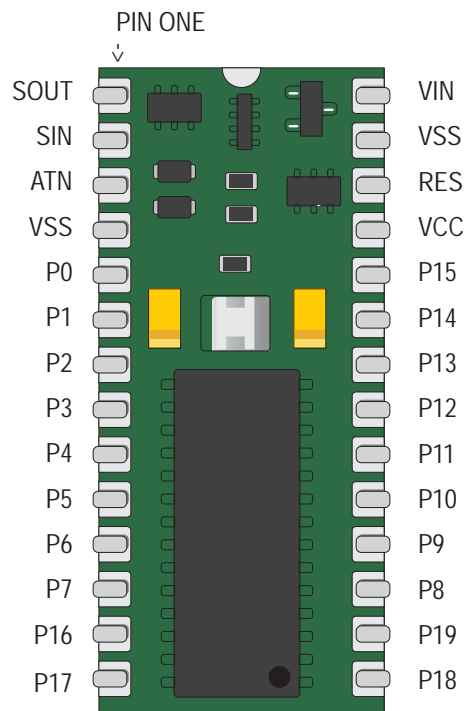
BasicATOM 24m

The BasicATOM 24m is a self contained microcontroller with all its support circuitry built in. It has 16 general purpose I/O, 8K of program memory, 300 Bytes of RAM, 256bytes of EEPROM. The module has 4 additional pads, 3 of which are Analog to Digital capable (AX3, AX2, AX1 and AX0) on its underside.



BasicATOM 28m

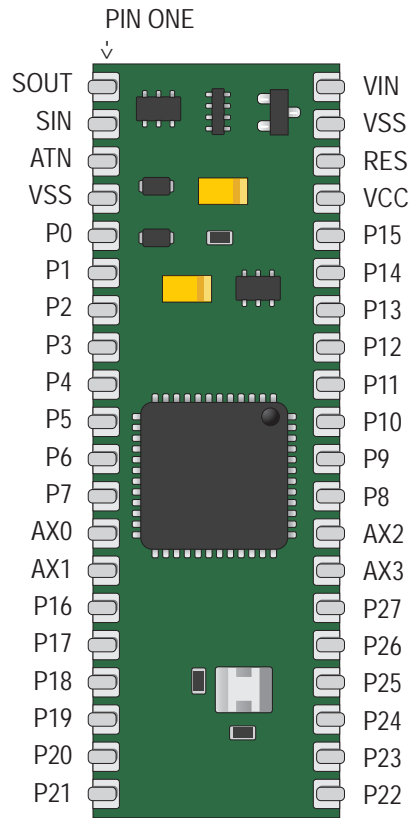
The BasicATOM 28m is a self contained microcontroller with all its support circuitry built in. It has 20 general purpose I/O, 8K of program memory, 300 Bytes of RAM, 256bytes of EEPROM with 10 Analog to Digital pins.





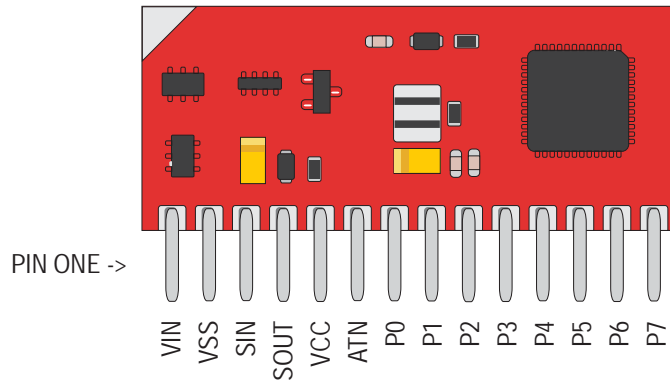
BasicATOM 40m

The BasicATOM 40m is a self contained microcontroller with all its support circuitry built in. It has 32 general purpose I/O, 8K of program memory, 300 Bytes of RAM, 256bytes of EEPROM with 14 Analog to Digital pins.



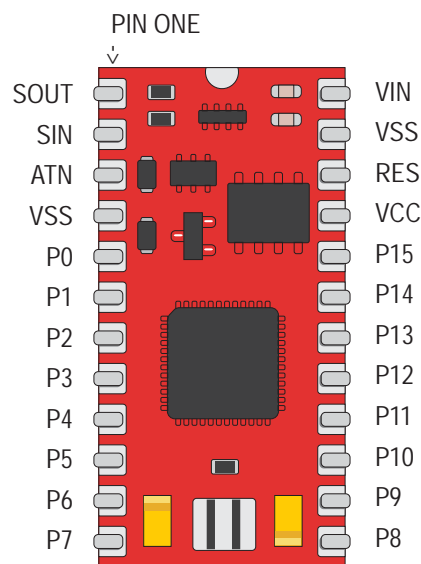
BasicATOM Pro ONEm

The BasicATOM Pro ONEm is a self contained microcontroller with all its support circuitry built in. It has 8 general purpose I/O, 32K of program memory, 2K of RAM and 4 Analog to Digital pins.



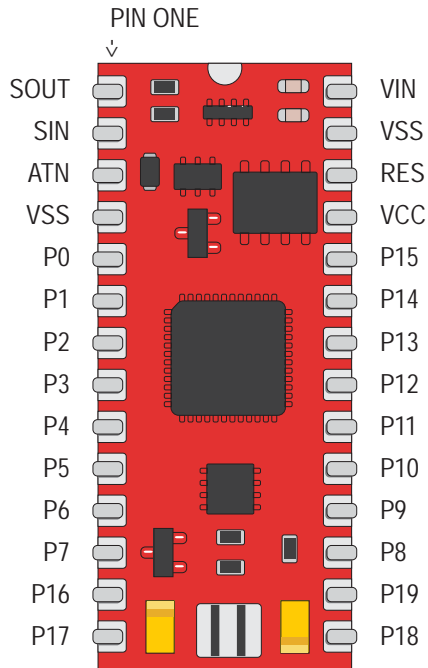
BasicATOM Pro 24m

The BasicATOM Pro 24m is a self contained microcontroller with all its support circuitry built in. It has 16 general purpose I/O, 32K of program memory, 2K of RAM and 4 Analog to Digital pins.



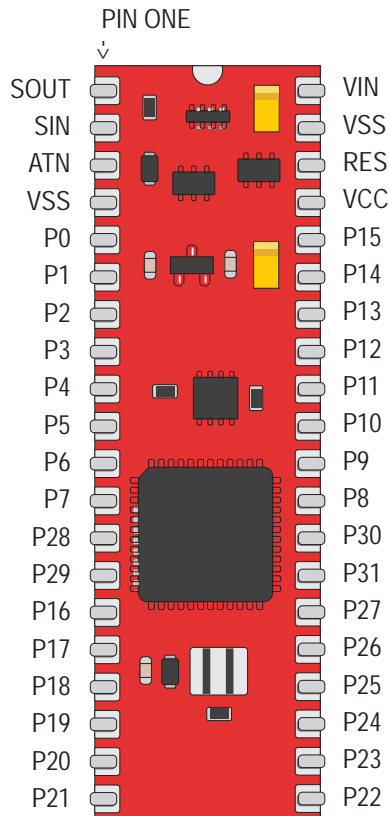
BasicATOM Pro 28m

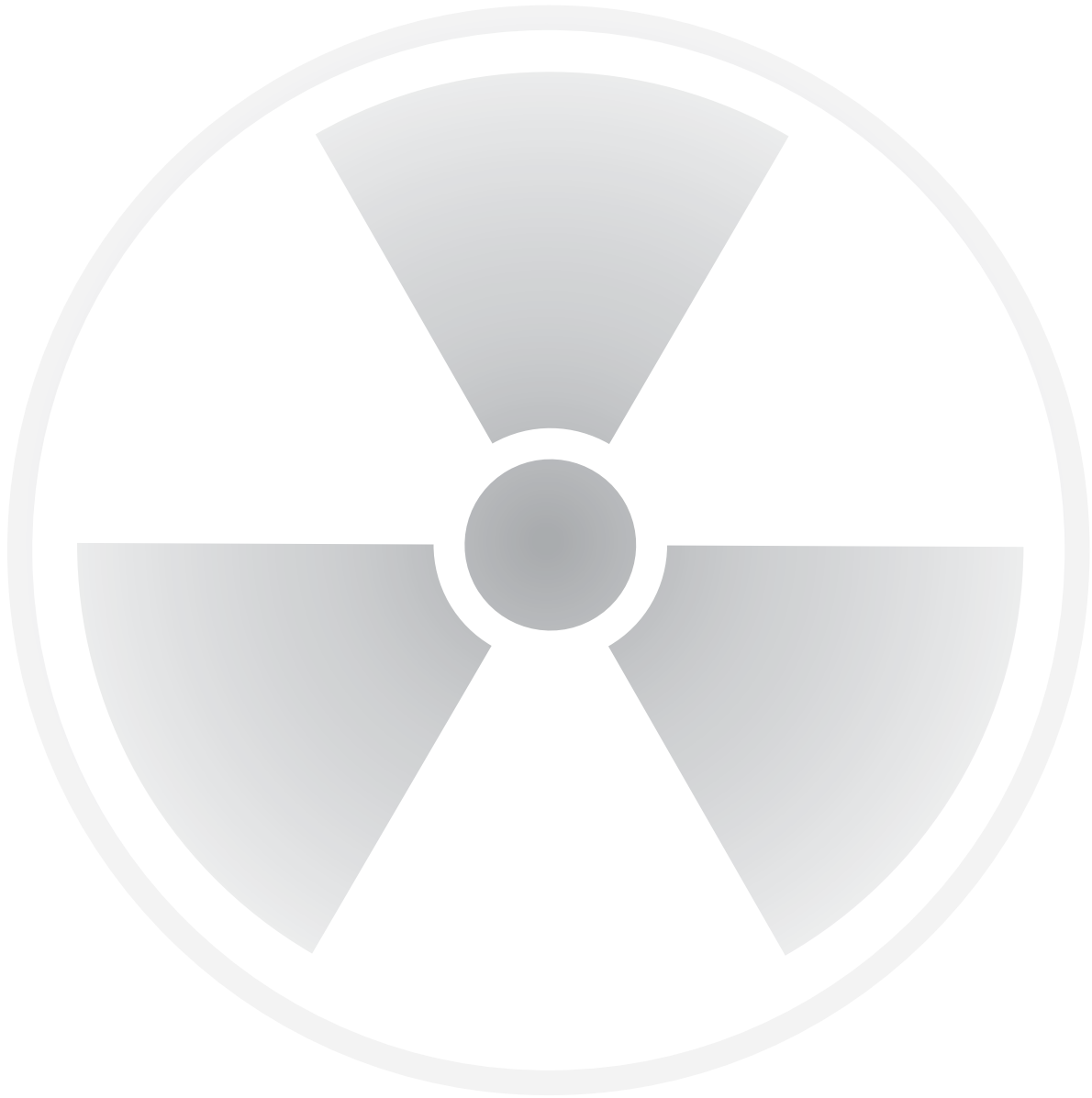
The BasicATOM Pro 28m is a self contained microcontroller with all its support circuitry built in. It has 20 general purpose I/O, 32K of program memory, 2K of RAM, 4K bytes of EEPROM and 8 Analog to Digital pins.



BasicATOM Pro 40m

The BasicATOM Pro 40m is a self contained microcontroller with all its support circuitry built in. It has 32 general purpose I/O, 56K of program memory, 4K of RAM, 4K bytes of EEPROM and 8 Analog to Digital pins.





## Quick Start Guide

The next few pages will go over setting up the hardware, software and creating your first program. This section is designed to teach you the basics of using the BasicATOM microcontroller line. If you are using the USB on the development board the drivers must be installed correctly first. See the development boards data sheet.

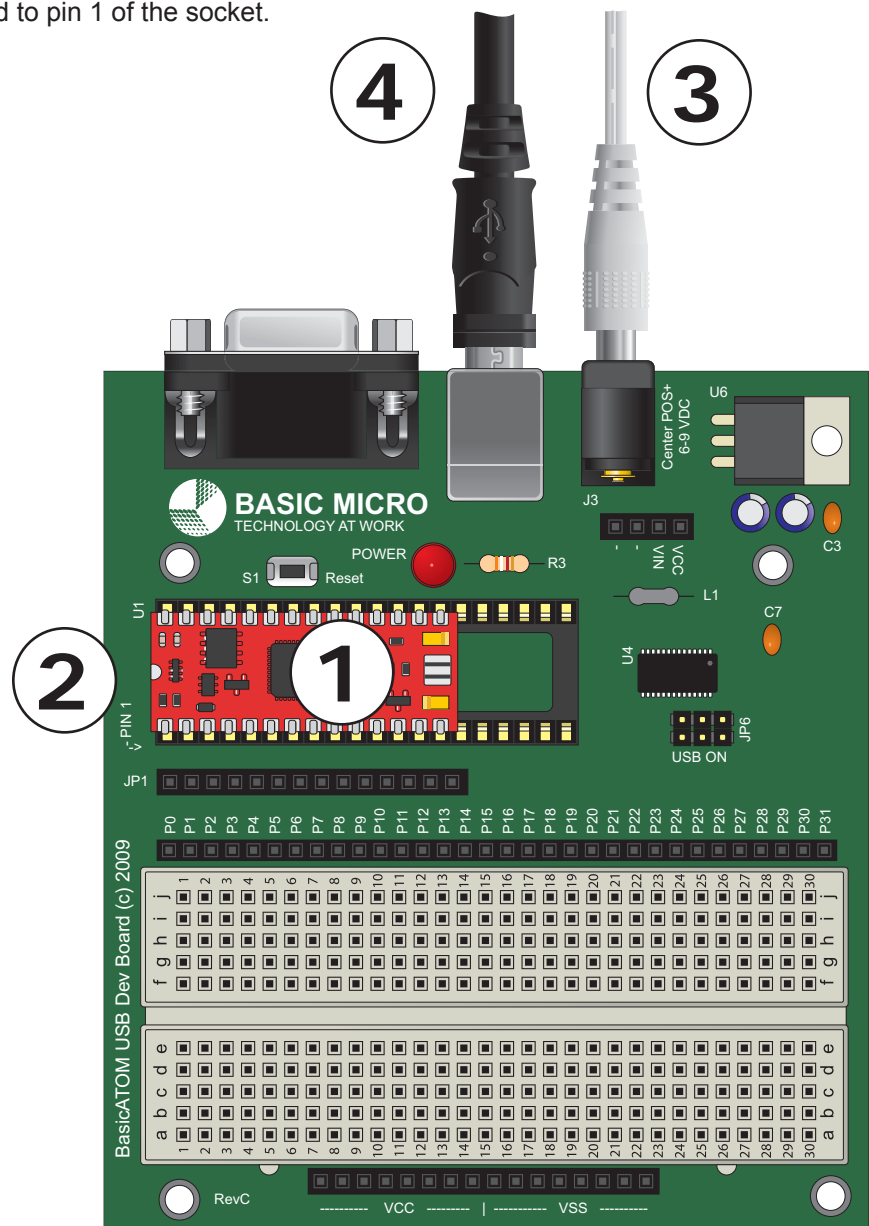
### Equipment Requirements

- BasicATOM Pro module
- USB Development Board or compatible
- PC running Windows XP or later
- USB cable

### BasicATOM USB Development Board

The first step is to insert the microcontroller into the development board socket. Make sure pin 1 of the microcontroller is aligned to pin 1 of the socket.

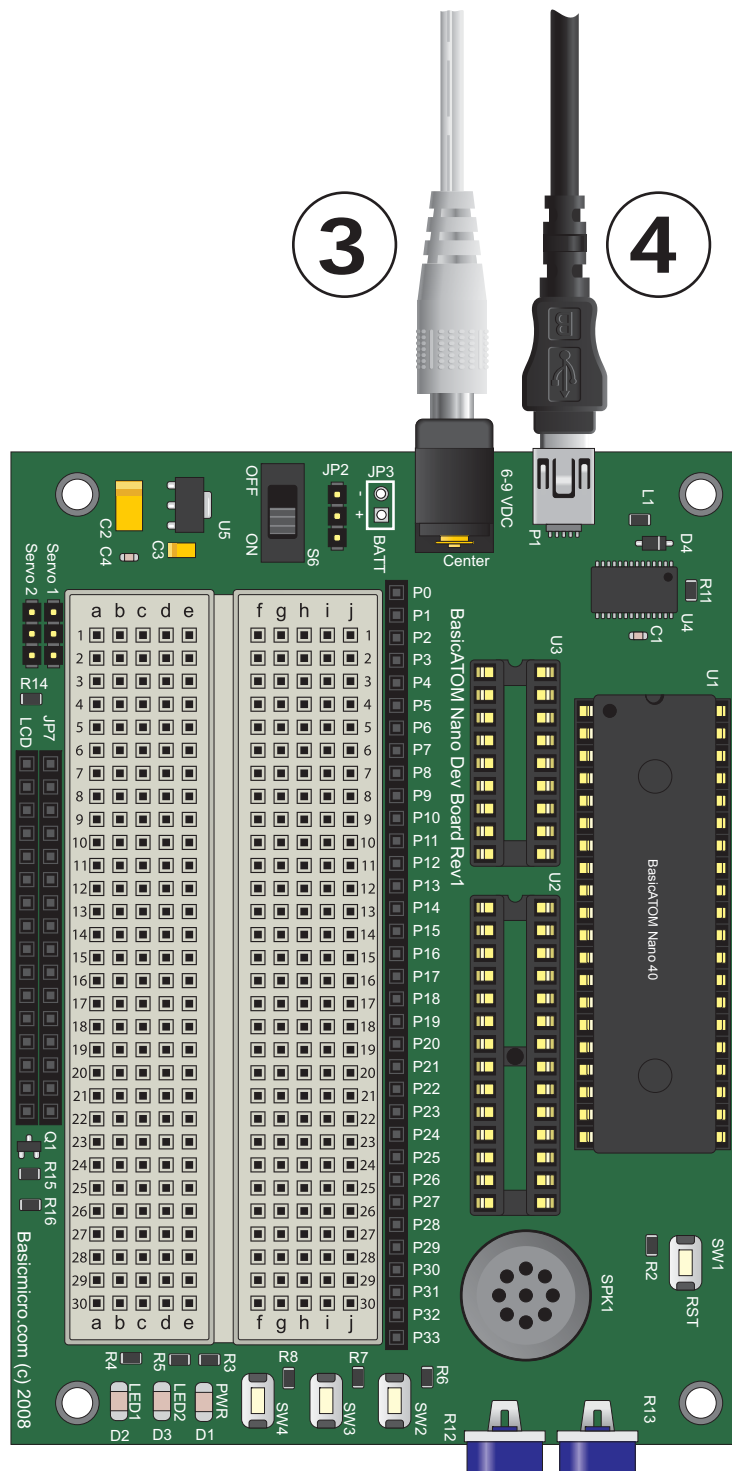
1. Insert the BasicATOM microcontroller.
2. Make sure pin 1 of the microcontroller is aligned to pin 1 of the socket.
3. Connect a 6-9V 500mA center positive wall adapter.
4. Connect the B side of a USB 2.0 compliant cable.



### Nano Development Board

The first step is to insert the microcontroller into the Nano Development board socket. Make sure pin 1 of the microcontroller is aligned to pin 1 of the socket. If you are using the USB on the development board the drivers must be installed correctly first. See the development boards data sheet.

1. Insert the Nano microcontroller.
2. Make sure pin 1 of the microcontroller is aligned to pin1 of the socket.
3. Connect a 6-9V 500mA center positive wall adapter.
4. Connect the B side of a USB 2.0 compliant cable.



## Quick Start

The first task before writing any program is installing and setting up Basic Micro Studio. Studio is the main piece of software that you will be using to develop your code for the BasicATOMs. It is commonly referred to as an Integrated Development Environment or IDE for short. The IDE contains 3 parts. A text editor for writing programs, a compiler to translate your program into something the microcontroller will understand and a loader to download your program to the microcontroller.

## Equipment Requirements

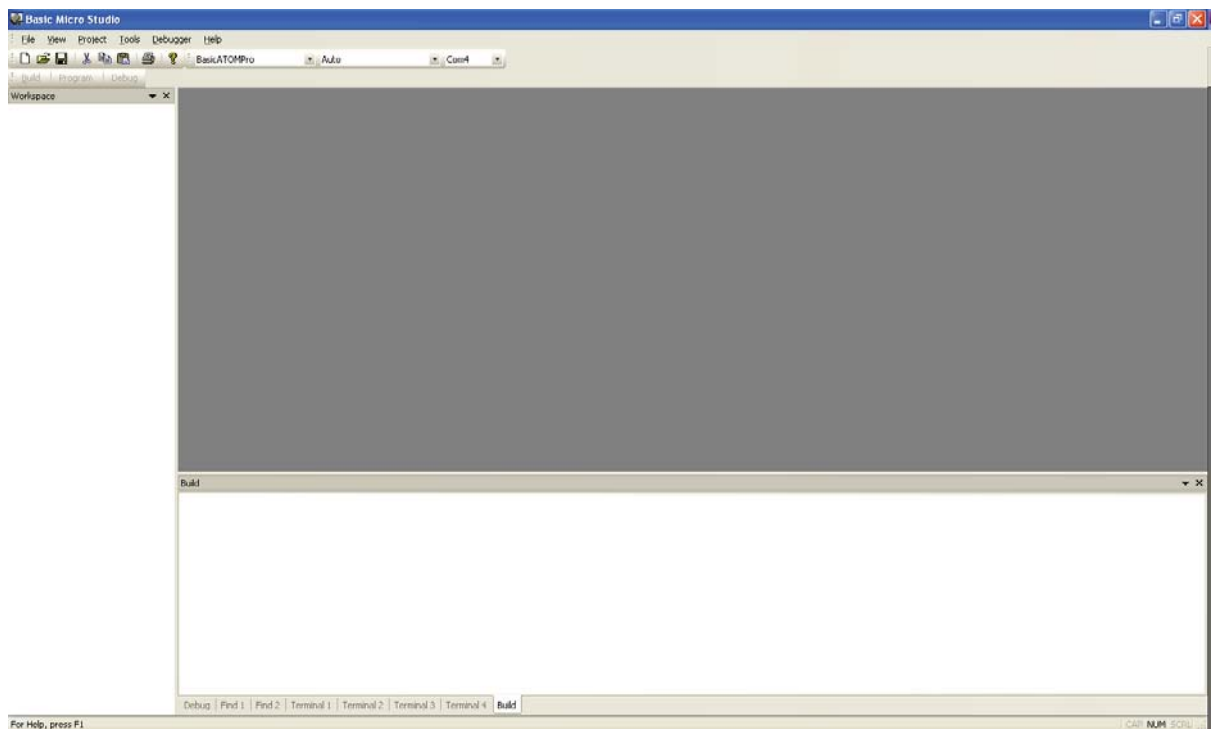
- Any BasicATOM module
- USB Development Board or compatible
- PC running Windows XP or later
- USB cable

## Basic Micro Studio Installation

You can download Basic Micro Studio from the downloads section at [Basicmicro.com](http://Basicmicro.com). Before installing Studio make sure you are logged in as an Administrator. Start installing Studio by double clicking the installer icon. The installation process is fairly quick and painless. Once the installation is complete, start Studio from the Start Menu -> Programs -> Basic Micro -> Basic Micro Studio.

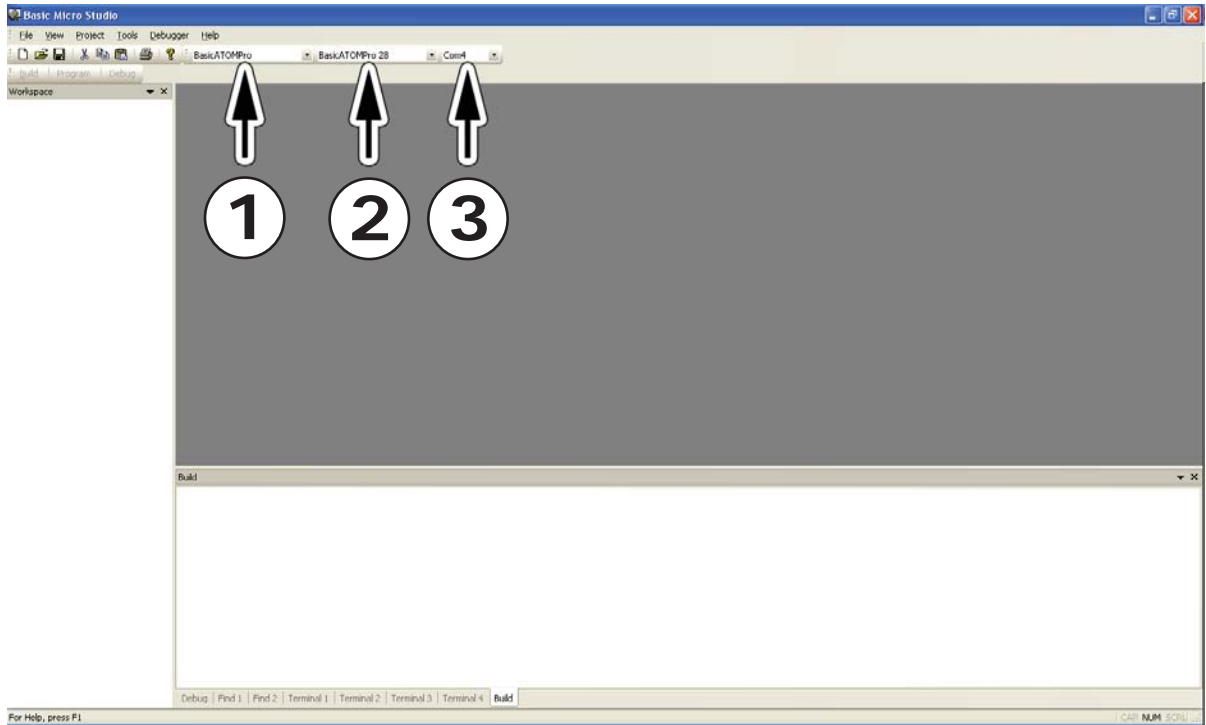
## Overview

To begin let's get familiar with some of Studio's features. The following screen is what you should see when first starting Basic Micro Studio.



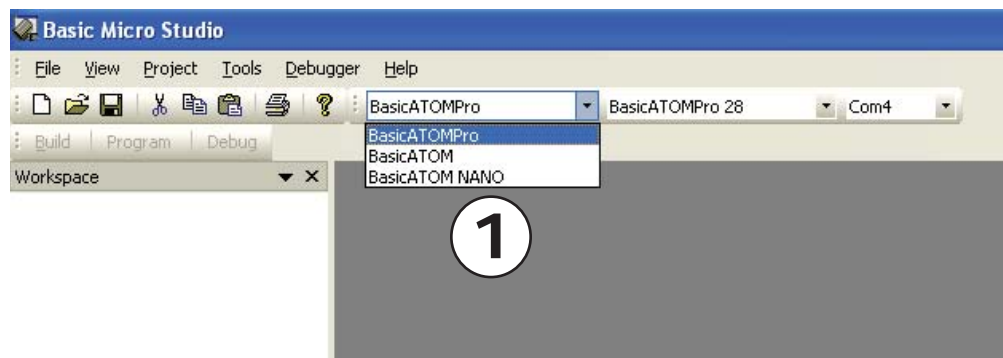
## Studio Setup

Before using Basic Micro Studio you must set the microcontroller that will be used. There are 3 drop down menus as shown below. The first sets the microcontroller family. The second sets the microcontroller type and the last sets what COM port the microcontroller is attached too.

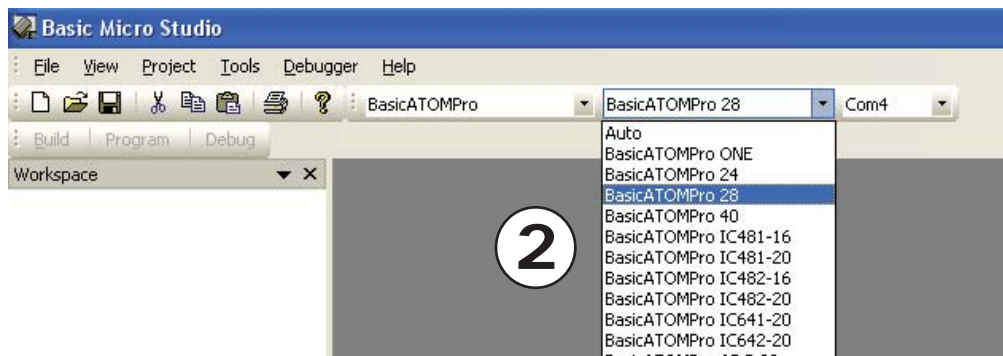




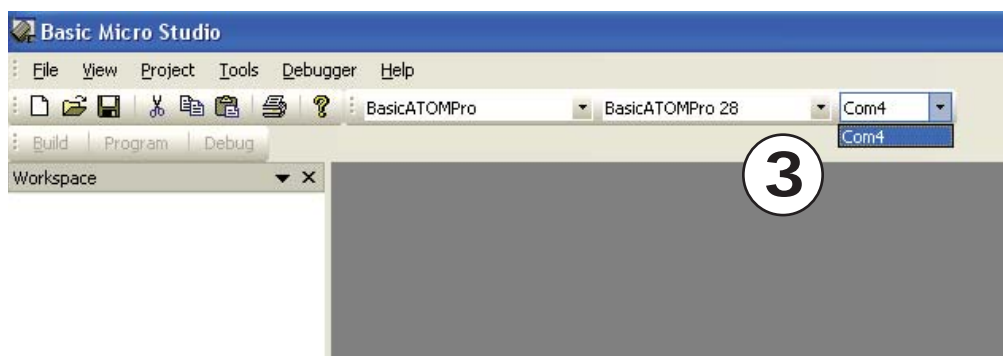
1. **Microcontroller Family** - is the main microcontroller family groups. BasicATOM Pro, BasicATOM Nano and BasicATOM. Choose the microcontroller family from drop down (1) as shown below:



2. **Microcontroller Type** - is the specific microcontroller such as BasicATOM Pro 24, BasicATOM 40 or BasicATOM Nano 18. Select the target module or chip you are using from the drop down menu. Auto can be used if you are unsure. If Studio is having issues detecting the microcontroller using Auto choose the microcontroller type from drop down (2) as shown below:

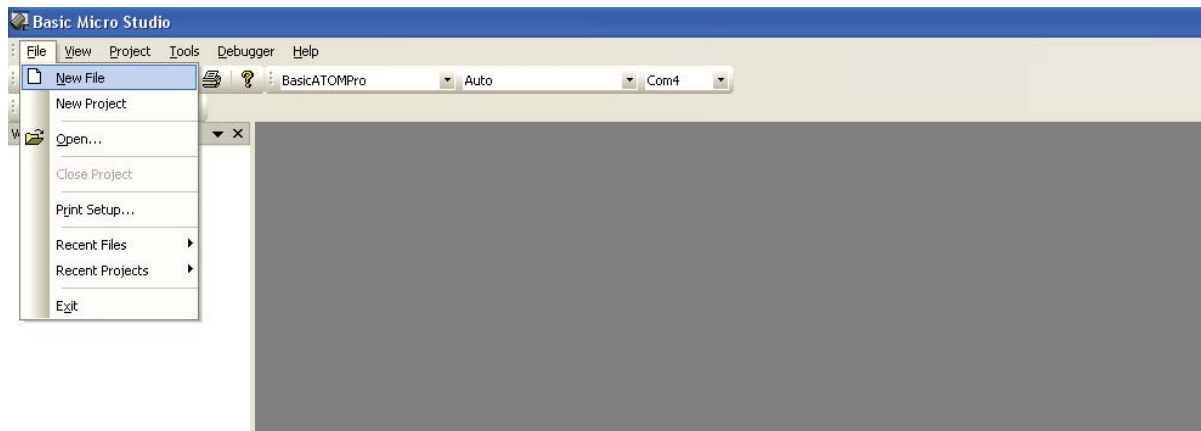


3. **COM Port** - is the communications port the device is attached to. This can be any port number from COM 1 to 255. Choose the COM ports from the drop down (3) as shown below:



## First Program

The next step is to create your first program. You can do this by selecting File-> New File. Then name your file *first.bas*. Choose a location to save your first program.



After you have created the file *first.bas*, type the following example code exactly as shown below. If you copy and paste the example code you will need to edit the quotes as they won't copy over properly.

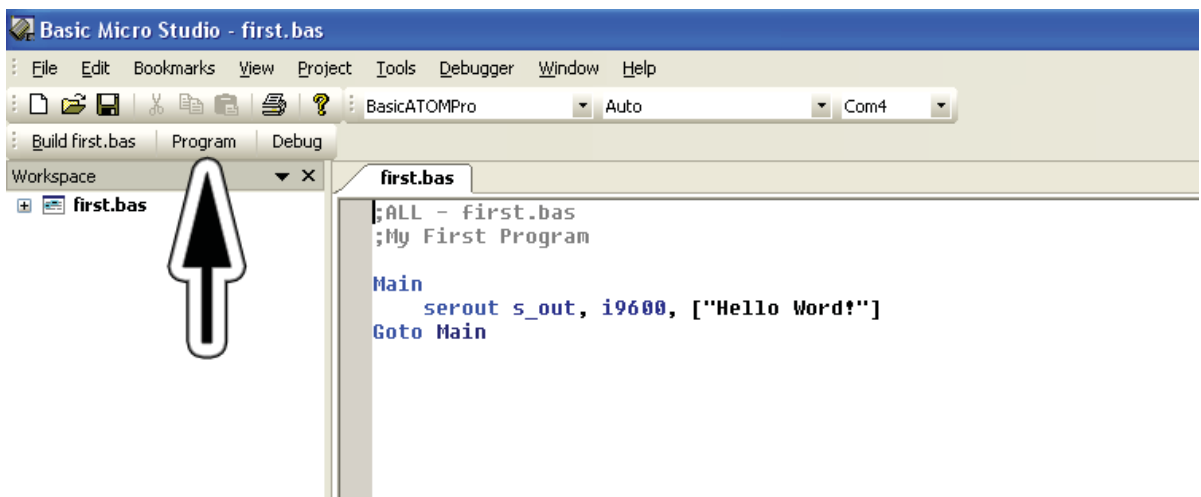
```
;ALL - first.bas
;My First Program

Main
    serout s_out, i9600, ["Hello Word!",13]
Goto Main
```

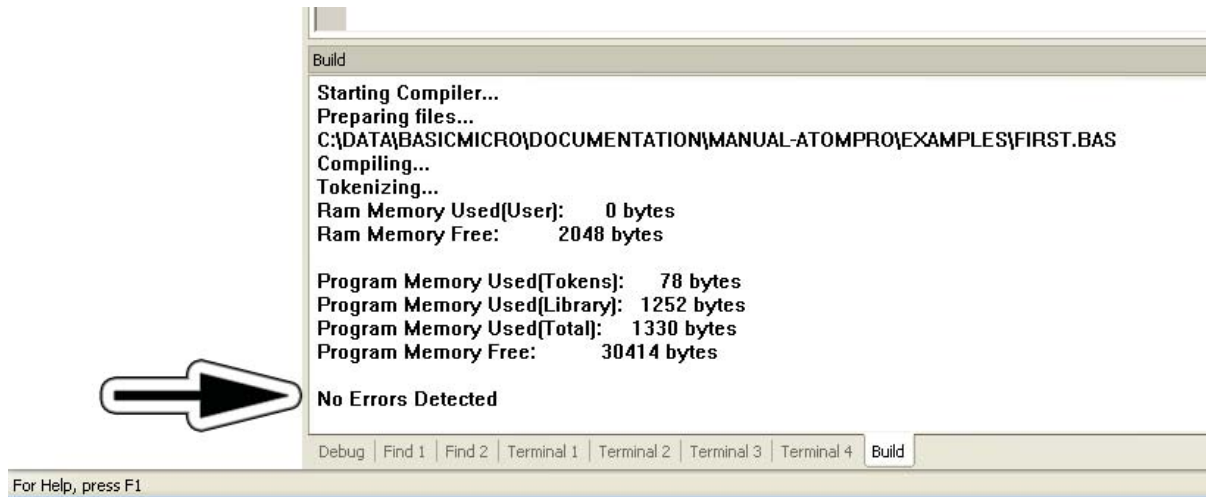
## Downloading The Program

By this time you should have the microcontroller installed on the development board. Apply power and connect the USB cable to your PC. If Basic Micro Studio has trouble detecting your microcontroller make sure the USB drivers were installed correctly and the correct COM port is selected.

Make sure the correct family, type and COM port has been set properly. Once you are ready click on the "Program" button shown by the arrow below.



If the program is correct with no syntax errors and the microcontroller is detected you should see a message “No Errors Detected” at the bottom of the build window as shown below.



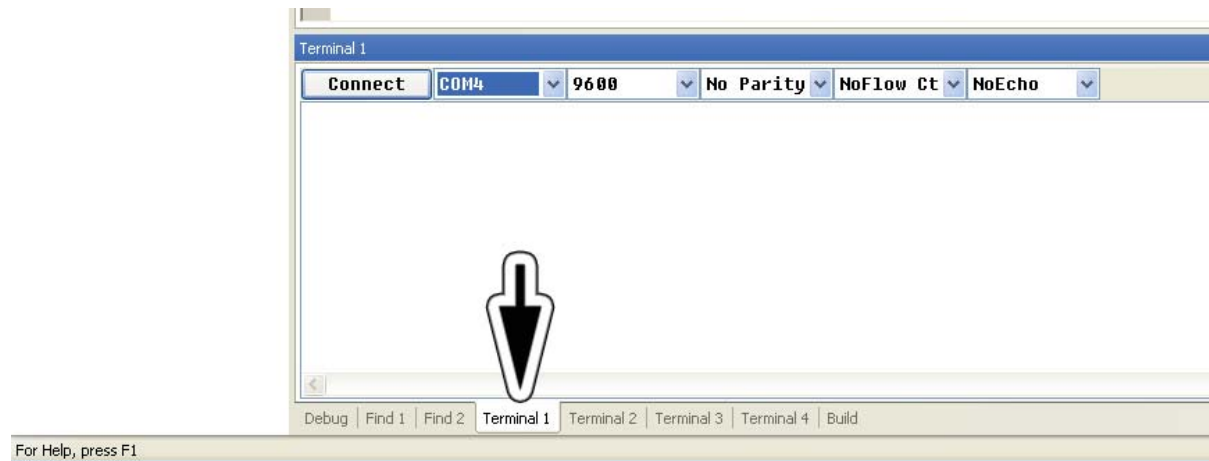
### The Download Begins

After the microcontroller is detected and your program compiles without errors, downloading will begin and you should see the progress dialog. Downloading the program can take up to 30 seconds on some modules. If downloading is slower than this you may not have your USB drivers installed correctly.



### Terminal Window

You have successfully written and downloaded your first program. Now its time to see what it does. We will connect to your program using Studios built in terminal windows. After programming is complete, at the bottom of the IDE window there are several tabs. These tabs set what control is displayed. Click on "Terminal 1".

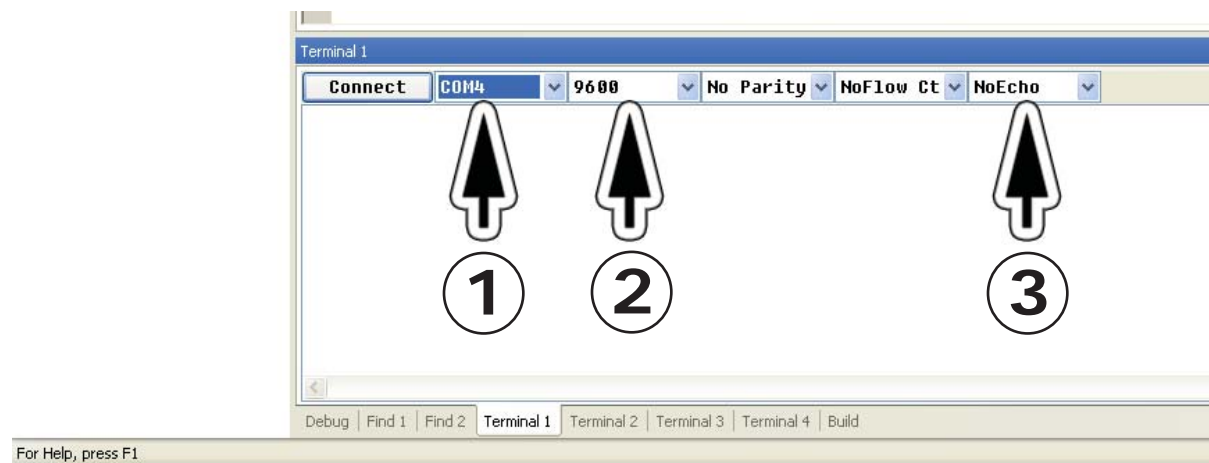


### Terminal Window Setup

The next step is setting the terminal window to match the SEROUT / SERIN command parameters you used in your program.

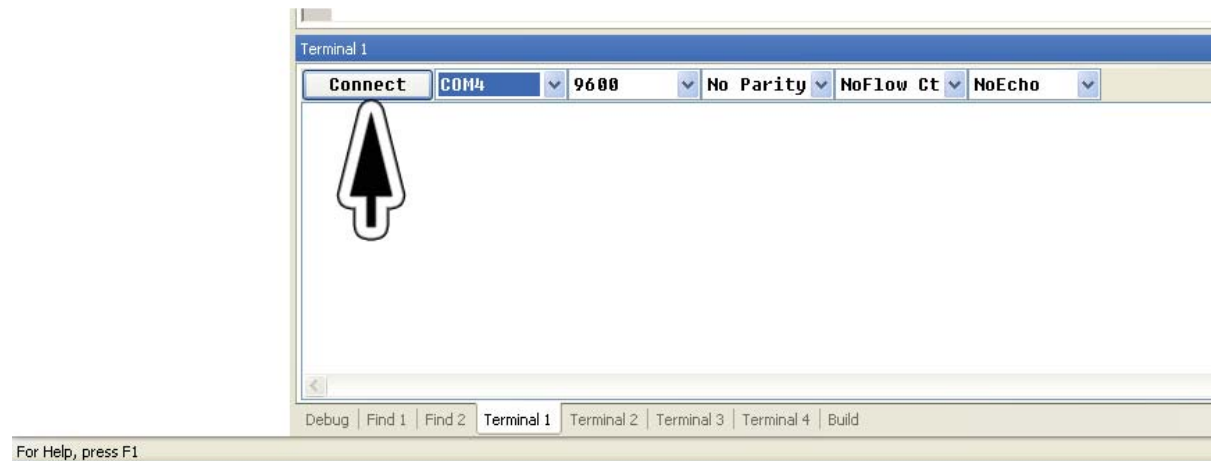
```
serout s_out, i9600, ["Hello Word!",13]
```

First select the COM port (1). For this example you will use the same COM port as you used in the drop down box at the top of Studio. Next i9600 was used so our baud rate (2) should be set to 9600. Last, set NoEcho (3).

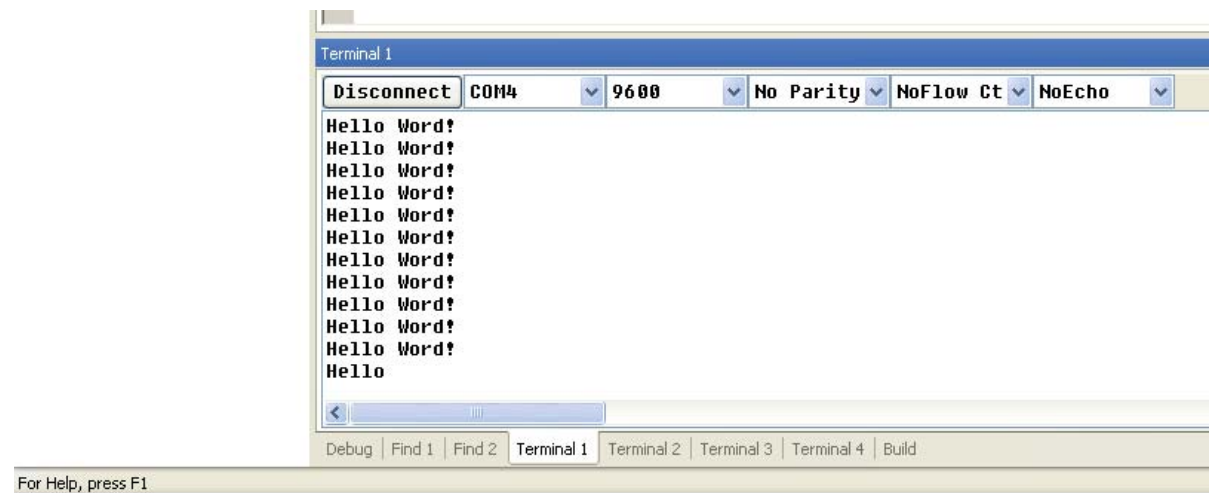


### Terminal Window Connect

Once you have the terminal window set properly its time to connect to your program. Click the connect button.



Once the terminal window successfully connects to your program you should see "Hello World!" displayed in the terminal window. Since our first program is a simple loop it will print "Hello World!" forever or until you disconnect it.



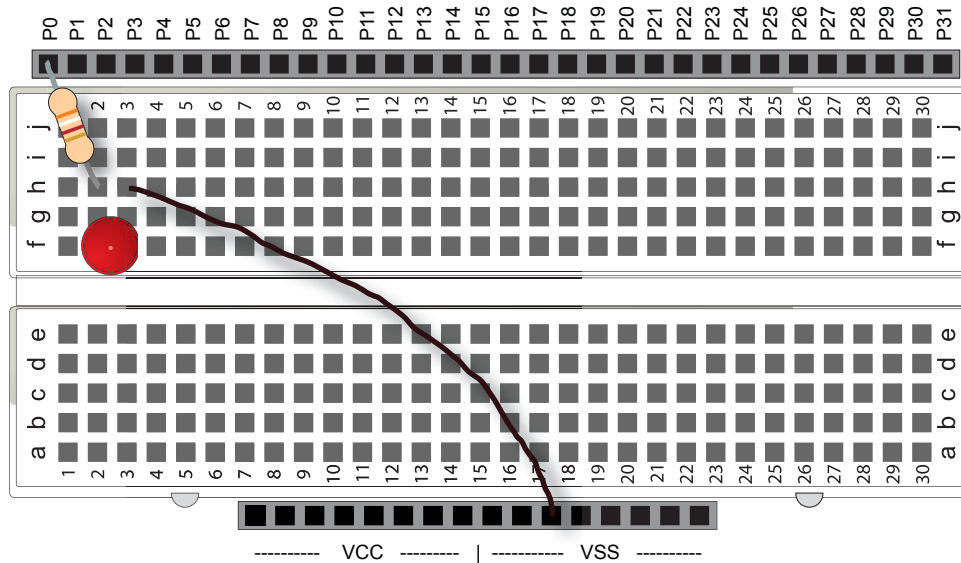
### Congratulations

You have successfully created and downloaded your first program. Welcome to the exciting world of microcontrollers!

## Blinking LED

Now that you know how to write a program and download it, lets make something we can see.

Everyone likes a blinking LED. Connect a 390 Ohm resistor to P0. Connect an LED anode (+) to the resistor. Connect the other side of the led cathode (-) to VSS (GND) which is marked by the small flat spot on the LED.



## The Program

Type or copy the below example exactly as shown. You can copy and paste the program into Studio. Once you have entered the program save it as *blink.bas*. Then download it to the BasicATOM as outline in the previous section of this manual.

```
;ALL - blink.bas
;My blinking LED program

Main
  High P0
  Pause 300
  Low P0
  Pause 300
Goto Main
```

## Congratulations

You have successfully blinked your first LED! You can now officially consider yourself a programmer.



## Variables

Variables are used to store values used in your program. The values stored typically will change during program run time. What was stored at the start of your program in a given variable can change by the time your program stops running.

To use a variable it must be defined in your program so Basic Micro Studio knows to set aside the required amount of space in RAM. The ATOM line is 32 bit so variables can be from 1 bit to 32bits in capacity. Typically most microcontroller systems are limited to 16bits capacity.

When creating a variable you specify what type of variable you will need in your program. You can define as many variables as you want. The only limit is the amount of RAM available which varies for each ATOM type.

### Variable Types

Type	Bits	Value Range
Bit	1	1 or 0
Nib	4	0 to 15
Byte	8	0 to 255
SByte	8	-128 to +127
Word	16	0 to 65,535
SWord	32	-32,768 to +32,767
Long	32	0 to 4,294,967,295
SLong	32	-2,147,483,647 to +2,147,483,648
Float	32	$\pm 2.0 \text{ EXP } -126$ to $\pm 2.0 \text{ EXP } 127$
Pointer	32	0 to 4,294,967,295

### Location

Variables should be defined in the beginning of your program if you want access to the variable anywhere in your program. Where variable are defined dictates where those variables can be used. Variables must be defined before they can be used.

### Signed

Variables can be signed. This keeps track if the variable is a negative or positive number. This is useful in many applications. A simple example would be controlling a DC motor. A negative number can indicate one direction of spin, 0 can be stop and a positive value can be the opposite direction of spin using one variable. Signed variables are created using Sbyte, Sword and Slong



### Defining Variables

Variables are defined with the statement VAR. You can declare your new variable as any type found in the Variable Types table. To define a variable use the syntax shown below.

#### Syntax:

```
VariableName VAR Type
```

#### Examples:

```
Red VAR Byte  
Tick VAR Nib  
Switch var Bit  
Totals var Sword
```

### Variable Names

Variables names must start with a letter but can be any combination of letters and numbers including some symbols that are not in the reserved word list. You can name your variables almost anything. However its a good idea to name them something useful that you will remember and understand for debugging your program. Variable names can be up to 1,024 characters long. The only names you can not use for your variables are reserved words. These are words used by Studio for commands or other syntax based names (see the manual appendix for a complete reserved word list).

#### Example:

```
MotorSensor VAR Word
```

### Aliases

Variables can be aliased. This is useful when creating a program for a processor with limited RAM (i.e. BasicATOM). Using an alias can help your code be more human readable without wasting RAM. If you have a variable that is only used in one particular section of code you can assign an alias pointing to a variable that was defined earlier.

#### Example:

```
MotorDirection VAR Word  
MaxCCSpeed VAR MotorDirection  
MaxCCWSpeed VAR MotorDirection.byte0
```

### Variable Modifiers

Variable modifiers can be used when only part of a variables value is needed. Most communication formats such as serial or I2C are byte driven. If you have a word sized variable and you are sending data to such a device, you will need to send one byte at a time of the word variable. The word value can be split using an alias with a modifier as shown below.

#### Example

```
MyData VAR Word  
FirstByte VAR MyData.HighByte  
SecondByte VAR MyData.LowByte
```

## Variable Modifier Type

Modifier	Description
LowBit	Returns the low bit of a variable (least significant bit).
HighBit	Returns the high bit of a variable (most significant bit).
Bit <i>n</i>	Returns the <i>N</i> th bit of a variable. From 0 to 31 depending on variable size.
LowNib	Returns the low nibble (4 bits) of a variable (least significant nib).
HighNib	Returns the high nibble (4 bits) of a variable (most significant nib).
Nib <i>n</i>	Returns the <i>N</i> th nib (4 bits) of a variable. From 0 to 7 depending on variable size.
LowByte	Returns the low byte (8 bits) of a variable (least significant byte).
HighByte	Returns the high byte (8 bits) of a variable (most significant byte).
Byte <i>n</i>	Returns the <i>N</i> th byte of a variable. From 0 to 3 depending on variable size.
SByte <i>n</i>	Returns the <i>N</i> th signed byte of a variable. From 0 to 3 depending on variable size.
LowWord	Returns the low word (8 bits) of a variable (least significant word).
HighWord	Returns the high word (8 bits) of a variable (most significant word)
Word <i>n</i>	Returns the <i>N</i> th word of a variable. 0 to 1 depending on variable size.
SWord <i>n</i>	Returns the <i>N</i> th signed word of a variable. 0 to 1 depending on variable size.

## Variable Arrays

Variable arrays can be used to store several values using one variable name. A good use of variable arrays would be taking multiple readings from a temperature sensor and storing the samples in a variable named Temperature. The reads can later be averaged for a more accurate temperature. Studio only supports one dimensional arrays.

Example:

```
Temperature VAR Byte(5)
```

The example above creates 5 byte sized RAM locations for the variable temperature. Each location is indexed and referenced by using a numerical value. The range is based on the array size that was defined. In this case the array size is 5. So an index value of 0 to 4 can be used.

Example:

```
Temperature(0) = 255 ;loads first position
Temperature(1) = 255 ;loads second position
Temperature(2) = 255 ;loads third position
Temperature(3) = 255 ;loads fourth position
Temperature(4) = 255 ;loads fifth position
```

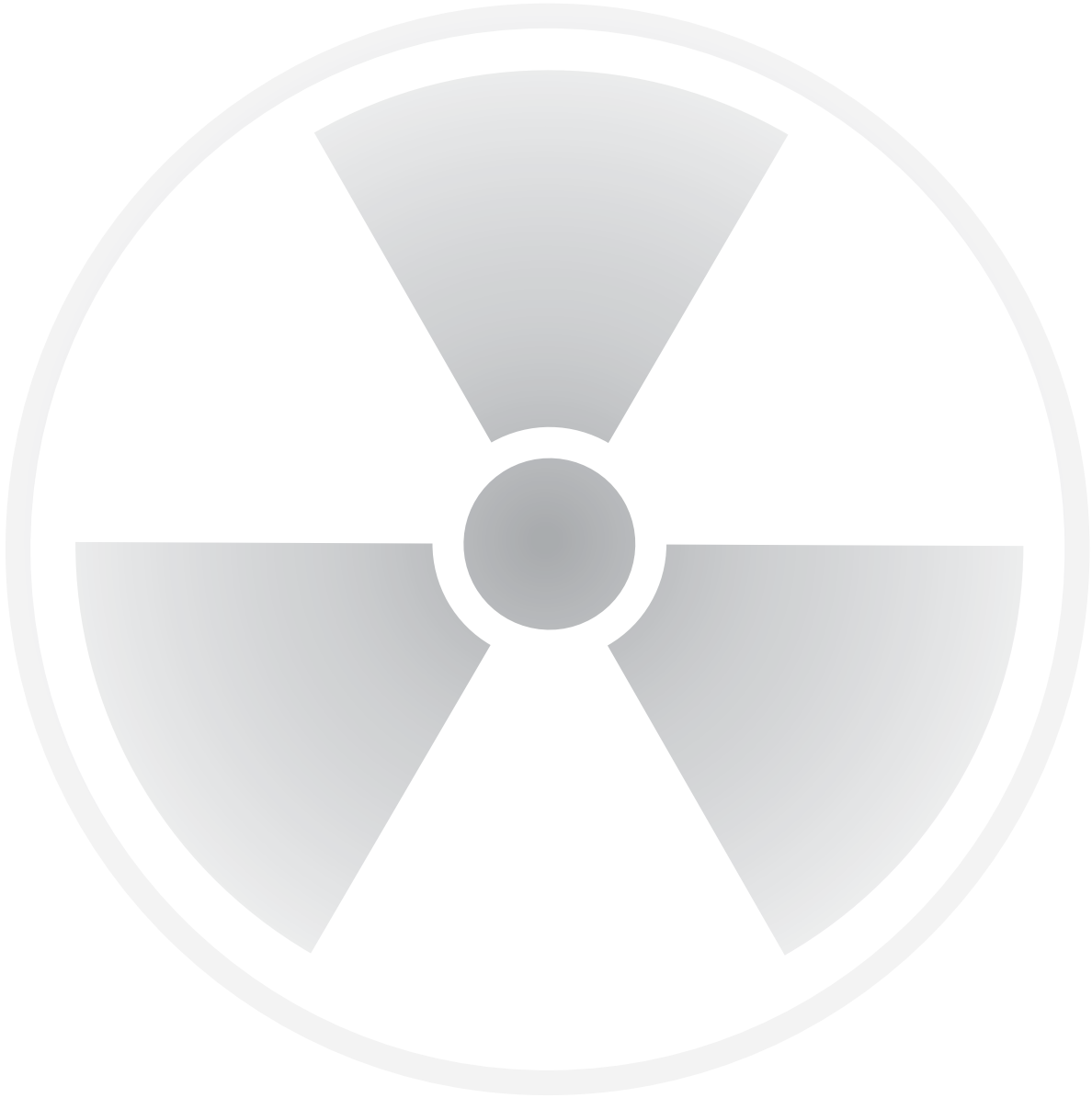
```
Temp = Temperature(0) ;Loads value from first position in the array
```

A common use of variable arrays is to store strings of ASCII based characters. This can be used to send an entire sentence of text to a byte driven device such as a computer terminal window using the STR command modifier shown later.

---

### Out of Range

When declaring variables, careful consideration should be given to the maximum value it will store. If a byte sized variable is declared but the result of some function is word sized then data will be lost. Basic Micro Studio has no way of knowing these possible conditions existing within your program. This situation has added some grey hair more times than most programmers will admit. So beware. Typically a beginner programmer will declare every variable as large as they can. While this sounds good in theory, in practice you'll run out of RAM space quickly. Plus it makes for an inefficient programming style.



## Constants

Constants are values defined that will never change. The assigned value is store in program memory and can not be changed. Constants are set during compile time by Studio. Constants are a convenient way to give a name to a numeric value. They are typically used to make a program more human readable. There are two types of constants. Normal constants defined using CON and a floating point constant defined by FCON. You can use almost any name for a constant, except reserved words (see manual appendix).

Example:

```
LimitTravel CON 255
MaxSpeed FCON 6.50
```

### Constant Tables

Tables are similar to arrays except tables only store constant values. You can store large strings of text in a constant table. Each position is accessed like arrays using an index number. Values assigned to a table are stored in program memory and can not be changed during program run time. A common use for constant tables is building interactive menu systems. If the user selects *n* reply with a string from table *n*. Constant tables are restricted to word boundaries if you have an odd number of bytes in a byte table an extra byte is added for padding automatically.

Example:

```
Sentence ByteTable "Hello World!"
```

The above example bytetable *Sentence* contains the string *Hello World!*. Each character including the space is a byte in the table. *Sentence* contains 12 bytes. To access the bytes you would use an index value of 0 to 11.

Example:

```
Temp VAR Byte
Sentence ByteTable "Hello World!"

Temp = Sentence(0)
```

The variable *Temp* now is equal to the ASCII value *H* since index 0 is the first byte in the defined table.

### Table Types

Type	Description
ByteTable	Each table index point is byte sized (8 bits).
SByteTable	Each table index point is sbyte sized(signed 8 bits).
WordTable	Each table index point is word sized (16 bits).
SWordTable	Each table index point is sword sized(signed 16 bits).
LongTable	Each table index point is long sized (32 bits).
SLongTable	Each table index point is slong sized (32 bits).
FloatTable	Each table index point is floating point sized (32 bits).



## Pins and Ports

Pins and ports is how your program interfaces with the outside world. Pins can be read, set as high or set as low. In some cases pins are what is known as open drain which can be a low or floating only, but not driven high.

Pins are accessed individually or by ports which are typically 8 pin groups. Pin names are treated as constants. However pins can be accessed like variables using pin variable names. Ports are accessed and handle just like variables.

### Pin Constants

Pin names are defined with a "P". The "P" stands for Pin followed by a pin number. The amount of pins and their names will vary depending on the processor you are using. The first 8 pins of a BasicATOM are:

```
P0
P1
P2
P3
P4
P5
P6
P7
```

Pin names are only limited by the number of accessible pins on the processor you are using. Pin names are normally used as an argument in commands. A simple example would be:

```
High P0
Low P1
```

The high command will set P0 high, which is I/O Pin 0 of the processor you are using. It is important to note pin names are constants. An example of this is shown below. *Temp* which is a variable would be made to equal the value of the P0 constant:

```
Temp = P0
```

All pin names are defined in sequence starting at P0 and ending at the highest pin name. This allows pin names to be used to initialize a for/next loop.

```
Temp var byte
for Temp = P0 to P31
  toggle Temp
next
```

### Pin Variables

Since pin names are constants they can't be used to read the state of a pin directly. Instead we use pin variables. Pin variables are important when your program needs to know something about the outside world. They can be used to determine the state of a pin or port:

```
Temp = IN0
Temp = OUT0
```

IN0 and OUT0 are pin variables. They both do the same thing. Both are defined to make reading your code easier. Using either IN or OUT will read the current state of the specific pin, assuming the pin is an input. *Temp* will either equal 0 or 1 depending on the pin state when its read. All pins start off as inputs during power up. However If a pin was last used by a command that set the pin to an output state the above expression would not work with out first using a DIR statement(or INPUT command) to set the pin back to being an input. The following table is a complete list of all the available pin and

port variable names. Using the variable names, individual pins can be accessed or up to 32 pins can be accessed at one time. You will need to check the data sheet for the processor you are using to determine how many pins are user accessible. Not all the pin or port variables will work on every processor, the limitation being the amount of available pins.

Pin and Port IN/OUT Variable Names

Variable Names	Variable Size	Bits	Affected Pins
INL0,OUTL0	Long	32	P0 - P31
INL1, OUTL1	Long	32	P32 - P63
INW0,OUTW0	Word	16	P0 - P15
INW1,OUTW1	Word	16	P16 - P31
INW2,OUTW2	Word	16	P32 - P47
INW3,OUTW3	Word	16	P48 - P63
INB0,OUTB0	Byte	8	P0 - P7
INB1,OUTB1	Byte	8	P8 - P15
INB2,OUTB2	Byte	8	P16 - P23
INB3,OUTB3	Byte	8	P24 - P31
INB4,OUTB4	Byte	8	P32 - P39
INB5,OUTB5	Byte	8	P40 - P47
INB6,OUTB6	Byte	8	P48 - P55
INB7,OUTB7	Byte	8	P56 - P63
INN0,OUTN0	Nib	4	P0 - P3
INN1,OUTN1	Nib	4	P4 - P7
INN2,OUTN2	Nib	4	P8 - P11
INN3,OUTN3	Nib	4	P12 - P15
INN4,OUTN4	Nib	4	P16 - P19
INN5,OUTN5	Nib	4	P20 - P23
INN6,OUTN6	Nib	4	P24 - P27
INN7,OUTN7	Nib	4	P28 - P31
INN8,OUTN8	Nib	4	P32 - P35
INN9,OUTN9	Nib	4	P36 - P39
INNA,OUTNA	Nib	4	P40 - P43
INNB,OUTNB	Nib	4	P44 - P47
INNC,OUTNC	Nib	4	P48 - P51
INND,OUTND	Nib	4	P52 - P55
INNE,OUTNE	Nib	4	P56 - P59
INNF,OUTNF	Nib	4	P60 - P63
IN0..IN63	Bit	1	Any
OUT0..OUT63	Bit	1	Any



### Direction Variables

The DIR variables are used to set pins and ports to inputs or outputs. Setting the bits of the direction variables sets the specified pins as inputs or outputs. Special attention must be paid to what module type you are using when utilizing the DIR variables. On the Nano and BasicATOM modules 0 represents an output state. The BasicATOM Pro is reverse where a 0 represents an input. The below table can be used to determine what zero means to the particular processor you are using.

### Input Output

Processor	Input	Output
Nano8	1	0
Nano18	1	0
Nano28	1	0
Nano40	1	0
BasicATOM 24-M	1	0
BasicATOM 28-M	1	0
BasicATOM 40-M	1	0
BasicATOM Pro One-M	0	1
BasicATOM Pro 24-M	0	1
BasicATOM Pro 28-M	0	1
BasicATOM Pro 40-M	0	1

All pins and ports are set as inputs by default on power up. When a pin is used in a command like High, the pin direction is set automatically. The pin is then left in that state unless otherwise specified in the command when it finishes.

In the below example the value of *Temp* will be incorrect:

```
High P0
Temp = IN0
```

This is due to the High command being an output command, the pin being left in an output state. So it can not be read as an input until its changed to an input. To fix this we use a DIR variable to change the pin to an input. The below program would now function correctly:

```
High P0
DIR0 = 0 ;sets pin as an input on BasicATOM Pro modules
Temp = IN0
```

Pin and Port DIR Variables Names

Variable Names	Variable Size	Bits	Affected Pins
DIRL0	Long	32	P0 - P31
DIRL1	Long	32	P32 - P63
DIRW0	Word	16	P0 - P15
DIRW1	Word	16	P16 - P31
DIRW2	Word	16	P32 - P47
DIRW3	Word	16	P48 - P63
DIRB0	Byte	8	P0 - P7
DIRB1	Byte	8	P8 - P15
DIRB2	Byte	8	P16 - P23
DIRB3	Byte	8	P24 - P31
DIRB4	Byte	8	P32 - P39
DIRB5	Byte	8	P40 - P47
DIRB6	Byte	8	P48 - P55
DIRB7	Byte	8	P56 - P63
DIRN0	Nib	4	P0 - P3
DIRN1	Nib	4	P4 - P7
DIRN2	Nib	4	P8 - P11
DIRN3	Nib	4	P12 - P15
DIRN4	Nib	4	P16 - P19
DIRN5	Nib	4	P20 - P23
DIRN6	Nib	4	P24 - P27
DIRN7	Nib	4	P28 - P31
DIRN8	Nib	4	P32 - P35
DIRN9	Nib	4	P36 - P39
DIRNA	Nib	4	P40 - P43
DIRNB	Nib	4	P44 - P47
DIRNC	Nib	4	P48 - P51
DIRND	Nib	4	P52 - P55
DIRNE	Nib	4	P56 - P59
DIRNF	Nib	4	P60 - P63
DIR0..DIR63	Bit	1	Any
DIR0..DIR63	Bit	1	Any

You can use binary, decimal or hexadecimal values to represent what pins are being set as inputs or outputs. The following statements all mean the same things. Based on the BasicATOM Pro 28-M P0 will be set to an output:

```
DIR0 = %0           ;Binary
DIR0 = 0            ;Decimal
DIR0 = 0x00        ;Hexadecimal
```



## Math

As with most BASIC implementations, Mbasic includes a full complement of math and comparison functions. MBasic supports 32 bit integer math, both signed and unsigned. It also supports floating point math, signed or unsigned. A signed value denotes whether the resulting value is positive or negative.

Floating point math can be processor intensive. This should be remembered any time making a decision when creating programs with math functions. There are numerous cases where floating point math is required. Hence the flexibility of the BasicATOM when compared to other available processors.

### Number Bases

Although all calculations are handled internally in binary, users can refer to numbers as decimal, hexadecimal or binary, whichever is most convenient for the programmer. For example, the number 2349 can be referred to as:

2349	Decimal
0x092D	Hexadecimal
%100100101101	Binary

Leading zeros are not required for hex or binary numbers, but may be used if desired. When using signed integers (sbyte, sword, slong) it's probably a good idea to stick to decimal notation to avoid confusion.

### Math and Operators

Operators are what makes math work, by performing a function. An example of an operator would be + (Addition), - (Subtraction), \* (Multiplication) and / (Division). All these symbols represent an operation to be performed. However the operators need something to do, so we add operands or better known as arguments. Math arguments are the values used in an expression.

In the following section you'll see the word "expression" used many times. This refers to something like 1+2 which is called an expression. The expression 1+2 has one operator (+) and two arguments(1 and 2) or operands.

## Operators

Operator	Description
-	Changes the value of an expression from positive to negative. Also used in subtraction.
ABS	Returns the absolute value of an expression.
SIN	Returns the integer sine of an expression.
COS	Returns the integer cosine of an expression.
DCD	Returns 2 to the power of an expression.
NCD	Returns the smallest power of 2 that is greater than the expression.
SQR	Returns the integer square root.
BIN2BCD	Converts expression from binary to packed binary coded decimal format.
BCD2BIN	Converts packed binary coded decimal format to a binary value.
RANDOM	Returns a random 32 bit number generated from a seed value.
-	Subtraction. Also used to sign a value.
+	Addition.
*	Multiplication, returns the low 32bits of a multiplication result.
/	Division.
**	Returns high 32 bits of a multiplication result.
*/	Fractional multiplication.
//	Remainder of Division.
MAX	limits the expression to a maximum value
MIN	limits the expression to a minimum value
DIG	returns the specified digits value
REV	reverses the order of the specified number of bits in an expression
<<	Shift left by specified amount.
>>	Shift right by specified amount.
&	Binary math AND.
	Binary math OR.
^	Binary math XOR.
&/	Binary math AND NOT.
	Binary math OR NOT.
^/	Binary math XOR NOT.
=	Is equal to.
<>	Is not equal to.
<	Is less than.
>	Is greater than.
<=	Is less than or equal to.
>=	Is greater than or equal to.
AND	Logical AND.
OR	Logical OR.
XOR	Logical XOR.
NOT	Logical NOT.
TOINT	Convert a Floating Point value to a Integer
TOFLOAT	Convert an Integer value to a Floating Point
FSQRT	Floating Point Square Root(BasicATOMPro Only)

Operator	Description
FSIN	Floating Point Sine (BasicATOMPro Only)
FCOS	Floating Point Cosine (BasicATOMPro Only)
FTAN	Floating Point Tangent (BasicATOMPro Only)
FASIN	Floating Point ArcSine (BasicATOMPro Only)
FACOS	Floating Point ArcCosine (BasicATOMPro Only)
FATAN	Floating Point Arc Tangent (BasicATOMPro Only)
FSINH	Floating Point Hyperbolic Sine (BasicATOMPro Only)
FCOSH	Floating Point Hyperbolic Cosine (BasicATOMPro Only)
FTANH	Floating Point Hyperbolic Tangent (BasicATOMPro Only)
FATANH	Floating Point Hyperbolic ArcTangent (BasicATOMPro Only)
FLN	Floating Point Natural Log (BasicATOMPro Only)
FEXP	Floating Point Exponent (BasicATOMPro Only)

### Operator Precedence

All math functions have a precedence order. This simply means each math function in an expression is calculated based on its precedence not based on the order in which it appears in the expression. This even holds true for math as it was taught in school. However the precedence of order may differ.

To solve the following equation  $2+2*5/10 = 3$  the ATOM would start with multiplication first since it has the higher precedence order.  $2*5$  will be calculated first which equals 10, then the divide  $10 / 10$  equals 1, then the addition of 2 which equals 3. The 2 was added last since it had the lowest precedence.

The multiply and divide operators have equal precedence, and both are higher precedence than addition and subtraction. Now you can change the order in which the math is performed by using parenthesis. This will force a specific order. Using parentheses, the following expression  $((2+2)*5) / 10$  would yield a result of 2.

### Precedence Table

Order	Operation
1st	NOT, ABS, SIN, COS, - (NEG), DCD, NCD, SQR, RANDOM, TOINT, TOFLOAT, BIN2BCD, BCD-2BIN, ~(Binary NOT), !(Binary NOT), NOT(Logical NOT), FSQRT, FSIN, FCOS, FTAN, FASIN, FACOS, FATAN, FSINH, FCOSH, FTANH, FATANH, FLN, FEXP
2nd	Rev, Dig
3rd	MAX, MIN
4th	*, **, */ , / , //
5th	+, -
6th	<<, >>
7th	<, <=, =, >=, >, <>
8th	&,  , ^, &/,  /, ^/
9th	And, Or, Xor

## - (Negative)

Signs an expression(integer or floating point) as its negative value.

### Example

```
Temp var Byte
Result var Byte
Temp = 1
Result = Temp + -1
```

*Temp* is first is set to equal 1. Then -1 is added. -1 is a signed integer. So *Result* now equals 0. Since 1 added to -1 equals 0.

## ABS

The Absolute Value converts a signed number to its absolute value. The absolute value of a number is the value that represents its difference from 0. The absolute value of -4 is 4. If the number is positive the result will always be the same number returned:

```
temp = abs(-1234)
temp = abs(1234)
```

The result will always be 1234 since the difference of 0 from -1234 is 1234.

## SIN, COS

For use with integer arithmetic, some modifications to the way sine and cosine work have been made. For example, in floating point math, the expression:

```
ans = sin(angle)
```

where angle is 1 radian, would return a value of 0.841... for ans. In fact, the sine of an angle must always be a fractional value between -1 and 1. MBasic can't deal with fractional values for integer math so SIN and COS are made to work with integers.

Since we are dealing with binary integers, we divide the circle into 256 (rather than 360) parts. This means that a right angle is expressed as 64 units, rather than 90 degrees. When working with Studio angular units give you a precision of about 1.4 degrees.

The result of the SIN or COS function is a signed number in the range of -127 to +128. This number divided by 128 gives the fractional value of SIN or COS.

### Example

In most "real world" applications, the angle does not need to be in degrees, nor the result be in decimal form. The following example shows a possible use of SIN with Studio values.

If a sensor returns the angle of a robotic control arm as a number from 0 to 64, where 0 is parallel and 64 is a right angle. We want to take action based on the sine of the angle.

```

    limit var byte
    angle var byte
loop
  (code that inputs the value of "angle")
  limit = sin angle
  if limit > 24 then first
  if limit > 48 then second
  goto loop
first
  code to warn of excessive angle
  goto loop
second
  code to shut down equipment
  etc...
```

This will warn the operator if the arm angle exceeds approximately 8 units (11.25 degrees) and shut down the equipment if the arm angle exceeds approximately 16 units (22.5 degrees). Most control examples don't need to work in actual degrees or decimal values of sine or cosine. To find the sine of a 60 degree angle, first convert the angle to MBasic units by multiplying by 256 and dividing by 360. For example:

```
angle = 60 * 256 / 360
```

will result in a value of 42. (It should actually be 42.667, which rounds to 43, but with integer arithmetic the decimal fraction is ignored, and the integer is not rounded up.) Then find the sine of this angle:

```
ans = sin angle
```

This will give a result of 109. Dividing this value by 128 will give the decimal value of 0.851 (compared to the correct floating point value which should be 0.866). You can't directly get the decimal value by doing this division within Studio Basic (you would get a result of 0). However, you could first multiply



by 1000, then divide by 128 to get 851 as your result.

## DCD

Similar to the “exp” function in some other BASIC implementations. Returns 2 to the power DCD.

### Example

If the value of “num” is 7, then the following statement will return a value of 2 to the power of 7, or 128.

```
answer = dcd num
```

Since the returned value increases exponentially, make sure your target variable (“answer” in this case) is correctly defined to accommodate the largest value anticipated. If the target variable is too small, only the low order bits of the result will be stored.

## NCD

This function returns the smallest power of 2 that is greater than the argument.

### Example

If the value of “num” is 51, the following statement will return the value of 6. Since  $2^5 = 32$  and  $2^6 = 64$ , 6 is the smallest power of 2 greater than 51.

```
answer = ncd num
```

## SQR (Square Root)

Returns the integer portion of the square root of the argument. Increased precision can be obtained by multiplying the argument by an even square of 10, such as 100 or 10000.

### Example 1

If the value of "num" is 64, the following statement will return the value of 8 (which is the square root of 64).

```
answer = sqr num
```

### Example 2

If the value of "num" is 220, the following statement will return the value 14, which is the integer portion of 14.832..., the square root of 220.

```
answer = sqr num
```

### Example 3

If more precision is required, multiply the argument by 100 or 10000. Using the example where "num" = 220 a value 148 is returned, which is 10 times the square root of 220.

```
answer = sqr (num * 100)
```

Alternately,

```
answer = sqr (num * 10000)
```

will return the value 1483, which is 100 times the square root of 220.

## BIN2BCD, BCD2BIN

These commands let you convert back and forth between binary and “packed” binary coded decimal (BCD). A BCD number is one in which each decimal digit is represented by a 4 bit binary number (from 0 to 9). Packed BCD packs two 4 bit decimal digits in a single byte of memory.

For example, the decimal number 93 is represented in binary as:

Values	128	64	32	16	8	4	2	1
Binary	0	1	0	1	1	1	0	1

The same number is expressed in packed BCD as:

Values	8	4	2	1	8	4	2	1
Binary	1	0	0	1	0	0	1	1

### Example

Assuming that “ans” is a byte variable and “num” has the decimal value of 93, the statement

```
answer = bin2bcd num
```

will set ans to a binary value of 10010011 (which is 93 in packed BCD).

## RANDOM

Generates a 32 bit (Long) random number from the seed value. As with most random number generators, the random numbers generated will follow a predictable pattern, and each time the program is run the random number sequence will be the same if the seed value is the same. The below code snippet will return a pseudo random set of numbers by re-seeding from the results:

```
seed var long
seed = 123456
seed = random seed
```

There are steps that can be taken to avoid repeating random number sequences. This is typically done using hardware based features. One common method is using an internal hardware timer for the seed value and asking the user to press a button at the beginning of a game. Each time the button is pressed the timer value will likely be different. Another method is reading an A/D pin that is left floating and near a noisy signal trace.

## - (Subtraction)

Subtract a value (integer or floating point) from another value. The resulting number is not signed unless a signed variable is used. An example of subtraction:

```
time var byte
time = 100
time = time - 1
```

The variable time will now equal 99 since we subtracted 1 from 100.

## + (Addition)

Add one value (integer or floating point) to another value. The resulting number is not signed unless a signed variable is used. An example of addition:

```
time var byte
time = 100
time = time + 1
```

The variable time will now equal 101 since we added 1 to 100.

## \* (Multiplication)

Multiply one value(integer or floating point) by another value. The resulting number is not signed unless a signed variable is used. An example of addition:

```
time var byte
time = 100
time = time * 1
```

The variable time will now equal 100 since we multiplied 100 x 1.

## / (Division)

Divide one value (integer or floating point) by another value. Integer division discards fractional results. For example:

```
result = 76/7
```

will set the variable "result" to a value of 10. (The actual decimal result should be 10.857, but the decimal part is discarded, rounding is not done.) If your application requires fractional results you can use floating point numbers or the following solution.

Use a floating point variable instead to get the full precision.

```
result var float  
result = 76.0/7.0
```

Alternatively when using integer variables multiply the dividend by 10, 100, 1000 etc. before dividing. The result will gain extra digits of precision but must be interpreted correctly. Using the previous example we can gain three digits of precision as follows. This is known as fixed point division:

```
temp = dividend * 1000 ;dividend is now 76000  
result = temp/7
```

Which sets "result" to a value of 10857.

## \*\* (High Multiplication)

If two long variables or constants are multiplied, the result may exceed 32 bits. Normally, the multiply function will return the least significant (lowest) 32 bits. The \*\* function will, instead, return the most significant 32 bits.

```
time = 80000 ** 80000 ; result returns high 32 bits
```

The value of time would be equal to 0x1 which is the high 32 bits of the result 6,400,000,000.

## \*/ (Fractional Multiplication)

Fractional multiplication will multiply by a number with a fractional part. The multiplier must be a long value, and it is handled in a special fashion. The high 16 bits are the integer portion of the multiplier, the low 16 bits are the fractional part (expressed as a fraction of 65536). The result, of course, will be an integer; any fractional part is discarded (not rounded).

## Example

Let us say we want to multiply the number 346 x 2.5. The multiplier must be constructed as follows: The high 16 bits will have a value of 2. We can do this with:

```
mult.highword = 2
```

The low 16 bits will have a value of half of 65535, or 32782, so:

```
mult.lowword = 32782
```

Then we do the fractional multiply:

```
a = 346 */ mult
```

which will give “a” the value 865. A similar procedure will let you multiply by any fraction; simply express that fraction with a denominator of 65535 as closely as possible.

Notice that half of 65535 is actually 32782.5; a number we can't enter as the fractional part. This means that multiplication by exactly  $\frac{1}{2}$  is not possible. However, the difference is so small that it has no effect on the actual outcome of the integer result.// (mod)

The mod function (short for “modulo”) returns the remainder after an integer division. So, for example, 13 modulo 5 is 3 (the remainder after dividing 13 by 5).

The mod function can be used to determine if a number is odd or even, as shown here:

```
x var word
  y var word
  (code that sets the value of x)
  y = x//2
  if y=0 goto even ;zero indicates an even number
  if y=1 goto odd  ;one indicates an odd number
even
  (more code)
odd
  (more code)
```

Similarly, the mod function can be used to determine if a number is divisible by any other number.

## MAX

The MAX function returns the smaller of two expressions(integer or floating point). For example:

```
x var word
y var word
code to set value of y
x = y max 13
```

The example will set x to the value of y or 13, whichever is smaller. Think of this as “x equals y up to a maximum value of 13”.

## MIN

The MIN function returns the larger of two expressions(integer or floating point). For example:

```
x var word
y var word
code to set value of y
x = y min 9
```

The example will set y to the value of x or 9, whichever is larger. Think of this as “x equals y down to a minimum value of 9”.

## DIG

The DIG (digit) function is used to isolate a single digit of a decimal number. For example:

```
x var word
y var byte
(code to set y) ;say the result is y=17458
x = y dig 4 ;gives the 4th digit of y, which is 7
```

Digits are counted from the right, starting with 1. The DIG function will work with numbers in decimal format only. If you need to find a specific digit in a hex or binary number, use a variable modifier.

## REV

The REV function works directly in binary, but the results may be expressed in any form. It is used to “reverse” the order of the bits of a number. Example:

```
x var byte
y var byte
x = %101110 ;this is decimal 46
y = x rev 3 ;gives g a value of %101011
```

## << (Shift Left)

The Shift Left operator shifts all the bits of a value to the left by a specified amount. Shifting left is the same as multiplying the value by 2 to the  $n$ th power. Bits shifted off the left end are lost. Zeros are added to the right for vacant bits. The example program will display the value of time before and after shifting left by 4. The results will be displayed in binary:

```
time var byte
serout sout, i9600,[bin time]
time = time << 4
serout sout, i9600,[bin time]
```

Important: The sign bit is not preserved so this function should not be used with signed numbers.

## >> (Shift Right)

The Shift Right operator shifts all the bits of a value to the right by a specified amount. Shifting right is the same as dividing the value by 2 to the  $n$ th power. Bits shifted off the right end are lost. Zeros are added to the left for vacant bits. The example program will display the value of time before and after shifting right by 4. The results will be displayed in binary:

```
time var byte
serout sout, i9600,[bin time]
time = time >> 4
serout sout, i9600,[bin time]
```

Important: The sign bit is not preserved so this function should not normally be used with signed numbers.



## & (AND)

The & function is a binary operator. It sets the result to 1 if both matching bits are 1's. Or to 0 if either or both bits are 0's.

$1 \& 1 = 1$   
 $1 \& 0 = 0$   
 $0 \& 1 = 0$   
 $0 \& 0 = 0$

Value1	0	1	0	1	1	1	0	1
Value2	1	0	0	1	0	0	1	1
Result	0	0	0	1	0	0	0	1

One useful function for AND is to “mask” certain bits of a number. For example, if we are interested only in the low 4 bits of a number, and want to ignore the high 4 bits, we could & the number with 00001111 as shown here:

Value1	0	1	0	1	1	1	0	1
Value2	0	0	0	0	1	1	1	1
Result	0	0	0	0	1	1	0	1

As you can see, the high 4 bits are now all set to 0's, regardless of their original state, but the low 4 bits retain their original state.

## | (OR)

The | function is a binary operator. It set the bit to 1 If either or both of the matching bits are 1. Or to 0 if both bits are 0's.

$1 | 1 = 1$   
 $1 | 0 = 1$   
 $0 | 1 = 1$   
 $0 | 0 = 0$

Value1	0	1	0	1	1	1	0	1
Value2	1	0	0	1	0	0	1	1
Result	1	1	0	1	1	1	1	1

$\wedge$  (Exclusive OR)

The  $\wedge$  function is a binary operator. It sets the resulting bits to a 1 if either, but not both, of the matching bits are 1. Or to 0 otherwise.

$1 \wedge 1 = 0$   
 $1 \wedge 0 = 1$   
 $0 \wedge 1 = 1$   
 $0 \wedge 0 = 0$

Value1	0	1	0	1	1	1	0	1
Value2	1	0	0	1	0	0	1	1
Result	1	1	0	0	1	1	1	0

 $\&/$  (AND NOT)

The AND NOT function is a binary operator. It compares the bits of two values. It sets the result bits to 1 if neither bit is set(0) or to a 1 for all other cases.

$1 \&/ 1 = 1$   
 $1 \&/ 0 = 1$   
 $0 \&/ 1 = 1$   
 $0 \&/ 0 = 0$

Value1	0	1	0	1	1	1	0	1
Value2	1	0	0	1	0	0	1	1
Result	1	1	1	0	1	1	1	0

 $|/$  (OR NOT)

The OR NOT function is a binary operator. It compares the bits of two values bit by bit and sets the result to a 1 if neither bit is a 1. All other conditions will return a 0.

$1 | / 1 = 0$   
 $1 | / 0 = 0$   
 $0 | / 1 = 0$   
 $0 | / 0 = 1$

Value1	0	1	0	1	1	1	0	1
Value2	1	0	0	1	0	0	1	1
Result	0	0	1	0	0	0	0	0

## $\wedge$ (XOR NOT)

The XOR NOT function is a binary operator, It compares two values bit by bit and sets the result to a 1 if neither bit is a 0 but not both. All other conditions will return a 0.

```
1 ^ 1 = 1
1 ^ 0 = 0
0 ^ 1 = 0
0 ^ 0 = 1
```

Value1	0	1	0	1	1	1	0	1
Value2	1	0	0	1	0	0	1	1
Result	0	0	1	1	0	0	0	1

## = (Equal)

The Equal (=) is a logic operator and lets something be equal.

```
temp = 10
```

The example code sets temp to 10.

## <> (NOT Equal To)

The NOT Equal (<>) is a logic operator and compares if something is not equal to some value.

```
if temp <> 10 then
```

The conditional statement will check to see if temp is not equal to 10. If the value of temp is lower or greater the comparison will be true.

## < (Less Than)

The Less Than (<) is a logic operator and compares if something is less than some value.

```
if temp < 10 then
```

The conditional statement will check to see if temp is less than 10. If the value of temp is lower the comparison is true. Any value equal to or over 10 will be false.

## > (Greater Than)

The Greater Than (>) is a logic operator and compares if something is greater than some value.

```
if temp > 10 then
```

The conditional statement will check to see if temp is greater than 10. If the value of temp is higher the comparison is true. Any value from 0 to 10 will be false.<= (Less Than or Equal To)

The Less Than or Equal To (<=) is a logic operator and compares if something is less than or equal to some value.

```
if temp <= 10 then
```

The conditional statement will check to see if temp is less than or equal to 10. If the value of temp is less than 10 or equal to 10 the comparison is true. Any value from 11 and up is false.

## >= (Greater Than or Equal To)

The Greater Than or Equal To (>=) is a logic operator and compares if something is greater than or equal to some value.

```
if temp >= 10 then
```

The conditional statement will check to see if temp is greater than or equal to 10. If the value of temp is greater than 10 or equal to 10 the comparison is true. Any value from 0 to 9 is false.

## AND

The AND operator is a logic comparison operator. It compares two conditions to make a single true or false statement. The AND operator will return a true only if both conditions are true. If one condition is false then a false is returned. The truth table demonstrates all combinations:

Condition 1	Condition 2	Result
True	True	True
True	False	False
False	True	False
False	False	False

The AND operator is used in decision making commands such as IF..THEN, DO..WHILE and so on. It differs from the & operator which is used in binary math functions. Example of the AND operator:

```
if minute = 10 AND hour = 1 then alarm
```

The conditional statement will check to see if both expressions are true before returning a true and jumping to the alarm label. If one of the expressions is not true a false is returned and the label is skipped. The IF..THEN only jumps to the label if the statement is true.

## OR

The OR operator is a logic comparison operator. It compares two conditions to make a single true or false statement. The OR operator will return a true if one or both conditions are true. If both conditions are false then a false is returned. The truth table demonstrates all combinations:

Condition 1	Condition 2	Result
True	True	True
True	False	True
False	True	True
False	False	False

The OR operator is used in decision making commands such as IF..THEN, DO..WHILE and so on. It differs from the | operator which is used in binary math functions. Example of the OR operator:

```
if hour = 12 OR minute = 30 then ding
```

The conditional statement will check to see if either expressions is true before returning a true and jumping to the ding label. If both of the expressions are false the label is skipped. The IF..THEN only jumps to the label if the statement is true.

## XOR

The XOR operator is a logic comparison operator. It compares two conditions to make a single true or false statement. The XOR operator will return a true if one but not both conditions are true. If both conditions are true or false then a false is returned. The truth table demonstrates all combinations:

Condition 1	Condition 2	Result
True	True	False
True	False	True
False	True	True
False	False	False

The XOR operator is used in decision making commands such as IF..THEN, DO..WHILE and so on. It differs from the ^ operator which is used in binary math functions. Example of the XOR operator:

```
if hour > 5 XOR hour = 5 then QuitTime
```

The conditional statement will check to see if either expressions is true before returning a true and jumping to the quittime label. If both of the expressions are false or true the label is skipped. The IF..THEN only jumps to the label if the statement is true.

---

## NOT

The NOT operator is a logic operator. It inverts a condition. When used, true becomes false and false becomes true. The truth table demonstrates all combinations:

Condition	Result
True	False
False	True

The NOT operator is used in decision making commands such as IF..THEN, DO..WHILE and so on. Example of the NOT operator:

```
if hour = 5 then Quit_Time  
if NOT hour < 5 then Over_Time
```

If hour is not equal to 5 the first conditional statement will skip Quit\_Time. In the second conditional statement if hour is not less than 5 it jumps to the label Over\_Time since the NOT operator inverted the result of the condition. The IF..THEN only jumps to the label if the statement is true.

## Floating Point Math

MBasic includes floating point math functions. These functions can be processor intensive. The time it takes will vary based on the processor used. The AtomPro40m being the fastest since it has built in 32 bit math hardware. This should be remembered any time making a decision when creating programs with floating point math functions.

### Floating Point Operators

Operator	Description
TOINT	Convert a Floating Point value to a Integer
TOFLOAT	Convert an Integer value to a Floating Point
FSQRT	Floating Point Square Root(BasicATOMPro Only)
FSIN	Floating Point Sine (BasicATOMPro Only)
FCOS	Floating Point Cosine (BasicATOMPro Only)
FTAN	Floating Point Tangent (BasicATOMPro Only)
FASIN	Floating Point ArcSine (BasicATOMPro Only)
FACOS	Floating Point ArcCosine (BasicATOMPro Only)
FATAN	Floating Point Arc Tangent (BasicATOMPro Only)
FSINH	Floating Point Hyperbolic Sine (BasicATOMPro Only)
FCOSH	Floating Point Hyperbolic Cosine (BasicATOMPro Only)
FTANH	Floating Point Hyperbolic Tangent (BasicATOMPro Only)
FATANH	Floating Point Hyperbolic ArcTangent (BasicATOMPro Only)
FLN	Floating Point Natural Log (BasicATOMPro Only)
FEXP	Floating Point Exponent (BasicATOMPro Only)

### TOINT

The TOINT operator explicitly converts a Floating Point value into an integer value. The decimal point of the Floating Point number is truncated.

```
myfloat var float
myint var long

myfloat = 10.0 ;myfloat now equals 10.0
myfloat = myfloat * 123.123 ;myfloat now equals 1231.23
myint = TOINT myfloat ;myint now equals 1231
serout s_out,i9600,["result = ",sdec myint,13]
```

### TOFLOAT

The TOFLOAT operator explicitly converts an Integer value into a Floating Point value.

```
myint var long
myfloat var float

myint = 10
myfloat = TOFLOAT myint / 100.0 ;myfloat now equals 0.1
serout s_out,i9600,["result = ",real myfloat,13]
```

## FSQRT

Returns the floating point square root of the argument. A square root is the number, that when multiplied by itself will equal the original argument.

If the value of "num" is 2, the following statement will return the value of 1.4142.

```
myfloat var float
myfloat = FSQRT 2.0
serout s_out,i9600,["result = ",real myfloat,13]
```

## FSIN

The FSIN operator calculates the Floating Point Sine of an angle in radians. FSIN gives the ratio of the length of the side opposite of the angle to the length of the hypotenuse in a right triangle.

```
myangle var float
mysin var float

myangle = 3.14159/2.0 ;myangle now equals PI/2 degrees in radians
mysin = FSIN myangle ;mysin now equals 1.0
serout s_out,i9600,["mysin = ",real mysin,13]
```

## FCOS

The FCOS operator calculates the Floating Point Cosine of an angle in radians. FCOS gives the ratio of the length of the side adjacent the angle to the length of the hypotenuse in a right triangle.

```
myangle var float
mycos var float

myangle = 0.0 ;myangle now equals 0 degrees in radians
mycos = FCOS myangle ;mycos now equals 1.0
serout s_out,i9600,["mycos = ",real mycos,13]
```

## FTAN

The FTAN operator calculates the Floating Point Tangent of an angle in radians. FTAN gives the ratio of the length of the side opposite the angle to the length of the side adjacent to the angle in a right triangle

```
myangle fcon 3.14159/4 ;myangle is 45 degrees in radians
myadj fcon 100.0
myopp var float

;This calculation finds the triangles opposite side length
;for a right triangle with angle of 45 degrees and
;adjacent side of 100.
myopp = FTAN myangle * myadj ;myopp now equals 100.0
serout s_out,i9600,["myopp = ",real myopp,13]
```



## FASIN

The FASIN operator calculates the Floating Point Arc Sine of a value. FASIN returns the angle in radians given the ratio of the length of the size opposite the angle and the length of the hypotenuse in a right triangle.

```
mysin fcon 1.0
myangle var float

myangle = FASIN mysin ;myfloat equals PI/2
serout s_out,i9600,["myangle = ",real myangle,13]
```

## FACOS

The FACOS operator calculates the Floating Point Arc Cosine of a value. FACOS returns the angle in radians given the ratio of the length of the size adjacent the angle and the length of the hypotenuse in a right triangle.

```
mycos fcon 0.0
myangle var float

myangle = FACOS mycos ;myfloat equals 0
serout s_out,i9600,["myangle = ",real myangle,13]
```

## FATAN

The FATAN operator calculates the Floating Point Arc Tangent of a value. FATAN returns the angle in radians given the ratio of the length of the size opposite the angle and the length of the side adjacent to the angle in a right triangle.

```
mytan fcon 1.0
myangle var float

myangle = FATAN mytan ; myfloat equals PI/4
serout s_out,i9600,["myangle = ",real myangle,13]
```

## FLN

The FLN operator calculates the Floating Point Natural Log of a value. The natural log is used to calculate the time it takes for compound growth to reach the specified amount. For example FLN 20.08 will equal approximately 3. This means it takes 3 growth cycles (the amount of time it takes to grow 100%) to reach 20.08 times the original amount.

```
result var float

result = FLN 2.0 ;result now equals 0.69315
serout s_out,i9600,["result = ",real result,13]
```

## FEXP

The FEXP operator calculates the Floating Point Natural Exponent of a value. The natural exponent calculates the inverse of the natural log. Given time how much will something grow. FEXP 3 will equal approximately 20.08 times the original quantity.

```
result var float
result = FEXP 0.693115 ;result now equals 2.0
serout s_out,i9600,["result = ",real result,13]
```

## Hyperbolic Functions

The hyperbolic trigonometric functions are found in many scientific calculations, including in Calculus, Special Relativity and calculating the Catenary of a hanging wire. Describing their uses in detail is beyond the scope of this manual.

## FSINH

The FSINH operator calculates the Floating Point Hyperbolic Sine of the value.

```
param var float
result var float
param = FLN 2.0
result = FSINH param
serout s_out,i9600,["The hyperbolic sine of ",real param," is ",real
result,13]
```

## FCOSH

The FCOSH operator calculates the Floating Point Hyperbolic Cosine of the value.

```
param var float
result var float
param = FLN 2.0
result = FCOSH param
serout s_out,i9600,["The hyperbolic cosine of ",real param," is ",real
result,13]
```

## FTANH

The FTANH operator calculates the Floating Point Hyperbolic Tangent of the value.

```
param var float
result var float

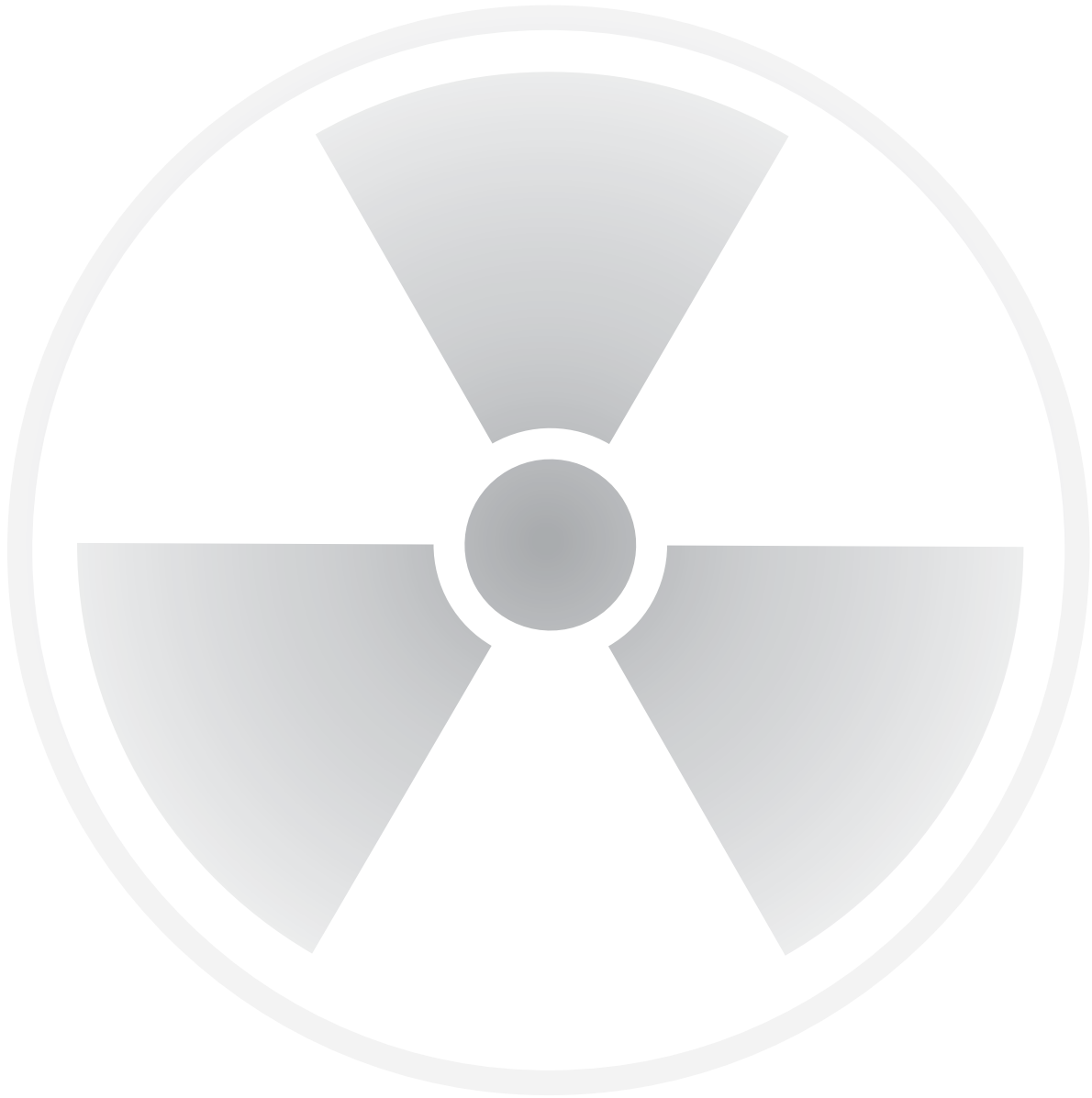
param = FLN 2.0
result = FTANH param
serout s_out,i9600,["The hyperbolic tangent of ",real param," is ",real
result",13]
```

## FATANH

The FATANH operator calculates the Floating Point Hyperbolic Arc Tangent of the value.

```
param var float
result var float

param = 0.6
result = FEXP (FATANH param)
serout s_out,i9600,["The result is ",real result,13]
```



## Modifiers

In MBasic all values are received and stored as binary. Modifiers were created for formatting the data used in commands that have input or output (SERIN / SEROUT). The modifiers are useful for formatting data being printed to a terminal window. All characters are represented in ASCII. Modifiers format the ASCII values to display properly based on the modifier used.

An example of a command modifier is formatting a decimal value. The decimal value 32 would output to a terminal window as a space character. Instead, to display the actual decimal value of the variable you would use the DEC modifier:

```
Temp Var Byte
Temp = 32

serout s_out, i9600, [DEC TEMP]
```

The code snippet above would display the value of temp which is set to 32. If the DEC modifier wasn't used a space character(" ") would be displayed instead.

Several modifiers are input only modifiers such as WAIT which would cause an input command to wait until the received data matches a specified string.

Some modifiers have variants. In SBIN the "S" stands for signed. Where in IBIN the "I" represents the indicator. The indicator is the symbol used to indicate the numeric type such as binary (%) or hexadecimal (\$).

Modifiers can be used with the following commands:

Output Modifiers	Input Modifiers
DEBUG	DEBUGIN
I2COUT	I2CIN
OWOUT	OWIN
SEROUT	SERIN
HSEROUT	HSERIN
LCDWRITE	LCDREAD
WRITEDM	READDM
DTMFOUT	
DTMFOUT2	

Output modifiers can also be used to modify array variables:

```
string var byte(100)

string = "Hello World" ;string(0-10)="Hello World"
string = dec 1234567    ;string(0-6)="1234567"
string = ihex 0x3456   ;string(0-4)="$3456"
```

## Modifiers

Name	Input	Output	Description
DEC	x	x	Decimal.
SDEC	x	x	Signed decimal.
HEX	x	x	Hexadecimal.
SHEX	x	x	Signed hexadecimal.
IHEX	x	x	Indicated (\$) hexadecimal.
ISHEX	x	x	Signed and indicated (\$) hexadecimal.
BIN	x	x	Binary.
SBIN	x	x	Signed binary.
IBIN	x	x	Indicated (%) binary.
ISBIN	x	x	Indicated (%) and signed binary.
REP		x	Repeat character <i>n</i> times.
REAL	x	x	Floating point number with decimal point.
STR	x	x	Read or write specified amount of characters and store in an array.
SKIP	x		Skip specified amount of characters.
WAIT	x		Wait for specified amount of characters.
WAITSTR	x		Compares specified amount of characters to array.

## DEC

DEC{#max} expression{\#min}

#max: optional maximum number of digits to send

#min: optional minimum number of digits to send

The DEC modifier when used in an output command will convert a stored values to ASCII characters. The example will format the value of *temp* so it prints out the number in a terminal window. The output would display 1234.

```
temp var word
temp = 1234

serout s_out, i9600, [DEC temp] ;prints "1234"
```

DEC(#max) variable

#max: optional maximum number of digits to receive

The DEC modifier for input commands will format incoming ASCII characters into a numeric value. The example will read in ASCII characters that represent decimal numbers up to 9 characters long and store the converted value in a variable. Until a numeral is received(eg "0" to "9") any incoming characters are ignored. Once a numeral has been received any character not a numeral will cause the conversion to finish before 9 characters have been received.

```
temp var word

serin s_out, i9600, [DEC temp]
```

## SDEC

SDEC{#max} expression{#min}

#max: optional maximum number of digits to send

#min: optional minimum number of digits to send

The DEC modifier when used in an output command will convert a stored values to ASCII characters. The example will format the value of *temp* so it prints out the number in a terminal window. The output would display 1234.

```
temp var sword
temp = -1234

serout s_out, i9600, [SDEC temp] ;prints "-1234"
```

## SDEC(#max) variable

#max: optional maximum number of digits to receive

The SDEC modifier for input commands will format incoming ASCII characters into a numeric value. The example will read in ASCII characters that represent decimal numbers up to 9 characters long and store the converted value in a variable. Until a negative sign or numeral is received(eg "0" to "9") any incoming characters are ignored. Once a numeral has been received any character not a numeral will cause the conversion to finish before 9 characters have been received.

```
temp var sword

serin s_out, i9600, [SDEC temp]
```



## HEX

HEX{#max} expression{\#min}

#max: optional maximum number of digits to send

#min: optional minimum number of digits to send

The HEX modifier, in output commands, converts stored values to ASCII characters. The example will format the value of *temp* so it prints out the number in a terminal window. The output would display 12ab.

```
temp var word
temp = 0x12ab

serout s_out, i9600, [HEX temp] ;prints "0x12ab"
```

HEX(#max) variable

#max: optional maximum number of digits to receive

The HEX modifier, for input commands, formats incoming ASCII characters into a numeric value. The example will read in ASCII characters that represent hexadecimal numbers up to 8 characters long and store the converted value in a variable. Until a numeral is received(eg "0" to "9" or "a" to "f") any incoming characters are ignored. Once a numeral has been received any character not a numeral will cause the conversion to finish before 8 characters have been received.

```
temp var word

serin s_out, i9600, [HEX temp]
```

## SHEX

SHEX{#max} expression{#min}

#max: optional maximum number of digits to send

#min: optional minimum number of digits to send

The SHEX modifier, in output commands, converts stored values to ASCII characters. The example will format the value of *temp* so it prints out the number in a terminal window. The output would display -12AB.

```
temp var word
temp = -0x12ab

serout s_out, i9600, [SHEX temp] ;prints "-12ab"
```

SHEX(#max) variable

#max: optional maximum number of digits to receive

The SHEX modifier, for input commands, formats incoming ASCII characters into a numeric value. The example will read in ASCII characters that represent hexadecimal numbers up to 8 characters long and store the converted value in a variable. Until a negative sign or numeral is received (eg "0" to "9" or "a" to "f") any incoming characters are ignored. Once a numeral has been received any character not a numeral will cause the conversion to finish before 8 characters have been received.

```
temp var sword

serin s_out, i9600, [SHEX temp]
```

## IHEX

IHEX{#max} expression{\#min}

#max: optional maximum number of digits to send

#min: optional minimum number of digits to send

The IHEX modifier, in output commands, converts stored values to ASCII characters. The example will format the value of *temp* so it prints out the number in a terminal window. The output would display \$12ab.

```
temp var word
temp = 0x12ab

serout s_out, i9600, [SHEX temp] ;prints "$12ab"
```

IHEX{#max} variable

#max: optional maximum number of digits to receive

The IHEX modifier, for input commands, formats incoming ASCII characters into a numeric value. The example will read in ASCII characters that represent hexadecimal numbers up to 8 characters long and store the converted value in a variable. Until the indicator is received, "\$", any incoming characters are ignored. Once the indicator has been received any character not a numeral will cause the conversion to finish before 8 characters have been received.

```
temp var word

serin s_out, i9600, [IHEX temp]
```

## ISHEX

ISHEX{#max} expression{\#min}

#max: optional maximum number of digits to send

#min: optional minimum number of digits to send

The ISHEX modifier, in output commands, converts stored values to ASCII characters. The example will format the value of *temp* so it prints out the number in a terminal window. The output would display `12ab`.

```
temp var sword
temp = -0x12ab

serout s_out, i9600, [ISHEX temp] ;prints "12ab"
```

ISHEX{#max} variable

#max: optional maximum number of digits to receive

The ISHEX modifier, for input commands, formats incoming ASCII characters into a numeric value. The example will read in ASCII characters that represent hexadecimal numbers up to 8 characters long and store the converted value in a variable. Until the indicator, "\$", is received any incoming characters are ignored. Once the indicator has been received any character not a numeral will cause the conversion to finish before 8 characters have been received.

```
temp var sword

serin s_out, i9600, [ISHEX temp]
```

## BIN

`BIN{#max} expression{#min}`

`#max`: optional maximum number of digits to send

`#min`: optional minimum number of digits to send

The BIN modifier, in output commands, converts stored values to ASCII characters. The example will format the value of `temp` so it prints out the number in a terminal window. The output would display 1100.

```
temp var word
temp = 0x0C

serout s_out, i9600, [HEX temp] ;prints "1100"
```

`BIN{#max} variable`

`#max`: optional maximum number of digits to receive

The BIN modifier, for input commands, formats incoming ASCII characters into a numeric value. The example will read in ASCII characters that represent hexadecimal numbers up to 8 characters long and store the converted value in a variable. Until a numeral is received(eg "0" to "1") any incoming characters are ignored. Once a numeral has been received any character not a numeral will cause the conversion to finish before 8 characters have been received.

```
temp var word

serin s_out, i9600, [BIN temp]
```

## SBIN

SBIN{#max} expression{\#min}

#max: optional maximum number of digits to send

#min: optional minimum number of digits to send

The SBIN modifier, in output commands, converts stored values to ASCII characters. The example will format the value of *temp* so it prints out the number in a terminal window. The output would display 1100.

```
temp var sword
temp = -0x0C

serout s_out, i9600, [HEX temp] ;prints "-1100"
```

SBIN{#max} variable

#max: optional maximum number of digits to receive

The SBIN modifier, for input commands, formats incoming ASCII characters into a numeric value. The example will read in ASCII characters that represent hexadecimal numbers up to 8 characters long and store the converted value in a variable. Until a negative sign or numeral is received (eg "0" to "1") any incoming characters are ignored. Once a numeral has been received any character not a numeral will cause the conversion to finish before 8 characters have been received.

```
temp var sword

serin s_out, i9600, [SBIN temp]
```

## IBIN

IBIN{#max} expression{#min}

#max: optional maximum number of digits to send

#min: optional minimum number of digits to send

The IBIN modifier, in output commands, converts stored values to ASCII characters. The example will format the value of *temp* so it prints out the number in a terminal window. The output would display %1100.

```
temp var word
temp = 0x0C

serout s_out, i9600, [IBIN temp] ;prints "1100"
```

IBIN{#max} variable

#max: optional maximum number of digits to receive

The IBIN modifier, for input commands, formats incoming ASCII characters into a numeric value. The example will read in ASCII characters that represent hexadecimal numbers up to 8 characters long and store the converted value in a variable. Until the indicator is received any incoming characters are ignored. Once the indicator has been received any character not a numeral will cause the conversion to finish before 8 characters have been received.

```
temp var word

serin s_out, i9600, [IBIN temp]
```

## ISBIN

ISBIN{#max} expression{#min}

#max: optional maximum number of digits to send

#min: optional minimum number of digits to send

The ISBIN modifier, in output commands, converts stored values to ASCII characters. The example will format the value of *temp* so it prints out the number in a terminal window. The output would display %-1100.

```
temp var sword
temp = -0x0C

serout s_out, i9600, [ISBIN temp] ;prints "-1100"
```

ISBIN(#max) variable

#max: optional maximum number of digits to receive

The ISBIN modifier, for input commands, formats incoming ASCII characters into a numeric value. The example will read in ASCII characters that represent hexadecimal numbers up to 8 characters long and store the converted value in a variable. Until the indicator is received any incoming characters are ignored. Once the indicator has been received any character not a numeral will cause the conversion to finish before 8 characters have been received.

```
temp var sword

serin s_out, i9600, [ISBIN temp]
```

## REP

The REP modifier will output the character *n* a specified number of times. The example will repeat the specified character "A" 10 times.

```
serout s_out, i9600, [REP "A"\10] ;prints A 10 times
```



## REAL

REAL{#maxint} expression{\#maxdec}

#maxint: optional maximum number of integer digits to send

#maxdec: optional maximum number of decimal point digits to send

The REAL modifier, in output commands, converts stored values to ASCII characters. The example will format the value of *temp* so it prints out the number in a terminal window. The output would display 3.14159.

```
temp var float
temp = 3.14159

serout s_out, i9600, [REAL temp] ;prints "3.14159"
```

## REAL variable

The REAL modifier, for input commands, formats incoming ASCII characters into a numeric value. The example will read in ASCII characters that represent floating point numbers and store the converted value in a variable. Until a numeral is received any incoming characters are ignored. Once a numeral has been received any character not a numeral will cause the conversion to finish before 8 characters have been received.

```
temp var float

serin s_out, i9600, [REAL temp]
```

## STR

STR array\length{\eol}

The STR modifier, in output commands, will output *Length* amount of characters from specified constant or variable array until the end of the array or until an optional specified *Eol* character is found.

```
temp var byte(20)
temp = "Hello world",0

serout s_out,i9600,[str temp\20\0] ;output "Hello world"
```

STR array\length{\eol}

The STR modifier, in input commands, will receive *Length* number of characters and store them in the variable array specified. An optional end of line(*Eol*) character can be specified to tell STR to stop receiving characters.

```
temp var byte(20)

serin s_in,i9600,[str temp\20\13] ; receive upto 20 characters
```

## SKIP

The SKIP modifier will skip *n* amount of characters.

```
serin s_in,i9600,[SKIP 10, temp] ;skip the first 10 values received
```

## WAIT

The WAIT modifier will wait for a specific constant string of characters.

```
serin s_in,i9600,[WAIT("Hello World")] ;wait for "Hello world"
```

## WAITSTR

WAITSTR array\length{\eol}

The WAITSTR modifier will wait for *Length* amount of characters comparing them to an array. Syntax is the same as the STR input modifier. The end of line character is compared against the existing array data. If it matches the current array character being compared the wait is finished.

```
temp var byte(10)
temp = "hello",13

serin s_in,i9600,[waitstr temp\10\13] ;waits until it receives "hello"
```



---

## Command Reference

The following section outlines the syntax and general use of each command supported in MBasic. Not all commands support all BasicATOM processors. Each command indicates what processors are supported with a simple “Supported” list. The supported processor list use abbreviations to indicate each processor family.

### Supported List Abbreviations

Abbreviation	Description
BA	All BasicATOM modules and Nano X chips.
BAN	Only standard Nano chips.
BAP	BasicATOM Pro One, 24m and 28m modules.
BAP40	BasicATOM Pro 40m and ARC32.

### Syntax

The syntax for each command is given in the command reference section. In addition Basic Micro Studio has built in syntax help. As a command is typed Basic Micro Studio will show the syntax for that command as its being typed.

## ADIN

### Syntax

adin pin, result

- **Pin** - is a variable or constant that specifies the pin to use for the analog reading. Must be an analog capable pin.
- **Result** - is a word(or larger) sized variable used to store the analog results.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The ADIN command directly accesses the built-in analog hardware. The analog hardware can read any analog voltage from 0 to 5VDC. This will give 1024 positions. GND = 0 and 5V = 1023. Depending on the module type you are using the speed at which analog samples are performed varies. Each processors analog pins differ. See the table below.

### Analog Capable Pins

Processor	Pins
Nano 18	P6 to P11
Nano 28	P0 to P5 and P16 to P20
Nano 40	P0 to P5 and P24 to P31
Atom 24m	AX0, AX1, AX3
Atom 28m	P0 to P5 and P16 to P19
Atom 40m	P0 to P5 and P24 to P31
AtomPro One	P0 to P3
AtomPro 24m	P0 to P3
AtomPro 28m	P0 to P3 and P16 to P19
AtomPro 40m	P0 to P3 and P28 to P31

### Notes

1. The ADIN conversion must complete before the program will process the next command. The AD conversion typically takes around 8 micro seconds.
2. The ADIN command only works with pins that are analog capable.
3. The analog pins for each module or Nano are listed in its corresponding data sheet and differ from module to module.

Examples

Run the example and connect to the terminal window at 9600. The ADIN command will read the analog pin P0 (AtomPro 28m) and print the result to the screen. Connect a potentiometer to P0 as shown in the schematic. You can connect any analog source and run the sample program.

```

;The analog pin number will change depending on module type

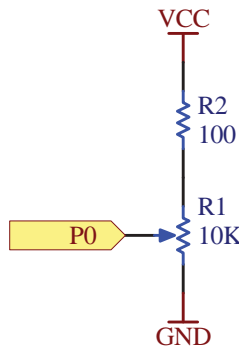
Pot var word

Main
  adin P0, pot ;read analog pin P0, load result into POT
  serout s_out, i9600, [0, "Analog Value = ", dec pot]
  Pause 50
  Goto Main

```

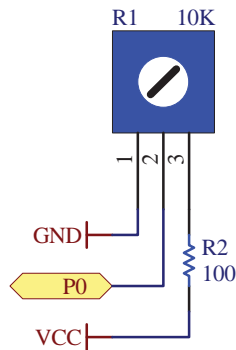
Schematic

The schematic is setup to use a potentiometer of 10K. Any value can be used. R2 the 100 ohm resistor is to prevent from shorting ground to power. Depending on what processor you are using you may need to change P0 to a different analog pin.



Schematic

This second schematic is basically the same as the first but shows an actually pot and how it would be wired up. R2 the 100 ohm resistor is to prevent from shorting ground to power. Depending on what processor you are using you may need to change P0 to a different analog pin.



---

## ADIN16

### Syntax

adin16 pin, result

- **Pin** - is a constant, variable or expression of an analog capable pin. See table from ADIN.
- **Result** - is a word(or larger) sized variable where the analog to digital sample will be store.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

ADIN16 was create to generate a more accurate reading from the analog to digital hardware. The typical A/D reading has a value range of 0 to 1024. ADIN16 drastically improves this by over sampling which results in a higher resolution A/D range. It sets up the A/D converter and stores the sum of 64 conversions in a result variable. The result variable will contain a 16 bit sum of all the conversions. This value can be used as a 16 bit resolution A/D reading which will be some what noisy. The noise can be cleaned up by right shifting the value to get to lower resolutions but with better readings than a standard A/D conversion value.

### Notes

1. 64 readings are taken which increase the time need to produce a result. The A/D is a separate piece of hardware which functions outside of the main MCU. ADIN16 will have little affect on the processing speed as the multiple conversions happen very fast relative to the entire instruction time.

### Example

The following example will load the result into the variable temp then right shift it 4 times. This gives an A/D range of 0 to 4095 which is 4x better than a standard 0 to 1023 reading.

```
temp var word
main
  adin16 p0,temp
  temp = temp>>4 ;temp will hold a 12bit decimated A/D conversion
  serout s_out,i9600,["Conversion = ",dec temp,13]
  pause 100
goto main
```

### Schematic

See ADIN.

## BRANCH

### Syntax

branch *index*, [*label1*, *label2*, *label3*, ..., *labelN*]

- **Index** - is a variable, constant or expression. It is used to reference a label in a list. The index is 0 based.
- **Label** - is a list of constant labels that are jump points in a program

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The BRANCH command will jump to a label defined within the brackets. The label used for the jump is determined by the pointer *index*. The only limit to the amount of the labels within the brackets are program memory.

### Notes

1. Labels are read from left to right with *label1* as position 0.
2. If a label is specified in the BRANCH command and not used in the program, an error will result during compiling.



## Examples

Connect to the following program with the terminal window at 9600 baud. Enter a value from 0 to 4. The program will jump to the label specified by the value typed in. The BRANCH command is a great way to build a long conditional jump list based on some value range. User interactive menu systems are one possibility.

```
;ALL - all_branch.bas
Index var word

Main
  Pause 1500
  serout s_out, i9600, [0, "Where should I jump to 0 to 4 ? -> "]
  Serin s_in, i9600, [dec Index]

  branch index, [label0, label1, label2, label3, label4]

Goto Main

Label0
  serout s_out, i9600, [13, "Label 0"]
  Goto Main

Label1
  serout s_out, i9600, [13, "Label 1"]
  Goto Main

Label2
  serout s_out, i9600, [13, "Label 2"]
  Goto Main

Label3
  serout s_out, i9600, [13, "Label 3"]
  Goto Main

Label4
  serout s_out, i9600, [13, "Label 4"]
  Goto Main
```

---

## BUTTON

### Syntax

button pin, downstate, delay, rate, workspace, targetstate, label

- **Pin** - is a variable or constant that specifies the pin to be used. The pin specified will be set to an input when the command runs.
- **Downstate** - is a variable or constant that specifies what state the pin should be to considered as a button press. High = 1 or Low = 0.
- **Delay** - is a byte sized variable or constant (0-255) that specifies the number of loops to execute before enter the auto repeat function (*rate*). Delay is the debounce function of the BUTTON command. If *delay* = 0 the command will run once with and *rate* is ignored. If *delay* = 255 the command will repeat 255 times but *rate* is ignored. Any value in between 1 - 254 will execute both debounce and *rate*.
- **Rate** - is a byte sized variable or constant (0-255) that specifies the number of loops to execute after delay has expired. This is the auto repeat function and only executes after delay has decremented to 0.
- **Workspace** - is a byte sized variable (0-255) that is an internal counter that tracks how many times the command has been run. Internally its set by delay, then rate and is decremented each loop through the command. A unique workspace variable must be created for each BUTTON command used in the program.
- **Targetstate** - is a byte sized variable or constant and determines what state the button must be in before the BUTTON command will jump to the *label*. 1 = pressed and 0 = not pressed. If target state is set to 1 (Pressed), the command will jump to the label if the button is pressed. Otherwise the next line of code is executed.
- **Label** - is where the command will jump to based on targetstate. If targetstate = 0 (Not Pressed) then the jump to *label* is made when no button press is detected. If targetstate = 1 (Pressed) then the jump to *label* is made when a button press is detected.

### Supported

- BA - Supported. X samples per second.
- BAN - Supported. X samples per second.
- BAP - Supported. X samples per second.
- BAP40 - Supported. X samples per second.

### Description

The BUTTON command when ran will first look at downstate and determine if Pin must be high (1) or low (0) to consider a button is in a pressed state. After a button is determined pressed, the command loads the delay value into the workspace variable. Any time a mechanical switch or button is pressed there is a tiny vibration within the device. This vibration will appear to a microcontroller I/O pin as several on / off cycles. To eliminate this unwanted noise we debounce the press using a delay. Once the delay value it loaded into workspace the command will look at the targetstate and either jump to the label or move onto the next line of code. This is the first time the command has run.

The BUTTON command must be nested inside of a loop to perform properly. This is typically accomplished with a GOTO statement. If a button press has occurred, the BUTTON command will then look at the state of pin again. With each pass through decrementing workspace. This cycle is repeated for as many times as delay was set to. At the end of this cycle, if pin is still consider pressed the button command acts on a pressed state.

#### Notes

1. The BUTTON command is non blocking. Each run through it will either jump to label or allow program execution to continue on the next line.
2. The BUTTON command automatically sets a pin to an input and will leave it in that state.

#### Examples

Connect to the following program with the terminal window set to 9600 baud. The BUTTON command is setup for an active low (Downstate = 0) on P0 and P1. Connect an active low button as shown in the below schematics. If a button press is detected on P0 or P1 its results are printed to the screen as "P0 = Pressed" or "P1 = Pressed". Once a button press is no longer detect the program returns to the main loop.

Targetstate is set to 1 (Pressed). When the command does not detect a press state the next line of code is ran. Which in this case is a SEROUT command used to clear [0] the terminal screen. If a button press is detect the BUTTON command will jump to label (pressed1, pressed2). Once at label, which button was pressed is printed to the screen.

```
;ALL - all_button.bas

Workspacel var byte
Workspace2 var byte

;watch P0 and P1 for an active low button press.
;clear the screen if no button press is detected.
;if button is pressed jump to label and tell
;us about it.

Main
  button P0,0,80,40,workspacel,1,pressed1
  serout s_out, i9600, [0]

  button P1,0,80,40,workspacel,1,pressed2
  serout s_out, i9600, [0]

goto main

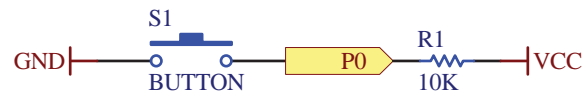
pressed1
  ;button press on P0 has been detected, print it to the scen.
  serout s_out, i9600,[0,"P0 = Pressed"]
  pause 300
goto main

pressed2
  ;button press on P1 has been detected, print it to the scen.
  serout s_out, i9600,[0,"P1 = Pressed"]
  pause 300
goto main
```

Schematics

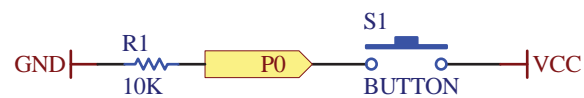
The first schematic is considered an active low setup. This means the pin is pulled high and the switch is normally open. When the switch is pressed it shorts the pin to GND. The switch is now "active".

Active Low



The second schematic is consider an active high setup. This means the pin is pulled low and the switch is normally open. When the switch is pressed it shorts the pin to VCC. The switch is now "active".

Active High



---

## CLEAR

### Syntax

clear

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The clear command sets all user memory to zeros. This includes variables and any command which relies on user accessible memory. The CLEAR function is typically used in the beginning of a program to set all memory to a known state. In some situations CLEAR is used in place of several statements like `variable = 0`.

### Notes

1. The CLEAR command will reset the buffers used in several hardware based commands such as HSERIAL and HSERVO. If the command is issued in the middle of program execution, data held in the buffers of these commands will be cleared to all zeros.

### Examples

Connect to the following program with the terminal window set to 9600 baud. The CLEAR command sets all variables and user memory to a known state before normal program execution begins. In some cases if your program is randomly restarted you would want to eliminate the potential for any random values left in user memory to interfere with normal operation. The following program demonstrates what happens to all defined variables after a CLEAR command is issued.

```
;ALL - all_clear.bas

Dog var byte
Cat var byte
Mop var byte
Top var byte

Dog = 1
Cat = 2
Mop = 3
Top = 4

Main
  pause 600
  serout s_out,i9600,[0,"Variables before CLEAR command: "]
  serout s_out,i9600,[13, "Dog = ", dec dog]
  serout s_out,i9600,[13, "Cat = ", dec cat]
  serout s_out,i9600,[13, "Mop = ", dec mop]
  serout s_out,i9600,[13, "Top = ", dec top]
  pause 1000

  CLEAR

  serout s_out,i9600,[13, 13, "Variables after CLEAR command: "]
  serout s_out,i9600,[13, "Dog = ", dec dog]
  serout s_out,i9600,[13, "Cat = ", dec cat]
  serout s_out,i9600,[13, "Mop = ", dec mop]
  serout s_out,i9600,[13, "Top = ", dec top]

End
```

---

## COUNT

### Syntax

count pin, time, cycles

- **Pin** - is a variable or constant that specifies which input pin to use. The pin is automatically set to input mode.
- **Time** - is a variable, constant or expression that specifies the amount of time to count *pin* clock cycles.
- **Cycles** - is a variable in which the total count is stored.

### Supported

- BA - Supported. Minimum pulse width 3.4us.
- BAN - Supported. Minimum pulse width 8.5us.
- BAP - Supported. Minimum pulse width 1.5us.
- BAP40 - Supported. Minimum pulse width 1.5us.

### Description

The COUNT command is used to count low to high transition (0-1-0). It will count a low-high-low as 1 cycle. COUNT is a blocking command and will only count the *pin* for the specified time. The COUNT command can be used to calculate the speed of a motor from an optical encoder. Or it can be used to determine lower range frequencies.

### Notes

1. Each module has a different minimum resolution, see supported chart.
2. Each module has a maximum speed it can count, see supported chart.
3. A cycle is considered a low-high-low transition.

### Examples

Connect to the following program with the terminal window set to 9600 baud. Pull-up P0 with a 10K resistor. Use a jumper wire with one end connected to ground. Short P0 with the jumper wire. See how many cycles you create. Each loop through the COUNT command will count for 1 second. Then print the results to the terminal window. Repeating forever.

```

;ALL - all_count.bas

cycles      var Word
time var Word

;lets count the low-high-low transitions on P0 for 1 second
;save the results to the variable cycles, then print it to
;the terminal window. Pull-up P0 with a 10K resistor. Short
;P0 to ground with a small wire and see how many cycles you
;can create per loop.

time = 1000
cycles = 0

Main
  ;lets count the low-high-low transitions for a second
  count p0, time, cycles
  pause 500

  ;lets print the results to the terminal window
  serout s_out, i9600, ["Count = ", dec cycles,13]
goto main

```

## DEBUG

### Syntax

debug [{modifiers}data1,...,{modifiers}dataN]

- **Data** - is a variable, constant or expression of data that will be display in the debug output window
- **Modifiers** - supports most all output modifiers. See Output Modifiers table below. For additional information regarding modifiers see the Modifier section of this manual.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Output Modifiers

Name	Description
DEC	Decimal.
SDEC	Signed decimal.
HEX	Hexadecimal.
SHEX	Signed hexadecimal.
IHEX	Indicated (\$) hexadecimal.
ISHEX	Signed and indicated (\$) hexadecimal.
BIN	Binary.
SBIN	Signed binary.
IBIN	Indicated (%) binary.
ISBIN	Indicated (%) and signed binary.
REP	Repeat character <i>n</i> times.
REAL	Floating point number with decimal point.
STR	Read specified amount of characters from an array.

### Description

The DEBUG command is a simplified version of a SEROUT command. It works only to help debug a program. Once the DEBUG command is used the results will be printed to the debug window only when the target processor has been programmed using the Debug mode in Studio.

Notes

1. The debug command is only included in your final code when your program is compiled using the DEBUG function in Basic Micro Studio.
2. The debug output window expects all output to be in ASCII. If variables are output directly without modifiers, their values will be truncated to 8bits and interpreted by the terminal window as ASCII characters, which may give unexpected results.
3. The Debug Output window accepts terminal window formatting commands.

Terminal Window Commands

Decimal Character	Command	Description
0	CLS	Clears the screen.
1	HOME	Moves cursor home.
3	MOVE LEFT	Moves cursor left.
4	MOVE RIGHT	Moves cursor right.
5	MOVE UP	Moves cursor up.
6	MOVE DOWN	Moves cursor down.
7	BELL	Make sound on PC.
8	BACK SPACE	Moves cursor back and delete.
9	HANDLE TAB	Add a standard tab.
10	LINEFEED	Move cursor to next line.
11	CLEAR RIGHT	Clear anything to the right of the cursor.
12	CLEAR DOWN	Clear anything below the cursor.
13	CARRIAGE RETURN	Move to the next line.

Example

The following example will out debug information to the debug window. The processor must be programmed using the Debug function of Basic Micro Studio.

```

value var long
fvalue var float

main
    value = value + 1
    fvalue = fvalue + 0.1

    debug ["Dec Value=",dec value," "]
    debug ["Hex Value=",hex value," "]
    debug ["Bin Value=",bin value," "]
    debug ["Real Value=",real fvalue,13]

goto main
    
```



## DEBUGIN

### Syntax

debugin [{modifiers}data1,...,{modifiers}dataN]

- **Modifiers** - supports most all input modifiers. See Input Modifiers table below. For additional information regarding modifiers see the Modifier section of this manual.
- **Data** - is a variable where data sent from the debug window will be stored.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Input Modifiers

Name	Description
DEC	Decimal.
SDEC	Signed decimal.
HEX	Hexadecimal.
SHEX	Signed hexadecimal.
IHEX	Indicated (\$) hexadecimal.
ISHEX	Signed and indicated (\$) hexadecimal.
BIN	Binary.
SBIN	Signed binary.
IBIN	Indicated (%) binary.
ISBIN	Indicated (%) and signed binary.
REAL	Floating point number with decimal point.
STR	Read or write specified amount of characters and store in an array.
SKIP	Skip specified amount of characters.
WAIT	Wait for specified amount of characters.
WAITSTR	Compares specified amount of characters to array.

### Description

Accepts keyboard input from the Debug Output window from Basic Micro Studio. The debug window is only available if the processor was programmed using the debug button. After a debug session is complete the processor needs to be reprogrammed using the normal program button otherwise it will not function properly.

### Note

1. The DEBUGIN command is only included in your final code when your program is compiled using the DEBUG function in Basic Micro Studio.
2. In the absence of modifiers DEBUGIN assigns each keystroke to a single variable.

---

### Example

The following example will send and receive data from the debug window. The processor must be programmed using the Debug function of Basic Micro Studio.

```
value var long
fvalue var float

debug ["Enter the starting integer value:"]
debugin [dec value]
debug ["Enter the starting real value:"]
debugin [real fvalue]

main
    value = value + 1
    fvalue = fvalue + 0.1

;output to Debug window using the DEC,HEX,BIN and REAL modifiers
    debug ["Dec Value=",dec value," "]
    debug ["Hex Value=",hex value," "]
    debug ["Bin Value=",bin value," "]
    debug ["Real Value=",real fvalue,13]
goto main
```

---

## DO - WHILE

### Syntax

```
do
    program statements
while condition
```

- **Statements** - any group of commands to be run inside the loop.
- **Condition** - can be a variable or expression

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The DO - WHILE loop executes commands nested inside of it will some condition is true. The condition can be any variable or expression that is tested every loop until it is false.

### Notes

1. In the programming world 0 is consider false. By default DO - WHILE will test this condition. If a stand alone variable is used for the test the loop will continue until its value equals 0.
2. DO - WHILE will always run at least once since the condition is checked last in the loop.
3. You can nest multiple DO - WHILE commands within each other. However you can not nest DO - WHILE with a WHILE - WEND together or the compiler will get the WHILE statements confused.

### Example

Connect to the following program with the terminal window set to 9600 baud. The program will start counting up from 0 to 100. Once index reaches a value of 100 the condition is no longer true. The less than symbol < was used for the condition and 100 is no longer less than 100 making the condition false. Since DO - WHILE loops while a statement is true the program exits.

```
;ALL - all_do_while.bas

Index var word

Main
    Index = 0

    Do
        index = index + 1
        serout s_out, i9600,[0, "Counting: ", dec index]
        pause 75

    While index < 100 ;repeat until index is no longer less than 100

        serout s_out, i9600, [13,13, "Index = ", dec index]
        serout s_out, i9600, [13, "My condition is no longer true."]
        serout s_out, i9600, [13, "Index is no longer less than 100"]

End
```

---

### Example

Connect to the following program with the terminal window set to 9600 baud. The program will start counting down from 100 until it equals 0. Once `index = 0` the program will quit because the condition is no longer true since `0 = false`. In programming 0 is always considered false. If a condition is tested and the results equal 0 then a false is returned.

```
;ALL - all_do_while2.bas
Index var word

Main
  Index = 100

  Do
    index = index - 1
    serout s_out, i9600,[0, "Counting: ", dec index]
    pause 75

  While index ;repeat until index is 0, which is a false expression

  serout s_out, i9600, [13,13, "Index = ", dec index]
  serout s_out, i9600, [13, "My condition is no longer true."]
  serout s_out, i9600, [13, "Index is now false"]

End
```

---

## DTMFOUT

### Syntax

dtmfout pin,{ontime,offtime,}[tone1, ..., toneN]

- **Pin** - is a variable, constant or expression that specifies the I/O pin to use. The pin will be set to an output during tone generation. After tone generation is complete the pin will be set to an input.
- **Ontime** - is an optional variable, constant or expression that specifies the duration of each tone in milliseconds. If not specified, default is 200 ms.
- **Offtime** - is an optional variable, constant or expression that specifies the length of silence after each tone in milliseconds. If not specified, default is 50 ms.
- **Tone** - is a constant, variable or expression that specifies the tone to be generated. Valid value ranges are 0 to 15. See Tone table below.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Tone Table

Value	Pins
0 to 9	Digits 0 through 9
10	* (Star)
11	# (Pound)
12	A
13	B
14	C
15	D

### Description

DTMFOUT is used to generate the standard 16 tones for phone lines. It can also be used to control any radio device that supports DTMF tones. The tones are generated by mixing internally two sine waves that were created mathematically. The results are then internal used to control the duty cycle of a PWM signal. The resulting sound must be filtered to remove the digitization that is created when the tones are generated.

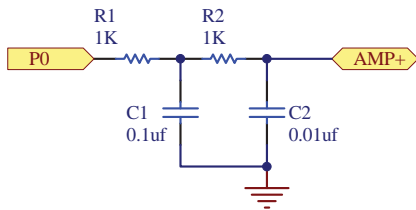
### Notes

1. In order for the tones to be recognized by the receiving device the output must be filtered and in many cases amplified.

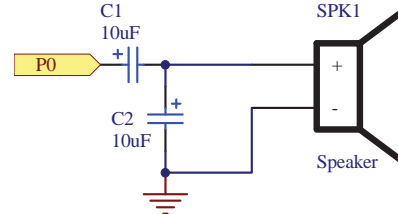
Schematic

There are a few ways to filter and interface devices to the processor used to create the DTMF tones. A speaker can be driven directly. A simple filter circuit can be used. Also shown is a method for connection the processor directly to a phone line.

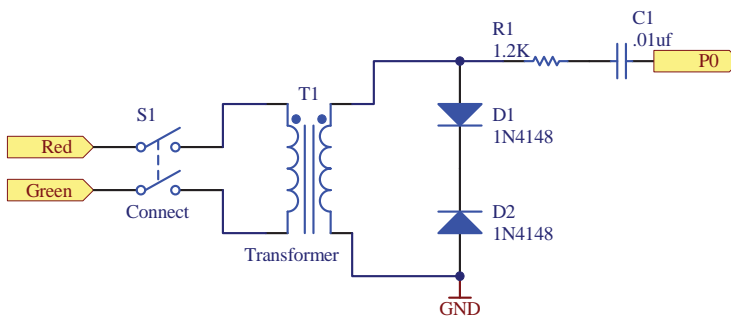
Passive Low Pass Filter



Direct Drive Speaker



Phone Line Interface



Example

The simple example program will dial out to Basic Micros main number using the phone line interface circuit.

```
main
  dtmfout p0,200,50,[8,0,0,5,3,5,9,1,6,1]
  pause 1000
  goto main
```

---

## DTMFOUT2

### Syntax

dtmfout2 Lpin\Hpin,{ontime, offtime,}[tone1, ..., toneN]

- **Lpin** - is a variable, constant or expression that specifies one of two I/O pins to used.
- **Hpin** - is a variable, constant or expression that specifies one of two I/O pins to used.
- **Ontime** - is an optional variable, constant or expression (0 – 65535) that specifies the duration of each tone in milliseconds. If not specified, default is 200 ms.
- **Offtime** - is a an optional variable, constant or expression (0 – 65535) that specifies the length of silence after each tone in milliseconds. If not specified, default is 50 ms.
- **Tone** - is a constant, variable or expression that specifies the tone to be generated. Tone can be any length. Valid value ranges are 0 to 15. See Tone table below.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Tone Table

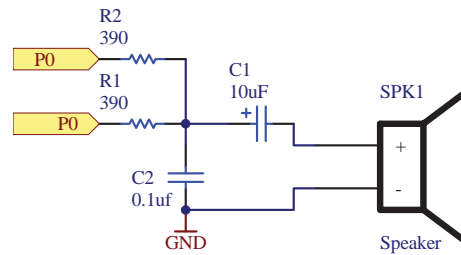
Value	Pins
0 to 9	Digits 0 through 9
10	* (Star)
11	# (Pound)
12	A
13	B
14	C
15	D

### Description

DTMFOUT2 uses square waves instead of sine waves like DTMFOUT. This will create louder tones that need less amplifying, but will still need to be filtered. DTMFOUT2 will generate the standard 16 tones for phone lines. It can also be used to control any radio device that supports DTMF tones. The tones are generated by created two square waves and out putting each on a separate pin.

## Schematics

The schematic can be used to combine the two pins into one out put without shorting each other. The speaker can be replaced with a filter or phone line interfacing circuit.



## Example

The simple example program will dial out to Basic Micros main number using the phone interface circuit.

```
main
  dtmfout2 p0\p1,200,50,[8,0,0,5,3,5,9,1,6,1]
  pause 1000
goto main
```



## END

Syntax  
end

Supported

- BA - supported
- BAN - supported
- BAP - supported
- BAP40 - supported

Description

END stops program execution until a reset occurs. The processor is put in a low power mode. I/O pins will not remain in their last state. Use STOP instead of END if you need the I/Os to be held in their last state at the end of your program.

Example

The following example will only run once. The program will only restart if reset is pressed or the power is cycled.

```
value var long  
  
serout s_out,i9600,["This program just ends",13]  
serout s_out,i9600,["Press reset to see it again.",13]  
end
```

---

## EXCEPTION

Syntax  
exception label

- **label** - is the label to continue execution from.

Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

Description

Since Mbasic allows nested GOSUB...RETURN commands in some cases you may need to return to a specific label and clear the last GOSUB call. The EXCEPTION command will exit and return to the label specified, in doing so it will clear all return addresses from the stack.

Notes

1. Any RETURN data in your GOSUB loop will be lost.

Examples

The following example will print to the terminal window at 9600 baud. The EXCEPTION command is used to exit the second nested GOSUB and return to the label main

```
Mode var byte
Serout s_out,i9600,["Starting",13]

Main
  Serout s_out,i9600,["Main",13]
  Mode = mode + 1
  Gosub mysub1
  Serout s_out,i9600,["Returned from MySub1",13]
  Goto main

Mysub1
  Serout s_out,i9600,["Entering MySub1",13]
  Gosub mysub2
  Serout s_out,i9600,["Returned from MySub2",13]
  Return

Mysub2
  Serout s_out,i9600,["Entering MySub2",13]
  If(mode.bit0)then
    Serout s_out,i9600,["Exception back to Main",13]
    Exception main
  Endif
  Return
```

See Also:

GOSUB  
RETURN

## FATAN2

## Syntax

fatn2 xval\yval,variable

- **Xval** - is a variable, constant or expression that specifies the X coordinate in radians
- **Yval** - is a variable, constant or expression that specifies the Y coordinate in radians
- **Results** - is a variable the result will be stored in. This variable should be defined as a floating point variable.

## Supported

- BA - Not Supported
- BAN - Not Supported
- BAP - Supported
- BAP40 - Supported

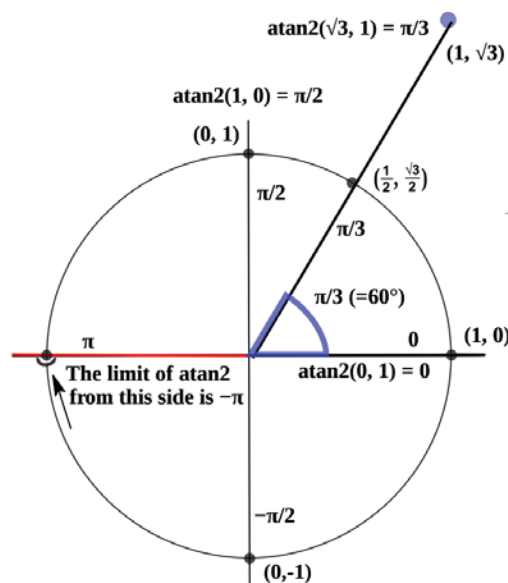
## Description

FATAN2 calculates the four quadrant ArcTangent of the specified X and Y values given. FATAN2 can be used to calculate the angle in +/- 180 degrees based on an X / Y coordinate. All results are given in radians. This function is typically used when calculating Inverse Kinematics.

## Example

The example calculates the solution for the graph shown. Where  $X = 1/2$  and  $Y = \sqrt{3}/2$  the result of FATAN2 would be 1.0471. To get the exact degree you would then take result  $(1.0471 \times 180) / \pi = 60^\circ$ .

```
result var float
Main
  fatan2 0.886\0.5,result
  serout s_out,i9600,[real result,13]
  pause 100
goto Main
```



Source: ATAN2 Wikipedia

---

## FOR...NEXT

### Syntax

```
for countVal = startVal to finishVal {step increment}
  ...code...
next
```

- **CountVal** - a variable used to store the current count
- **StartVal** - the starting value to count from
- **FinishVal** - a value to count up or down to
- **Increment** - the value to increment the variable by through each loop

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

Repeats a block of instructions and increments a variable counter with a value specified each time through the loop. The FOR...NEXT loop will exit when the count value is no longer between start and finish.

### Notes

1. When no step increment is defined the default increment value is 1.
2. When the increment is positive the start must be less than or equal to finish or the FOR..NEXT loop will be skipped entirely.
3. If the increment is negative the start value must be greater then or equal to finish or the FOR...NEXT loop will be skipped entirely.
4. If variable is modified by the instructions in the FOR...NEXT block the loop can be forced to exit early.
5. You can safely jump out of a FOR...NEXT loop.

## Example

The below example will print the results to a terminal window at 9600 baud. It will count up from 0 to 9 then exit.

```
value var long

main
  for value = 0 to 9
    pause = 1000
    serout s_out,i9600,["Value=",dec value,13]
  next
end
```

## Example

The below example will print the results to a terminal window at 9600 baud. It will count down from 0 to 9 then exit.

```
value var long

main
  for value = 9 to 0 step -1
    pause = 1000
    serout s_out,i9600,["Value=",dec value,13]
  next
end
```

## Example

The below example will print the results to a terminal window at 9600 baud. It will count up from 0 to 54 using 5 as the increment then exit.

```
value var long

main
  for value = 0 to 54 step 5
    pause = 1000
    serout s_out,i9600,["Value=",dec value,13]
  next
end
```

---

## FREQOUT

### Syntax

freqout pin, duration, freq1{,freq2}

- **Pin** - is a variable, constant or expression that specifies the I/O pin to be used. This pin will be set to output mode during tone generation and left in that state after output is completed.
- **Duration** - is a variable, constant or expression that specifies the duration of the output tone in milliseconds.
- **Freq1** - is a variable, constant or expression that specifies the frequency in Hz of the first tone which can be between 0 – 32767Hz.
- **Freq2** - is an optional variable, constant or expression that specifies the frequency in Hz of the second tone which can be between 0 – 32767Hz.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

This command generates one or two tones that are output on a single I/O pin. The signal generated is a pulse width modulated signal. This command can be used to create music, tones or general output signals. The maximum supported range is limited to what is practical. There aren't many speakers or piezo devices on the market that will work well outside of 30Khz. Which is why MBasic is limited to a high of 32Khz.

### Notes

1. The tones are generated mathematically and output as a pulse width modulated (PWM) signal. The signal must be converted to a sine wave (or a pair of sine waves) by passing it through an integrator which is a low pass filter.

### Example

The code snippet will generate a 1000Hz tone for 5 seconds on P1.

```
freqout p1, 5000, 1000
```

Now let's generate two tones 1000Hz and 2000Hz for 5 seconds on P1.

```
freqout p1, 5000, 1000, 2000
```

The next program will increment through several tones holding each for 3 seconds. Try adjusting the initial values tone1 and tone2 are set to. Each loop through the program tone1 will increment by 500 and tone2 will increment by 750.

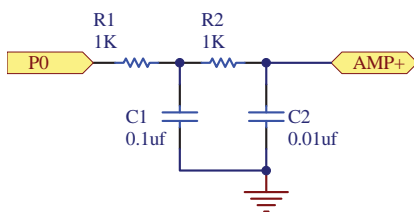
```
tone1 var long
tone2 var long
clear

main
tone1 = tone1 + 500
tone2 = tone2 + 750
  freqout p0, 3000, tone1, tone2
Goto Main
```

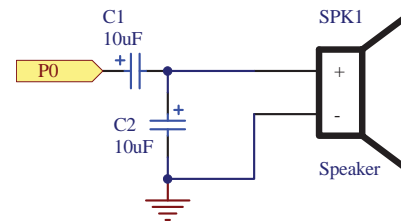
### Schematics

The sine wave output from FREQOUT is generated using a high frequency PWM. The higher frequency need to be filtered to generate a cleaner tone. There are a number of way this can be accomplished. The following circuits are provided as examples. P1 is used but any specified pin can be connected.

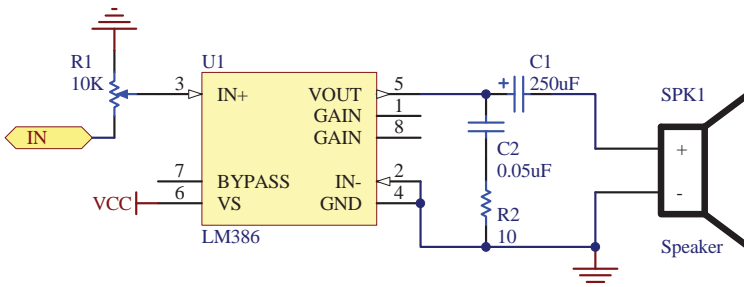
#### Passive Low Pass Filter



#### Direct Speaker



#### Simple Audio Amplifier



## GOSUB

### Syntax

gosub label{[argument1,...,argumentN]},{DataResult}

- **Label** - the go to label of the subroutine to be execute.
- **Argument** - is user defined arguments to send to the called subroutine. The only limit to the amount of arguments used is program memory.
- **DataResult** - is an optional variable to store the value returned from called subroutine.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The GOSUB command will jump to a specified label. After executing the code at the jump label a RETURN command is then used to return the program to the next command after the last called GOSUB.

If you are familiar with some BASIC languages there is typically a limit to what is called nesting. This is where multiple GOSUB...RETURN statements are nested inside of each other. There is no limit to this with Studio other than the size of the available stack memory.

GOSUB stores the address of the next command on the stack and jumps to the specified label. User specified arguments can be defined in the subroutine and a return value from the subroutine called can be stored in the variable that is specified by the GOSUB DataResult argument.

### Notes

1. Subroutines should exit via the RETURN command, which clears the saved address from the stack and returns to the command following the GOSUB. Do not use BRANCH or GOTO when exiting a subroutine.
2. User defined arguments must match the number of arguments defined at the subroutine. If they do not match, a stack overflow or underflow will happen.
3. If subroutines return a value the GOSUB is not required to use it or specify a return value variable



## Example

The below program will print the results to the terminal window at 9600 baud. The results will be 110. The GOSUB command has two arguments and includes DataResult variable. The values 10 and 100 are passed to the subroutine MyAdd. The values are then loaded into the variables *arg1* and *arg2*. Since RETURN can have an expression the variables *arg1* and *arg2* are added and returned to the variable *result*.

```
Result var long
Main
  Gosub myadd[10,100],result
  Serout s_out,i9600,["Result =",dec result]
End

Arg1 var long
Arg2 var long

MyAdd [arg1,arg2]
Return arg1+arg2
```

See Also  
RETURN  
EXCEPTION

---

## GOTO

### Syntax

goto label

- **Label** - is a label the program will jump to.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The GOTO command tells the program to jump to some label.

### Notes

1. Can not use reserved words or variables as label names.

### Examples

Connect to the following program with the terminal window set to 9600 baud. The following program is a simple loop using GOTO that will repeat for ever.

```
;Demo Program - GOTO.bas
basic
  Pause 800
  serout s_out,i9600,[0, "Basic"]
  pause 800
  goto micro
  goto basic

micro
  serout s_out,i9600,[2, " Micro"]
  pause 800
  Goto rules
  goto basic

rules
  serout s_out,i9600,[2, " Rules!"]
  pause 800
  goto basic
```

---

## HIGH

Syntax  
high pin

- **PIN** - is a variable, constant or expression that specifies which pin to go high.

Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

Description

The HIGH command is an output command that will set a pin HIGH. All pins can be in 3 states. HIGH, LOW and FLOAT (input). The HIGH command will change any of these states to an output and set the pin HIGH (5VDC)

Notes

1. If a variable is used, the variables value will be directly translated into a pin number. If the variable equals 0 then P0 will go high. If the value is out of range to the pins that are present nothing will happen. If you have 10 pins and the variable equals 11 then nothing will happen.

Examples

The following program is a simple loop using HIGH and LOW commands that will repeat for ever. If you connect an LED to P0 it will blink at half second intervals forever.

```
;ALL - all_high.bas
Main
  High p0
  Pause 500
  Low p0
  Pause 500
  Goto Main
```

---

## HPWM

### Syntax

hpwm pin,period,duty{,wait}

- **Pin** - hardware capable PWM pin number. See table below.
- **Period** - the time in clock cycles for a single pulse (high and low part).
- **Duty** - the time in clock cycles for the high side of the pulse.
- **Wait** - an optional boolean (true or false) argument that will cause the command to wait until the current pulse has finished before setting the new period and duty. This eliminates the possibility of a glitch in the pulses.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### HPWM Capable Pins

Processor	Pins
Nano 18	P3
Nano 28	P9 / P10
Nano 40	P9 / P10
Atom 24m	P9 / P10
Atom 28m	P9 / P10
Atom 40m	P9 / P10
AtomPro One	P4 / P5 / P6
AtomPro 24m	P10 / P11 / P12
AtomPro 28m	P10 / P11 / P12
AtomPro 40m	P9 / P10 / P11 / P13 / P14 / P15

### Description

The HPWM command generates a pulse width modulated signal. HPWM is based on internal hardware and is processor independent. Meaning it will run in the back ground while your program performs additional task. HPWM has several uses, some of which are generating an analog voltage, DC motor control, servo control or generating a frequency.

To create an analog voltage the pin is transition from high to low quickly. During the transitions leaving the pin high for a certain amount of time versus low and then averaging the output will cause a voltage change. If the pin was high (5V) 50% of the time and then low (0V) the other 50% the voltage output would be around 2.5V. By adjusting the ratio of how long the pin is high versus low we can control the output voltage with a simple RC circuit.

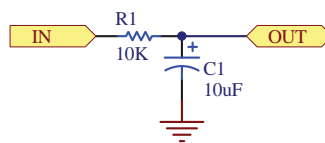
The rate of the high side to the low side of a single pulse is called the duty cycle. The higher the duty cycle the higher the overall output voltage would be. Period specifies how long a pulse is in total. A pulse is made up of one high to low transition. So the period specifies the frequency. When dealing with generating an analog voltage using the RC circuit the duty and period will determine at what voltage and how much current the circuit can provide.

Notes

1. Nano and BasicATOM processors the highest Period allowed is 16383
2. BasicATOM Pro processors the highest Period is 524287.

Schematic

The schematic is a simple RC filter. The resistor and capacitors values can be changed to affect the final output.

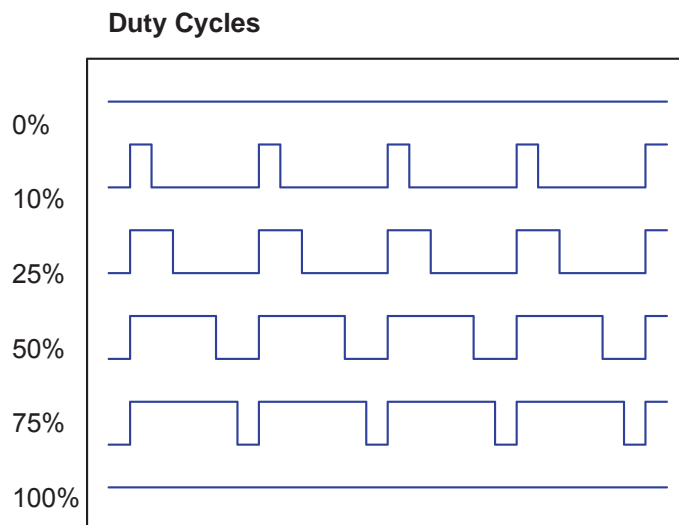


The code snippet will generate a 50% duty cycle. Which, with no load, will output 2.5V on the output side of our RC filter. Build the circuit, run this program and attach a volt meter probe to the output. Try adjusting the duty cycle to see the results.

```
hpwm p10, 10000, 5000
```

Duty Cycle

The following chart is what the signal would look like on an oscilloscope at specific duty cycles. This will give you an idea of what the duty cycle looks like. You can see their on times. More on time would be mean more charge for our RC circuit which would increase its output voltage. At 0% duty cycle the pin is always low. At 100% duty cycle the pin is always high.



## HSERIN

### Syntax

hserin uart, {tlabel, timeout,}{modifiers} InputData1, ..., {modifiers} InputDataN]

- **UART** - is an optional argument on a 1 UART processor. It is a constant of 1 or 2 that specifies what UART to use. See HSERIAL Pin table below.
- **Tlabel** - is an optional label the program will jump to if the time out value is exceeded. Tlabel must be specified if a Timeout value is set.
- **Timeout** - is an optional constant, variable or expression that specifies the time in milliseconds to wait for data to be received.
- **Modifiers** - supports most all input modifiers. See Input Modifiers table below. For additional information regarding modifiers see the Modifier section of this manual.
- **InputData** - is a variable were incoming data will be stored. The only limit to the amount of InputData variables in the list is the amount of user ram on the processor. The incoming data can be formatted with optional input modifiers. See the Modifier section of this manual.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

HSERIN / HSEROUT use a special hardware feature called a UART. The UART allows the processor to perform other task while receiving or sending serial data. The hardware runs in the background and is processor independent. Data that is received, is stored in a buffer, regardless of what the processor is doing. The HSERIN / HSEROUT commands then retrieve this data from the buffer. Some processors have two UARTS that run independent of each other. The UART output is inverted. Which is setup to work with a RS232 circuit. Non inverted mode is not support. A simple inverter circuit could be used if needed. For all HSERIN / HSEROUT capable pins see the table below.

### HSERIAL Capable Pins

Processor	HSEROUT (TX)	HSERIN (RX)	HSEROUT (TX2)	HSERIN (RX2)
Nano 18	P5	P2		
Nano 28	P14	P15		
Nano 40	P14	P15		
Atom 24m	P14	P15		
Atom 28m	P14	P15		
Atom 40m	P14	P15		
AtomPro One	S_OUT	S_IN		
AtomPro 24m	P15	P14		
AtomPro 28m	P15	P14		
AtomPro 40m	S_OUT	S_IN	P15	P14
ARC32	S_OUT	S_IN	P26	P38

### SETHSERIAL

MBasic must create a buffer in memory for each UART and specify the baud rate for the UART. The directive SETHSERIAL sets the available parameters for the specified UART. This is only required once at the beginning of a program. There are two UARTs available on some processors. To set UART1 use SETHSERIAL1 and to set UART2 use SETHSERIAL2.

The SETHSERIAL1 command is used to set UART1 as shown below.

```
setserial1, baudrate {,databits, parity, stopbits} ;Set UART1
```

The SETHSERIAL2 command is used to set UART2 as shown below.

```
setserial2, baudrate {,databits, parity, stopbits} ;Set UART2
```

### Baud Rate

SETHSERIAL is also used to set the baud rate for HSERIAL. The other arguments shown below are optional. They are typically set when attached to a proprietary device. Rarely would these be set when communicating to a PC.

- **Baudrate** - is a predefined value that specifies the transmit and receive rate. Different processors support different baud rates. See Supported Baud Rate table.
- **DataBits** - is an optional argument that defines how many bits are used. The default is 8. See Supported Options table.
- **Parity** - is an optional argument that sets what the parity bit is. No parity, even parity or odd parity are supported. The default is no parity (N). See Supported Options table.
- **Stopbits** - is an optional argument that specifies 1 or 2 stop bits in the serial format. The default is 1. See Options table.

### Options

DataBits	Parity	StopBits
H8DATABITS	HNOPARITY	H1STOPBIT
H7DATABITS	HEVENPARITY	H2STOPBIT
	HODDPARITY	

### Multiple UARTs

When two UARTs are available they can be setup independently of each other. The UARTS can be set to different or the same baud rates. The second UART is only available on some processors. See HSERIAL table. To setup both UARTS follow the example below:

```
sethserial1, H9600      ;Sets UART 1 to 9600 Baud
sethserial2, H38400    ;Sets UART 2 to 38400 Baud
```

### Example

The following example is a loop back. Wire the TX pin to the RX pin. It will receive and send to its self to demonstrate how the buffer of the UARTS. Unlike normal SERIAL commands the HSERIAL commands do not need to be running to receive data. Its done in the background.

```
sethserial1 h57600

string var byte(20)
integer var long

main
  hserout ["Type a string upto 20 characters long and hit enter",13]
  hserin [str string\20\13]
  hserout ["You string is: ",str string\20\13,13]

  hserout ["Type a decimal number and hit enter",13]
  hserin [dec integer]
  hserout ["Your number is:",dec integer,13]
  hserout ["Your number in Hexidecimal is:",hex integer,13]
  hserout ["Your number in Bianry is:",bin integer,13]

goto main
```

### Supported Baud Rates

BAP	Error	BAP40	Error	BA	Error	BAN	Error
H300	0.16%	H300	0.16%	H2400	0.16%	H600	0.16%
H600	0.16%	H600	0.16%	H4800	0.16%	H1200	0.16%
H1200	0.16%	H1200	0.16%	H7200	0.22%	H2400	0.16%
H2400	0.16%	H2400	0.16%	H9600	0.16%	H4800	0.16%
H4800	0.16%	H4800	0.16%	H12000	0.16%	H7200	0.64%
H7200	0.64%	H7200	0.22%	H14400	0.22%	H9600	0.16%
H9600	0.16%	H9600	0.16%	H16800	0.55%	H12000	0.79%
H12000	0.79%	H12000	0.16%	H19200	0.16%	H14400	0.79%
H14400	0.79%	H14400	0.94%	H21600	0.22%	H16800	0.79%
H16800	0.79%	H16800	0.55%	H24000	0.16%	H19200	0.16%
H19200	0.16%	H19200	1.36%	H26400	0.74%	H21600	0.64%
H21600	0.64%	H21600	0.22%	H28800	0.94%	H24000	0.79%
H24000	0.79%	H24000	0.16%	H31200	0.16%	H26400	0.32%
H26400	0.32%	H26400	1.36%	H33600	0.55%	H28800	2.12%
H28800	2.12%	H28800	1.36%	H36000	0.79%	H31200	0.16%
H31200	0.16%	H31200	0.16%	H38400	1.36%	H33600	0.79%
H33600	0.79%	H33600	2.10%	H57600	1.36%	H36000	0.79%
H36000	0.79%	H36000	2.12%	H115200	1.36%	H38400	0.16%



BAP	Error	BAP40	Error	BA	Error	BAN	Error
H38400	0.16%	H38400	1.73%				
H40800	2.12%	H40800	2.12%				
H45600	0.32%	H45600	2.10%				
H50400	0.79%	H48000	0.16%				
H55200	0.64%	H52800	1.36%				
H62400	0.16%	H57600	1.36%				
H72000	0.79%	H62400	0.16%				
H81600	2.12%	H69600	0.22%				
H84000	0.79%	H76800	1.73%				
H98400	1.63%	H79200	1.36%				
H100800	0.79%	H88800	0.55%				
H55555	0.00%	H91200	2.10%				
H62500	0.00%	H103200	0.94%				
H71428	0.00%	H105600	1.36%				
H83333	0.00%	H69444	0.00%				
H100000	0.00%	H78125	0.00%				
H125000	0.00%	H89285	0.00%				
H166666	0.00%	H104166	0.00%				
H250000	0.00%	H125000	0.00%				
H500000	0.00%	H156250	0.00%				
		H208333	0.00%				
		H312500	0.00%				
		H625000	0.00%				

HSERINNEXT

hserinnext mode

Is a function. Reads the next bytes available in the specified UARTs buffer without removing it from the buffer. The available mode values are:

Mode	Definition
0x01	Read UART1 and wait for byte
0x02	Read UART2 and wait for byte
0x81	Read UART1 and return -1 if no byte available
0x82	Read UART2 and return -1 if no byte available

```
temp var byte
temp = hserinnext 0x01
```

## HSEROUT

### Syntax

hserout {uart,}{{modifiers}data1,...,{modifiers}dataN]

- **UART** - is an optional argument on a 1 UART processor. It is a constant of 1 or 2 that specifies what UART to use. See HSERIAL Pin table below.
- **Modifiers** - supports all output modifiers. See Input Modifiers table below. For additional information regarding modifiers see the Modifier section of this manual.
- **Data** - is a variable where data sent from the debug window will be stored.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

HSERIN / HSEROUT use a special hardware feature called a UART. The UART allows the processor to perform other task while receiving or sending serial data. The hardware runs in the background and is processor independent. Data that is received, is stored in a buffer, regardless of what the processor is doing. The HSERIN / HSEROUT commands then retrieve this data from the buffer. Some processors have two UARTS that run independent of each other. The UART output is inverted. Which is setup to work with a RS232 circuit. Non inverted mode is not support. A simple inverter circuit could be used if needed. For all HSERIN / HSEROUT capable pins see the table below.

### HSERIAL Capable Pins

Processor	HSEROUT (TX)	HSERIN (RX)	HSEROUT (TX2)	HSERIN (RX2)
Nano 18	P5	P2		
Nano 28	P14	P15		
Nano 40	P14	P15		
Atom 24m	P14	P15		
Atom 28m	P14	P15		
Atom 40m	P14	P15		
AtomPro One	S_OUT	S_IN		
AtomPro 24m	P15	P14		
AtomPro 28m	P15	P14		
AtomPro 40m	S_OUT	S_IN	P15	P14
ARC32	S_OUT	S_IN	P26	P38

### Notes

Most of the information for HSEROUT would simply be duplicated from HSERIN. See HSERIN.

---

## HSERVO

### Syntax

hservo [pin\pos\spd, ..., pinN\posN\spdN]

- **Pin** - are constants, variables or expressions that specify the pin numbers connected to servos.
- **Pos** - are constants, variables or expressions that specify the desired positions for each of the specified servos.
- **Spd** - are constants, variables or expressions that specify the speed used to move each servo to its new position (default 0).

### Supported

- BA - NOT Supported.
- BAN - NOT Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

HSERVO uses a back ground hardware system to control one or several servos at a time. HSERVO is a non blocking command. Program execution will continue as normal while the command is executed. HSERVO can control the position and the speed of up to 32 servos. Before using this command the ENABLEHSERVO compile time directive must be included in your program.

### Notes

1. Analog servos may need to be deactivated. To accomplish this set its position to -24000 on the BAP 24 and BAP 28 modules and to -30000 on the BAP 40 module and ARC32 dev board.
2. The first time a servo is started its speed will be set to 0 even if you specify a speed.

### Example

The example HSERVO program is a very basic program that will control 2 servos connected to P0 and P1 of the microcontroller you are using.

```
hservo [p0\0,p1\0] ;set initial servo positions(centered)
main
  hservo [p0\1000\100,p1\1000\100]
  pause 1000
  hservo [p0\1000\100,p1\1000\100]
  pause 1000
goto main
```

### HSERVO System

The HSERVO system has several supporting commands that can be used to control the interaction between the program and the servo. The following commands require the HSERVO system is loaded by using the HSERVO command somewhere in your program.

HSERVOWAIT  
hservowait [pin1, ..., pinN]

In some situations it may be necessary to wait for the HSERVO command to finish updating the servo position before program execution resumes. The HSERVOWAIT command is used to delay program execution until HSERVO has finished moving the servos specified.

#### Example

The example will pause program execution until servos attached to PO through P4 are finished being moved to new positions.

```
hservowait [p0,p1,p2,p3,p4]
```

HSERVOPOS  
hservopos pin

To determine the current position a particular servo is set to the HSERVOPOS command is used. The value returned is the position the HSERVO system is currently driving the servo at. Typically there is no feedback from a servo for true position.

#### Example

The example sets a word sized variable and loads it with the last position used for a servo attached to P0.

```
temp var word  
temp = hservopos p0
```

HSERVOIDLE  
hservoidle pin

To determine if a servo is still being moved to a new position by HSERVO the HSERVOIDLE command is used. It will return a value of 0 if the servo specified is not idle. If the specified servo has reached its new position a value not equal to 0 (0xFFFFFFFF) will be returned.

#### Example

The example will update a servo on PO. The program will loop and report until the servo is idle.

```
hservo [p0\0]  
pause 1000  
hservo [p0\1000\100]  
  
main  
  if (NOT hservoidle p0) then  
    serout s_out,i9600,["Servo P0 is not idle",13]  
  endif  
goto main
```

---

## IF...THEN...ELSEIF...ELSE...ENDIF

### Simple Syntax

```
if expression then label  
if expression then goto label  
if expression then gosub label
```

### Extended Syntax

```
if expression then  
...code...  
endif
```

```
if expression then  
...code...  
else  
...code...  
endif
```

```
if expression then  
...code...  
elseif expression  
...code...  
endif
```

```
if expression then  
...code...  
elseif expression  
...code...  
else  
...code...  
endif
```

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

IF..THEN are the decision makers of MBasic. It evaluates a condition to determine if it is true. If you want to test the value of a variable against another variable or a known value you would use the IF..THEN. When the results are true the code after THEN is executed. If it returns false the code after THEN will be ignored. A simple example would be to increment a variable in a loop and each time through the loop test if our variable equals 10. This lets us control how many times through the loop we want to run. We can also test if our variables is greater than, less than or even not equal too. Several math expressions can be used as the condition to test.

```
main  
if temp = 10 then label  
goto main
```

The above statement is looking for our variable temp to equal 10. If it is true then we will jump to label. If its not true then lets keep looping. Since there is nothing to make our variable equal to 10 this loop would run forever.

#### Notes

1. Multiple ELSEIF blocks may be used in a single IF...THEN block.
2. ELSE will only execute following code if no conditions were true.
3. ENDIF is required to close a block of conditionals.

#### Example

This first example demonstrates using the IF...THEN argument with a go to label. If something is true jump to the label after the THEN statement. Otherwise, if the condition is false execute the commands on the next line after the THEN statement. You can follow the program flow with a terminal window connected at 9600 baud.

```
value var long
value = 0

main
    value = value+1
    if value = 10 then reset

        ;display the value on the PC terminal window
        serout s_out,i9600,["Value=",dec value,13]
        pause 1000
    goto main

reset
    value = 0
    goto main
```

#### GOSUB Example

A GOSUB statement can be used after a THEN. When the condition is true a GOSUB will send the program to the GOSUB label. Eventually a RETURN statement is expected. This will return the program to the next line of code after the GOSUB statement was used. Its an easy way to create conditional branching in a main program loop. The program will increment value by 1 each loop through. Once value is equal to 10 the condition becomes true and the GOSUB label is executed. Which in turns resets value to 0 starting the process over. This program was design to be followed using a terminal window connected to it at 9600 baud. Follow the results until you understand the decision making process. Can you guess what the terminal window will show?

```
value var long
value = 0

main
    value = value+1
    if value = 10 then gosub reset

        ;display the value on the PC terminal window
        serout s_out,i9600,["Value=",dec value,13]
        pause 1000
    goto main

reset
    value = 0
    return
```

### Advance Arguments

Now that we understand the basics of IF..THEN we can explore optional arguments. The optional arguments may not make sense at first. The next section will explain each with sample code that will display the result so you can follow along.

### ENDIF Example

Not always will you want to jump to a label or GOSUB. In some cases you may want to run a block of code when the condition is true. MBasic can execute a command or commands directly after the THEN statement instead of jumping to a label. The ENDF is used to tell MBasic run the following commands after THEN if the condition is true. So ENDF literally mean what it says, lets end the IF. When the IF..THEN condition is false ENDF tells MBasic to instead, run the commands after ENDF. This simply resumes normal program operation. ENDF is an easy way to execute a group of commands based on some condition returning true or completely skipping them if the condition is false. This program was design to be followed using a terminal window connected at 9600 baud. See if you can you guess the results?

```
value var long
value = 0

main
  value = value+1
  if value = 10 then
    value = 0
  endif

  ;display the value on the PC terminal window
  serout s_out,i9600,["Value=",dec value,13]
  pause 1000
goto main
```

### ELSEIF Example

In some cases you may want to test for several possible conditions. ELSEIF allows you to check multiple possible conditions until one of them is true in a single IF..ENDIF block. Until one of the ELSEIF conditions is found to be true each condition will be tested. As soon as one is found to be true the code in that section will execute and then program execution will continue starting after the ENDF.

```
if temp = 10 then
temp = 20
elseif temp = 20
temp = 10
endif
```

There is no limit to how many ELSEIF statements you can have in any single IF...THEN block. The following program shows an example of a program using multiple ELSEIF statements. Can you guess what the output will be?

This program was design to be followed using a terminal window connected at 9600 baud. See if you can you guess the results?

```
value var long
value = 0

main
  if value = 0 then
    value = 1

  elseif value = 1
    value = 2

  elseif value = 2
    value = 3

  elseif value = 3
    value = 4

  elseif value = 10
    value = 0
  endif

  ;display the value on the PC terminal window
  serout s_out,i9600,["Value=",dec value,13]
  pause 1000
goto main
```

## ELSE

So we tested a bunch of conditions and everything was false which means the program will resume normal operation, skipping any code found enclosed within the IF..THEN / ENDIF statements unless we add an ELSE block. The ELSE block will execute if everything else was FALSE. This program was design to be followed using a terminal window connected at 9600 baud. See if you can you guess the results?

```
value var long
value = 0

main
  if value = 1 then
    value = 1

  elseif value = 2
    value = 2

  else
    value = 3

endif

  ;display the value on the PC terminal window
  serout s_out,i9600,["Value=",dec value,13]
  pause 1000
goto main
```



---

## INPUT

Syntax  
input pin

- **Pin** - is any expression, constant or variable that specifies an input capable pin number to use.

Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

Description

In order for a pin to “see” something from the outside world we need to set it as an input. When a pin is set as an input, it can be read as 1 (high) or 0 (low). All input based commands will automatically set the pins state and after execution leave the pin in an input state. During start up all pins are in an input state. Typically the INPUT statement would be used in the beginning of your program to set specific pins to known states.

Notes

1. On power up, all pins by default are inputs.

Example

Connect to the running program example below using the terminal window set to a baud rate of 9600. As the program runs it will print the state of pins 0 to the terminal window. P1 is then set high and the first report is P0 set as a 1 (high). Next command sets it to an input and P0 is now 0 (low). P1 is then set back to an output. Since its last known state was high it will return to this state when set to an output again so P0 will read 1 (high) again. See if you can follow the program flow to see when the pin state changes.

```
;Connect p0 to p1

Input p0
  High p1 ;Set P1 to an output and high
  Serout s_out,i9600,["P0 state is ",dec IN0,13]
  Pause 1000

Input p1
  Serout s_out,i9600,["P0 state is ",dec IN0,13]
  Pause 1000

;Set P1 high, will remember its last set state
Output p1
  Serout s_out,i9600,["P0 state is ",dec IN0,13]
  Pause 1000
```

---

## I2COUT

### Syntax

i2cout sda, scl, controlbyte, [{modifiers}databyte1, ..., {modifiers}databyteN]

- **SDA** - is a variable or constant that defines the serial data pin (SDA).
- **SCL** - is a variable or constant that defines the serial clock pin (SCL).
- **ControlByte** - is a byte sized variable or constant (0-255) that sets the device ID and the device address of the I2C device.
- **Modifiers** - supports all output modifiers. See modifier section of this manual.
- **DataBytes** - is a list of byte sized variables or expressions with optional modifiers that are the values to be written or sent to an attached I2C device. The only limit to the number of data bytes in one command is the device its self.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The I2COUT command is generic to all I2C devices and not specific to I2C EEPROMs. There are many I2C devices available as the I2C format is fairly popular. The *controlbyte* is used to indicate the device type and address (A0,A1,A2). The high nib (reading from left to right, first 4 bits in a byte) control the device type ID. An I2C EEPROM would be %1010. This information is typically found in the device data sheet. The next 3 bits represent A0, A1, A2 pins and if they are set high or low. The last bit can be ignored.

A *controlbyte* of %10100000 indicates a device type ID of 1010 for I2C EEPROM, The next 3 zeros indicate A0, A1 and A2 are tied to ground for address 000. The last bit is ignored.

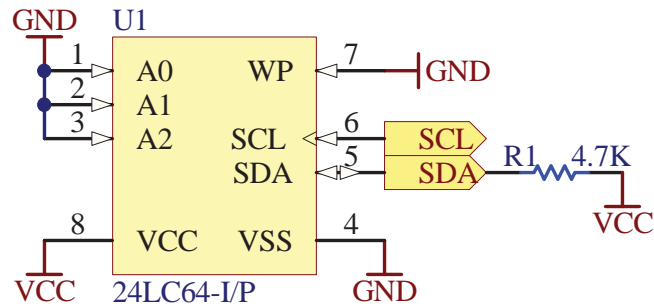
There are many different I2C devices. Each can have different control bytes. If you are using an I2C not mentioned in this manual you can refer to the device data sheet for information on addressing the device.

### Notes

1. To read or write any I2C eeprom an I2COUT command must be used to specify the address.
2. I2C devices other than I2C EEPROM will have a different device type ID.
3. A0, A1 and A2 are the physical address pins. If only 1 device exist on the I2C buss, address pins are usually connected to GND.
4. More than one I2C device can be on the same SDA and SCL pins. Each device requires a unique address by changing A0,A1 and A2 pins.
5. Some I2C EEPROM internally use A0, A1 and A2 to change memory banks. Commonly 24LC04 (24C04) and 24LC08 (24C08).

## Schematic

The schematic illustrates how to connect an I2C EEPROM for use with the sample code. A0,A1 and A2 are all tied to GND. This sets the device address to 000. WP is the write protect pin and can not be left floating. SCL is the clock pin and SDA is the data pin. For low speed EEPROM 10K pull-up resistor will work. However for faster I2C devices you may need to increase this value up to 1K.



## Examples

Connect to the following program with the terminal window set to 9600 baud. The program will write to the first 10 bytes of a 24LC01 EEPROM using byte addressing. Each loop *databyte* is added by 7. The values that will be written are 0, 7, 14, 21, 28, 35, 42, 49, 56 and 63. The loop pauses so you can see the data being printed to the screen.

```
;ALL - all_i2c_byte.bas

;This program will write and read to the first
;10 locations of a 24LC01 I2C eeprom using
;byte address.

databyte var byte
address var byte

databyte = 0
address = 0

pause 500

main
  serout s_out, i9600, [13, "Writing:",13]

  for address = 0 to 9

    i2cout p0, p1, %10100000, [address, databyte]
    Serout s_out, i9600, [dec address, " = ", dec databyte, 13]
    databyte = databyte + 7
    pause 100
  Next

      databyte = 0
      address = 0
      pause 100
      serout s_out, i9600, [13, "Reading:",13]

  for address = 0 to 9

    i2cout p0, p1, %10100000, [address]
    i2cin p0,p1, %10100000, [databyte]
    serout s_out, i9600, [dec address, " = ", dec databyte, 13]
    pause 100
  next

end
```

---

## I2CIN

### Syntax

i2cin sda, scl, controlbyte, [{modifiers}databyte1,...,{modifiers}databyteN]

- **SDA** - is a variable or constant that defines the serial data pin (SDA).
- **SCL** - is a variable or constant that defines the serial clock pin (SCL).
- **ControlByte** - is a byte sized variable or constant (0-255) that sets the device ID and the device address of the I2C device.
- **Modifiers** - readdm supports all input modifiers. See modifier section of this manual.
- **DataBytes** - is a list of byte sized variables with optional modifiers that store the values read from an attached I2C device. The only limit to the number of data bytes in one command is the device its self.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The I2CIN command is generic to all I2C devices and not specific to I2C EEPROMs. There are many I2C devices available as the I2C format is fairly popular. The *controlbyte* is used to indicate the device type and address (A0,A1,A2). The high nib (reading from left to right, first 4 bits in a byte) control the device type ID. An I2C EEPROM would be %1010. This information is typically found in the device data sheet. The next 3 bits represent A0, A1, A2 pins and if they are set high or low. The last bit can be ignored.

A *controlbyte* of %10100000 indicates a device type ID of 1010 for an I2C EEPROM, The next 3 bits are set to zeros which indicates address pins A0, A1 and A2 are tied to ground for address 000. The last bit is ignored.

### Notes

1. To read or write any I2C EEPROM an I2COUT command must be used to specify the address.
2. I2C devices other than I2C EEPROM will have a different device type ID.
3. A0, A1 and A2 are the physical address pins. Typically if only 1 device exist on the I2C buss the address pins are all tied to GND.
4. You can connect several I2C devices together on the same SDA and SCL pins by changing the A0,A1 and A2 address pins.
5. Some I2C EEPROM internally use A0, A1 and A2 to change memory banks. Commonly 24LC04 (24C04) and 24LC08 (24C08).

### Examples

See I2COUT.

## Examples

Connect to the following program with the terminal window set to 9600 baud. This next example will write to the first 10 locations of a 24LC512 using word sized addressing. *Address* is set as a word sized variable. For the *Address* data we use a high and low byte modifier to load the address values from *Address*. Each loop we add 7 to *databyte* which is what we are writing to the EEPROM. The values written are 0, 7, 14, 21, 28, 35, 42, 49, 56 and 63. The program will print to the terminal window what it is writing and then what is being read. If the values above are not returned then you have a bad EEPROM or you don't have something wired correctly (see schematic).

```
;ALL - all_i2c_word.bas

;This program will write and read to the first
;10 locations of a 24LC512 I2C eeprom using
;word addressing.

databyte var byte
address var Word

databyte = 0
address = 0

pause 500

main
  serout s_out, i9600, [13, "Writing:",13]

  for address = 0 to 9

    i2cout p0, p1, %10100000, [address.byte1, address.byte0, databyte]
    Serout s_out, i9600, [dec address, " = ", dec databyte, 13]
    databyte = databyte + 7
    pause 100
  Next

  databyte = 0
  address = 0
  pause 100
  serout s_out, i9600, [13, "Reading:",13]

  for address = 0 to 9

    i2cout p0, p1, %10100000, [address.byte1,address.byte0]
    i2cin p0,p1, %10100000, [databyte]
    serout s_out, i9600, [dec address, " = ", dec databyte, 13]
    pause 100
  next

end
```

## Examples

Connect to the following program with the terminal window set to 9600 baud. The 24LC08 has 4 pages of 255 bytes each. The example will write to the first 10 memory locations of all 4 pages then read back and print to the terminal window. Each page is accessed with A0,A1 and A2 internally. So the control bits change for each page.

```

;ALL - all_i2c_paging.bas

;This program will write and read to the first
;10 memory locations of all 4 pages of a 24LC08
;I2C eeprom using byte addressing.

databyte var byte
address var Byte

databyte = 10
address = 0

pause 500

main
  serout s_out, i9600, [13, "Writing:",13]

  for address = 0 to 9      ;increment through the first 10 locations

    i2cout p0, p1, %10100000, [address, databyte]    ;page 1
    i2cout p0, p1, %10100010, [address, databyte]    ;page 2
    i2cout p0, p1, %10100100, [address, databyte]    ;page 3
    i2cout p0, p1, %10100110, [address, databyte]    ;page 4

    ;lets see what we are writing to each address on each page
    serout s_out, i9600, ["Page 1 to 4-> ",dec address," = ",dec |
    databyte, 13]

    databyte = databyte + 10
    pause 100
  Next

  databyte = 0
  address = 0
  serout s_out, i9600, [13, "Reading:",13]

  for address = 0 to 9      ;increment through the first 10 locations

    i2cout p0, p1, %10100000, [address] ;page 1
    i2cin p0,p1, %10100000, [databyte]

    i2cout p0, p1, %10100010, [address] ;page 2
    i2cin p0,p1, %10100010, [databyte]

    i2cout p0, p1, %10100100, [address] ;page 3
    i2cin p0,p1, %10100100, [databyte]

    i2cout p0, p1, %10101100, [address] ;page 4
    i2cin p0,p1, %10101100, [databyte]

    ;lets read and print each address of each page.
    serout s_out, i9600, ["Page 1 to 4-> ",dec address," = ",dec |
    databyte, 13]

    pause 100
  next

end

```

---

## LCDINIT

### Syntax

lcdinit RS\ E\ D7\ D6\ D5\ D4 {,RW}

- **RS** - is any expression, constant or variable that specifies the I/O pin connected to the LCD's R/S pin.
- **E** - is any expression, constant or variable that specifies an I/O pin connected to the LCD's enable pin.
- **D7** - is any expression, constant or variable that specifies an I/O pin connected to the LCD's data bit 7 pin.
- **D6** - is any expression, constant or variable that specifies an I/O pin connected to the LCD's data bit 6 pin.
- **D5** - is any expression, constant or variable that specifies an I/O pin connected to the LCD's data bit 5 pin.
- **D4** - is any expression, constant or variable that specifies an I/O pin connected to the LCD's data bit 4 pin.
- **RW** - is any expression, constant or variable that specifies an I/O pin connected to the LCD's read or write enable pin.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

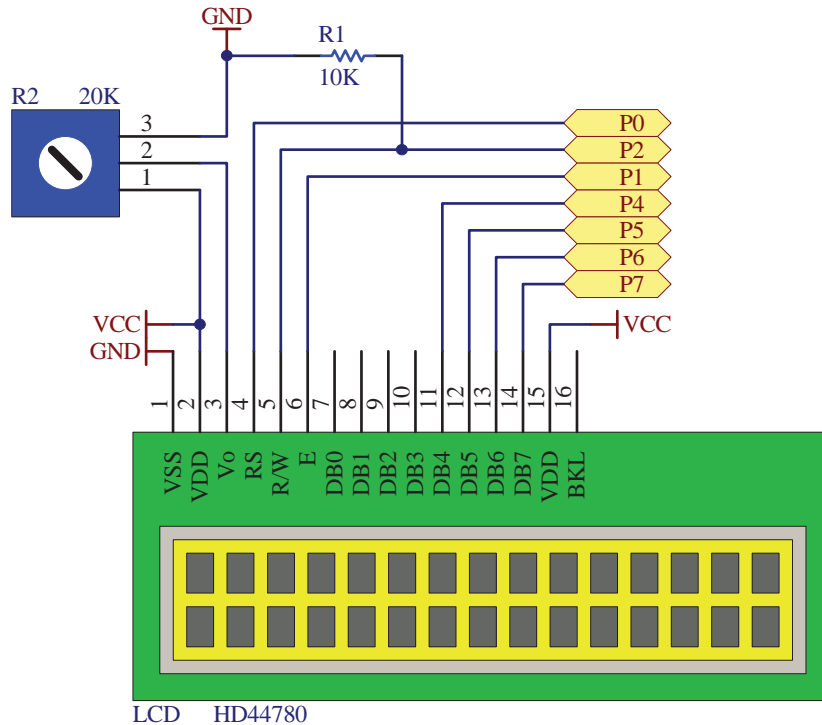
### Description

There are 3 LCD commands available in MBasic and are used to interface to a Hitachi HD44780 controller based LCD. This is probably one of the most common LCDs available. 1x16 all the way up to 4x40 size screens are supported. The HD44780 controller does require special initialization commands so the LCDINIT command was added to handle this automatically. The LCDINIT command is only needed once in a program, typically to the beginning of a program. The LCD does require some time to power up before running the LCDINIT command. A short pause placed before the LCDINIT command will handle any power up issues. Otherwise the LCDINIT command can be ran later in a program to provide the same power up time.



Schematic

The schematic shown is a common HD44780 wiring. 7 I/O pins are required to drive it in 4 bit mode. To control the screen contrast a 20K potentiometer R2 is used. Optionally the backlight can be controller by driving BKL to GND either hard wired or through a transistor circuit for optional software control.



Example

The following program allows the LCD time to power up with a short pause. The LCD is then initialized. After which "Hello World!" is printed to the screen using the LCDWRITE command.

```

;DEMO PROGRAM - LCDINIT.BAS

Pause 500
lcdinit p0\p1\p7\p6\p5\p4,p2
lcdwrite p0\p1\p7\p6\p5\p4,p2, [CLEARLCD,HOMELCD,SCR,TWOLINE,"Hello World"]
    
```

---

## LCDWRITE

### Syntax

lcdwrite RS\ E\ D7\ D6\ D5\ D4 {,RW}, [(modifiers) expression1, ..., (modifiers) expressionN]

- **RS** - is any expression, constant or variable that specifies the I/O pin connected to the LCD's R/S pin.
- **E** - is any expression, constant or variable that specifies an I/O pin connected to the LCD's enable pin.
- **D7** - is any expression, constant or variable that specifies an I/O pin connected to the LCD's data bit 7 pin.
- **D6** - is any expression, constant or variable that specifies an I/O pin connected to the LCD's data bit 6 pin.
- **D5** - is any expression, constant or variable that specifies an I/O pin connected to the LCD's data bit 5 pin.
- **D4** - is any expression, constant or variable that specifies an I/O pin connected to the LCD's data bit 4 pin.
- **RW** - is any expression, constant or variable that specifies an I/O pin connected to the LCD's read or write enable pin.
- **Modifiers** - are any output modifiers as shown in the modifiers section of this manual.
- **Expression** - is any expression, constant or variable that specifies data to be displayed to the lcd screen. In addition screen locations can be specified (SCRRAM).

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The LCD commands are setup to work with the Hitachi HD44780 controller. This is probably one of the most common LCDs available. 1x16 all the way up to 4x40 size screens are supported. Before using the LCDWRITE command see LCDINIT command.

The LCDWRITE command is used to address the HD44780 ram. Depending on the display size there is one byte of screen ram for each display character. Most displays will include additional off screen ram. The off screen ram locations can be used to store characters that are then later shifted onto the screen. LCDWRITE includes several formatting and cursor control commands. These are used to position the cursor or set features on the HD44780 controller. The following table includes all the LCD commands MBasic supports.

LCD Command Table

Command	Value	Function
Lcdclear	\$101	Clear display. Clears all on and off screen ram.
Lcdhome	\$102	Return to home position.
Inccur	\$104	Auto increment cursor (default).
Incscr	\$105	Auto increment display.
Deccur	\$106	Auto decrement cursor.
Decscr	\$107	Auto decrement display.
Off	\$108	Display, cursor and blink OFF.
Scr	\$10C	Display ON, cursor and blink OFF.
Scrbk	\$10D	Display and blink ON, cursor OFF.
Scrcur	\$10E	Display and cursor ON, blink OFF.
Scrcurblk	\$10F	Display, cursor and blink ON.
Curleft	\$110	Move cursor left.
Currright	\$114	Move cursor right.
Oneline	\$120	Set display for 1 line LCDs.
Twoline	\$128	Set display for 2 line LCDs.
Cgram   address	\$140	Set CGRAM address for reading or writing.
Scrram   address	\$180	Set display RAM address for reading or writing.

### LCD Commands

The LCD commands are used to setup the display. This is done if the first LCDWRITE command issued. It only needs to be done once unless something is needs to be changed the next time the LCDWRITE command is used. If you are using a 2x16 LCD display you would need to issue the Twoline command. One of the SCR commands are required to turn the display on. Otherwise data will be written to the DDRAM but not displayed. The following code snippet turns the screen on, clears it and moves the cursor to home before printing "Hello World!".

```
;DEMO PROGRAM - LCDINIT.BAS
Pause 500
lcdinit p0\p1\p7\p6\p5\p4,p2
lcdwrite p0\p1\p7\p6\p5\p4,p2, [CLEARLCD,HOMELCD,SCR,TWOLINE,"Hello World"]
```

### LCD DDRAM

The HD44780 uses DDRAM to store ASCII characters. The one HD44780 control can handle up to 4x20 ram locations. This means one controller can drive up to a 4x20 screen. A 4x40 display would use to HD44780 controllers. With this in mind a 1x16 or 2x16 screen still has all the ram locations to make up a 4x20 display. Only certain locations will actually print to the display while the unused can be written to an characters can be stored for later shifting into the display area. The diagrams below illustrate how the memory map (DDRAM) is handled for each display.

### 2x16 LCD Memory Map

The HD44780 controller has the same memory locations regardless of display size. Displays that are smaller than the amount of DDRAM available will allow writes and reads to off screen locations. The 2x16 display uses only 0-15 and 64-79 for on screen. As you can see the memory locations are not in order. This is due to how the HD44780 controller was designed.

0	1	2	3	4	5	6	7	8	9	0	11	12	13	14	15	16	17	18	19
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83
20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103

### 4x20 LCD Memory Map

The HD44780 controller has the same memory locations regardless of display size. The 4x20 display uses all the memory locations to display on the screen. Again you can see memory address locations are not in order.

0	1	2	3	4	5	6	7	8	9	0	11	12	13	14	15	16	17	18	19
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83
20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103

### Cursor Positioning

The cursor is set to the position the screen is being written to. The cursor can be positioned anywhere in the 80 locations. With this control the cursor can be moved around the screen and individual characters can be changed or erased at will. Characters can also be loaded into the off screen memory and then shifted onto the display area.

This following code snippet is setup to work with the schematic shown for LCDINIT. It moves the cursor to the home position and clears the memory. Returning the cursor home with LCDHOME does not clear the memory. The LCDCLEAR command is used in conjunction with sending the cursor home:

```
lcdwrite p0\p1\p7\p6\p5\p4,p2,[lcdc clear,lcdhome]
```

The next line will print an ASCII "A" to an off screen location on a 2x16 LCD Display using the screen ram command with an address. If you look at the 2x16 memory map you will see location 16 is off screen:

```
lcdwrite p0\p1\p7\p6\p5\p4,p2,[scrram+16, "A"]
```

### Custom Characters

Most HD44780 controllers provides room for 8 user definable characters. They are ASCII values 0 to 7. The programmable characters are 5x8 pixels. There are 8 bytes for each custom character. The bytes are arranged in a grid to define the new character. The custom characters can be used to make sprites that can then be used to form larger characters that are made up of multiple locations and spread across more than one display line. The first 3 bits are ignored (7-5). Only bits 4-0 are used to define the character. An on pixel is express with a 1 and off is 0.

Bits ->	4	3	2	1	0	Binary	Hex	Decimal
Byte 0						00000000	00	00
Byte 1		■		■		00001010	0A	10
Byte 2		■		■		00001010	0A	10
Byte 3		■		■		00001010	0A	10
Byte 4						00000000	00	00
Byte 5	■				■	00010001	11	17
Byte 6		■	■	■		00001110	0E	14
Byte 7						00000000	00	00

### Example

To write and display the custom character shown above you would use the LCDWRITE command with the CGRAM function. The following code snippet will write and display the custom character shown above on a 2x16 LCD display. Connect LCD as shown for LCDINIT.

```
lcdinit p0\p1\p7\p6\p5\p4
lcdwrite p0\p1\p7\p6\p5\p4,[cgram+0,0x00,0x0A,0x0A,0x0A,0x00,0x11,0x0E,0x00,scrram+0]
```

The sample code below will clear the screen and starting from home print "Hello Word!" with our newly created custom character shown above.

```
lcdinit p0\p1\p7\p6\p5\p4
;create custom character
lcdwrite p0\p1\p7\p6\p5\p4,[cgram+0,0x00,0x0A,0x0A,0x0A,0x00,0x11,0x0E,0x00,scrram+0]
;clear screen, home cursor, print "Hello World!" and custom character
lcdwrite p0\p1\p7\p6\p5\p4,[lcdclear,lcdhome,"Hello World! ",0]
```

### Custom Shapes

Custom characters can be created and used as sprites to form large graphics. The following example will display a 2 line 3 character wide heart in the middle of a 2x16 LCD display. Experiments see what creative characters and sprites you can come up with. The leading 3 zeros have been left off the binary value due to space constraints. When creating the custom characters you will need to add the 3 leading zeros. See example code.

Bits ->	4	3	2	1	0	Bin	4	3	2	1	0	Bin	4	3	2	1	0	Bin
Byte 0						00000						00000						00000
Byte 1				■	■	00011						00000	■	■				11000
Byte 2			■	■	■	00111	■				■	10001	■	■	■			11100
Byte 3		■	■	■	■	01111	■	■		■	■	11011	■	■	■	■		11110
Byte 4		■	■	■	■	01111	■	■	■	■	■	11111	■	■	■	■		11110
Byte 5	■	■	■	■	■	11111	■	■	■	■	■	11111	■	■	■	■	■	11111
Byte 6	■	■	■	■	■	11111	■	■	■	■	■	11111	■	■	■	■	■	11111
Byte 7	■	■	■	■	■	11111	■	■	■	■	■	11111	■	■	■	■	■	11111
Bits ->	4	3	2	1	0	Bin	4	3	2	1	0	Bin	4	3	2	1	0	Bin
Byte 0	■	■	■	■	■	11111	■	■	■	■	■	11111	■	■	■	■	■	11111
Byte 1		■	■	■	■	01111	■	■	■	■	■	11111	■	■	■	■		11110
Byte 2			■	■	■	00111	■	■	■	■	■	11111	■	■	■			11100
Byte 3				■	■	00011	■	■	■	■	■	11111	■	■				11000
Byte 4					■	00001	■	■	■	■	■	11111	■					10000
Byte 5						00000	■	■	■	■	■	11111						00000
Byte 6						00000		■	■	■		01110						00000
Byte 7						00000			■			00100						00000

### Example

The following program will load all 6 custom characters and display them on a 2x16 LCD display using the schematics shown for LCDINIT. The program uses the hex values which can be generated by converting from binary to hex.

```

lcdinit p0\p1\p2\p3\p4\p5

;create custom heart character
lcdwrite p0\p1\p2\p3\p4\p5,[cgram+0,0x00,0x03,0x07,0x0F,0x0F,0x1F,0x1F,0x1F]
lcdwrite p0\p1\p2\p3\p4\p5,[0x00,0x00,0x11,0x1B,0x1F,0x1F,0x1F,0x1F]
lcdwrite p0\p1\p2\p3\p4\p5,[0x00,0x18,0x1C,0x1E,0x1E,0x1F,0x1F,0x1F]
lcdwrite p0\p1\p2\p3\p4\p5,[0x1F,0x0F,0x07,0x03,0x01,0x00,0x00,0x00]
lcdwrite p0\p1\p2\p3\p4\p5,[0x1F,0x1F,0x1F,0x1F,0x1F,0x1F,0x0E,0x04]
lcdwrite p0\p1\p2\p3\p4\p5,[0x1F,0x1E,0x1C,0x18,0x10,0x00,0x00,0x00,scrram+0]

lcdwrite p0\p1\p2\p3\p4\p5,[TWOLINE,SCR,0,1,2,scrram+0x40,3,4,5]

end

```

---

## LCDREAD

### Syntax

```
lcdread RS\ E\ D7\ D6\ D5\ D4 {,RW}, {cgram+} address,  
      [{modifiers} expression1, ..., {modifiers} expressionN]
```

- **RS** - is any expression, constant or variable that specifies the I/O pin connected to the LCD's R/S pin.
- **E** - is any expression, constant or variable that specifies an I/O pin connected to the LCD's enable pin.
- **D7** - is any expression, constant or variable that specifies an I/O pin connected to the LCD's data bit 7 pin.
- **D6** - is any expression, constant or variable that specifies an I/O pin connected to the LCD's data bit 6 pin.
- **D5** - is any expression, constant or variable that specifies an I/O pin connected to the LCD's data bit 5 pin.
- **D4** - is any expression, constant or variable that specifies an I/O pin connected to the LCD's data bit 4 pin.
- **RW** - is any expression, constant or variable that specifies an I/O pin connected to the LCD's read or write enable pin.
- **Cgram** - LCDREAD defaults to reading screen ram. CGRAM is an optional argument which specifies custom character ram (CGRAM). If CGRAM is used the LCDREAD command will return data store in the CGRAM and not SCRRAM.
- **Address** - specifies what address to start the read at.
- **Modifiers** - are any output modifiers as shown in the modifiers section of this manual. They are used to format the out going data (Hex, Bin, Dec and so on).
- **Expression** - is any expression, constant or variable that specifies data to be displayed to the lcd screen.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The LCD commands are setup to work with the Hitachi HD44780 controller. This is probably one of the most common text LCDs available. 1x16 all the way up to 4x40 size screens are supported. Before using the LCDREAD command see LCDINIT command.

The LCDREAD command is used to read the memory addresses of the HD44780 ram. The ram memory map is standard across LCD display sizes. The only difference being what values are actual display ram and what values are off screen ram. This is outlined in the LCDWRITE section of this manual under the memory map tables. The LCDREAD command can access both screen ram and the custom character area (CGRAM).

#### Example

If you are running a large menu system and some of the information is generated on the fly the LCDREAD command can be used to retrieve screen data to determine what is being displayed. It can also be used to temporarily store data in the off screen ram. The following program will read the first location of screen ram on a 2x16 LCD display.

```
character var byte
lcdinit p0\p1\p7\p6\p5\p4,p2
lcdread p0\p1\p7\p6\p5\p4,p2,0,[character]
serout s_out, i9600,[character,13]
```

The next program will read the CGRAM where custom characters are created. Since it takes 8 bytes to define 1 custom character we will need to read all 8 bytes in and create a variable array to do so. After reading the first custom character location the program will display the results to a terminal window connected at 9600 baud.

```
cch var byte(8)
lcdinit p0\p1\p2\p3\p4\p5,p6
lcdread p0\p1\p2\p3\p4\p5,p6,cgram+0,[cch( 0 ),cch( 1 ),cch( 2 ),cch( 3 ),|
cch( 4 ),cch( 5 ),cch( 6 ),cch( 7 )]
serout s_out,i9600,[hex cch( 0 )," ",hex cch( 1 )," ",hex cch( 2 )," ",|
hex cch( 3 )," ",hex cch( 4 )," ",hex cch( 5 )," ",hex cch( 6 )," ",hex cch( 7 )]
end
```



---

## LOOKDOWN

### Syntax

lookdown value,{operator,} [list],target

- **Value** - is a variable or constant that will be used to compare to the *list*.
- **Operator** - is the comparison operator which defines how *value* is applied to *list*. If no operator is given the default operator is equals (=). Comparison operators are =, <, >, <>, >=, <=.
- **List** - is any combination of variables, constants or expressions that will be used to compare against. Variables can be any type variable including floating point, signed, longs.
- **Target** - is where the result of a comparison are store. The index position of a match from *list*, is loaded into *target*.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The LOOKDOWN command checks through a list of variables or constants looking for the first match to *value* using the comparison operator. Each item in *list* is indexed starting from 0. If you have 10 variable or constants in the list then the expected results will be from 0 to 9.

### Notes

1. The LOOKDOWN command will stop on the first result that is true. If there is a possibility that two or more *values* in *list* will return true the LOOKDOWN command returns the first match.
2. The first item in *list* has an index value of 0. If the first item is true in *list*, a 0 is returned.
3. If no results are true then *target* is left unchanged.

### Examples

The LOOKDOWN command can be used to determine the relevance of an unknown value by comparing it to a known list of known values. An example would be determining a range of something such as low, medium or high.

Connect to the following program with the terminal window set to 9600 baud. The DEC modifier was used to convert the variable to real numbers. If the value 1 is sent, the program will return 1:Low. Since we used the comparison operator less than (<) and equals (=), 1 is less than and not equal to anything in the list. Type 100 in the terminal window and press enter. 100: Low is returned since 100 is equal to something in the list. If 101 was sent 101: Medium would be returned. 101 is greater than 100 and less than 512. A good programming technique is coding to anticipate errors or unexpected results. Try sending a value of 1025.

```
Value      Var   Long
Target     Var   Long

Main
  Value = 0
  Target = 3

  Serin S_IN, i9600, [DEC Value]

  LOOKDOWN value, <=, [100, 512, 1024], target
  IF target = 0 Then
    Serout S_OUT, i9600, [0, DEC value, " : Low",13]
  ENDIF
  IF target = 1 Then
    Serout S_OUT, i9600, [0, DEC value, " : Medium",13]
  ENDIF
  IF target = 2 Then
    Serout S_OUT, i9600, [0, DEC value, " : High",13]
  ENDIF
  IF target = 3 Then
    Serout S_OUT, i9600, [0, DEC value, " : Out of range",13]
  ENDIF

  Goto Main ;Repeat forever
```

---

## LOOKUP

### Syntax

lookup index,[list],target

- **Index** - is a variable, constant or expression that will be used to select a position in *list*.
- **List** - is any combination of variables, constants or expressions. Variables can be any type of variable including floating point, signed, longs.
- **Target** - is the variable where the value returned from *list* is stored.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The LOOKUP command use the *index* value to locate a value from *list* and load it into *target*. Each item in *list* is indexed starting from 0. If you have 10 variable or constants in *list* then *index* can be a value from 0 to 9.

### Notes

1. The first item in *list* has an index value of 0.
2. If *index* has a value that does not match a position in *list* then *target* is left unchanged.

### Examples

Connect to the following program with the terminal window set to 9600 baud. Enter a value from 0 to 8. The number received will be stored in *index*. The LOOKUP command will then use this number and move to that position in the list. The results are loaded into *target*. In the program there are 9 values in *list*. Since the first position is 0, only 0 to 8 are valid input commands. Any other values will leave *target* unchanged. For an example of LOOKUP in use see SOUND command example.

```
;ALL - all_lookup.bas

Index var Word
Target var Word

Main
  Index = 0
  Target = 0

  serout s_out, i9600,[1, "Type a value from 0 to 8: "]
  Serin s_in, i9600,[dec Index]

  lookup Index,[10,20,30,40,50,60,70,80,90],Target

  serout s_out, i9600,[0, 13, 13, "Index Position= ", dec Index]
  serout s_out, i9600,[13, "Result= ", dec Target,13]

Goto Main
```

---

Constant strings can also be used. The following program will print out one character per loop. Since there is 18 characters we use a for next loop to increment through the LOOKUP *list*.

```
;ALL - all_lookup2.bas

index var word
target var byte

main
  serout s_out,i9600,[0]
  for index = 0 to 17
    lookup index,["Basic Micro Rules!"],target
    serout s_out,i9600,[target]
    pause 100
  next
  goto main
```

---

## LOW

Syntax  
low pin

- **Pin** - is a variable or constant that specifies which pin to go high.

Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

Description

The LOW command is an output command that will set a pin LOW. All pins can be in 3 states. HIGH, LOW and FLOAT (input). The LOW command will change any of these states to an output and set the pin LOW (GND)

Notes

1. If a variable is used, the variables value will be directly translated into a pin. If the variable equals zero then P0 will go low. If the value is out of range to the pins that are present nothing will happen. If you have 10 pins and the variable equals 11 then nothing will happen.

Examples

The following program is a simple loop using HIGH and LOW commands that will repeat for ever. Connect an LED from P0 to P7. They will blink in sequence, repeating forever.

```
Pins var byte
Main
  Pins = 0
  For Pins = 0 to 7
    High Pins
    Pause 200
    Low Pins
    Pause 200
  Next
  Goto Main
```

---

## NAP

Syntax  
nap period

- **Period** - is a variable, constant or expression that determines the duration of nap. Valid range is 0 to 7.

Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

Nap Period

Nap Value	Time
0	18ms
1	36ms
2	72ms
3	144ms
4	288ms
5	576ms
6	1152ms
7	2304ms

Notes

1. The period can vary slightly with temperature, supply voltage and manufacturing tolerances.
2. The Atom Pro will immediately wake up from a nap if an interrupt occurs.
3. The NAP command does not affect internal registers so your program will continue executing when the time expires.

Description

The NAP command executes the processor's internal sleep mode for a specified time (Period). Power consumption is reduced in sleep mode. NAP is a simplified version of the SLEEP command. NAP is an easy way to conserve power on battery powered application when the processor is idle. The AtomPro can conserve power until an outside trigger occurs (Interrupt) to wake the device.

Example

The following example will put the processor in a lower power state for 2,304 milliseconds.

```
main
  serout s_out, i9600, ["Going to NAP"]
  nap 7
  serout s_out, i9600, ["I'm wake!"]
  goto main
```

---

## OWIN

### Syntax

owin pin,mode,{FailLabel,} [{modifiers}InData1, ..., {modifiers}InDataN]

- **Pin** - is a variable, constant or expression that specifies the pin used for 1-wire data transfer.
- **Mode** - is a variable, constant or expression the specifies the data transfer mode as described in the table below.
- **FailLabel** - is a label the program will jump to if communications fails (No Chip present).
- **InData** - is a list of variables with optional modifiers to store incoming data from an attached 1-wire device.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Modes

Mode	Reset	Byte / Bit	Speed
0	None	Byte	Low
1	Before Data	Byte	Low
2	After Data	Byte	Low
3	Before and After	Byte	Low
4	None	Bit	Low
5	Before Data	Bit	Low

### Description

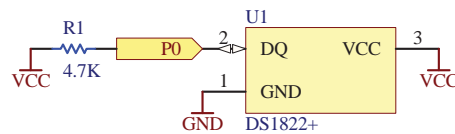
The 1-wire protocol was developed by Maxim (Dallas Semiconductor). It is a 1 wire asynchronous serial protocol that does not require a clock. Most 1-wire devices can optionally be powered from the data line. This is known as parasitic power. This means the device is powered from a extended data high (1) state on its data pin. A high (1) state is held briefly to charge a small internal capacitor. One wire is setup to have a single master device which communications with one or more 1-wire devices over a single data line. This network is dubbed a "MicroLan". The master initiates and controls all activities on the 1-wire bus.

### Notes

1. The 1-wire parts use CMOS/TTL logic levels with open collector outputs. The data line requires a 4.7K pull-up.

## Schematics

The following schematic shows how to wire up a DS1822+ temperature device.



## Example

This example shows how to read the temperature sensor DS1822+. The commands used for each 1-wire device will differ. To find the list of supported commands for the device being used see the device data sheet. Wire up the DS1822+ as shown in the schematics. Load the program below and connect using Studio terminal window at 9600 baud. The program will update the current temperature reading from the 1-wire part. Place your fingers on the device and watch the temperature reading increase.

```
temp var word
convert var float
counter var byte

main
  owout P0,1,main,[$cc,$44]

wait
  owin P0,0,[temp]
  if temp = 0 then wait
  owout P0,1,main,[$cc,$be]
  owin P0,0,[temp.byte0,temp.byte1]
  convert = tofloat temp / 2.0
  serout s_out,i9600,["Temperature = ",real convert," C",13]
goto main
```

If you are only using one device on the MicroLan then you can send a command to skip the unique ID. Using mode 1 (byte mode, low speed, and reset before data) we send the command \$CC (Skip ROM) which sets the DS1822+ to accept commands regardless of its unique ID code. The next command is \$44 (Convert Temperature) which initiates the temperature conversion and stores the result in the DS1822's scratch pad memory.

Using mode 0 (byte mode, low speed, and no reset) we read back the conversion status. While a conversion is in progress the read will be a 0. When data is ready it changes to a 1. The example program will loop until a conversion is finished.

Command \$CC is sent, then \$BE (Read Scratch pad) which tells the DS1822 to send the two bytes from its scratch pad on the next read. The the two bytes from the DS1820's scratch pad are read and stores them in the variable temp. The program makes use of the variable modifiers byte0 and byte1 to load the word variable temp with the low and high bytes of the temperature.

The program then converts the temperature to floating point format and divides by 2.0 because the DS1822+'s output is in 0.5°C steps. The result is then sent to a terminal window set to 9600 baud.



---

## OWOUT

### Syntax

owout pin,mode,{NCLabel,}[Data1, ..., DataN]

- **Pin** - is a variable, constant or expression that specifies the pin used for 1-wire data transfer.
- **Mode** - is a variable, constant or expression the specifies the data transfer mode as described in the table below.
- **FailLabel** - is a label the program will jump to if communications fails (No Chip present).
- **Data1..Data2** - is a variable, constant or expressions that specifies the data to be send to the 1-wire device.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Modes

Mode	Reset	Byte / Bit	Speed
0	None	Byte	Low
1	Before Data	Byte	Low
2	After Data	Byte	Low
3	Before and After	Byte	Low
4	None	Bit	Low
5	Before Data	Bit	Low

### Description

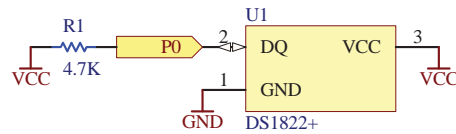
The information here is duplicated from OWIN. The 1-wire protocol was developed by Maxim (Dallas Semiconductor). It is a 1 wire asynchronous serial protocol that does not require a clock. Most 1-wire devices can optionally be powered from the data line. This is known as parasitic power. This means the device is powered from a extended data high (1) state on its data pin. A high (1) state is held briefly to charge a small internal capacitor. One wire is setup to have a single master device which communications with one or more 1-wire devices over a single data line. This network is dubbed a "MicroLan". The master initiates and controls all activities on the 1-wire bus.

### Notes

1. The 1-wire parts use CMOS/TTL logic levels with open collector outputs. The data line requires a 4.7K pull-up.

## Schematics

The following schematic shows how to wire up a DS1822+ temperature device.



## Example

This example shows how to read the temperature sensor DS1822+. The commands used for each 1-wire device will differ. To find the list of supported commands for the device being used see the device data sheet. Wire up the DS1822+ as shown in the schematics. Load the program below and connect using Studio terminal window at 9600 baud. The program will update the current temperature reading from the 1-wire part. Place your fingers on the device and watch the temperature reading increase.

```
temp var word
convert var float
counter var byte

main
  owout P0,1,main,[$cc,$44]

wait
  owin P0,0,[temp]
  if temp = 0 then wait
  owout P0,1,main,[$cc,$be]
  owin P0,0,[temp.byte0,temp.byte1]
  convert = tofloat temp / 2.0
  serout s_out,i9600,["Temperature = ",real convert," C",13]
goto main
```

If you are only using one device on the MicroLan then you can send a command to skip the unique ID. Using mode 1 (byte mode, low speed, and reset before data) we send the command \$CC (Skip ROM) which sets the DS1822+ to accept commands regardless of its unique ID code. The next command is \$44 (Convert Temperature) which initiates the temperature conversion and stores the result in the DS1822's scratch pad memory.

Using mode 0 (byte mode, low speed, and no reset) we read back the conversion status. While a conversion is in progress the read will be a 0. When data is ready it changes to a 1. The example program will loop until a conversion is finished.

Command \$CC is sent, then \$BE (Read Scratch pad) which tells the DS1822 to send the two bytes from its scratch pad on the next read. The the two bytes from the DS1820's scratch pad are read and stores them in the variable temp. The program makes use of the variable modifiers byte0 and byte1 to load the word variable temp with the low and high bytes of the temperature.

The program then converts the temperature to floating point format and divides by 2.0 because the DS1822+'s output is in 0.5°C steps. The result is then sent to a terminal window set to 9600 baud.

---

## OUTPUT

Syntax  
output pin

- **Pin** - is a variable, constant or expression that specifies an output capable pin to use.

Supported

- BA - supported
- BAN - supported
- BAP - supported
- BAP40 - supported

Description

In order for a pin to control something in the outside world we need to set it as an output. When a pin is set as an output, only then can we specify whether it is high (1) or low (0). All output based commands will automatically set the pins state. But during start up all pins are in an input state and after an input command is ran the pin is left in an input state. The OUTPUT command can be used to set a pin as an output. Typically the OUTPUT statement would be used in the beginning of your program to set specific pins to known states.

Notes

1. On power, all pins by default are inputs.

Example

Connect to the running program example below using the terminal window set to a baud rate of 9600. As the program runs it will print the state of pins 0 to the terminal window. P1 is then set high and the first report is P0 set as a 1 (high). Next command sets it to an input and P0 is now 0 (low). P1 is then set back to an output. Since its last known state was high it will return to this state when set to an output again so P0 will read 1 (high) again. See if you can follow the program flow to see when the pin state changes.

```
;Connect p0 to p1

Input p0
  High p1 ;Set P1 to an output and high
  Serout s_out,i9600,["P0 state is ",dec IN0,13]
  Pause 1000

Input p1
  Serout s_out,i9600,["P0 state is ",dec IN0,13]
  Pause 1000

;Set P1 high, will remember its last set state
Output p1
  Serout s_out,i9600,["P0 state is ",dec IN0,13]
  Pause 1000
```

---

## PAUSE

Syntax  
pause time

- **Time** - is a variable, constant or expression that specifies number of milliseconds to wait.

Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

Description

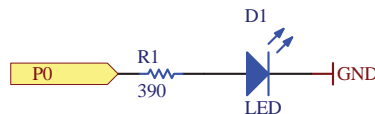
The PAUSE command is used to create a predetermined delay in a program. The amount of delay is specified in milliseconds. There are 1000 milliseconds in 1 second. So if a value of 500 is used for time, it would be a half second. If 1000 is used for time, the program would wait 1 second.

Notes

1. PAUSE is fairly accurate. However over time the count will be off. This will vary from processor to processor. The Nano uses an internal clock which is less accurate than the external clocks used on the modules. Therefore the error rate would be higher with the Nano.

Schematic

Connect a LED as shown below. You can substitute the resistor for anything from 330 ohms to 470 ohms.



Example

The following program will blink an led at approximately once second intervals forever or until power is removed. To increase the speed at which the LED blinks change both PAUSE time values from 1000 to something smaller.

```
;Wire LED to P0
Main
  High P0
  Pause 1000
  Low P0
  Pause 1000
Goto Main
```

---

## PAUSEUS

### Syntax

pauseus time

- **Time** - is a variable, constant or expression that specifies the number of half microseconds to wait.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

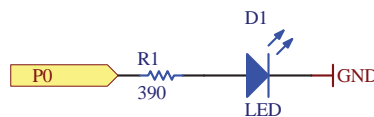
The PAUSEUS command is used to create a predetermined delay in a program. The amount of delay is specified in microseconds. The PAUSEUS command differs from the PAUSE command in that it uses half microsecond increments. There are 1000 microseconds in 1 millisecond. So a value of 2,000,000 would wait a full second. The PAUSEUS command is used when high resolution delays are required.

### Notes

1. PAUSEUS is fairly accurate. The Nano uses an internal clock which is less accurate than the external clocks used on the modules. Therefore the error rate would be higher with the Nano.

### Schematic

Connect a LED as shown below. You can substitute the resistor for anything from 330 ohms to 470 ohms.



### Example

The following program will blink an led at approximately once second intervals forever or until power is removed. To increase the speed at which the LED blinks change both PAUSEUS time values from 1000000 to something smaller.

```
;Wire LED to P0

Main
  High P0
  Pause 1000000
  Low P0
  Pause 1000000
Goto Main
```

---

## PAUSECLK

### Syntax

pauseclk time

- **Time** - is a variable, constant or expression that specifies the number of clock cycles to wait.

### Supported

- BA - Supported with timing differences.
- BAN - Supported with timing differences.
- BAP - Supported with timing differences.
- BAP40 - Supported with timing differences.

### Description

The PAUSECLK command is used to create a high resolution delay in a program. The delay is specified in clock cycles. The PAUSECLK command differs from the PAUSE and PAUSEUS commands in that it uses clock cycles. The clock cycle across processor lines are different so the same value on one processor applied to another one will change the delay. The PAUSECLK command can be used in generating precise timing signals.

### Notes

1. The Nano processors runs at 8Mhz internally. It takes 4 clock cycles to complete one instruction. So it runs at 1/4 clock cycles. One millisecond is equal to 2000 clock cycles on the Nano.
2. The BasicATOM processors runs at 20Mhz. It takes 4 clock cycles to complete one instruction. So it runs at 1/4 clock cycles. One millisecond is equal to 5000 clock cycles on the ATOM.
3. The ATOM Pro runs at 16Mhz. Where the BasicATOM Pro 40 runs at 20Mhz. One millisecond is equal to 16000 on the ATOM Pro and 20000 on the ATOM Pro 40.

### Example

The program below will function slightly differently depending on what processor you are using. Separate programs have been provided to demonstrate a one second timing on all 4 different processors. The program will print to the terminal window set at 9600 baud. To test the examples only load the portion that corresponds to the processor you are using.

```
;Nano pauseclk example

  Serout s_out,i9600,["Starting",13]
  Pauseclk 2000000
  Serout s_out,i9600,["One second later"]
  End

;Atom pauseclk example

  Serout s_out,i9600,["Starting",13]
  Pauseclk 5000000
  Serout s_out,i9600,["One second later"]
  End
```

```
;AtomPro pauseclk example  
    Serout s_out,i9600,["Starting",13]  
    Pauseclk 16000000  
    Serout s_out,i9600,["One second later"]  
    End  
  
;AtomPro40 pauseclk example  
    Serout s_out,i9600,["Starting",13]  
    Pauseclk 20000000  
    Serout s_out,i9600,["One second later"]  
    End
```

---

## PULSIN

### Syntax

pulsin pin, direction, {Tlabel, Timeout,} result

- **Pin** - is variable, constant or expression that specifies the pin to be used for the input pulse.
- **Direction** - is variable, constant or expression (0 or 1) that specifies the pulse direction start. If direction is set to 0 a high state will trigger the timer. If direction is set to a 1 a low state will trigger the timer.
- **Tlabel** - is an optional argument that specifies a label the program will jump to if a timeout occurs. If no Tlabel or Timeout value is specified the command will wait about 32ms for a pulse to occur. If no pulse occurs a result of 0 is returned and program execution will resume.
- **Timeout** - is variable, constant or expression that specifies the number of microseconds for a time out. Timeout is a 32-bit integer value (0 - 4,294,967,295). Timeout is measured in 0.5 microsecond intervals. A timeout of 1000 microseconds would be entered as 2000.
- **Result** - is a variable that stores the pulse duration in 0.5 microseconds increments. If the result variable specified is too small only the least significant bits will be stored. If no pulse is detected within the time out value the result will be set to 0. See the table below for the minimum time resolution of each processor.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Resolution

Processor	Time
BA	1.4us
BAN	3.5us
BAP	1.5us
BAP40	1.5us

### Description

PULSIN will measure the time from edge to edge of the pulse. It will start the count on a high to low transition or a low to high transition depending on what direction is set to. PULSIN measures the width of the pulse. PULSIN can be used to measure a servo signal or how long a button was press. PULSIN is a blocking command that will wait for 32ms until a pulse is detected. If no pulse is detected the command returns a 0 and continues program execution.

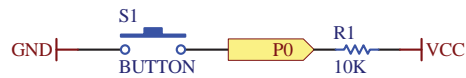
### Notes

1. Each processor has a differ minimum resolution of time. After the minimum resolution all time values are added by 0.5us increments.
2. The value stored in result is in 0.5us increments. If 1000 is returned this represents 500us.



Schematics

Connect a momentary switch as shown below. R1 can be a 10K or a 4.7K resistor.



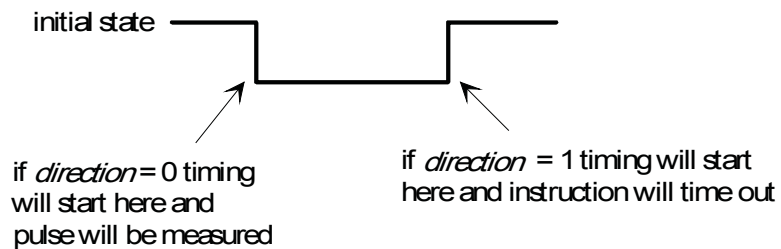
Examples

The example program will time how long a button is pressed, displaying the results to a terminal window at 9600 baud. The direction is set to 0 since our initial state will be high from the pull-up resistor.

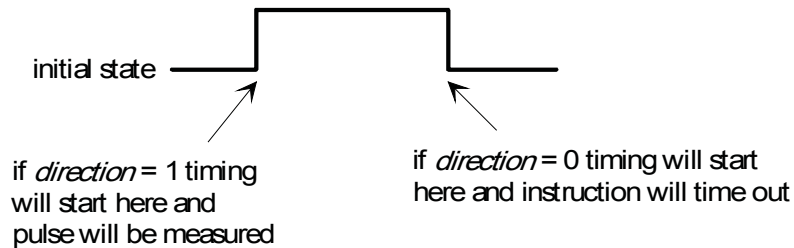
```
temp var long
main
  pulsln p40,1,main,2000000,temp
  serout s_out,i9600,[dec temp,13]
  pause 100
  goto main
```

The direction 0 or 1 is important. If it is not set correctly the results will not be accurate. If the initial state is missed PULSIN may not time any pulse. Or time a second pulse as one pulse skewing the results. The two pulses below illustrate this point.

Initial State 1 (Direction 0)



Initial State 0 (Direction 1)



---

## PULSOUT

### Syntax

pulsout pin, time

- **Pin** - is a variable, constant or expression that specifies the pin to read a pulse.
- **Time** - is a variable, constant or expression that specifies the duration of the pulse. Time is measured in 0.5 microseconds increments. The smallest unit of time differs for each processor family. See Time Units table.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Time Units

Processor	Time
BA	5us
BAN	12us
BAP	3us
BAP40	2.5us

### Description

PULSOUT toggles the pin state from high to low to equal one pulses. You can use the high or low commands to set the initial state of the pin, which controls the polarity of the pulse. The pin will remain in the last known state after the PULSOUT command has ran. This allows successive use of the PULSOUT command to produce pulses of the same polarity.

### Notes

1. Each processor has a differ minimum resolution of time. After the minimum resolution all time values are added by 0.5us increments.
2. Temperature will effect the pulse. The affect is greater the closer to the minimum resolution increments.

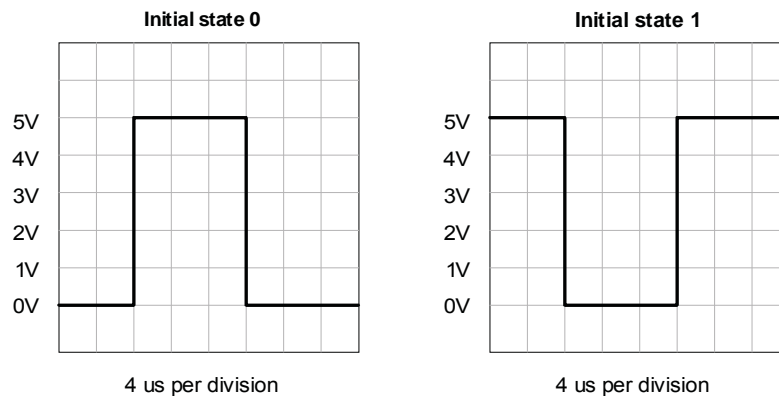
### Examples

The example program will produce a pulse to that shown below for an initial state of 0. If the “low p0” is changed to “high p0” the pulse will be similar to that shown for initial state of 1.

The pulse will vary slightly. This will be due to the temperature and other factors such as components values used to build the BasicATOM or BasicATOM Pro. The closer the desired pulse is to the minimum resolution the more noticeable the effect. Large values see no loss.

```
time var word
time=120 ; 0.5us increments, pulse width = 60
main
  low p0
  pulsout p0, time
goto main
```

### Pulse Initial states



---

## PWM

### Syntax

`pwm pin, period, duty, cycles`

- **Pin** - is a variable, constant or expression that specifies the pin to use.
- **Period** - is a variable, constant or expression that specifies the period of the pulse width signal in half microsecond increments.
- **Duty** - is a variable, constant or expression that specifies the period of the duty cycle pulse width signal in half microsecond increments.
- **Duration** - is a variable, constant or expression that specifies the number of pulses to output.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

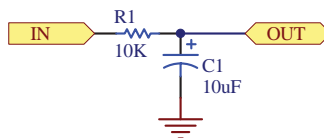
### Description

The PWM command generates a pulse width modulated signal. PWM has several use some of which are generating an analog voltage, DC motor control, servo control or generating a frequency.

To create an analog voltage the pin is switched from high to low. During the transitions leaving the pin high for a certain amount of time versus low and then averaging the output will cause a voltage change. If the pin was high (5V) 50% of the time and then low (0V) the other 50% the voltage output would be around 2.5V. By adjusting how long the pin is high or low we can control the output voltage with a simple RC circuit.

### Schematic

The schematic is a simple RC filter. The resistor and capacitors values can be changed to affect the final output.

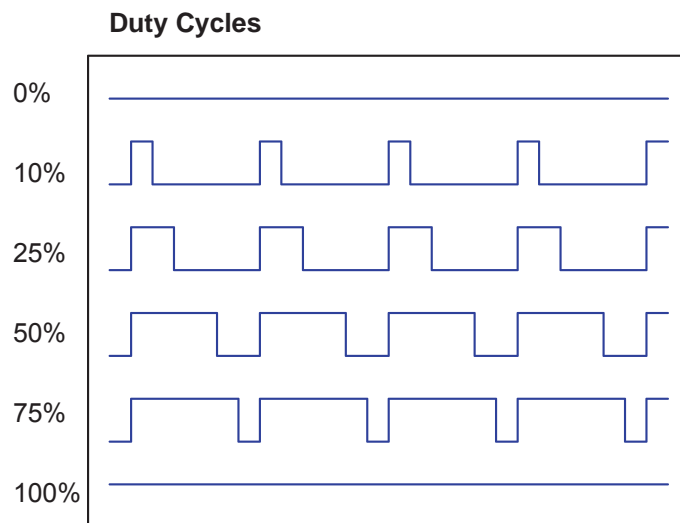


The code snippet will generate a 50% duty cycle. Which, with no load, will output 2.5V on the output side of our RC filter. Build the circuit, run this program and attach a volt meter probe to the output. Try adjusting the duty cycle to see the results.

```
pwm p0,1000,500,1000
```

### Duty Cycle

The following chart is what the signal would look like on an oscilloscope at specific duty cycles. This will give you an idea of what the duty cycles look like. You can see their on times. More on time would mean more charge for our RC circuit which would increase its output voltage. At 0% duty cycle the pin is always low. At 100% duty cycle the pin is always high.



---

## RCTIME

### Syntax

rctime pin,state,{tlabel,timeout,}result

- **Pin** - is a variable, constant or expression that specifies the pin to use.
- **State** - is a variable, constant or expression (1 or 0) that specifies the state which will end the timing period.
- **Tlabel** - is an optional argument that specifies a label the program will jump to if no transition occurs. If Tlabel and timeout are not specified, RCTIME will wait 32ms for a transition to occur. If no transitions occurs 0 is returned and program execution will resume.
- **Timeout** - is variable, constant or expression that specifies the number of microseconds for a time out. Timeout is measured in 0.5 microsecond intervals. A timeout of 1000 microseconds would be entered as 2000.
- **Result** - is a variable that stores the pulse duration in 0.5 microseconds increments. If the result variable specified is too small only the least significant bits will be stored. If no pulse is detected within the timeout value the result will be set to 0. See the table below for the minimum resolution of each processor.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Resolution

Processor	Time
BA	4.2us
BAN	10.5us
BAP	4.5us
BAP40	4.5us

### Description

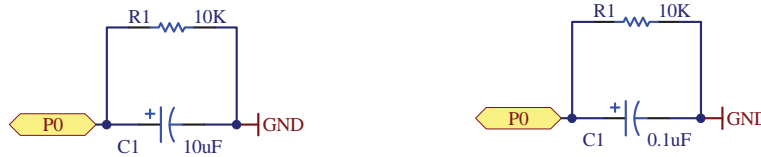
RCTIME is typically used to measure the charge and or discharge time of a resistor / capacitors circuit. It can also be used in precise triggered timing measurements like ultra sonic sensors. When using RCTIME with an R/C circuit it can measure the amount of time the capacitor takes to discharge. RCTIME is a blocking command and will wait for a transition to occur before continue program execution unless a timeout is set.

### Notes

1. RCTIME can only measure up to its maximum value, 4,294,967,295.
2. Each processor has a minimum measurement resolution. The example program is timed for the BAP.
3. Analog should be measured using the ADin command.

## Schematic

The schematic can use any value capacitor. Two different values are shown. The resistor must be a 10K in order for the sample program to work properly.



## Example

The example program will measure the discharge rate of a capacitor and determine its value. This is possible since we use a known resistor value of 10K. RCTIME can also be used to calculate a resistor or a potentiometer value by using a known capacitance and changing the formula used to calculate RC. The pin is first set to an output and high. The small pause allows the capacitor time to fully charge. The RCTIME command will then count how long it takes for the capacitor to discharge through a 10K resistor. When the capacitor voltage level falls below the TTL threshold of a logical 0, RCTIME will return the result.

```

capacitance var float
results var long

main
  pause 320 ;charge for 1 second
  rctime P0, 0, NoTime, 2000000, results ;result is time in .5us
  high P0
  capacitance = TOFLOAT(results) / 12800.0
  serout s_out, i9600, ["Time = ",dec results/2,"us ", real
capacitance\3,"uF", 13]
  goto main

NoTime
  serout s_out, i9600, ["RCTIME timed out!", 13]
  goto main

```

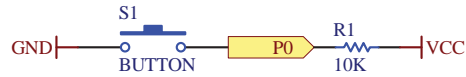
The program sets P0 to output, high state, and then waits one second for the capacitor to charge fully. The RCTIME command then changes P0 to an input and waits for a low state, which occurs when the capacitor discharges to about 0.8V. The time, in microseconds, is stored in the results variable.

The resistance is then calculated using the formula:  $\text{resistance} = \text{results} / (1.83 * \text{capacitance})$ . However, to accommodate the integer arithmetic, the 1.83 and capacitance are each multiplied by 100 (giving 183 and 2, respectively), so the numerator must be multiplied by 10000 to compensate.

If you see the RCTIME timeout error message, then the capacitor is too large for the program or you have something wired wrong.

## Example

The next example is a small game that can be played by connecting a button as show to P0. Its a reaction time game. See how fast you can get. Use the button schematic shown with the following example program to play the game.



```
temp var long
main
  if IN0=0 then
    serout s_out,i9600,["Please release the button to continue.",13]
  endif
  while IN0=0
    wend

  serout s_out,i9600,["Get ready",13]
  pause 1000

  serout s_out,i9600,["Get set",13]
  pause 1000

  serout s_out,i9600,["GO!!!",13]
  rctime p0,0,timeout,200000000,temp

  serout s_out,i9600,["Your time was:",real TOFLOAT(temp)/2000000.0\4," seconds",13]
  pause 2000

goto main

timeout
  serout s_out,i9600,["You waited too long. Starting over.",13]
  goto main
```



---

## READ

### Syntax

read address, databyte

- **Address** - is a byte sized variable or constant that specifies what address to read the on board eeprom from.
- **DataByte** - is a byte sized variable (0-255) which stores the data returned from the on board EEPROM.

### Supported

- BA - supported
- BAN - supported
- BAP - supported
- BAP40 - supported

### Description

All modules except the BasicATOM Pro 24 and BasicATOM Pro ONE come with built in eeprom. The READ / WRITE commands were created to access the built in memory. READ will read a single byte from the built in eeprom at the address specified in the command.

### Notes

1. The AtomPro24 does not have built in eeprom. An external 32kbit or larger eeprom can be added. Connect the eeprom SCL pin to P10 and the eeprom SDA pin to P11 with a 10K pull up resistor. The address pins on the eeprom (A0,A1 and A2) are tied to ground.
2. READ will only read one byte at a time. To read multiple locations a simple loop can be used or see READDMM.

### Example

The example program will write the string "Hello" starting at the first location to the on board eeprom. Next, it will read the eeprom locations 0 to 10 and print the contents to the terminal window.

```
index var byte
char var byte

write 0,"H"
write 1,"e"
write 2,"l"
write 3,"l"
write 4,"o"

for index = 0 to 10
  read index,char
  serout s_out,i9600,[char]
next
end
```

---

## READDM

### Syntax

```
readdm address, [{modifiers}databyte1, ..., {modifiers}databyteN]
```

- **Address** - the starting address of the onboard eeprom to read from.
- **DataByte** - is a list of variables with optional modifiers in which data read from the eeprom will be stored in.
- **Modifiers** - are used in the command syntax to format data. Refer to the modifier section of this manual. All input modifiers are supported.

### Supported

- BA - supported
- BAN - supported
- BAP - supported
- BAP40 - supported

### Description

The READDM differs from the READ command in that it auto increments to the next read address. It can be ran once and read multiple bytes from the onboard eeprom using only a starting address. READDM will continue to read based on how many variables are specified. Modifiers can be used to format the data or add functionality to the command. See table above.

### Notes

1. The AtomPro24 does not have built in eeprom. An external 32kbit or larger eeprom can be added. Connect the eeprom SCL pin to P10 and the eeprom SDA pin to P11 with a 10K pull up resistor. The address pins on the eeprom (A0,A1 and A2) are tied to ground.
2. The ReadDM command differs from the Read command in that it auto increments the address. It can read multiple bytes each time it is executed. The Read command can only read one byte each time it is executed.

### Example

The example program will first write the string "Hello" using *Writedm* to the on board eeprom. Next, it will read the eeprom locations starting from 0 using *Readdm*. It will continue to read starting at location 0 until all specified databytes are loaded. Run the program once, then comment out the *Writedm* command. The on board eeprom will retain all the data last written to it.

```
;Program will not work with BasicATOM Pro 24.

writedm 0,["Hello world"]

readloop
  index var byte
  string var byte(11)
  readdm index,[str string\11]
  serout s_out,i9600,[str string\11]
end
```

---

## REPEAT - UNTIL

### Syntax

```
repeat
    program statements
until condition
```

- **Statements** - any group of commands to be run inside the loop.
- **Condition** - can be a variable or expression

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The REPEAT - UNTIL loop executes commands nested inside of it until some condition is false. This is the opposite of DO - WHILE and WHILE - WEND. The condition can be any variable or expression and is tested every loop until true.

### Notes

1. In the programming world 0 is consider false. By default REPEAT - UNTIL will test this condition. If a stand alone variable is used for the test, the loop will continue until its value NOT equal to 0.
2. WHILE - WEND checks the condition first. If the condition is false the WHILE - WEND statements and all program code nested within them will not run.
3. You can nest multiple REPEAT - UNTIL commands within each other. You can nest DO - WHILE or WHILE - WEND loops within a REPEAT - UNTIL loop.

### Example

Connect to the following program with the terminal window set to 9600 baud. The program will start counting up from 0 to 100. Once index reaches a value of 101 the condition is no longer false. The greater than symbol > was used for the condition and 101 is now greater than 100 making the condition true. Since REPEAT - UNTIL loops while a statement is false the program exits.

```
;ALL - all_repeat_until.bas

Index var word

Main
    Index = 0

    Repeat
        index = index + 1

        ;lets print the value index
        serout s_out, i9600,[0, "Couting: ", dec index]
        pause 75

    Until index > 100 ;run until index is greater than 100

    serout s_out, i9600, [13,13, "Index = ", dec index]
    serout s_out, i9600, [13, "My condition is no longer false."]
    serout s_out, i9600, [13, "Index is now greater than 100"]

End
```

---

## RETURN

### Syntax

return {DataResult}

- **DataResult** - an optional value to return to the gosub statement. Can be an expression, constant or variable.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

RETURN removes from the stack the address stored by GOSUB, and resumes program execution on the line following the calling GOSUB command. A return value may optionally be returned to the calling GOSUB.

The GOSUB command will jump to a specified label. After executing the code at the jump label a RETURN command is then used to return the program to the next command after the last called GOSUB.

If you are familiar with some BASIC languages there is typically a limit to what is called nesting. This is where multiple GOSUB...RETURN statements are nested inside of each other. The only limit to MBasic is the available stack ram on the MCU that is in use.

GOSUB stores the address of the next command on the stack and jumps to the specified label. User specified arguments can be defined in the subroutine and a return value from the subroutine called can be stored in variable that is then loaded into the GOSUB DataResult argument.

### Notes

1. Subroutines should exit via the RETURN command, which clears the saved address from the stack and returns to the command following the GOSUB. Do not use BRANCH or GOTO when exiting a subroutine.
2. User defined arguments must match the number of arguments defined at the subroutine. If they do not match, a stack overflow or underflow will happen.
3. If subroutines returns a value the GOSUB is not required to use it or specify a return value variable

---

### Example

The below program will print the results to the terminal window at 9600 baud. The results will be 110. The GOSUB command has two arguments and includes a DataResult variable. The values 10 and 100 are passed to the subroutine MyAdd. The values are then loaded into the variables *arg1* and *arg2*. Since RETURN can have an expression the variables *arg1* and *arg2* are added and returned to the variable *result*.

```
Result var long
Main
  Gosub myadd[10,100],result
  Serout s_out,i9600,["Result =",dec result]
End

Arg1 var long
Arg2 var long

MyAdd [arg1,arg2]
Return arg1+arg2
```

See Also:  
GOSUB  
EXCEPTION

---

## REVERSE

### Syntax

reverse pin

- **Pin** - is any expression, constant or variable that specifies a pin to use.

### Supported

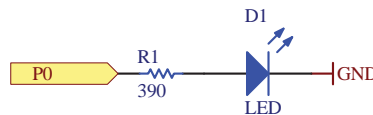
- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

REVERSE will switch the input/output direction of a pin. If pin was an input it would reverse to an output. If the pin was an output it would reverse to an input. A pins state determines whether it can be read or set. To read a pin and determine if it has a high (1) or low (0) signal applied to it, the pin must be set to an input. A pin can only be set to high (1) or low (0) if the pin is in an output state.

### Schematic

Connect a LED as shown below. You can substitute the resistor for anything from 330 ohms to 470 ohms.



### Example

The program turn the LED on using the High command which is an output command. Then the LED will turn off using the Reverse command which changes P0 from an output to an input. When P0 is an input the internal circuitry is disconnected and the pin can no longer supply the 5V need to illuminate the LED.

```
Loop
  High p0
  Pause 500
  Reverse p0
  Pause 500
Goto Loop
```

---

## SERIN

### Syntax

```
serin rxpin{\fpin},baudmode,{plabel,}{timeout,tlabel,}  
      [{modifiers} InputData1, ..., {modifiers} InputDataN]
```

- **Rxpin** - is a variable, constant or expression that specifies the receive data pin. This pin will switch to input mode and remain in that state at the end of the command.
- **\fpin** - is an optional variable, constant or expression that specifies a pin for flow control (the “\” is required). Flow control is used primarily with PC COM ports and conforms to RS232 serial port specifications. The flow control pin will switch to output mode and remain in that state at the end of the command.
- **Baudmode** - is a 32 bit variable, constant or expression that specifies how fast to receive data and in what format. Several predefined baud modes are available however custom modes are allowed. See the baud mode table for predefined modes.
- **Plabel** - is an optional label. The program will jump to plabel if parity is used and there was a parity error.
- **Timeout** - is an optional 32 bit variable, constant or expression that specifies the time to wait for incoming data. The value is measured in 0.5 microsecond increments. If data does not arrive within this time, the program will jump to tlabel.
- **Tlabel** - is an optional label the program will jump to if a *timeout* is specified and has occurred.
- **InputData** - is a list of variables or constants that tell SERIN what to do with incoming data. The incoming data can be formatted with optional input modifiers. See the Modifier section of this manual.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The SERIN command is designed to work with the RS232 serial specifications. The command is bit banded, which means they are created in software running on the processor. When SERIN is ran the processor can not do anything but receive data.

RS232 is a simple communication format supported by the PC and is referred to as a COM port. The COM port can send and receive data simultaneously with each being a separate circuit that works in only one direction. Each circuit can function independently to transmit and received data at the same time. This means the COM port is full duplex. The SERIN command is not capable of full duplex since it is software based.

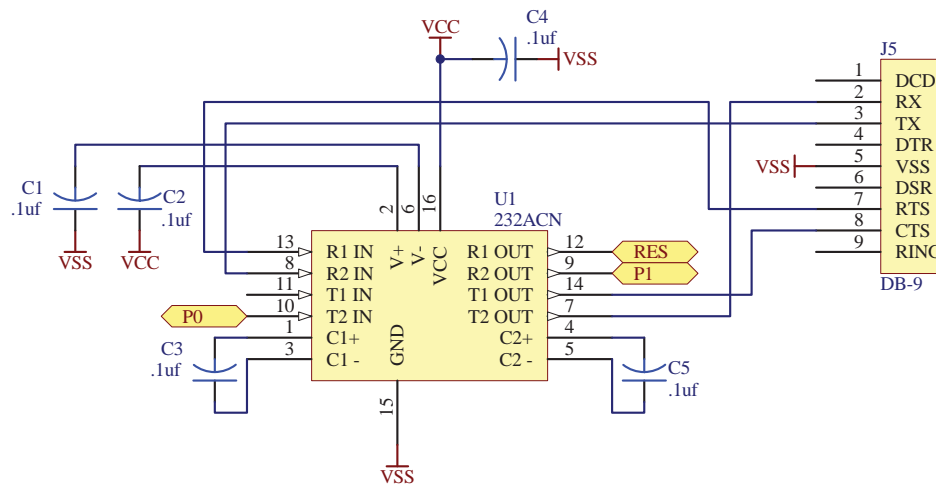
A 12 volt system was develop to work over long wire lengths and electrically noisy environments. The low state (0) is defined by +3 to +12 volts while a high state (1) is defined by a -3 to -12 volts. Modern computer equipment ignores the negative level and accepts a zero voltage level as a 1. This means circuits powered by 5V are capable of driving RS232 circuits directly, however, the overall range that the RS232 signal may be transmitted and received is dramatically reduced.

All of the Basic Micro modules have specific pins meant to interface directly to a RS232 COM port on any device. These pins are labeled SIN / SOUT and have a circuit to handle the -12V and +12V signals. The SERIN / SEROUT commands will work with any pin. However connecting a standard I/O pin directly to a 12V circuit would damage the I/O.

The Nano and BasicATOM Chips require a translation circuit like the one shown below. This circuit uses a standard IC that is made to interface to RS232 COM ports. Mostly commonly a PC COM port. Most of the development boards meant for these chips will either have an RS232 translation circuit like the one below or will have a USB translation circuit. The Nano USB programmer can also be used as a simple USB translation circuit to any of the pins for communication.

Schematic

The schematic is a standard circuit for RS232 interface. It uses a MAX232A version. This version requires only .1uf caps. A cheaper alternate to the MAX232A is a HIN232ACN. The RES pin should be connected to the reset (RES) pin of the micro. The T2IN connected to P0 will be the receive (RX) pin. The R2OUT pin connected to P1 is the transmit (TX) pin. The circuit below or one like it should be used when interfacing generic I/O pins to a PC COM port.



Baud Modes

The baud modes define how SERIN / SEROUT operate. There are two parts to a predefined baud mode. The first part indicates the mode in which to receive or send the data. The second part defines the rate (speed) the data is sent or received. An example would be n9600. The N being the mode and the 9600 being the speed.

The two main modes supported are non inverted (N) and inverted (I). Inverted (I) is typically required when a translation circuit is used since the signal will be inverted. When the SIN / SOUT pins are used inverted will typically be used. If a standard I/O pin is used with no translation circuit non inverted (N) is typically used. Inverted or non inverted modes default to 8 bits, no parity bit and 1 stop bit. This means 8 bits of data will be sent or received without parity and with 1 stop bit which indicates the end of the 8 bits.



### Parity

Parity is used to determine if there was a communication error. If parity is set using a predefined mode like IE (Inverted, even parity, 7 data bits) then the *plabel* is required. If an error is detected the command will exit and jump to *plabel*. The parity system is very rudimentary but in some situations is a better than nothing approach. The parity works by truncating the 8 data bits to 7 bits (0 to 127 for printable text characters). Even parity simply looks for an even amount of 1s received in the 8 bits.

Since the data is truncated to 7 bits the 8th bit is used to ensure there is always an even number of 1s. If the value being sent has three 1s in it, then the 8th bit is set to a 1 to make the total number of ones four which is even. If there were only two 1s in the value being sent then the 8th bit would be 0 since two 1s would be an even amount.

### Supported Modes

Mode	Description
I	Inverted.
N	Non Inverted.
IE	Inverted, Even Parity.
NE	Non Inverted, Even Parity.
IO	Inverted, Open Drain.
NO	Non Inverted, Open Drain.
IEO	Inverted, Even Parity, Open Drain.
NEO	Non Inverted, Even Parity, Open Drain.

### Baud Rates

The rate determines how fast the data is transmitted. A fairly standard speed is 9600. This means 9,600 bits per second will be sent or received. There are several predefined speeds. However custom ones can be created. Custom speeds allow SERIN / SEROUT to work with just about any RS232 devices. On occasion you do find devices you may want to interface with that support odd ball speeds. MBasic can easily support these by defining your own baud modes.

There are some limits with the SERIN / SEROUT commands. There is a lot going on when SERIN is ran, so the max speed at which they can receive or send serial data is limited. The Nano is limited to a maximum speed of 38400. The BasicATOM Pro can handle speeds over 115200.

The predefined modes are listed below. They are used in combination with the predefined modes. An example would be n9600, i38400, ieo1200 and so on.

### Predefined Baud Rates

Predefined Rate	Predefined Rate	Predefined Rate
300	16800	115200
600	19200	
1200	21600	
2400	24000	
4800	28800	
7200	31200	
9600	33600	
12000	38400	
14400	57600	

### Limitations

The Nano maximum baud mode is 38400. The BasicATOM is limited to 57600 and the BasicATOM Pro support over 115200. These are standard PC COM port baud rates. Since the SERIN command is software based it will not exit unless the data it is waiting for is received or the *timeout* option is specified. The *timeout* option will wait the time specified and then jump to the timeout label if nothing has been received.

### Modifiers

The SERIN command will store the ASCII value of a character received. If a capital “A” is received a value of 65 is stored. Since the value is stored as binary it can be read back in decimal, hexadecimal or binary. Any character received is store as the ASCII equivalent value. Characters are case sensitive. “A” will have a different value from “a”. See the ASCII table in this manual. In some cases you may want to send or receive hexadecimal values or raw binary. Modifiers allow you to format the data.

### Input Modifiers

Name	Description
DEC	Decimal.
SDEC	Signed decimal.
HEX	Hexadecimal.
SHEX	Signed hexadecimal.
IHEX	Indicated (\$) hexadecimal.
ISHEX	Signed and indicated (\$) hexadecimal.
BIN	Binary.
SBIN	Signed binary.
IBIN	Indicated (%) binary.
ISBIN	Indicated (%) and signed binary.
REAL	Floating point number with decimal point.
STR	Read specified amount of characters and store in an array.
SKIP	Skip specified amount of characters.
WAIT	Wait for specified amount of characters.
WAITSTR	Compares specified amount of characters to array.

The DEC/Hex/BIN modifiers will store all incoming numeric values as their binary equivalents. However the DEC/HEX/BIN modifiers will ignore all preceding non-numeral character data and the commands will exit on the first non-numeral character received after it received numeral characters. The example below will read in one byte and store it in MyData. If a “1A” is received the command stores a “1” and exits when it receives the “A” since we only were looking for a number. If an “AA” was sent it would ignore both and continue to wait for numeral characters.

```
MyData var byte
serin p0, i9600, [DEC MyData]
```

In some case you may want to wait for a specific string of text. This is done with the WAIT modifier. This example will wait until the exact string of characters, “TEXT”, is received. The quotes designate ASCII characters. One or more characters can be specified.

```
MyData var byte
serin p0, i9600, [WAIT ("TEXT")]
```

In other cases you may want to ignore most of the incoming data and receive only a specific value in a sequence. The SKIP modifier will allow you to ignore as much incoming data as specified before reading a value. The example will skip the first 3 values received and exit on the 4th value. If the word "TEXT" is sent the character "T" would be stored in MyData.

```
MyData var byte
serin p0, i9600, [SKIP 3 MyData]
```

For an example of how all input modifiers work see the modifier section of this manual. The modifiers work the same regardless of what command they are used in.

#### Notes

1. The Nano maximum baud rate is 38400.
2. BasicATOM maximum is 57600 and the BasicATOM Pro maximum baud rate is over 115200.
3. When SERIN is running the processor can not do anything else. If SERIN is not running and data is received it will be ignored/lost.

#### Examples

This example is meant to work with a Basic Micro development board using the SIN / SOUT pins. This setup requires no wiring it use the default configuration. Simply power up the board and plug in the USB or Serial port.

The following program will return the ASCII value in decimal and hexadecimal of any character you send to it. Once you have the program loaded connect to it with the terminal window built into Studio. Connect using 9600 baud. Type the capital letter "A" in and you should receive a hex value of 0x41 and a decimal value of 65.

```
temp var byte
main
  serin s_in,i9600,[temp]
  serout s_out,i9600,[0,temp," = Hex:0x",hex2 temp\2," Dec:",dec temp]
  goto main
```

---

## SEROUT

### Syntax

```
serout txpin{fpin},baudmode,{pace,}{timeout,tlabel,}  
      [{modifiers} OutputData1, ..., {modifiers} OutputDataN]
```

- **Txpin** - is a variable, constant or expression that specifies the transmit data pin. This pin will switch to output mode and remain in that state at the end of the command.
- **Fpin** - is an optional variable, constant or expression that specifies a pin for flow control (the “\” is required). Flow control is used primarily with PC COM ports and conforms to RS232 serial port specifications. The flow control pin will switch to input mode and remain in that state at the end of the command.
- **Baudmode** - is a 32 bit variable, constant or expression that specifies how fast to receive data and in what format. Several predefined baud modes are available however custom modes are allowed. See the baud mode table for predefined modes.
- **Pace** - is an optional variable, constant or expression that specifies the length of an added pause in between transmitting bytes. Pace can not be used with Timeout and Fpin. Pace is expressed in .5us increments.
- **Timeout** - is an optional 32 bit variable, constant or expression that specifies the time to wait for Fpin before jumping to Tlabel. The value is measured in 0.5 microsecond increments. If Fpin does not grant permission to send within this time, the program will jump to tlabel.
- **Tlabel** - is an optional label the program will jump to if Fpin and *timeout* are given and the amount of time specified has elapsed.
- **OutputData** - is a list of variables or constants that tell SERIN what to do with incoming data. The incoming data can be formatted with modifiers. See the Modifier section of this manual.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The SEROUT command is designed to work with the RS232 serial specifications. The command is bit banded, which means they are created in software running on the processor. When SEROUT is ran the processor can not do anything but transmit data.

RS232 is a simple communication format supported by the PC and is referred to as a COM port. The COM port can send and receive data simultaneously with each being a separate circuit that works in only one direction. Each circuit can function independently to transmit and received data at the same time. This means the COM port is full duplex. The SEROUT command is not capable of full duplex since it is software based.

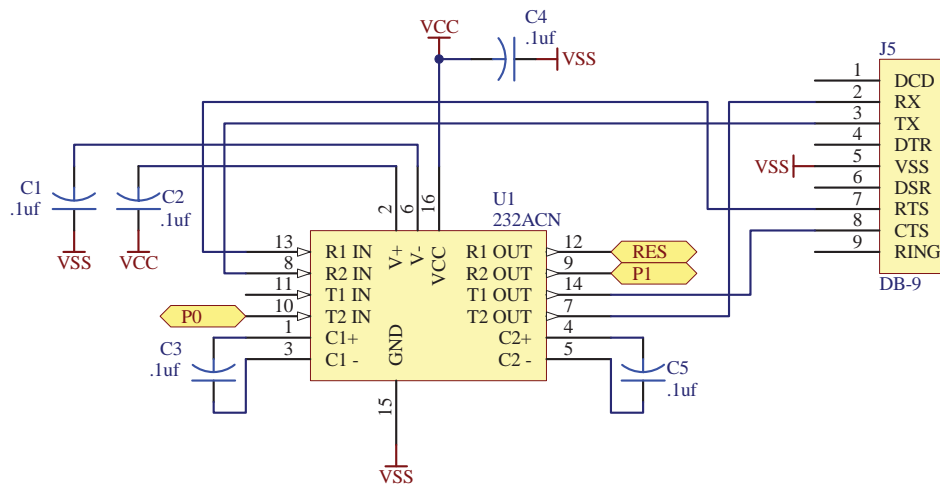
A 12 volt system was develop work over long wire lengths and electrically noisy environments. The low state (0) is defined by +3 to +12 volts while a high state (1) is defined by a -3 to -12 volts. Modern computer equipment ignores the negative level and accepts a zero voltage level as a 1. This means circuits powered by 5V are capable of driving RS232 circuits directly, however, the overall range that the RS232 signal may be transmitted and received is dramatically reduced.

All of the Basic Micro modules have specific pins meant to interface directly to a RS232 COM port on any device. These pins are labeled SIN / SOUT and have a circuit to handle the -12V and +12V signals. The SERIN / SEROUT commands will work with any pin. However connecting a standard I/O pin directly to a 12V circuit would damage the I/O.

The Nano and BasicATOM Chips require a translation circuit like the one shown below. This circuit uses a standard IC that is made to interface to RS232 COM ports. Mostly commonly a PC COM port. Most of the development boards meant for these chips will either have an RS232 translation circuit like the one below or will have a USB translation circuit. The Nano USB programmer can also be used as a simple USB translation circuit to any of the pins for communication.

Schematic

The schematic is a standard circuit for RS232 interface. It uses a MAX232A version. This version requires only .1uf caps. A cheaper alternate to the MAX232A is a HIN232ACN. The RES pin should be connected to the reset (RES) pin of the micro. The T2IN connected to P0 will be the receive (RX) pin. The R2OUT pin connected to P1 is the transmit (TX) pin. The circuit below or one like it should be used when interfacing generic I/O pins to a PC COM port.



Baud Modes

The baud modes define how SERIN / SEROUT operate. There are two parts to a predefined baud mode. The first part indicates the mode in which to receive or send the data. The second part defines the rate (speed) the data is sent or received. An example would be n9600. The N being the mode and the 9600 being the speed.

The two main modes supported are non inverted (N) and inverted (I). Inverted (I) is typically required when a translation circuit is used since the signal will be inverted. When the SIN / SOUT pins are used inverted will typically be used. If a standard I/O pin is used with no translation circuit non inverted (N) is typically used. Inverted or non inverted modes default to 8 bits, no parity bit and 1 stop bit. This means 8 bits of data will be sent or received without parity and with 1 stop bit which indicates the end of the 8 bits.

### Parity

Parity is used to determine if there was a communication error. If parity is set using a predefined mode like IE (Inverted, even parity, 7 data bits) then the *plabel* is required. If an error is detected the command will exit and jump to *plabel*. The parity system is very rudimentary but in some situations is a better than nothing approach. The parity works by truncating the 8 data bits to 7 bits (0 to 127 for printable text characters). Even parity simply looks for an even amount of 1s received in the 8 bits.

Since the data is truncated to 7 bits the 8th bit is used to ensure there is always an even number of 1s. If the value being sent has three 1s in it, then the 8th bit is set to a 1 to make the total number of ones four which is even. If there were only two 1s in the value being sent then the 8th bit would be 0 since two 1s would be an even amount.

### Supported Modes

Mode	Description
I	Inverted.
N	Non Inverted.
IE	Inverted, Even Parity.
NE	Non Inverted, Even Parity.
IO	Inverted, Open Drain.
NO	Non Inverted, Open Drain.
IEO	Inverted, Even Parity, Open Drain.
NEO	Non Inverted, Even Parity, Open Drain.

### Baud Rates

The rate determines how fast the data is transmitted. A fairly standard speed is 9600. This means 9,600 bits per second will be sent or received. There are several predefined speeds. However custom ones can be created. Custom speeds allow SERIN / SEROUT to work with just about any RS232 devices. On occasion you do find devices you may want to interface with that support odd ball speeds. MBasic can easily support these by defining your own baud modes.

There are some limits with the SERIN / SEROUT commands. There is a lot going on when SERIN is ran, so the max speed at which they can receive or send serial data is limited. The Nano is limited to a maximum speed of 38400. The BasicATOM Pro can handle over 115200.

The predefined modes are listed below. They are used in combination with the predefined modes. An example would be n9600, i38400, ieo1200 and so on.

### Predefined Baud Rates

Predefined Rate	Predefined Rate	Predefined Rate
300	16800	115200
600	19200	
1200	21600	
2400	24000	
4800	28800	
7200	31200	
9600	33600	
12000	38400	
14400	57600	

### Limitations

The Nano maximum baud mode is 38400. The BasicATOM maximum is 57600 and the BasicATOM Pro supports over 115200. These are standard PC COM port baud rates.

### Modifiers

In some cases you may want to send or receive hexadecimal values or raw binary. Modifiers allow you to format the data.

### Output Modifiers

Name	Description
DEC	Decimal.
SDEC	Signed decimal.
HEX	Hexadecimal.
SHEX	Signed hexadecimal.
IHEX	Indicated (\$) hexadecimal.
ISHEX	Signed and indicated (\$) hexadecimal.
BIN	Binary.
SBIN	Signed binary.
IBIN	Indicated (%) binary.
ISBIN	Indicated (%) and signed binary.
REP	Repeat character <i>n</i> times.
REAL	Floating point number with decimal point.
STR	Read specified amount of characters from an array.

### Terminal Window

Basic Micro Studio has 4 built in terminal windows. The windows support formatting commands. These commands control the cursor positioning and other functions. The example shown will clear the screen and move the cursor to the home position before the word "Test" is printed.

```
serout s_out, i9600, [0,"Test"]
```

### Terminal Window Commands

Decimal Character	Command	Description
0	CLS	Clears the screen.
1	HOME	Moves cursor home.
3	MOVE LEFT	Moves cursor left.
4	MOVE RIGHT	Moves cursor right.
5	MOVE UP	Moves cursor up.
6	MOVE DOWN	Moves cursor down.
7	BELL	Make sound on PC.
8	BACK SPACE	Moves cursor back and delete.
9	HANDLE TAB	Add a standard tab.
10	LINEFEED	Move cursor to next line.
11	CLEAR RIGHT	Clear anything to the right of the cursor.
12	CLEAR DOWN	Clear anything below the cursor.
13	CARRIAGE RETURN	Move to the next line.

---

### Notes

1. The Nano maximum baud rate is 38400.
2. The BasicATOM maximum is 57600 and the BasicATOM Pro maximum baud rate is over 115200.
3. When SEROUT is running the processor can not do anything else.

### Examples

This example is meant to work with a Basic Micro development board using the SIN / SOUT pins. This setup requires no wiring it use the default configuration. Simply power up the board and plug in the USB or Serial port.

The following program will return the ASCII value in decimal and hexadecimal of any character you send to it. Once you have the program loaded connect to it with the terminal window built into Studio. Connect using 9600 baud. Type the capital letter "A" in and you should receive a hex value of 0x41 and a decimal value of 65.

```
temp var byte
main
  serin s_in,i9600,[temp]
  serout s_out,i9600,[0,temp," = Hex:0x",hex2 temp\2," Dec:",dec temp]
goto main
```

### Examples

This example shows how to use the string modifier. We setup an array, load it with the text string "Hello World". The SEROUT command with the string modifier will increment through the variable array and send the character from each position in the array. Connect to this program with the terminal window at 9600 baud to see the results.

```
string var byte(20)
string = "Hello World",0
main
  serout s_out,i9600,[str string\20\0,13]
goto main
```



---

## SERVO

### Syntax

```
servo pin,pos{,repeat}
```

- **Pin** - is a variable, constant or expression that indicates what processor pin to use.
- **Pos** - is a variable, constant or expression that indicates servo position. The value used must be signed to indicate left or right swing. Valid real world ranges are +/- 3000 with 0 being center. The ranges will vary from servo to servo.
- **Repeat** - is a variable, constant or expression that is optional and specifies how many times a pulse is repeated before exiting the command. A repeat value of 0 is a special case and will only wait until the high side of the pulse is finished before resuming normal program execution.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The SERVO commands will generate pulses to control a standard R/C hobby servo. The range is -3000 to +3000. Most servos will run from -2200 to +2200. Center is typically 0. The command is a blocking command and will wait until the pulse is finished.

The repeat argument can be used to drive up to 6 servos by setting it to 0. The time it takes to update 6 SERVO commands in a row with repeat = 0 is about 18ms. Leaving time for the program to perform additional tasks before more servo pulses need to be sent.

### Notes

1. Setting the repeat argument to 0 means this servo command will output 1 pulse and continue on to the next command without waiting for the low part of the pulse to finish
2. Setting the repeat argument to 1 or more will cause the SERVO command to output the pulse the number of times value is set to. Repeat = 20 would generate 20 pulses and take 400ms.

### Examples

The program demonstrates how to update more than one servo at a time using the optional repeat argument. Setting the first repeat to 0 will allow the second SERVO command to run while the first would normally be waiting for the low side of its pulse to finish. Each servo command takes approximately 3ms when repeat is 0 so we need a pause after them to add up to approximately 20ms total.

```
pos var sword
main
  for pos = -500 to 500
    servo p0,pos,0
    servo p1,-pos,0
    pause 14
  next
  for pos = -500 to 500
    servo p0,-pos,0
    servo p1,pos,0
    pause 14
  next
  goto main
```

## SHIFTIN

### Syntax

shiftin dpin,cpin,mode,[InputData1{bits}, ..., InputDataN{bits}]

- **Dpin** - is a variable, constant or expression that specifies the Data input pin. This pin will switch to an input pin.
- **Cpin** - is a variable, constant or expression that specifies the Clock output pin. This pin will switch to an output.
- **Mode** - is a value (0 to 7) or a predefined constant that sets the incoming data conditions.
- **InputData** - is a variable where incoming data is stored. The only limit to the amount of variables that can be used is the available ram on the processor being used.
- **Bits** - is an optional entry (1 – 32) defining the number of bits that will be written to each variable in the list. The default is 8 bits. Some devices will send back more than 8 bits or interleave from 8 to 16 bits. The optional bits argument allows the number of bits to be defined that will be stored in each variable in the list.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The SHIFTIN and SHIFTOUT commands can send or received data from synchronous serial devices like an SPI device. Synchronous serial differs from asynchronous (SERIN / SEROUT) in that there is a clock line. This clock line is used to time the data bits sent or received. There are several settings which make the SHIFTIN / SHIFTOUT commands flexible enough to work with most any synchronous serial device.

The following tables provides all supported modes and maximum speed in kilo bits per second that each processor family can handle. There are 8 modes. The first 4 modes (0-3) are standard modes. The second set of 4 modes (4-7) are basically duplicates of the first 4 modes but for high speed operation.

### Modes

Mode	Value	Description
MSBPRES	0	Sample bits msb first, before clock pulse.
LSBPRES	1	Sample bits lsb first, before clock pulse.
MSBPOST	2	Sample bits msb first, after clock pulse.
LSBPOST	3	Sample bits lsb first, after a clock pulse.
FASTMSBPRES	4	Faster sampling, msb first, before clock pulse
FASTLSBPRES	5	Faster sampling, lsb first, before clock pulse
FASTMSBPOST	6	Faster sampling, msb first, after clock pulse
FASTLSBPOST	7	Faster sampling, lsb first, after clock pulse

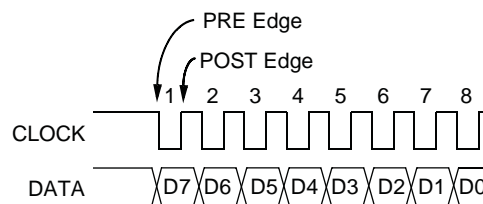
Speeds

Processor	Mode 0-4	Mode 5-7
BA	50 Kbps	100 Kbps
BAN	20 Kbps	40 Kbps
BAP	100 Kbps	380 Kbps
BAP40	100 Kbps	400 Kbps
ARC32	100 Kbps	400 Kbps

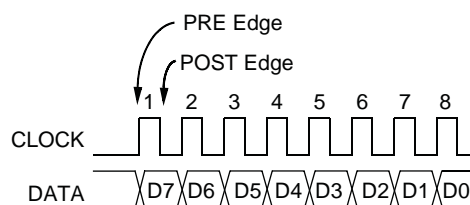
All SPI devices fall into 4 main transmission modes. They are specific to the timings of the writing and reading of data. The write and read of data at the first edge of the clock pulse is PRE. The second edge of the clock pulse is POST. The PRE and POST edge of the clock will change depending on whether the clock starts high(1) or low(0).

Most SPI device data sheets will indicate if they are PRE, POST, MSB or LSB. However in some cases the data sheet might not be clear. So first we must determine if the data is read PRE or POST clock edge. In almost all cases data can be written and read on the rising and falling edge of the clock signal. Looking at the timing diagrams below the clock can start from one of two states high(1) or low(0). The diagrams below illustrate the PRE and POST edge based on a high(1) or low(0) clock start.

**High(1) Clock Start**



**Low(0) Clock Start**



The diagrams for clock start show the data in both cases being written on the POST edge of the clock. The next thing to determine is the bit order. Which from the diagrams starts with bit 7 (D7) and ends with bit 0 (D0), this is MSB (Most Significant Bit First). If the byte order were reversed and bit 0 (D0) was first with bit 7 (D7) last that would be LSB (Least Significant Bit First)

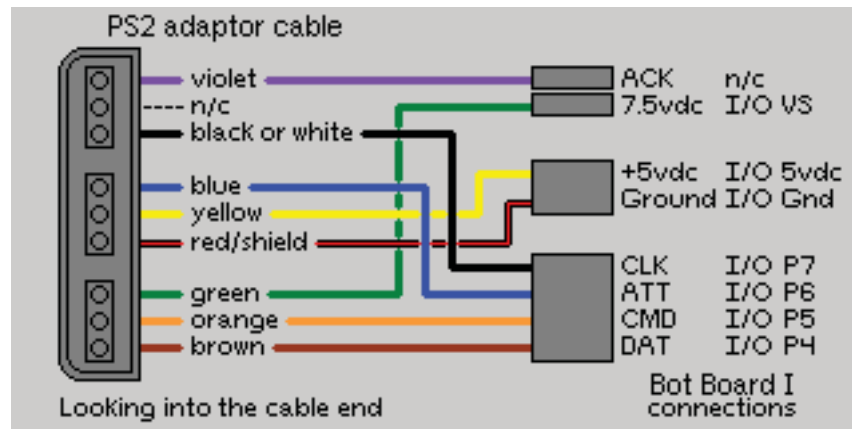
Notes

1. SHIFTIN and SHIFTOUT only work as a master device, the clock pin will always be an output for both SHIFTIN and SHIFTOUT.

Schematics

The wiring diagram below illustrates how to connect a PS2 controller for the given example program. The connector shown is a standard PS2 cable that can be made or purchased from Lynxmotion. The DAT pin requires a 10K pull-up resistor to work properly.

**PS2 Wiring Diagram**



Examples

The example program on the following pages can be copied and pasted into Studio. However you must watch for the header and page number from being copied with it. Copy the first page into a new bas file then copy and past the second page into the same file.

After the program is loaded connect to it with the terminal window at 38400 baud. The program will display on the screen what control is being pressed from the PS2 controller and how hard. All buttons on most PS2 controllers have analog sensors that sense how hard a button is being pressed. These result will be displayed to the terminal window. Connect the PS2 controller as shown in the above wiring diagram. Don't forget the 10K pull-up resistor on the DAT pin.

```

;[PS2 Controller Constants]
DAT                con P0                ;PS2 Controller DAT (Brown)
CMD                con P1                ;PS2 controller CMD (Orange)
ATT                con P2                ;PS2 Controller SEL (Blue)
CLK                con P3                ;PS2 Controller CLK (White)

PadMode            con $79
;-----

;[Ps2 Controller Variables]
DualShock          var Byte(18)
LastButton         var Byte(2)
DS2Mode            var Byte
PS2Index           var byte
;-----

main
  gosub ControlInput[dualshock(16),dualshock(17)]

  serout s_out,i38400,[0," DPad-Left: ",bin1 dualshock(0).bit7\1," ",dec dualshock(7),13]
  serout s_out,i38400,[ " DPad-Down: ",bin1 dualshock(0).bit6\1," ",dec dualshock(9),13]
  serout s_out,i38400,[ " DPad-Right: ",bin1 dualshock(0).bit5\1," ",dec dualshock(6),13]
  serout s_out,i38400,[ " DPad-Up: ",bin1 dualshock(0).bit4\1," ",dec dualshock(8),13]

  serout s_out,i38400,[ "Right XAxis: ",sdec 128-dualshock(2),13]
  serout s_out,i38400,[ "Right YAxis: ",sdec 128-dualshock(3),13]
  serout s_out,i38400,[ " Left XAxis: ",sdec 128-dualshock(4),13]
  serout s_out,i38400,[ " Left YAxis: ",sdec 128-dualshock(5),13]

  serout s_out,i38400,[ "      Start: ",bin1 dualshock(0).bit3\1,13]
  serout s_out,i38400,[ " RAxis(R3): ",bin1 dualshock(0).bit2\1,13]
  serout s_out,i38400,[ " LAxis(L3): ",bin1 dualshock(0).bit1\1,13]
  serout s_out,i38400,[ "      Select: ",bin1 dualshock(0).bit0\1,13]

  serout s_out,i38400,[ "      Square: ",bin1 dualshock(1).bit7\1," ",dec dualshock(13),13]
  serout s_out,i38400,[ "      Cross: ",bin1 dualshock(1).bit6\1," ",dec dualshock(12),13]
  serout s_out,i38400,[ "      Circle: ",bin1 dualshock(1).bit5\1," ",dec dualshock(11),13]
  serout s_out,i38400,[ "      Triangle: ",bin1 dualshock(1).bit4\1," ",dec dualshock(10),13]

  serout s_out,i38400,[ "      R1: ",bin1 dualshock(1).bit3\1," ",dec dualshock(15),13]
  serout s_out,i38400,[ "      L1: ",bin1 dualshock(1).bit2\1," ",dec dualshock(14),13]
  serout s_out,i38400,[ "      R2: ",bin1 dualshock(1).bit1\1," ",dec dualshock(17),13]
  serout s_out,i38400,[ "      L2: ",bin1 dualshock(1).bit0\1," ",dec dualshock(16),13]

  pause 100

  goto main

ps2_motorww var byte
ps2_motoryy var byte
ControlInput [ps2_motorww,ps2_motoryy]
  high CLK
ControlInput_Retry
  LastButton(0) = DualShock(1)
  LastButton(1) = DualShock(2)
  low ATT
  shiftout CMD,CLK,FASTLSBPRES,[$1\8]
  shiftin DAT,CLK,FASTLSBPOST,[DS2Mode\8]
  high ATT
  pause 1

  if DS2Mode <> PadMode THEN
    low ATT
    shiftout CMD,CLK,FASTLSBPRES,[$1\8,$43\8,$0\8,$1\8,$0\8] ;enter config mode
    high ATT
    pause 1

    low ATT
    shiftout CMD,CLK,FASTLSBPRES,[$01\8,$44\8,$00\8,$01\8,$03\8,$00\8,$00\8,$00\8,$00\8]

```

```

;set and lock analog mode
    high ATT
    pause 1

    low ATT
    shiftout CMD,CLK,FASTLSBPRE,[$01\8,$4D\8,$00\8,$00\8,$01\8,$FF\8,$FF\8,$FF\8,$FF\8]
;Enable Vibration motors
    high ATT
    pause 1

    low ATT
    shiftout CMD,CLK,FASTLSBPRE,[$01\8,$4F\8,$00\8,$FF\8,$FF\8,$03\8,$00\8,$00\8,$00\8]
;set 18byte reply
    high ATT
    pause 1

    low ATT
    shiftout CMD,CLK,FASTLSBPRE,[$01\8,$43\8,$00\8,$00\8,$5A\8,$5A\8,$5A\8,$5A\8,$5A\8]
;exit config mode
    high ATT
    pause 1

    DualShock(1) = 255,255

    goto ControlInput_Retry
else
    low ATT
    shiftout CMD,CLK,FASTLSBPRE,[$1\8,$42\8,$00\8]
    shiftin DAT,CLK,FASTLSBPOST,[DualShock(0)\8,DualShock(1)\8]
    high ATT
    low ATT
    shiftout CMD,CLK,FASTLSBPRE,[$1\8,$42\8,$00\8,ps2_motorww\8,ps2_motoryy\8]
    shiftin DAT,CLK,FASTLSBPOST,[DualShock(2)\8,DualShock(3)\8,DualShock(4)\8,
DualShock(5)\8, |
                                DualShock(6)\8,DualShock(7)\8,DualShock(8)\8,DualShock(9)\8, |
                                DualShock(10)\8,DualShock(11)\8,DualShock(12)\8,DualShock(13)\8, |
                                DualShock(14)\8,DualShock(15)\8,DualShock(16)\8,DualShock(17)\8]
    high ATT
endif
return
;-----

```

## SHIFTOUT

### Syntax

shiftout dpin,cpin,mode,[OutputData1{\bits}, ..., OutputDataN{\bits}]

- **Dpin** - is a variable, constant or expression that specifies the Data input pin. This pin will switch to an input pin.
- **Cpin** - is a variable, constant or expression that specifies the Clock output pin. This pin will switch to an output.
- **Mode** - is a value (0 to 7) or a predefined constant that sets the incoming data conditions.
- **OutputData** - is a variable, constant or expression that specifies the data to be sent. The only limit to the amount of data is the available ram on the processor being used.
- **Bits** - is an optional entry (1 – 32) defining the number of bits that will be sent.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The SHIFTIN and SHIFTOUT commands can send or received data from synchronous serial devices like an SPI device. Synchronous serial differs from asynchronous (SERIN / SEROUT) in that there is a clock line. This clock line is used to time the data bits sent or received. There are several settings which make the SHIFTIN / SHIFTOUT commands flexible enough to work with most any synchronous serial device.

The following tables provides all supported modes and maximum speed in kilo bits per second that each processor family can handle. There are 8 modes. The first 4 modes (0-3) are standard modes. The second set of 4 modes (4-7) are basically duplicates of the first 4 modes but for high speed operation.

### Modes

Mode	Value	Description
MSBPRES	0	Sample bits msb first, before clock pulse.
LSBPRES	1	Sample bits lsb first, before clock pulse.
MSBPOST	2	Sample bits msb first, after clock pulse.
LSBPOST	3	Sample bits lsb first, after a clock pulse.
FASTMSBPRES	4	Faster sampling, msb first, before clock pulse
FASTLSBPRES	5	Faster sampling, lsb first, before clock pulse
FASTMSBPOST	6	Faster sampling, msb first, after clock pulse
FASTLSBPOST	7	Faster sampling, lsb first, after clock pulse

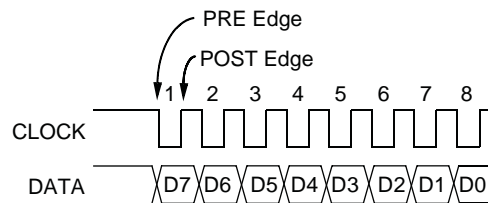
Speeds

Processor	Mode 0-4	Mode 5-7
BA	50 Kbps	100 Kbps
BAN	20 Kbps	40 Kbps
BAP	100 Kbps	380 Kbps
BAP40	400 Kbps	400 Kbps
ARC32	400 Kbps	400 Kbps

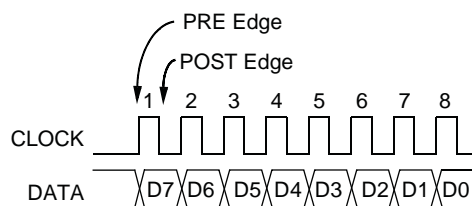
All SPI devices fall into 4 main transmission modes. They are specific to the timings of the writing and reading of data. The write and read of data at the first edge of the clock pulse is PRE. The second edge of the clock pulse is POST. The PRE and POST edge of the clock will change depending on whether the clock starts high(1) or low(0).

Most SPI device data sheets will indicate if they are PRE, POST, MSB or LSB. However in some cases the data sheet might not be clear. So first we must determine if the data is read PRE or POST clock edge. In almost all cases data can be written and read on the rising and falling edge of the clock signal. Looking at the timing diagrams below the clock can start from one of two states high(1) or low(0). The diagrams below illustrate the PRE and POST edge based on a high(1) or low(0) clock start.

**High(1) Clock Start**



**Low(0) Clock Start**



The diagrams for clock start show the data in both cases being written on the POST edge of the clock. The next thing to determine is the bit order. Which from the diagrams starts with bit 7 (D7) and ends with bit 0 (D0), this is MSB (Most Significant Bit First). If the byte order were reversed and bit 0 (D0) was first with bit 7 (D7) last that would be LSB (Least Significant Bit First)

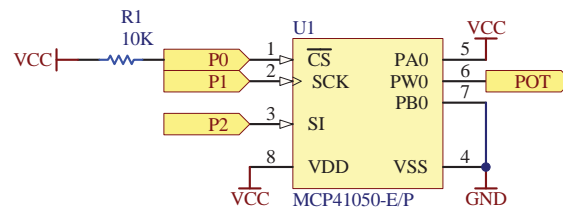


## Notes

1. SHIFTIN and SHIFTOUT only work as a master device, the clock pin will always be an output for both SHIFTIN and SHIFTOUT.

## Schematics

The digital potentiometer is controlled by SHIFTOUT command using SPI communications. The 10K pull-up resistor on CS is require to disable the part when its in an idle state. P1 is connected to the clock pin and P2 is connected to the serial data input pin.



## Examples

The following example will cycle the digital potentiometer from 0K to 50K. Connect the potentiometer as shown above and run the example code. Connect a multimeter to ground and the PW0 / POT pin on the digital potentiometer. As the code cycles the pot the multimeter reading will corollate.

```
cs con p0
clk con p1
si con p2

temp var byte
temp = 0

high clk
main
    temp = temp + 1
    low cs
    shiftout si,clk,MSBPRES,[(0x11<<8) | temp\16]
    high cs
    pause 25
goto main
```

---

## SLEEP

Syntax  
sleep time

- **Time** - is a variable, constant or expression that specifies the amount of time to sleep.

Supported

- BA - Supported with 1 second timings.
- BAN - Supported with 1 second timings.
- BAP - Supported with 1ms timings.
- BAP40 - Supported with 1ms timings.

Description

The SLEEP command is similar to the NAP command. It use the same behind the scene method to put the processor in a lower power state. SLEEP however supports timings in increments of 1. The BasicATOM and Nano supports 1 second timings. The BasicATOM Pro and BasicATOM Pro 40 support 1 millisecond timing increments.

Notes

1. BasicATOM and Nano support 1 second timing increments.
2. BasicATOM Pro and BasicATOM Pro 40 support 1 millisecond timings.

Example

The example program shown will sleep for 1 minute on a BasicATOM and Nano. Increase 60 to 60000 to sleep for the same time on the BasicATOM Pro and BasicATOM Pro 40 .

```
main
  serout s_out, i9600, ["Going to Sleep"]
  sleep 60 ;change to 6000 for ATOM Pro
  serout s_out, i9600, ["I'm awake!"]
goto main
```

---

## SOUND

### Syntax

sound pin, [duration\note, ...]

- **Pin** - is a variable or constant that specifies the pin used to generate the sounds on.
- **Duration** - is a variable, constant or expression that specifies how long to play the note following the backslash (\). Duration is in milliseconds.
- **Note** - is a variable constant or expression that specifies the frequency to generate in Hz. Notes and their durations can be a list separated by commas and only limited to the amount of user ram available.

### Supported

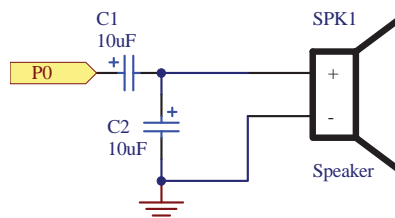
- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The SOUND command generates a square wave output. It can drive a small piezo speaker without amplification. To increase the volume output add a low pass filter such as a 10uf capacitor in-line.

### Schematic

The schematic outlines a simple method of connecting a piezo speaker. The 10uf capacitor is typically an electrolytic but can be a tantalum. You can experiment by omitting the capacitor when a song is being played to see the difference it makes.



## Example

Connect a piezo speaker to P0. The speakers GND should be connected to GND and the positive side connected through a 10uf capacitor to P0 (See schematic). The following program will play "Mary Had a Little Lamb" looping forever. The constant table in the program is setup to correctly create a given note. This makes writing the song easier since its now human readable. See what songs you can make.

```

;ALL - all_sound.bas

;The below table defines all the notes needed.
;It assigns the numeric value to an easily readable
;constant to make the song easier to create.

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

;A lookup table is used to load the notes into the sound command.
;With a FOR/NEXT loop to increment through the note table in lookup.
;A slight pause is added to prevent the notes from overlapping. The
;loop will repeat forever.

Position var byte
Note var long

playsong
  for Position = 0 to 33
    lookup Position,[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],Note
    if Note then
      sound p0,[300\Note]
    else
      pause 10
    endif
  next
  pause 5000
  goto playsong

```

## SOUND2

### Syntax

sound2 pin1\pin2,[duration\note1\note2, ...]

- **Pin** - is a variable, constant or expression that specifies the first pin used for generating note1 on.
- **Pin** - is a variable, constant or expression that specifies the first pin used for generating note2 on.
- **Duration** - is a variable, constant or expression that specifies how long to play the note (following the \). Duration is in milliseconds.
- **Note1** - is a variable constant or expression that specifies the first frequency to generate in Hz. Notes and their durations can be a list separated by commas and only limited to the amount of user ram available.
- **Note2** - is a variable constant or expression that specifies the second frequency to generate in Hz. Notes and their durations can be a list separated by commas and only limited to the amount of user ram available.

### Supported

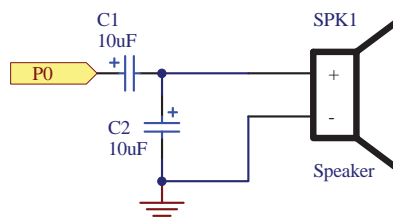
- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The SOUND2 command generates two square waves. Each wave is output on a unique pin. It can drive a small piezo speaker without amplification. To increase the volume output add a low pass filter such as a 10uf capacitor in-line.

### Schematic

The schematic outlines a simple method of connecting a piezo speaker. The 10uf capacitor is typically an electrolytic but can be a tantalum. You can experiment by omitting the capacitor when a song is being played to see the difference it makes.



### Examples

The following example will generate a two tone sound from P0 and P1. To connect a speaker see the circuits shown for SOUND.

```
main
  sound2 p0,p1,[500\500\1000,500\600\900]
goto main
```

## STOP

Syntax  
stop

Supported

- BA - supported
- BAN - supported
- BAP - supported
- BAP40 - supported

Description

Stops program execution until a reset occurs. All I/O pins will remain in their last known state. Typically STOP is used at the very end of a program to halt all processes and cleanly shut down operation. STOP works almost the same as END. Unlike END, STOP does not put the processor in a low power mode.

Example

The following example will only run once. The program will only restart if reset is pressed of the power is cycled.

```
value var long
    serout s_out,i9600,["This program just stops.",13]
    serout s_out,i9600,["Press reset to see it again.",13]
stop
```

---

## SWAP

### Syntax

swap variable1, variable2

- **Variable** - is any variable defined in the program.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The SWAP command is a convenient way to switch the value of one variable with another.

### Notes

1. SWAP will truncate a value if it is larger than the destination variable is setup for. If a WORD value is loaded into a BYTE variable it will truncate and load the low byte of the WORD.

### Examples

Connect to the following program with the terminal window set to 9600 baud. Variable1 and variable2 values will be printed to the screen. The variables are then printed again, after the SWAP command is issued. You can see values of the variables have been swapped.

```
;ALL - all_swap.bas

Variable1 var word
Variable2 var word

Variable1 = 256
Variable2 = 512

Main
  Pause 500
  Serout s_out, i9600, [0, "Variable1 = ", dec variable1, 13]
  Serout s_out, i9600, ["Variable2 = ", dec variable2, 13 ]

  Pause 1000
  Serout s_out, i9600, [13, "Swapping Variables", 13]
  Pause 1000

  swap variable1, variable2 ;lets swap var1 with var2

  ;Lets see what happened
  Serout s_out, i9600, [13, "Variable1 = ", dec variable1, 13]
  Serout s_out, i9600, ["Variable2 = ", dec variable2]

End
```

---

## TOGGLE

Syntax  
toggle pin

- **Pin** - is any expression, constant or variable that specifies an I/O pin to use.

Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

Description

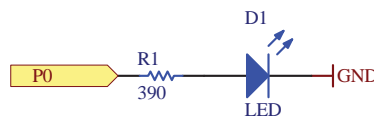
TOGGLE configures a pin as an output and toggles it's state from low (0) to high (1) or high (1) to low (0). The TOGGLE command is not like the REVERSE command since TOGGLE only deals with the output state of the pin low or high.

Notes

1. All pins on power up are inputs by default with a default output state of low if made outputs.

Schematic

Connect a LED as shown below. You can substitute the resistor for anything from 330 ohms to 470 ohms.



Example

The following program demonstrates another way to blink an LED attached to pin 0. The TOGGLE command changes the state of the pin from low as defined by LOW P0 to a high since it “toggles” the pin state.

```
;Wire an LED to P0  
  
Low p0  
  
Main  
  Toggle p0  
  Pause 500  
  Goto main
```



## WHILE - WEND

### Syntax

```
while condition
    program statements
wend
```

- **Condition** - can be a variable or expression
- **Statements** - any group of commands to be run inside the loop.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The WHILE - WEND loop executes commands nested inside of it while some condition is true. The condition is tested before the WHILE - WEND loop is ran. This is opposite of DO - WHILE which will test the condition for true after running once. The condition can be any variable or expression and is tested every loop until false. A simple example would be using the WHILE - WEND loop to test the state of an input pin. If the pin is low go do something until pin is high.

### Notes

1. In the programming world 0 is consider false. By default WHILE - WEND will test this condition. If a stand alone variable is used for the test the loop will continue until its value equals 0.
2. WHILE - WEND checks the condition first. If the condition is false the WHILE - WEND statements and all program code nested within them will not run.
3. You can nest multiple WHILE - WEND commands within each other. However you can not nest DO - WHILE with a WHILE - WEND together or the compiler will get the WHILE statements confused.

---

**Example**

Connect to the following program with the terminal window set to 9600 baud. The program will start counting up from 0 to 100. Once index reaches a value of 100 the condition is no longer true. The less than symbol < was used for the condition and 100 is no longer less than 100 making the condition false. Since WHILE - WEND loops if a statement is true the program exits.

```
;ALL - all_while_wend2.bas
Index var word
Main
  Index = 0
  While Index < 100 ;run until no longer less than 100
    index = index + 1
    ;print the value of index
    serout s_out, i9600,[0, "Couting: ", dec index]
    pause 75
  wend
  serout s_out, i9600, [13,13, "Index = ", dec index]
  serout s_out, i9600, [13, "My condition is no longer true."]
  serout s_out, i9600, [13, "Index is no longer less than 100"]
End
```

---

## WRITE

### Syntax

write address,data

- **Address** - address of eeprom to store data
- **Data** - is a byte value that will be written to the address specified. It can be an expression, constant or variable.

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

All modules except the BasicATOM Pro 24 come with built in eeprom. The READ / WRITE commands were created to access the built in memory. WRITE will write a signal byte to the built in eeprom at the address specified in the command.

### Notes

1. The AtomPro24 does not have built in eeprom. An external 32kbit or larger eeprom can be added. Connect the eeprom SCL pin to P10 and the eeprom SDA pin to P11. A 10k ohm pull up resistor is required on P11. Ground the eeproms A0,A1 and A2 address pins.

### Example

The example program will write the string "Hello" starting at the first location to the built in memory. Next, it will read built in memory locations 0 through 10 and print the contents to the terminal window.

```
index var byte
char var byte

write 0,"H"
write 1,"e"
write 2,"l"
write 3,"l"
write 4,"o"

for index = 0 to 10
  read index,char
  serout s_out,i9600,[char]
next
end
```

---

## WRITEDM

### Syntax

write address, [{modifiers}data1, ..., {modifiers}dataN]

- **Address** - memory location to start the write at. Can be an expression, constant or variable.
- **Data** - a list of values and optional modifiers that will be stored in the built in eeprom. Values larger than a byte will be truncated to a byte.
- **Modifiers** - are used in the command syntax to format data. Refer to the modifier section of this manual. All output modifiers are supported

### Supported

- BA - Supported.
- BAN - Supported.
- BAP - Supported.
- BAP40 - Supported.

### Description

The READDM differs from the READ command in that its auto incrementing. It can be ran once and read multiple bytes from the onboard eeprom using only a starting address. READDM will continue to read based on how many variables are specified. Modifiers can be used to format the data or add functionality to the command. See table above.

### Notes

1. The AtomPro24 does not have built in eeprom. An external 32kbit or larger eeprom can be added. Connect the eeprom SCL pin to P10 and the eeprom SDA pin to P11. A 10k ohm pull up resistor is required on P11. Ground the eeproms A0,A1 and A2 address pins.
2. The WriteDM command differs from the Write command in that auto increments the address. It can write multiple bytes each time it is executed. The Write command can only write one byte each time it is executed.

### Example:

The program will write "Hello World" to the on board eeprom starting at location 0. Then using the ReadDM command will read the on board eeprom starting at location 0 then print the results to the terminal window using 9600 baud.

```
writedm 0,["Hello world"]

index var byte
string var byte(11)

    readdm index,[str string\11]
    serout s_out,i9600,[str string\11]
end
```



---

## Interrupts

While the program is executing it is possible to interrupt this execution to handle another event. The events can be things like a pin changing from high to low or low to high. Other events can be received data or timers counting down or counting up. There are several types of interrupts. The interrupt system is an advance feature of MBasic and should only be explored by an advance users. A basic understanding of interrupts and there limitations is required to utilize them properly. MBasic interrupts are only available on BasicATOMPro processors(ProONE,Pro24,Pro28,Pro40 and ARC32)

### ONINTERRUPT

oninterrupt interrupt, handler

- **Interrupt** - one of the listed interrupt types. See Interrupt Types table.
- **Handler** - the interrupt handler label to jump to when the interrupt triggers.

This is a directive to the compiler to specify the label to jump to when a specific interrupt triggers. The interrupt handler label must use RESUME to exit the interrupt handler.

### ONASMINTERRUPT

onasminterrupt interrupt, handler

- **Interrupt** - one of the listed interrupt types. See Interrupt Types table.
- **Handler** - the assembly interrupt handler label to jump to when the interrupt triggers.

This is a directive to the compiler to specify the label to jump to when a specific interrupt triggers. The assembly interrupt handler must use the RTE assembly command to exit the interrupt handler.

### ENABLE

enable {interrupt}

- **Interrupt** - an optional argument of one of the listed interrupt types. See Interrupt Types table.

Enables one or all interrupts. When used with no arguments the global interrupt enable flag is set. When used with an interrupt argument, that interrupt's enable flag is set. Only when both the global interrupt flag and specific interrupt flags are set will those interrupts be allowed to trigger.

### DISABLE

disable {interrupt}

- **Interrupt** - an optional argument of one of the listed interrupt types. See Interrupt Types table.

Disables one or all interrupts. When used with no arguments the global interrupt enable flag is cleared. When used with an interrupt argument, that interrupt's enable flag is cleared. If either the global interrupt flag or the specific interrupt flag is cleared that interrupt will not trigger.

RESUME  
resume

This command is used to exit the Basic interrupt handler and return to the point in your program where execution was interrupted due to an interrupt trigger.

#### Interrupt Example

The following example uses Time V and creates a counter that will blink an LED every second regardless of what the program is doing. This is a very simple way to create a back ground task that can easily indicate the device is still running.

```
;calculates the number of interrupts per 1/100th of a second
;as a floating point constant.

interval fcon MHZ/100/256/128

counter var float
counter = 0.0

milliseconds var long
milliseconds = 0

TCRV0 = 0x03      ;Sets Timer V to count once every 128 OSC clocks
TCRV1 = 0x01

;Tells the processor where to jump to when the timer V
;overflow interrupt triggers.

ONINTERRUPT TIMERVINT_OVF,handler

ENABLE TIMERVINT_OVF  ;enables the timer V overflow interrupt
ENABLE                ;enables interrupts in general

main
  if(milliseconds >= 500)then
    toggle p0
    milliseconds = milliseconds - 500
  endif
  goto main

handler
  ;this interrupt is executed once per 256*128 clock cycles

  counter = counter + 1.0
  if(counter>interval)then
    counter=counter-interval
    milliseconds = milliseconds + 10
  endif
  resume
```

### Interrupt Types

There are several types of interrupts available on the BasicATOM Pro processors. The following tables list all the types available in MBasic. The types of interrupts will trigger based on different events. The processor at the heart of the BasicATOM Pro is an H8Tiny. The following chart details what is available. Information on how to utilize the interrupt can be found in the H8Tiny hardware manuals.

Interrupt Table

Interrupt	Description	Renesas Hardware Manual Pages
IRQ0INT	Irq0 pin interrupt	
IRQ1INT	Irq1 pin interrupt	
IRQ2INT	Irq2 pin interrupt	
IRQ3INT	Irq3 pin interrupt	
WKPINT_0	WKP0 pin onchange interrupt	
WKPINT_1	WKP1 pin onchange interrupt	
WKPINT_2	WKP2 pin onchange interrupt	
WKPINT_3	WKP3 pin onchange interrupt	
WKPINT_4	WKP4 pin onchange interrupt	
WKPINT_5	WKP5 pin onchange interrupt	
TIMERVINT_OVF	TimerV overflow interrupt	
TIMERVINT_CMEB	TimerV compare match A int	
TIMERVINT_CMEA	TimerV compare match B int	
SCI3INT_TDRE	Transmit Data Register Empty interrupt	
SCI3INT_RDRF	Read Data Register Full interrupt	
SCI3INT_TEND	Transmit End interrupt	
SCI3INT_OER	Overflow Error interrupt	
SCI3INT_FER	Frame Error interrupt	
SCI3INT_PER	Parity Error interrupt	
IICINT	I2C interrupt	
ADINT	Analog conversion complete int	
HSERIALINT_TDRE	Transmit Data Register Empty interrupt	
HSERIALINT_RDRF	Read Data Register Full interrupt	
HSERIALINT_TEND	Transmit End interrupt	
HSERIALINT_OER	Overflow Error interrupt	
HSERIALINT_FER	Frame Error interrupt	
HSERIALINT_PER	Parity Error interrupt.98	
HSERVOINT	Servo Handler interrupt.	
HSERVOINT_IDLE#	Servo Idle interrupt. Replace # with servo number.	



BAP ONE/24/28 Only Interrupt Types

Interrupt	Description	Renesas Hardware Manual Pages
TIMERAIN	Overflow interrupt	
TIMERWINT_OVF	Overflow interrupt	
TIMERWINT_IMIEA	Capture/Compare Match A int	
TIMERWINT_IMIEB	Capture/Compare Match B int	
TIMERWINT_IMIEC	Capture/Compare Match C int	
TIMERWINT_IMIED	Capture/Compare Match D int	

BAP40/ARC32 Only Interrupt Types

Interrupts	Description	Renesas Hardware Manual Pages
RTCINT	Real time clock interrupt	
TIMERZ0INT_OVF	Overflow interrupt	
TIMERZ0INT_IMIEA	Capture/Compare Match A int	
TIMERZ0INT_IMIEB	Capture/Compare Match B int	
TIMERZ0INT_IMIEC	Capture/Compare Match C int	
TIMERZ0INT_IMIED	Capture/Compare Match D int	
TIMERZ1INT_UDF	Underflow interrupt	
TIMERZ1INT_OVF	Overflow interrupt	
TIMERZ1INT_IMIEA	Capture/Compare Match A int	
TIMERZ1INT_IMIEB	Capture/Compare Match B int	
TIMERZ1INT_IMIEC	Capture/Compare Match C int	
TIMERZ1INT_IMIED	Capture/Compare Match D int	
TIMERB1INT	Overflow interrupt	
SCI3_2INT_TDRE	Transmit Data Register Empty interrupt	
SCI3_2INT_RDRF	Read Data Register Full interrupt	
SCI3_2INT_TEND	Transmit End interrupt	
SCI3_2INT_OER	Overflow Error interrupt	
SCI3_2INT_FER	Frame Error interrupt	
SCI3_2INT_PER	Parity Error interrupt	
HSERIAL2INT_TDRE	Transmit Data Register Empty interrupt	
HSERIAL2INT_RDRF	Read Data Register Full interrupt	
HSERIAL2INT_TEND	Transmit End interrupt	
HSERIAL2INT_OER	Overflow Error interrupt	
HSERIAL2INT_FER	Frame Error interrupt	
HSERIAL2INT_PER	Parity Error interrupt.	

# Basic Stamp Conversion



## Basic Stamp Conversion

The following section covers some of the differences you need to be aware of when porting over from a Basic Stamp. Several commands will have speed differences. The speed difference can cause timing issues with your program. Simply making changes to the arguments for those commands will solve most porting issues. The following is a list of compiled notes we have collect from customers and our own experiences when porting programs from the Basic Stamp.

### Timings

On average most commands will execute considerably faster. In some cases you may need to add pauses in between commands if you have timing critical functions.

### Math

MBasic treats math correctly signed or unsigned. So normal math rules apply.

### AUXIO

The AUXIO command has no MBasic equivalent since all I/O are available at all times. No banking of pins is required.

### IN / OUT / DIR

On the Basic Stamp IN / OUT / DIR variables only support up to 16 I/Os. MBasic has pin variables for all pins including the AtomPro 40 pin module.

### BRANCH

The BRANCH command in Mbasic has no label limits. In PBasic its 255 max.

### BUTTON

The BUTTON command is functionally the same.

### COMPARE

The COMPARE command has no MBasic equivalent. RCTIME and or hardware analog functions can be used as a possible alternate.

### CONFIGPIN

The CONFIGPIN command has no MBasic equivalent. Pull-ups can be added to an existing circuit.

### COUNT

The COUNT command is the same but can be 32bits on the AtomPro. Duration is in clock cycles.

### DATA

The DATA command has no equivalent on the BasicATOM Pro processors. Data can be loaded to the on board eeprom using a Write command.

### DEBUG / DEBUGIN

The debug commands are very different. They uses the same syntax as SEROUT / SERIN inside the brackets.

### DO

The DO loop command is similar to MBasic DO..WHILE, WHILE..WEND or REPEAT..UNTIL.

### DTMFOUT

The DTMFOUT command on time and off time use 1ms increments on all modules/processors.

### EEPROM

The EEPROM command has no MBasic equivalent.

### EXIT

The EXIT command has no MBasic equivalent. To exit form a loop add an exit label to the loop.

```
do
    ...doing stuff...
    if wanttoexit then goto loopexit
while 1
loopexit:
```

### FOR...NEXT

The FOR..NEXT loop functions the same.

### FREQOUT

The FREQOUT command is similar but the duration is in milliseconds in MBasic. Freq1 and Freq2 are in Hz.

### GET

The GET command has no MBasic equivalent. An alternate method would be to define a variable and access directly instead of using PBasic scratch pad memory.

### GOSUB

The GOSUB command is functionally the same except MBasic supports calling subroutines with arguments. There is no limit on the number of total GOSUBS. Nested GOSUBS are only limited by the total stack memory available.

### GOTO

The GOTO command functions the same.

### HIGH

The HIGH command functions the same.

### I2CIN - I2COUT

The I2C commands in MBasic are generic and will work with any I2C device. We've removed the eeprom address argument and a I2COUT command is used to send the address when talking to an eeprom. I2CIN data rates are 70kbps on the AtomPro 24, 28 and ONE. 87kbps on the BasicAtom Pro 40 and ARC32.

### IF..THEN

The IF..THEN commands are mostly the same except MBasic doesn't support IF Condition(s) THEN Statement(s) { ELSE Statement(s) } directly. To do this you have to do this:

```
IF Condition(s) THEN : Statement(s) { : ELSE Statement(s) } : ENDIF
```

You have to add the ":" because that makes the compiler see a new line. There is no max nested limit There is no max ELSEIF limit.

### INPUT

The INPUT command functions the same.

### IOTERM

The IOTERM command has no MBasic equivalent. There is no need to switch between pin banks on the Atom.

### LCDCMD

The LCD commands function differently. LCDCMD is replaced with LCDWRITE to send LCD commands. To send a command with LCDWrite send 0x100 + command number. See LCDWRITE and LCDREAD command section of this manual.

### LCDIN

The BS2 LCDIN command is replaced with the MBasic LCDREAD command.

### LCDOUT

The BS2 LCDOUT command is replaced with the MBasic LCDWRITE command.

### LET

The LET command is supported with the use of modifiers or loading array variables with a single command.

```
let myarray = 1,2,3,4,5,6.
```

### LOOKDOWN

The LOOKDOWN command syntax is similar to the BS2. No entry limit. MBasic LOOKDOWN supports 32bit expressions and values .

### LOOKUP

The LOOKUP command syntax is similar to the BS2. No entry limit. MBasic LOOKDOWN supports 32bit expressions and values .

### LOW

The LOW command functions the same.

### MAINIO

The MAINIO command has no MBasic equivalent and is not required because all pins are directly accessible.

### NAP

The NAP command is the same on all processors.

### ON

The ON command has no MBasic equivalent. Here is the way to get the same functionality.

```
main
gosub branchsub

goto main

branchsub
  branch index,[label1,label2,label3,label4]
  ..do default stuff..
  return

label1
..do stuff..
return

label2
..do stuff..
return

label3
..do stuff..
return

label4
..do stuff..
return
```

### OUTPUT

The OUTPUT command functions the same.

### OWIN - OWOUT

The One Wire commands functions the same except Mbasic supports a no connection label. Used to jump to if 1-wire device is not connected. Mbasic only supports normal speed.

### PAUSE

The PAUSE command will run at the same speed since its arguments are based on time. If you are using PAUSE in a loop you may need to increase the value used for PAUSE since the loop will execute faster because other commands execute faster.

### POLLIN

The POLLIN command has no MBasic equivalent. However the functionality can be replicated in code due to the speed on the Atom processors. Optionally Interrupts can be used.

### POLLMODE

The POLLMODE command has no MBasic equivalent. However the functionality can be replicated in code due to the speed on the Atom processors. Optionally Interrupts can be used.

### POLLOUT

The POLLOUT command has no MBasic equivalent. However the functionality can be replicated in code due to the speed on the Atom processors. Optionally Interrupts can be used.

### POLLRUN

The POLLRUN command has no MBasic equivalent. However the functionality can be replicated in code due to the speed on the Atom processors. Optionally Interrupts can be used.

### POLLWAIT

The POLLWAIT command has no MBasic equivalent. However the functionality can be replicated in code due to the speed on the Atom processors. Optionally Interrupts can be used.

### POT

The POT command is replaced by RCTIME. Which provides the same functionality. ADIN would be a better choice in most cases though.

### PULSIN..PULSOUT

The pulse commands functions the same. Units are in .5us with 32bit range or about 35 minutes max pulse width(when setting the time out to the maximum). Default time out is 65536 units or ~32ms.

### PUT

The PUT command has no MBasic equivalent. The same functionality is available by using standard user defined variables instead.

### PWM

The PWM command for MBasic has an additional argument for the period. Use a period of 256 in Mbasic to duplicate the BS2 PWM command. Units are .5us. With a period of 256 cycles are .1285ms

### RANDOM

The RANDOM command in MBasic is a function. eg temp = random seed. MBasic will produce a 32bit random number.

### RCTIME

The RCTIME command is functionally the same. Units are .5us. The maximum range is 32bits. Default time out is 65536 or approximately 32ms.

### READ

The READ command is functionally the same. Read address 0 to 4095 on the AtomPro 28/40. Read addresses 0 to 32,767 on ARC32

### RETURN

The RETURN command is functionally the same. We support returning a value.

### REVERSE

The REVERSE command is functionally the same.

### RUN

The RUN command has no MBasic equivalent. Can replace with a GOTO command in order to jump to another section of code. There is no 2K program limits in MBasic.

### SELECT..CASE

The SELECT..CASE commands can be replaced with MBasic IF..THEN..ELSEIF..ELSE..ENDIF.

### SERIN..SEROUT

The serial commands are functionally the same. The maximum standard baudrate is over 115200bps on all AtomPro modules. no limit to qualifiers. Time out is in .5us units

### SHIFTIN..SHIFTOUT

The shift commands are functionally the same except for speed and extended modes

SLEEP

The SLEEP command is functionally the same but it is in ~1ms units on BasicATOMPro.

SOUND

The SOUND command is functionally the same but Mbasic allows multiple sounds in a sequence. Units are in ms and Hz.

STOP

The STOP command is functionally the same.

STORE

The STORE command has no MBasic equivalent. This command is not necessary with MBasic. Since all memory is accessible at all times.

TOGGLE

The TOGGLE command is functionally the same.

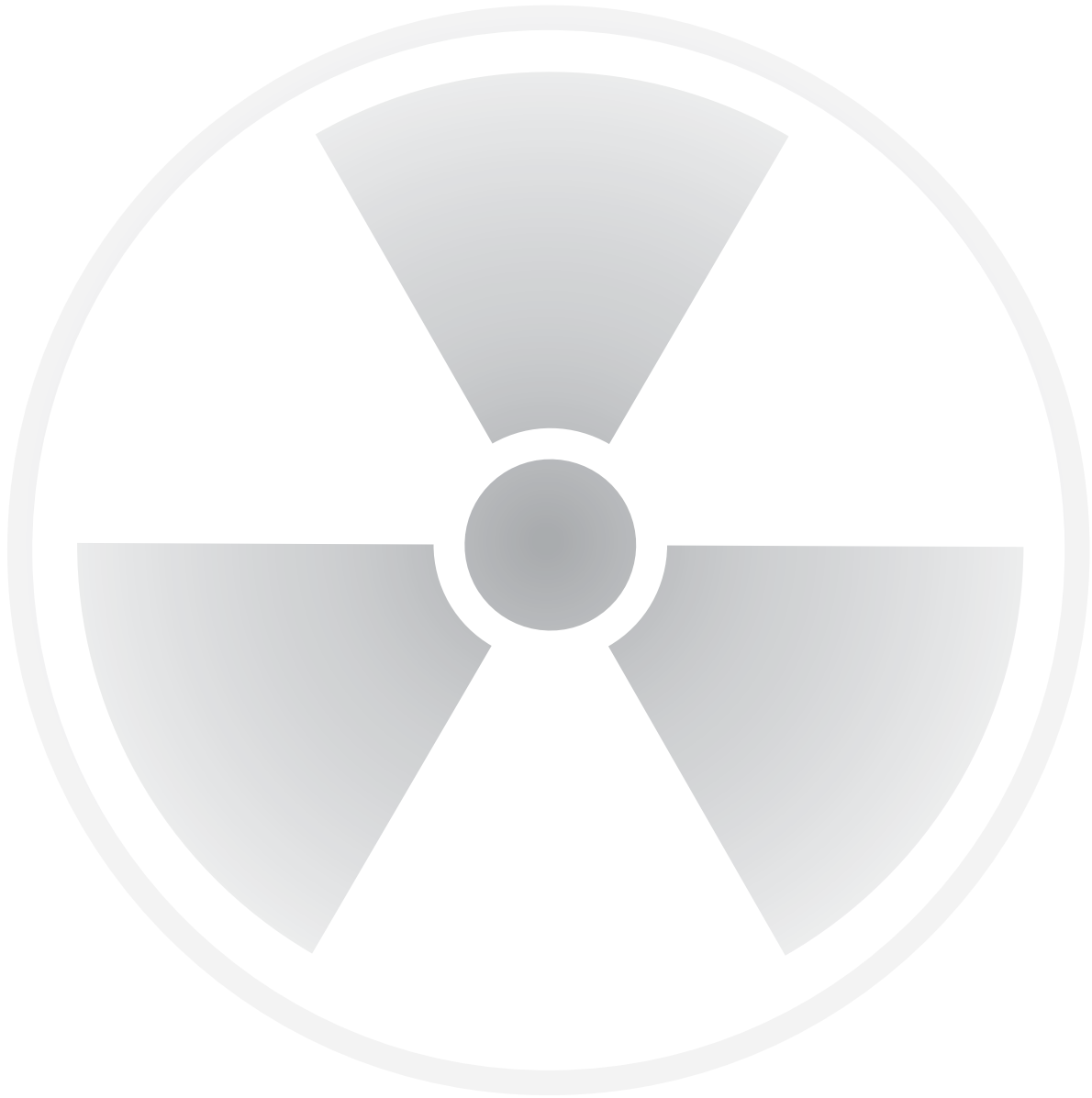
WRITE

The WRITE command is functionally the same. Same memory size as listed for READ

XOUT

The X10 commands have no MBasic equivalent. Most X10 units now use a serial protocol making the XOUT / XIN commands obsolete.





## Compiler Directives

MBasic supports compile time directives. Compile time directives can be used to selectively include or exclude parts of a program which can be very useful. An example would be a product that has several configurations but only requires one program with some minor changes. During compile time you can define what the master program is compiled for allowing MBasic to conditionally compile certain parts of the program.

This lets you keep a collection of frequently used program modules (“snippets” of code) and include them whenever you need them. It also provides conditional tests (IF ... THEN) to modify the code to compile different versions of a program.

### Including Files

Perhaps you’ve written a subroutine to control an LCD display, and you’d like to use this subroutine in various different programs. You can save the subroutine on disk, and “include” it whenever you need it. The #include directive is used to “paste” program modules at compile time. Modules are pasted at the location of the #include directive.

```
#Include
#include "filename"
#include "partial or complete path to file"
```

### Example

Assume that your LCD subroutine is called “lcd.bas” and is in the same directory as your main program. The subroutine contains the label “displaywrite”. You can include this in your program as shown:

```
main
(some code)
    gosub displaywrite
(some more code)
#include "lcd.bas"
end
```

The #include directive simply pastes in the code from lcd.bas as if it was part of your program. If lcd.bas is in a subdirectory of your program directory, just put the partial path, for example:

```
#include "modules\lcd.bas"
```

If it’s in another directory, you can include the relative or absolute path, using normal Windows notation.

### Conditional Compiling

Sometimes the same program may be used for slightly different applications. For example, if you’ve written a program to display temperature from a sensor, you may want versions for Celsius and Fahrenheit degrees, or perhaps you want one version to use an LCD display and a different one to output serial data to your computer. Most of the code is identical, but some constants, variables and subroutines may differ.

Conditional compiling lets you set a “switch” in your program (usually a constant, but not necessarily) that controls compiling. You can have different constants, variables, or even different sections of code compiled depending on the switch or switches that you set.

```
#IF .. #ENDIF
#IF expression
    optional code
#ENDIF
```

#### Example

Similar to Mbasic IF..THEN conditional branch but specifies code to be compiled if the expression is true. In the example below the constant temp is set to 1. During compile time the #IF will test temp to see if it is true. Which in the example below will return true so the following block of code is included during compile time.

```
temp con 1
#IF temp=1
    ..optional code..
#ENDIF
..rest of program..
```

```
#IFDEF .. #ENDIF
#IFDEF name
    ..optional code..
#ENDIF
..rest of program..
```

#### Example

Compiles the code (up to #ENDIF) if the constant or variable (name) is defined, or if the label appears previously in the code.

```
temperature var byte
#IFDEF temperature
    ..optional code..
#ENDIF
..rest of program..
```

This will compile “optional code” because “temperature” has been defined.

```
#IFNDEF .. #ENDIF
#IFNDEF name
    ..optional code..
#ENDIF
..rest of program..
```

#### Example

Compiles the code between #IFNDEF and #ENDIF only if the constant or variable has not been defined, or the label has not been previously used in the program. In effect, it’s the opposite of #IFDEF.

```
temperature var byte
#IFNDEF temperature
    ..optional code..
#ENDIF
..rest of program..
```

This will NOT compile “optional code” because “temperature” has been defined.

```
#ELSE
#IF expression
    ..optional code..
#ELSE
    ..use this code if the other code wasnt used..
#ENDIF
..rest of program..
```

#### Example

Allows you to have two code snippets, and compile one or the other depending on the result of the #IF, #IFDEF or #IFNDEF directive.

```
temp con 1
#IF temp=1
    ..optional code..
#ELSE
    ..more optional code..
#ENDIF
    ..rest of program..
```

Compiles “optional code” if “temp = 1” and “more optional code” if “temp” is equal to any other value.

```
#ELSIF
#IF expression
    ..optional code..
#ELSEIF
    ..more optional code..
#ELSEIF
    ..even more optional code..
#ELSE
    ..if nothing else was used, then use this code..
#ENDIF
..rest of program..
```

#### Example

Allows multiple snippets of code to be compiled based on multiple tests. ELSEIF is an extension of #ELSE and allows multiple test to be ran during compile time.

```
screentype con 1
#IF screentype=1
    ..optional code..
#ELSEIF screentype=2
    .. more optional code..
#ELSEIF screentype=3
    ..even more optional code..
#ENDIF
... rest of program ...
```

Compiles “optional code”, “more other code”, or “even more optional code” depending on what the constant “screentype” is set to (1, 2 or 3). If “screentype” has some other value than 1,2 or 3 compilation simply continues with “rest of program” and none of the optional code is compiled.

`#ELSEIFDEF, #ELSEIFNDEF`

Equivalents of `#ELSEIF` for the `#IFDEF` and `#FNDEF` directives.

Syntax

`#ELSEIFDEF` name

`#ELSEIFNDEF` name

Example

Similar to the example given for `#ELSEIF`.

`BEGINASMSUB, ENDASMSUB`

This is a special directive that tells the compiler not to add special entry code to inline assembly code. This is necessary when writing ASM subroutines which are called by other ASM code and ASM interrupt handlers.



## Reserved Words

The following section list all the names and words used internally by MBasic. These words can not be used in your program for constants and variable names. Otherwise MBasic will mistake them for commands or other internal names.

## Reserved Words

Name	Name	Name	Name
DIRS	DIR24	IN14	OUT4
DIRL	DIR25	IN15	OUT5
DIRH	DIR26	IN16	OUT6
DIRA	DIR27	IN17	OUT7
DIRB	DIR28	IN18	OUT8
DIRC	DIR29	IN19	OUT9
DIRD	DIR30	IN20	OUT10
DIREA	DIR31	IN21	OUT11
DIREB	DIR32	IN22	OUT12
DIREC	DIR33	IN23	OUT13
DIRED	INS	IN24	OUT14
DIR0	INL	IN25	OUT15
DIR1	INH	IN26	OUT16
DIR2	INA	IN27	OUT17
DIR3	INB	IN28	OUT18
DIR4	INC	IN29	OUT19
DIR5	IND	IN30	OUT20
DIR6	INEA	IN31	OUT21
DIR7	INEB	IN32	OUT22
DIR8	INEC	IN33	OUT23
DIR9	INED	OUTS	OUT24
DIR10	IN0	OUTL	OUT25
DIR11	IN1	OUTH	OUT26
DIR12	IN2	OUTA	OUT27
DIR13	IN3	OUTB	OUT28
DIR14	IN4	OUTC	OUT29
DIR15	IN5	OUTD	OUT30
DIR16	IN6	OUTEA	OUT31
DIR17	IN7	OUTEB	OUT32
DIR18	IN8	OUTEC	OUT33
DIR19	IN9	OUTED	INDF
DIR20	IN10	OUT0	TMR0
DIR21	IN11	OUT1	PCL
DIR22	IN12	OUT2	STATUS
DIR23	IN13	OUT3	FSR

## Reserved Words

Name	Name	Name	Name
PORTA	SSPADD	RBIE	SSPM1
PORTB	SSPSTAT	T0IF	SSPM0
PORTC	WPUB	TMR0IF	P1M1
PORTD	IOCB	INTF	P1M0
PORTE	VRCON	RBIF	DC1B1
PCLATH	TXSTA	ADIF	CCP1X
INTCON	SPBRG	RCIF	DC1B0
PIR1	SPBRGH	TXIF	CCP1Y
PIR2	PWM1CON	SSPIF	CCP1M3
TMR1L	ECCPAS	CCP1IF	CCP1M2
TMR1H	PSTRCON	TMR2IF	CCP1M1
T1CON	ADRESL	TMR1IF	CCP1M0
TMR2	ADCON1	OSPIF	SPEN
T2CON	WDTCON	C2IF	RX9
SSPBUF	CM1CON0	C1IF	RC9
SSPCON	CM2CON0	EEIF	NOT_RC8
CCPR1L	CM2CON1	BCLIF	RC8_9
CCPR1H	EEDATA	ULPWUIF	SREN
CCP1CON	EEADR	CCP2IF	CREN
RCSTA	EEDATH	T1CKPS1	ADDEN
TXREG	EEADRH	T1CKPS0	FERR
RCREG	SRCON	T1OSCEN	OERR
CCPR2L	BAUDCTL	NOT_T1SYNC	RX9D
CCPR2H	ANSEL	T1INSYNC	RCD8
CCP2CON	ANSELH	T1SYNC	CCP2X
ADRESH	EECON1	TMR1CS	DC2B1
ADCON0	EECON2	TMR1ON	CCP2Y
OPTION_REG	IRP	TOUTPS3	DC2B0
TRISA	RP1	TOUTPS2	CCP2M3
TRISB	RP0	TOUTPS1	CCP2M2
TRISC	NOT_TO	TOUTPS0	CCP2M1
TRISD	NOT_PD	TMR2ON	CCP2M0
TRISE	Z	T2CKPS1	ADCS1
PIE1	DC	T2CKPS0	ADCS0
PIE2	C	WCOL	CHS3
PCON	GIE	SSPOV	CHS2
OSCCON	PEIE	SSPEN	CHS1
OSCTUNE	T0IE	CKP	CHS0
SSPCON2	TMR0IE	SSPM3	GO
PR2	INTE	SSPM2	NOT_DONE



## Reserved Words

Name	Name	Name	Name
GO_DONE	TUN0	IOCB4	PDC6
ADON	GCEN	IOCB3	PDC5
NOT_RBPU	ACKSTAT	IOCB2	PDC4
INTEDG	ACKDT	IOCB1	PDC3
T0CS	ACKEN	IOCB0	PDC2
T0SE	RCEN	VREN	PDC1
PSA	PEN	VROE	PDC0
PS2	RSEN	VRR	ECCPASE
PS1	SEN	VRSS	ECCPAS2
PS0	SMP	VR3	ECCPAS1
ADIE	CKE	VR2	ECCPAS0
RCIE	D	VR1	PSSAC1
TXIE	I2C_DATA	VR0	PSSAC0
SSPIE	NOT_A	CSRC	PSSBD1
CCP1IE	NOT_ADDRESS	TX9	PSSBD0
TMR2IE	D_A	NOT_TX8	STRSYNC
TMR1IE	DATA_ADDRESS	TX8_9	STRD
OSFIE	P	TXEN	STRC
C2IE	I2C_STOP	SYNC	STRB
C1IE	S	BRGH	STRA
EEIE	I2C_START	TRMT	ADFM
BCLIE	R	TX9D	VCFG0
ULPWUIE	I2C_READ	TXD8	VCFG1
CCP2IE	NOT_W	BRG7	WDTPS3
ULPWUE	NOT_WRITE	BRG6	WDTPS2
SBOREN	R_W	BRG5	WDTPS1
NOT_POR	READ_WRITE	BRG4	WDTPS0
NOT_BO	UA	BRG3	SWDTEN
NOT_BOR	BF	BRG2	C1ON
IRCF2	WPUB7	BRG1	C1OUT
IRCF1	WPUB6	BRG0	C1OE
IRCF0	WPUB5	BRG15	C1POL
OSTS	WPUB4	BRG14	C1R
HTS	WPUB3	BRG13	C1CH1
LTS	WPUB2	BRG12	C1CH0
SCS	WPUB1	BRG11	C2ON
TUN4	WPUB0	BRG10	C2OUT
TUN3	IOCB7	BRG9	C2OE
TUN2	IOCB6	BRG8	C2POL
TUN1	IOCB5	PRSEN	C2R

## Reserved Words

Name	Name	Name	Name
C2CH1	P0	P40	D2
C2CH0	P1	P41	D3
MC1OUT	P2	AX0	D4
MC2OUT	P3	AX1	D5
C1RSEL	P4	AX2	D6
C2RSEL	P5	AX3	D7
T1GSS	P6	S_IN	E0
C2SYNC	P7	S_OUT	E1
SR1	P8	GP0	E2
SR0	P9	GP1	E3
C1SEN	P10	GP2	E4
C2REN	P11	GP3	E5
PULSS	P12	GP4	E6
PULSR	P13	GP5	E7
FVREN	P14	A0	F0
ABDOVF	P15	A1	F1
RCIDL	P16	A2	F2
SCKP	P17	A3	F3
BRG16	P18	A4	F4
WUE	P19	A5	F5
ABDEN	P20	A6	F6
ANS7	P21	A7	F7
ANS6	P22	B0	G0
ANS5	P23	B1	G1
ANS4	P24	B2	G2
ANS3	P25	B3	G3
ANS2	P26	B4	G4
ANS1	P27	B5	G5
ANS0	P28	B6	G6
ANS13	P29	B7	G7
ANS12	P30	C0	H300
ANS11	P31	C1	H600
ANS10	P32	C2	H1200
ANS9	P33	C3	H2400
ANS8	P34	C4	H4800
EEPGD	P35	C5	H7200
WRERR	P36	C6	H9600
WREN	P37	C7	H12000
WR	P38	D0	H14400
RD	P39	D1	H16800

## Reserved Words

Name	Name	Name	Name
H19200	TMR0INT128	TMR2PRE1POST7	TMR2PRE16POST15
H21600	TMR0INT256	TMR2PRE1POST8	TMR2PRE16POST16
H24000	TMR0EXTL1	TMR2PRE1POST9	CAPTUREOFF
H26400	TMR0EXTL2	TMR2PRE1POST10	CAPTURE1H2L
H28800	TMR0EXTL4	TMR2PRE1POST11	CAPTURE1L2H
H31200	TMR0EXTL8	TMR2PRE1POST12	CAPTURE4L2H
H33600	TMR0EXTL16	TMR2PRE1POST13	CAPTURE16L2H
H36000	TMR0EXTL32	TMR2PRE1POST14	COMPAREOFF
H38400	TMR0EXTL64	TMR2PRE1POST15	COMPARESETHIGH
H57600	TMR0EXTL128	TMR2PRE1POST16	COMPARESETLOW
H115200	TMR0EXTL256	TMR2PRE4POST1	COMPAREINT
H250000	TMR0EXTH1	TMR2PRE4POST2	COMPARESPECIAL
H312500	TMR0EXTH2	TMR2PRE4POST3	X_A
H625000	TMR0EXTH4	TMR2PRE4POST4	X_B
H1250000	TMR0EXTH8	TMR2PRE4POST5	X_C
MSBPRE	TMR0EXTH16	TMR2PRE4POST6	X_D
LSBPRE	TMR0EXTH32	TMR2PRE4POST7	X_E
MSBPOST	TMR0EXTH64	TMR2PRE4POST8	X_F
LSBPOST	TMR0EXTH128	TMR2PRE4POST9	X_G
FASTMSBPRE	TMR0EXTH256	TMR2PRE4POST10	X_H
FASTLSBPRE	TMR1OFF	TMR2PRE4POST11	X_I
FASTMSBPOST	TMR1INT1	TMR2PRE4POST12	X_J
FASTLSBPOST	TMR1INT2	TMR2PRE4POST13	X_K
SLOWMSBPRE	TMR1INT4	TMR2PRE4POST14	X_L
SLOWLSBPRE	TMR1INT8	TMR2PRE4POST15	X_M
SLOWMSBPOST	TMR1EXT1	TMR2PRE4POST16	X_N
SLOWLSBPOST	TMR1EXT2	TMR2PRE16POST1	X_O
MSBFIRST	TMR1EXT4	TMR2PRE16POST2	X_P
LSBFIRST	TMR1EXT8	TMR2PRE16POST3	X_1
PU_OFF	TMR1ASYNC1	TMR2PRE16POST4	X_2
PU_ON	TMR1ASYNC2	TMR2PRE16POST5	X_3
EXT_H2L	TMR1ASYNC4	TMR2PRE16POST6	X_4
EXT_L2H	TMR1ASYNC8	TMR2PRE16POST7	X_5
TMR0INT1	TMR2OFF	TMR2PRE16POST8	X_6
TMR0INT2	TMR2PRE1POST1	TMR2PRE16POST9	X_7
TMR0INT4	TMR2PRE1POST2	TMR2PRE16POST10	X_8
TMR0INT8	TMR2PRE1POST3	TMR2PRE16POST11	X_9
TMR0INT16	TMR2PRE1POST4	TMR2PRE16POST12	X_10
TMR0INT32	TMR2PRE1POST5	TMR2PRE16POST13	X_11
TMR0INT64	TMR2PRE1POST6	TMR2PRE16POST14	X_12

## Reserved Words

Name	Name	Name	Name
X_13	HOMELCD	ONELINE5X11	
X_14	LCDCLEAR	CGRAM	
X_15	LCDHOME	SCRRAM	
X_16	INCCUR	HSERSTAT_INCLEAR	
X_UNITS_ON	INCSCR	HSERSTAT_OUTCLEAR	
X_LIGHTS_ON	DECCUR	HSERSTAT_CLEAR	
X_ON	DECSCR	HSERSTAT_INDATA	
X_OFF	LCDOFF	HSERSTAT_INNODATA	
X_DIM	SCR	HSERSTAT_OUTDATA	
X_BRIGHT	SCRBLK	HSERSTAT_OUTNODATA	
X_LIGHTS_OFF	SCRCUR		
X_HAIL	SCRCURBLK		
X_STATUS_ON	CURLEFT		
X_STATUS_OFF	CURRIGHT		
X_STATUS_REQUEST	SCRLEFT		
INITLCD1	SCRRIGHT		
INITLCD2	ONELINE		
CLEARLCD	TWOLINE		



Dec	Hex	Char	Function
0	0x00	NUL	Null
1	0x01	SOH	Start of Heading
2	0x02	STX	Start of Text
3	0x03	ETX	End of Text
4	0x04	EOT	End of Transmit
5	0x05	ENQ	Enquiry
6	0x06	ACK	Acknowledge
7	0x07	BEL	Bell
8	0x08	BS	Backspace
9	0x09	HT	Horizontal Tab
10	0x0A	LF	Line Feed
11	0x0B	VT	Vertical Tab
12	0x0C	FF	Form Feed
13	0x0D	CR	Carriage Return
14	0x0E	SO	Shift Out
15	0x0F	SI	Shift In
16	0x10	DLE	Data Line Escape
17	0x11	DC1	Device Cntrl 1
18	0x12	DC2	Device Cntrl 2
19	0x13	DC3	Device Cntrl 3
20	0x14	DC4	Device Cntrl 4
21	0x15	NAK	Non Acknowledge
22	0x16	SYN	Synchronous Idle
23	0x17	ETB	End Transmit Block
24	0x18	CAN	Cancel
25	0x19	EM	End of Medium
26	0x1A	SUB	Substitute
27	0x1B	ESC	Escape
28	0x1C	FS	File Separator
29	0x1D	GS	Group Separator
30	0x1E	RS	Record Separator
31	0x1F	US	Unit Separator
32	0x20	SPACE	
33	0x21	!	
34	0x22	"	
35	0x23	#	
36	0x24	\$	
37	0x25	%	
38	0x26	&	
39	0x27	'	
40	0x28	(	
41	0x29	)	
42	0x2A	*	

Dec	Hex	Char
43	0x2B	+
44	0x2C	,
45	0x2D	-
46	0x2E	.
47	0x2F	/
48	0x30	0
49	0x31	1
50	0x32	2
51	0x33	3
52	0x34	4
53	0x35	5
54	0x36	6
55	0x37	7
56	0x38	8
57	0x39	9
58	0x3A	:
59	0x3B	;
60	0x3C	<
61	0x3D	=
62	0x3E	>
63	0x3F	?
64	0x40	@
65	0x41	A
66	0x42	B
67	0x43	C
68	0x44	D
69	0x45	E
70	0x46	F
71	0x47	G
72	0x48	H
73	0x49	I
74	0x4A	J
75	0x4B	K
76	0x4C	L
77	0x4D	M
78	0x4E	N
79	0x4F	O
80	0x50	P
81	0x51	Q
82	0x52	R
83	0x53	S
84	0x54	T
85	0x55	U

Dec	Hex	Char
86	0x56	V
87	0x57	W
88	0x58	X
89	0x59	Y
90	0x5A	Z
91	0x5B	[
92	0x5C	\
93	0x5D	]
94	0x5E	^
95	0x5F	_
96	0x60	`
97	0x61	a
98	0x62	b
99	0x63	c
100	0x64	d
101	0x65	e
102	0x66	f
103	0x67	g
104	0x68	h
105	0x69	i
106	0x6A	j
107	0x6B	k
108	0x6C	l
109	0x6D	m
110	0x6E	n
111	0x6F	o
112	0x70	p
113	0x71	q
114	0x72	r
115	0x73	s
116	0x74	t
117	0x75	u
118	0x76	v
119	0x77	w
120	0x78	x
121	0x79	y
122	0x7A	z
123	0x7B	{
124	0x7C	
125	0x7D	}
126	0x7E	~
127	0x7F	Delete

