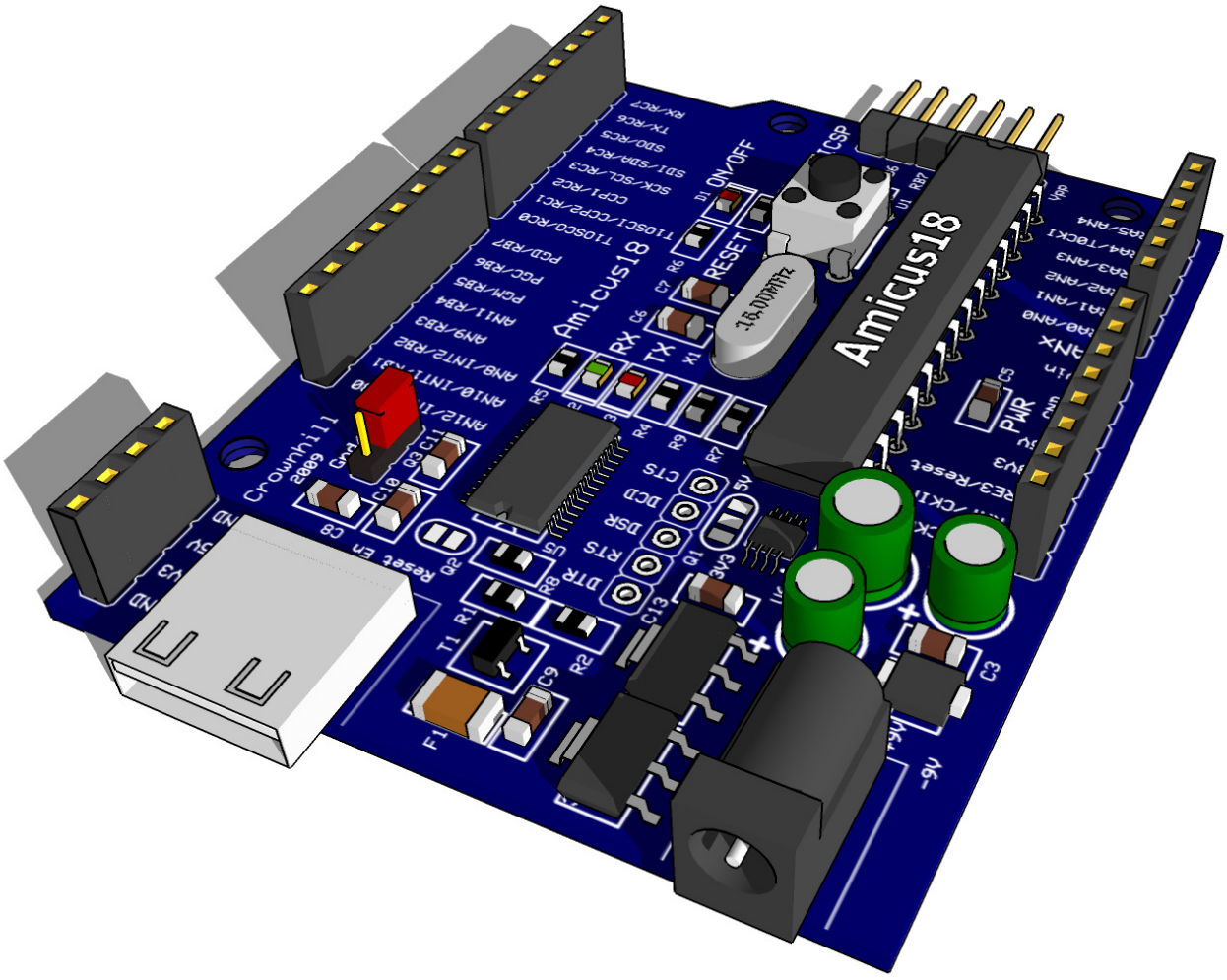
Proton Amicus18 BASIC Compiler Language Overview



Language Overview 0

Identifiers 7
Line Labels 7
Variables 8
Intuitive Variable Handling. 8
RAM space required. 9
Floating Point Math 10
Floating Point Format 10
Variables Used by the Floating Point Libraries. 11
Floating Point Rounding 12
Aliases 15
Finer points for variable handling. 16
Constants 17
Numeric Representations 18
Quoted String of Characters 18
Ports and other Registers 18
General Format 19
A Typical basic Program Layout 20
Inline Commands within Comparisons 21
Creating and using Arrays 22
Using Arrays in Expressions. 23
Arrays as Strings 23
Creating and using Strings 27
Loading a String Indirectly 30
Slicing a String 31
Creating and using Virtual Strings with Cdata 33
Creating and using Virtual Strings with Edata 35
String Comparisons 37
Relational Operators 40
Boolean Logic Operators 41
Math Operators 42
Add '+' 43
Subtract '-'. 43
Multiply '*' 44
Multiply High '**'. 45
Multiply Middle '*/'. 45
Divide '/'. 46
Modulus '//'. 47
Logical and '&'. 48
Logical or '|'. 48
Logical xor '^' 49
BitWise Shift Left '<<'. 49
BitWise Shift Right '>>'. 50
BitWise Complement '~' 50
Abs 51
Acos 52
Asin 53
Atan 54
Cos 55

Dcd 56
Dig (BASIC Stamp version) 56
Exp 57
ISqr 58
Log 59
Log10 60
Max 61
Min 61
Ncd 61
Pow 62
Rev 63
Sin 64
Sqr 65
Tan 66
Div32 67

Commands and Directives 68

Adin 72
Asm..EndAsm 74
Box 75
Branch 76
Break 77
Bstart 78
Bstop 79
Brestart 79
BusAck 79
BusNack 79
Busin 80
Busout 83
Button 87
Call 89
Cdata 90
Cerase 94
Circle 95
Clear 96
ClearBit 97
Cls 98
Config_Start – Config_End 99
Counter 105
Cread 106
Cursor 107
Cwrite 108
Dec 109

Declare 110
Misc Declares. 110
Trigonometry Declares. 113
Adin Declares. 113
Busin - Busout Declares. 114
Hbusin - Hbusout Declare. 114
Hserin, Hserout, HRsin and HRsout Declares. 115
Hpwm Declares. 116
Alphanumeric (Hitachi) LCD Print Declares. 116
Graphic LCD Declares. 117
Samsung KS0108 Graphic LCD specific Declares. 117
Toshiba T6963 Graphic LCD specific Declares. 118
Keypad Declare. 120
Rsin - Rsout Declares. 120
Serin - Serout Declare. 121
Shin - Shout Declare. 122
Crystal Frequency Declare. 123
DelayCs 124
DelayMs 125
DelayUs 126
Dig 127
Dim 128
DTMFout 131
Edata 132
End 137
Eread 138
Ewrite 139
For...Next...Step 140
FreqOut 142
GetBit 144
GoSub 145
What is a Stack? 147
Popping 148
GoTo 149
HbStart 150
HbStop 151
HbRestart 151
HbusAck 151
HbusNack 151
Hbusin 152
Hbusout 155
High 158
Hpwm 159
HRsin 160
HRsout 165
Hserin 169
Hserout 174
I2Cin 178
I2Cout 180
If..Then..ElseIf..Else..EndIf 183
Include 185
Inc 187
Inkey 188

Input....189
LCDread....190
LCDwrite....192
Len....194
Left$....195
Line....197
LineTo....198
LoadBit....199
LookDown....200
LookDownL....201
LookUp....202
LookUpL....203
Low....204
Lread....205
Lread8, Lread16, Lread32....207
Mid$....209
On GoTo....211
On GoSub....212
On_Hardware_Interrupt....213
Context Save....213
Context Restore....213
Managed Hardware Interrupts....215
On_Low_Interrupt....217
Context Save....219
Context Restore....219
Managed Low-Priority Hardware Interrupts.....220
Output....223
Org....224
Oread....225
Owrite....230
Pixel....232
Plot....233
Pop....235
Pot....237
Print....238
Using a Samsung KS0108 Graphic LCD....244
Using a Toshiba T6963 Graphic LCD....248
PulsIn....251
PulseOut....252
Push....253
Pwm....258
Random....259
RC5in....260
RCin....261
Repeat...Until....264
Return....265
Right$....267
Rsin....269
Rsout....274
Seed....279
Select..Case..EndSelect....280

Serin 282
Serout 289
Servo 297
SetBit 299
Set 300
Shin 301
Shout 303
Snooze 305
Sleep 306
SonyIn 308
Sound 309
Sound2 310
Stop 311
Strn 312
Str$ 313
Swap 315
Symbol 316
Toggle 317
ToLower 318
ToUpper 320
Toshiba_Command 322
Toshiba_UDG 326
UnPlot 328
Val 329
VarPtr 331
While...Wend 332
Xin 333
Xout 335

Using the Optimiser 337

Using the Preprocessor 339
Preprocessor Directives 339
Definition File 345

Built in Peripheral Macros 346
ADC macro introduction 347
BusyADC 347
CloseADC 347
ConvertADC 347
OpenADC 348
ReadADC 349
SetChanADC 350
SelChanConvADC 350
ADC_IntEnable 351
ADC_IntDisable 351
Timer macros Introduction 352
CloseTimer0 352
CloseTimer1 352
CloseTimer2 352
CloseTimer3 352
OpenTimer0 353
OpenTimer1 354
OpenTimer2 355
OpenTimer3 356
ReadTimer0 357
ReadTimer1 357
ReadTimer2 357
ReadTimer3 357
WriteTimer0 358
WriteTimer1 358
WriteTimer2 358
WriteTimer3 359
SetTmrCCPSrc 359
T3_OSC1EN_ON 360
T3_OSC1EN_OFF 360
SPI macros Introduction 361
CloseSPI 361
DataReadySPI 361
OpenSPI 362
ReadSPI 363
WriteSPI 363
Analogue Comparator macro Introduction 364
CloseComp1 364
CloseComp2 364
Comp1_IntEnable 364
Comp2_IntEnable 364
Comp1_IntDisable 364
Comp2_IntDisable 364
OpenComp1 365
OpenComp2 366
Hardware PWM macro Introduction 367
CloseAnalog1 367
CloseAnalog2 367
OpenAnalog1 367
OpenAnalog2 368
WriteAnalog1 368
WriteAnalog2 370

Protected Compiler Words 372

Identifiers

An identifier is a technical term for a name. Identifiers are used for line labels, variable names, and constant aliases. An identifier is any sequence of letters, digits, and underscores, although it must not start with a digit. Identifiers are not case sensitive, therefore label, LABEL, and Label are all treated as equivalent. And while labels might be any number of characters in length, only the first 32 are recognised.

Line Labels

In order to mark statements that the program may wish to reference with the **GoTo**, **Call**, or **GoSub** commands, the compiler uses line labels. Unlike many older BASICs, the compiler does not allow or require line numbers and doesn't require that each line be labelled. Instead, any line may start with a line label, which is simply an identifier followed by a colon ':'.

```
Label:
   Hrsout "Hello World"
   GoTo Label
```


Variables

Variables are where temporary data is stored in a BASIC program. They are created using the **Dim** keyword. Choosing the correct size variable for a specific task is important. Variables may be Bits, Bytes, Words, Dwords or Floats.

Space for each variable is automatically allocated in the microcontroller's RAM area. The format for creating a variable is as follows:

```
Dim Label as Size
```

Label is any identifier, (excluding keywords). Size is Bit, Byte, Word, Dword or Float. Some examples of creating variables are:

```
Dim Dog as Byte          ' Create an 8-bit unsigned variable (0 to 255)
Dim Cat as Bit           ' Create a single bit variable (0 or 1)
Dim Rat as Word          ' Create a 16-bit unsigned variable (0 to 65535)
Dim Large_Rat as Dword   ' Create a 32-bit signed variable (-2147483648 to
                         ' +2147483647)
Dim Pointy_Rat as Float ' Create a 32-bit floating point variable
```

The number of variables available depends on the amount of RAM on a particular device and the size of the variables within the BASIC program. The compiler will create additional System variables for use when calculating complex equations, or more complex command structures. Especially if floating point calculations are carried out.

Intuitive Variable Handling.

The compiler handles its System variables intuitively, in that it only creates those that it requires. Each of the compiler's built in library subroutines i.e. **Print**, **Rsout** etc, require a certain amount of System RAM as internal variables.

Try the following program, and look at the RAM usage message on the bottom Status bar.

```
   Dim WordVar as Word                ' Create a Word variable i.e. 16-bits
Loop:
   High PortB.0                       ' Set bit 0 of PortB high
   For WordVar = 1 to 20000 : Next    ' Create delay without using library call
   Low PortB.0                        ' Set bit 0 of PortB high
   For WordVar = 1 to 20000 : Next    ' Create delay without using library call
   GoTo Loop                          ' Do it forever
```

Only two bytes of RAM were used, and those were the ones declared in the program as variable WordVar1.

The compiler will increase it's System RAM requirements as programs get larger, or more complex structures are used, such as complex expressions, inline commands used in conditions, Boolean logic used etc.

There are certain reserved words that cannot be used as variable names, these are the system variables used by the compiler.

The following reserved words should not be used as variable names, as the compiler will create these names when required:

PP0, PP0H, PP1, PP1H, PP2, PP2H, PP3, PP3H, PP4, PP4H, PP5, PP5H, PP6, PP6H, PP7, PP7H, PP8, PP9H, GEN, GENH, GEN2, GEN2H, GEN3, GEN3H, GEN4, GEN4H, GPR, BPF, BPFH.

RAM space required.

Each type of variable requires differing amounts of RAM memory for its allocation. The list below illustrates this.

- Float Requires 4 bytes of RAM.
- Dword Requires 4 bytes of RAM.
- Word Requires 2 bytes of RAM.
- Byte Requires 1 byte of RAM.
- Bit Requires 1 byte of RAM for every 8 Bit variables used.

Each type of variable may hold a different minimum and maximum value.

Float type variables may theoretically hold a value from -1e37 to +1e38, but because of the 32-bit architecture of the compiler, a maximum and minimum value should be thought of as -2147483646.999 to +2147483646.999 making this the most accurate of the variable family types. However, more so than Dword types, this comes at a price as Float calculations and comparisons will use more code space within the microcontroller. Use this type of variable sparingly, and only when strictly necessary. Smaller floating point values offer more accuracy.

Dword type variables may hold a value from -2147483648 to +2147483647 making this the largest of the variable family types. This comes at a price however, as Dword calculations and comparisons will use more code space within the microcontroller. Use this type of variable sparingly, and only when necessary.

Word type variables may hold a value from 0 to 65535, which is usually large enough for most applications. It still uses more memory, but not nearly as much as a Dword type.

Byte type variables may hold a value for 0 to 255, and are the usual work horses of most programs. Code produced for Byte sized variables is very low compared to Word, Float, or Dword types, and should be chosen if the program requires faster, or more efficient operation.

Bit type variables may hold a 0 or a 1. These are created 8 at a time, therefore declaring a single Bit type variable in a program will not save RAM space, but it will save code space, as Bit type variables produce the most efficient use of code for comparisons etc.

Dimensioning a variable with the text **Symbol** following it will ensure that the variable is created in the Access RAM area of the microcontroller, making the variable bankless. However, this only occupies the first 96 bytes of RAM so there is a limit to the amount of system variables that can be created. **Bit** type variables cannot be declared as system types, nor can **Byte** or **Word Arrays**.

See also : **Aliases, Arrays, Dim, Symbol, Floating Point Math.**

Floating Point Math

The compiler can perform 32 x 32 bit IEEE 754 'Compliant' Floating Point calculations.

Declaring a variable as Float will enable floating point calculations on that variable.

```
Dim FloatVar as Float
```

To create a floating point constant, add a decimal point. Especially if the value is a whole number.

```
Symbol PI = 3.14    ' Create an obvious floating point constant

Symbol FlNum = 5.0 ' Create a floating point value of a whole number
```

Please note.
Floating point arithmetic is not the utmost in accuracy, it is merely a means of compressing a complex or large value into a small space (4 bytes in the compiler's case). Perfectly adequate results can usually be obtained from correct scaling of integer variables, with an increase in speed and a saving of RAM and code space. 32 bit floating point math is extremely microcontroller intensive since the microcontroller is only an 8 bit processor. It also consumes large amounts of RAM, and code space for its operation, therefore always use floating point sparingly, and only when strictly necessary.

Floating Point Format

The Proton Amicus18 compiler uses the Microchip variation of IEEE 754 floating point format. The differences to standard IEEE 745 are minor, and well documented in Microchip application note AN575 (downloadable from www.microchip.com).

Floating point numbers are represented in a modified IEEE-754 format. This format allows the floating-point routines to take advantage of the microcontroller's architecture and reduce the amount of overhead required in the calculations. The representation is shown below compared to the IEEE-754 format: where s is the sign bit, y is the lsb of the exponent and x is a placeholder for the mantissa and exponent bits. The compiler's floating point is big-endian.

The two formats may be easily converted from one to the other by manipulation of the *Exponent* and *Mantissa 0* bytes. The following assembly code shows an example of this operation.

Format	Exponent	Mantissa 0	Mantissa 1	Mantissa 2
IEEE-754	sxxx xxxx	yxxx xxxx	xxxx xxxx	xxxx xxxx
Microchip	xxxx xxxy	sxxx xxxx	xxxx xxxx	xxxx xxxx

IEEE-754 to Microchip

```
Rlf MANTISSA0,f
Rlf EXPONENT,f
Rrf MANTISSA0,f
```

Microchip to IEEE-754

```
Rlf MANTISSA0,f
Rrf EXPONENT,f
Rrf MANTISSA0,f
```


The conversion process can be streamlined by using the preprocessor:

```
' Microchip 32-bit Floating Point to IEEE754 conversion
$define FloatToIEEE754(a) rol a.Byte1 : ror a.Byte0 : ror a.Byte1
' IEEE754 to Microchip 32-bit Floating Point conversion
$define IEEE754ToFloat(a) rol a.Byte1 : rol a.Byte0 : ror a.Byte1
```

To use the above defines place the variable requiring altering as the parameter. Note that the variable itself will be altered:

```
' Convert Microchip 32-bit Floating Point to IEEE754
FloatToIEEE754(FloatVar)
```

Variables Used by the Floating Point Libraries.

Several 8-bit RAM registers are used by the floating point math routines to hold the operands and results of floating point operations. Since there may be two operands required for a floating point operation (such as multiplication or division), there are two sets of exponent and mantissa registers reserved (A and B). For argument A, PP_AARGHHH holds the exponent and PP_AARGHH, PP_AARGH and PP_AARG hold the mantissa. For argument B, PP_BARGHHH holds the exponent and PP_BARGHH, PP_BARGH and PP_BARG hold the mantissa.

Floating Point Example Programs.

```
' Multiply two floating point values

  Dim FloatVar as Float
  Symbol FlNum = 1.234    ' Create a floating point constant value

  FloatVar = FlNum * 10
  HRsout Dec FloatVar, 13


' Add two floating point variables

  Dim FloatVar as Float
  Dim FloatVar1 as Float
  Dim FloatVar2 as Float

  FloatVar1 = 1.23
  FloatVar2 = 1000.1
  FloatVar = FloatVar1 + FloatVar2
  HRsout Dec FloatVar, 13
```



```
' A digital voltmeter, using the on-board 10-bit ADC
  Include "ADC.inc"            ' Load the AD macros into the program
  Dim Raw as Word
  Dim Volts as Float
  Symbol Quanta = 3.3 / 1023   ' Calculate the quantising value
' Open the ADC:
'               Fosc to Fosc / 32
'               Right justified for 10-bit operation
'               Tad value of 2
'               Vref+ at Vcc : Vref- at Gnd
'               Make AN0 an analogue input
'
  OpenADC(ADC_FOSC_32 & ADC_RIGHT_JUST & ADC_2_TAD, ADC_REF_VDD_VSS, ADC_1ANA)
  While 1 = 1
    Raw = Adin 0
    Volts = Raw * Quanta
    HRsout Dec2 Volts, "V\r"
    DelayMs 400
  Wend
```

Notes.
Floating point expressions containing more than 3 operands are not supported.

Any expression that contains a floating point variable or value will be calculated as a floating point. Even if the expression also contains a Byte, Word, or Dword value or variable. If the assignment variable is a Byte, Word, or Dword variable, but the expression is of a floating point nature. Then the floating point result will be converted into an integer.

```
  Dim DwordVar as Dword
  Dim FloatVar as Float
  Symbol PI = 3.14
  FloatVar = 10
  DwordVar = FloatVar + PI    ' Calc will result 13.14,reduced to integer 13
  HRsout Dec DwordVar,13      ' Display the integer result 13
  Stop
```

For a more in-depth explanation of floating point, download the Microchip application notes AN575, and AN660. These can be found at www.microchip.com.

Floating Point Rounding

Assigning a floating point variable to an integer type will be rounded to the nearest value by default. For example:

```
  FloatVar = 3.9
  DwordVar = FloatVar
```

The variable DwordVar will hold the value of 4.

This behaviour can be altered by issuing the Float_Rounding = Off declare before the conversion takes place. For example:

```
  Declare Float_Rounding = Off    ' Disable Floating Point Rounding
  FloatVar = 3.9                  ' Load FloatVar with the value of 3.9
  DwordVar = FloatVar             ' Truncate 3.9 into FloatVar
```

The variable DwordVar will hold the value of 3. i.e truncated

The Float_Rounding setting will be remembered, as none of the compiler's floating point library routines alter it. However, remember that Floating Point rounding will effect Addition, Subtraction, Division, and Multiplication accuracy. It is therefore recommended to re-enable rounding after it has been disabled.

```
Declare Float_Rounding = Off    ' Disable Floating Point Rounding
FloatVar = 3.9                  ' Load FloatVar with the value of 3.9
DwordVar = FloatVar             ' Truncate 3.9 into FloatVar
Declare Float_Rounding = On     ' Enable Floating Point Rounding
```

Note that the Float_Rounding declare will not effect loading a floating point constant value into an integer. This will always be truncated. For example:

```
WordVar = 3.9
```

The variable WordVar will contain the value 3.

Floating Point Exception Flags

The floating point exception flags are accessible from within the BASIC program via the system variable _FP_FLAGS. This must be brought into the BASIC program for the code to recognise it:

```
Dim _FP_FLAGS as Byte System
```

The exceptions are:

```
_FP_FLAGS.1      ' Floating point overflow
_FP_FLAGS.2      ' Floating point underflow
_FP_FLAGS.3      ' Floating point divide by zero
_FP_FLAGS.5      ' Domain error exception
```

The bit that enables/disables floating point rounding to the nearest value is also contained within the _FP_FLAGS variable:

```
_FP_FLAGS.6    ' Floating point rounding
               ' 0 = Truncation
               ' 1 = Unbiased Rounding to Nearest LSB
```

The exception bits can be aliased for more readability within the program:

```
Symbol FpOverflow    = _FP_FLAGS.1   ' Floating point overflow
Symbol FpUnderFlow   = _FP_FLAGS.2   ' Floating point underflow
Symbol FpDiv0        = _FP_FLAGS.3   ' Floating point divide by zero
Symbol FpDomainError = _FP_FLAGS.5   ' Domain error exception
Symbol FpRounding    = _FP_FLAGS.6   ' Floating point rounding
' Setting FpRounding = 0: Truncation
' Setting FpRounding = 1: Unbiased rounding to nearest LSB
```

After an exception is detected and handled in the program, the exception bit should be cleared so that new exceptions can be detected, however, exceptions can be ignored because new operations are not affected by old exceptions.

More Accurate Display or Conversion of Floating Point values.

By default, the compiler uses a relatively small routine for converting floating point values to decimal, ready for **Rsout**, **Print**, **Str$** etc. However, because of its size, it is only capable of converting relatively small values. i.e. approx 6 digits of accuracy. In order to produce a more accurate result, the compiler needs to use a larger routine. This is implemented by using a **Declare**:

```
Declare Float_Display_Type = Fast or Standard
```

Using the Fast model for the above declare will trigger the compiler into using the more accurate floating point to decimal routine. Note that even though the routine is larger than the standard converter, it operates much faster.

The compiler defaults to Standard if the **Declare** is not issued in the BASIC program.

See also : **Dim, Symbol, Aliases, Arrays, Constants.**

Aliases

The **Symbol** directive is the primary method of creating an alias, however **Dim** can also be used to create an alias to a variable. This is extremely useful for accessing the separate parts of a variable.

```
Dim Fido as Dog             ' Fido is another name for Dog
Dim Mouse as Rat.LowByte    ' Mouse is the first byte (low byte) of word Rat
Dim Tail as Rat.HighByte    ' Tail is the second byte(high byte) of word Rat
Dim Flea as Dog.0           ' Flea is bit-0 of Dog
```

There are modifiers that may also be used with variables. These are **HighByte**, **LowByte**, **Byte0**, **Byte1**, **Byte2**, **Byte3**, **Word0**, and **Word1**.

Word0, **Word1**, **Byte2**, and **Byte3** may only be used in conjunction with a 32-bit Dword type variable.

HighByte and **Byte1** are one and the same thing, when used with a Word type variable, they refer to the High byte of a Word type variable:

```
Dim WordVar as Word          ' Create a Word sized variable
Dim WordVar_Hi as WordVar.HighByte
' WordVar_Hi now represents the High Byte of variable WordVar
```

Variable WordVar_Hi is now accessed as a Byte sized type, but any reference to it actually alters the high byte of WordVar.

However, if **Byte1** is used in conjunction with a Dword type variable, it will extract the second byte. **HighByte** will still extract the high byte of the variable, as will **Byte3**.

The same is true of **LowByte** and **Byte0**, but they refer to the Low Byte of a Word type variable:

```
Dim WordVar as Word          ' Create a Word sized variable
Dim WordVar_Lo as WordVar.LowByte
' WordVar_Lo now represents the Low Byte of variable WordVar
```

Variable WordVar_Lo is now accessed as a Byte sized type, but any reference to it actually alters the low byte of WordVar.

The modifier **Byte2** will extract the 3rd byte from a 32-bit Dword type variable, as an alias. Likewise **Byte3** will extract the high byte of a 32-bit variable.

```
Dim DwordVar as Dword          ' Create a 32-bit variable named DwordVar
Dim Part1 as DwordVar.Byte0    ' Alias Part1 to the low byte of DwordVar
Dim Part2 as DwordVar.Byte1    ' Alias Part2 to the 2nd byte of DwordVar
Dim Part3 as DwordVar.Byte2    ' Alias Part3 to the 3rd byte of DwordVar
Dim Part4 as DwordVar.Byte3    ' Alias Part3 to the high (4th) byte of DwordVar
```

The **Word0** and **Word1** modifiers extract the low word and high word of a Dword type variable, and is used the same as the Byte*n* modifiers.

```
Dim DwordVar as Dword          ' Create a 32-bit variable named DwordVar
Dim Part1 as DwordVar.Word0    ' Alias Part1 to the low word of DwordVar
Dim Part2 as DwordVar.Word1    ' Alias Part2 to the high word of DwordVar
```


RAM space for variables is allocated within the microcontroller in the order that they are placed in the BASIC code. For example:

```
Dim Var1 as Byte
Dim Var2 as Byte
```

Places Var1 first, then Var2:

```
Var1 equ n
Var2 equ n
```

This means that the first 95 variables will always be in BankA (Access RAM).

Finer points for variable handling.

The position of the variable within Banks is usually of little importance if BASIC code is used, however, if assembler routines are being implemented, always assign any variables used within them first.

Problems may also arise if a Word, or Dword variable crosses a Bank boundary. If this happens, a warning message will be displayed in the error window. Most of the time, this will not cause any problems, however, to err on the side of caution, try and ensure that Word, or Dword type variables are fully inside a Bank. This is easily accomplished by placing a dummy Byte variable before the offending Word, or Dword type variable, or relocating the offending variable within the list of **Dim** statements.

Word type variables have a low byte and a high byte. The high byte may be accessed by simply adding the letter H to the end of the variable's name. For example:

```
Dim WordVar as Word
```

Will produce the assembler code:

```
WordVar equ n
WordVarH equ n
```

To access the high byte of variable WordVar, use:

```
WordVarH = 1
```

This is especially useful when assembler routines are being implemented, such as:

```
Movlw 1
Movwf WordVarH      ' Load the high byte of WordVar with 1
```

Dword type variables have a low, mid1, mid2, and hi byte. The high byte may be accessed by using **Byte0**, **Byte1**, **Byte2**, or **Byte3**.. For example:

```
Dim DwordVar as Dword
```

To access the high byte of variable DwordVar, use:

```
DwordVar.Byte3 = 1
```


Constants

It can be more informative to use a constant name instead of a constant number. Once a constant is declared, it cannot be changed later, hence the name 'constant'.

The **Symbol** directive provides a method for aliasing variables and constants. **Symbol** cannot be used to create a variable.

Constants declared using **Symbol** do not use any RAM within the Amicus18 Hardware.

```
Symbol Cat = 123
Symbol Tiger = Cat              ' Tiger now holds the value of Cat
Symbol Mouse = 1                ' Same as Dim Mouse as 1
Symbol TigOuse = Tiger + Mouse  ' Add Tiger to Mouse to make Tigouse
```

Floating point constants may also be created using **Symbol** by simply adding a decimal point to a value.

```
Symbol PI = 3.14    ' Create a floating point constant named PI
Symbol FlNum = 5.0  ' Create a floating point constant holding the value 5
```

Floating point constant can also be created using expressions.

```
' Create a floating point constant holding the result of the expression
  Symbol Quanta = 3.3 / 1024
```

If a variable or register's name is used in a constant expression then the variable's or register's address will be substituted, not the value held in the variable or register:

```
Symbol Con = (PORTA + 1)   ' Con will hold the value 6 (5+1)
```

Symbol is also useful for aliasing Ports and Registers:

```
Symbol LED = PORTA.1       ' LED now references bit-1 of PortA
Symbol T0IF = INTCON.2     ' T0IF now references bit-2 of INTCON register
```

The equal sign between the Constant's name and the alias value is optional:

```
Symbol LED PORTA.1         ' Same as Symbol LED = PortA.1
```


Numeric Representations

The compiler recognises several different numeric representations:

Binary is prefixed by %. i.e. %0101
Hexadecimal is prefixed by $ or 0x. i.e. $0A , 0x0A
Character byte is surrounded by quotes. i.e. "a" represents a value of 97
Decimal values need no prefix.
Floating point is created by using a decimal point. i.e. 3.14

Quoted String of Characters

A Quoted String of Characters contains one or more characters (maximum 250) and is delimited by double quotes. Such as "Hello World"

The compiler also supports a subset of C language type formatters within a quoted string of characters. These are:

\a	Bell (alert) character	$07
\b	Backspace character	$08
\f	Form feed character	$0C
\n	New line character	$0A
\r	Carriage return character	$0D
\t	Horizontal tab character	$09
\v	Vertical tab character	$0B
\\	Backslash	$5C
\"	Double quote character	$22

Example:

```
Hrsout "HELLO WORLD\n\r"
```

Strings are usually treated as a list of individual character values, and are used by commands such as **Print**, **Rsout**, **Busout**, **Ewrite** etc. And of course, String variables.

Null Terminated

Null is a term used in computer languages for zero. So a null terminated String is a collection of characters followed by a zero in order to signify the end of characters. For example, the string of characters "HELLO", would be stored as:

```
"H" , "E" , "L" , "L" , "O" , 0
```

Notice that the terminating null is the value 0 not the character "0".

Ports and other Registers

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

```
PORTA = %01010101        ' Write value to PortA

Var1 = WordVar * PORTA   ' Multiply WordVar with the contents of PortA
```


The compiler can also combine 16-bit registers such as TMR1L into a Word type variable. Which makes loading and reading these registers simple:

```
' Combine TMR1L, and TMR1H into Word variable wTimer1
   Dim wTimer1 as TMR1L.Word

   wTimer1 = 12345       ' Load TMR1 with value 12345
or
   WordVar1 = wTimer1    ' Load WordVar1 with contents of TMR1
```

The **.Word** extension links registers TMR1L and TMR1H.

Any SFR that can hold a 16-bit result can be assigned as a Word type variable:

```
' Combine ADRESL, and ADRESH into Word variable wAD_Result
   Dim wAD_Result as ADRESL.Word
' Combine PRODL, and PRODH into Word variable wMulProd
   Dim wMulProd as PRODL.Word
```

General Format

The compiler is not case sensitive, except when processing string constants such as "hello".

Multiple instructions and labels can be combined on the same line by separating them with colons ':'.

The examples below show the same program as separate lines and as a single-line:

Multiple-line version:

```
   TRISB = %00000000     ' Make all pins on PortB outputs
   For Var1 = 0 to 100   ' Count from 0 to 100
     PORTB = Var1        ' Make PortB = count (Var1)
   Next                  ' Continue counting until 100 is reached
```

Single-line version:

```
   TRISB = %00000000 : For Var1 = 0 to 100 : PORTB = Var1 : Next
```

Do not try to cram as much as you can on a single line, as the compiler has a line limit of 255 chacarters.

Line Continuation Character '_'

Lines that are too long to display, may be split using the continuation character '_'. This will direct the continuation of a command to the next line. It's use is only permitted after a comma delimiter:

```
   Var1 = LookUp Var2,[1,2,3,_
                       4,5,6,7,8]
or
   Print At 1,1,_
   "HELLO WORLD",_
   Dec Var1,_
   Hex Var2
```


A Typical basic Program Layout

The compiler is very flexible, and will allow most types of constant, declaration, or variable to be placed anywhere within the BASIC program. However, it may not produce the correct results, or an unexpected syntax error may occur due to a variable being declared after it is supposed to be used.

The recommended layout for a program is shown below.

```
{
   Declares
}
{
   Includes
}
{
   Constants and Variables
}

   GoTo Main                          ' Jump over the subroutines (if any)

{
   Subroutines go here
}
{
   Main:
   Main Program code goes here
}
```

For example:

```
' Adjust the baud rate for Hrsout/Hrsin
   Declare Hserial_Baud = 9600
'----------------------------------------
' Load the ADC include file (if required)
   Include "ADC.inc"
'----------------------------------------
' Define Variables
   Dim WordVar as Word                ' Create a Word size variable
'----------------------------------------
' Define Constants and/or aliases
   Symbol Value = 10                  ' Create a constant
'----------------------------------------
   GoTo Main                          ' Jump over the subroutine/s (if any)
'----------------------------------------
' Simple Subroutine
AddIt:
   WordVar = WordVar + Value          ' Add the constant to the variable
   Return                             ' Return from the subroutine
'----------------------------------------
' Main Program Code
Main:
   WordVar = 10                       ' Pre-load the variable
   GoSub AddIt                        ' Call the subroutine
   Hrsout Dec WordVar, 13             ' Display the resault on the serial terminal
```

Of course, it depends on what is within the include file as to where it should be placed within the program, but the above outline will usually suffice. Any include file that requires placing within a certain position within the code should be documented to state this fact.

Inline Commands within Comparisons

A very useful addition to the compiler is the ability to mix most *inline* commands into comparisons. For example:

Adin, **Busin**, **Counter**, **Dig**, **Eread**, **Hbusin**, **Inkey**, **LCDread**, **LookDown**, **LookDownL**, **LookUp**, **LookUpL**, **Pixel**, **Pot**, **PulsIn**, **Random**, **Shin**, **Rcin**, **Rsin** etc.

All these commands may be used in an **If**-**Then**, **Select**-**Case**, **While**-**Wend**, or **Repeat**-**Until** structure. For example, with the previous versions of the compiler, to read a key using the **Inkey** command required a two stage process:

```
Var1 = Inkey
If Var1 = 12 Then { do something }
```

Now, the structure:

```
If Inkey = 12 Then { do something }
```

is perfectly valid. And so is:

```
If Adin 0 = 1020 Then { do something }   ' Test the ADC from channel 0
```

The new structure of the in-line commands does not always save code space, however, it does make the program easier to write, and a lot easier to understand, or debug if things go wrong.

The **LookUp**, **LookUpL**, **LookDown**, and **LookDownL** commands may also use another in-line command instead of a variable. For example, to read and re-arrange a key press from a keypad:

```
KEY = LookUp Inkey, [1,2,3,15,4,5,6,14,7,8,9,13,10,0,11,12,255]
```

In-line command differences do not stop there. They may now also be used for display purposes in the **Rsout**, **Serout**, **HRsout**, and **Print** commands:

```
Label:
   Rsout LookUp Inkey, [1,2,3,15,4,5,6,14,7,8,9,13,10,0,11,12,255]
   GoTo Label
```

How's that for a simple serial keypad program. Or:

```
While 1 = 1 : Print Rsin : Wend
```

Believe it or not, the above single line of code is a simple serial LCD controller. Accepting serial data through the **Rsin** command, and displaying the data with the **Print** command.

Note.
Inline commands cannot be nested too deeply because the compiler was not designed for such situations.

Creating and using Arrays

The Proton Amicus18 compiler supports multi part Byte, and Word variables named arrays. An array is a group of variables of the same size (8-bits wide, or 16-bits wide), sharing a single name, but split into numbered cells, called elements.

An array is defined using the following syntax:

```
Dim Name[length] as Byte

Dim Name[length] as Word
```

where Name is the variable's given name, and the new argument, [length], informs the compiler how many elements you want the array to contain. For example:

```
Dim MyArray[10] as Byte    ' Create a 10 element byte array
Dim MyArray[10] as Word    ' Create a 10 element word array
```

The compiler will allow up to 256 elements within a Byte array, and 128 elements in a Word array.

Once an array is created, its elements may be accessed numerically. Numbering starts at 0 and ends at n-1. For example:

```
MyArray[3] = 57
HRsout "MyArray[3] = ", Dec MyArray[3],13
```

The above example will access the fourth element in the Byte array and display "MyArray[3] = 57" on the LCD. The true flexibility of arrays is that the index value itself may be a variable. For example:

```
Dim MyArray[10] as Byte           ' Create a 10-byte array
Dim Index as Byte                 ' Create a normal Byte variable
For Index = 0 to 9                ' Repeat with Index= 0,1,2...9
  MyArray[Index] = Index * 10     ' Write Index*10 to each element of array
Next
For Index = 0 to 9                ' Repeat with Index= 0,1,2...9
   HRsout Dec MyArray[Index],13   ' Show the contents of each element
   DelayMs 500                    ' Wait long enough to view the values
Next
```

If the above program is run, 10 values will be displayed, counting from 0 to 90 i.e. Index * 10.

A word of caution regarding arrays: If you're familiar with other languages and have used their arrays, you may have run into the "subscript out of range" error. Subscript is simply another term for the index value. It is considered 'out-of range' when it exceeds the maximum value for the size of the array.

For example, in the example above, MyArray is a 10-element array. Allowable index values are 0 through 9. If your program exceeds this range, the compiler will not respond with an error message. Instead, it will access the next RAM location past the end of the array.

If you are not careful about this, it can cause all sorts of subtle anomalies, as previously loaded variables are overwritten. It's up to the programmer (you!) to help prevent this from happening.

Even more flexibility is allowed with arrays because the index value may also be an expression.

```
Dim MyArray[10] as Byte           ' Create a 10-byte array
Dim Index as Byte                 ' Create a normal Byte variable
For Index = 0 to 8                ' Repeat with Index= 0,1,2...8
MyArray[Index + 1] = Index * 10   ' Write Index*10 to each element of array
Next
For Index = 0 to 8                ' Repeat with Index= 0,1,2...8
  HRsout Dec MyArray[Index + 1], 13   ' Show the contents of elements
  DelayMs 500                     ' Wait long enough to view the values
Next
```

The expression within the square braces should be kept simple, and arrays are not allowed as part of the expression.

Using Arrays in Expressions.

Of course, arrays are allowed within expressions themselves. For example:

```
Dim MyArray[10] as Byte    ' Create a 10-byte array
Dim Index as Byte          ' Create a Byte variable
Dim Var1 as Byte           ' Create another Byte variable
Dim Result as Byte         ' Create a variable to hold result of expression
Index = 5                  ' And Index now holds the value 5
Var1 = 10                  ' Variable Var1 now holds the value 10
MyArray[Index] = 20        ' Load the 6th element of MyArray with value 20
Result = (Var1 * MyArray[Index]) / 20 ' Do a simple expression
HRsout Dec Result, 13      ' Display result of expression
```

The previous example will display 10 on the serial terminal, because the expression reads as:

(10 * 20) / 20

Var1 holds a value of 10, MyArray[Index] holds a value of 20, these two variables are multiplied together which will yield 200, then they're divided by the constant 20 to produce a result of 10.

Arrays as Strings

Arrays may also be used as simple strings in certain commands, because after all, a string is simply a byte array used to store text.

For this, the **Str** modifier is used.

The commands that support the **Str** modifier are:

Busout – **Busin**
Hbusout – **Hbusin**
HRsout – **Hrsin**
Owrite – **Oread**
Rsout – **Rsin**
Serout – **Serin**
Shout – **Shin**
Print

The **Str** modifier works in two ways, it outputs data from a pre-declared array in commands that send data i.e. **Rsout**, **Print** etc, and loads data into an array, in commands that input information i.e. **Rsin**, **Serin** etc. The following examples illustrate the **Str** modifier in each compatible command.

Using Str with the Busin and Busout commands.

Refer to the sections explaining the **Busin** and **Busout** commands.

Using Str with the Hbusin and Hbusout commands.

Refer to the sections explaining the **Hbusin** and **Hbusout** commands.

Using Str with the Rsin command.

```
  Dim Array1[10] as Byte      ' Create a 10-byte array named Array1
  Rsin Str Array1             ' Load 10 bytes of data directly into Array1
```

Using Str with the Rsout command.

```
  Dim Array1[10] as Byte      ' Create a 10-byte array named Array1
  Rsout Str Array1            ' Send 10 bytes of data directly from Array1
```

Using Str with the HRsin and HRsout commands.

Refer to the sections explaining the **HRsout** and **HRsin** commands.

Using Str with the Shout command.

```
  Symbol DTA = PortA.0        ' Alias the two lines for the Shout command
  Symbol CLK = PortA.1
  Dim Array1[10] as Byte      ' Create a 10-byte array named Array1
' Send 10 bytes of data from Array1
  Shout DTA, CLK, MSBFIRST, [Str Array1]
```

Using Str with the Shin command.

```
  Symbol DTA = PortA.0        ' Alias the two lines for the Shin command
  Symbol CLK = PortA.1
  Dim Array1[10] as Byte      ' Create a 10-byte array named Array1
' Load 10 bytes of data directly into Array1
  Shin DTA, CLK, MSBPRE, [Str Array1]
```

Using Str with the Print command.

```
  Dim Array1[10] as Byte      ' Create a 10-byte array named Array1
  Print Str Array1            ' Send 10 bytes of data directly from Array1
```

Using Str with the Serout and Serin commands.

Refer to the sections explaining the **Serin** and **Serout** commands.

Using Str with the Oread and Owrite commands.

Refer to the sections explaining the **Oread** and **Owrite** commands.

The **Str** modifier has two forms for variable-width and fixed-width data, shown below:

Str ByteArray ASCII string from ByteArray until byte = 0 (null terminated).

Or array length is reached.

Str ByteArray\n ASCII string consisting of n bytes from bytearray.

null terminated means that a zero (null) is placed at the end of the string of ASCII characters to signal that the string has finished.

The example below is the variable-width form of the **Str** modifier:

```
Dim MyArray[5] as Byte  ' Create a 5 element array
MyArray[0] = "A"        ' Fill the array with ASCII
MyArray[1] = "B"
MyArray[2] = "C"
MyArray[3] = "D"
MyArray[4] = 0          ' Add the null Terminator
HRsout Str MyArray      ' Display the string
```

The code above displays "ABCD" on the serial terminal. In this form, the **Str** formatter displays each character contained in the byte array until it finds a character that is equal to 0 (value 0, not ASCII "0"). Note: If the byte array does not end with 0 (null), the compiler will read and

output all RAM register contents until it cycles through all RAM locations for the declared length of the byte array.

For example, the same code as before without a null terminator is:

```
Dim MyArray[4] as Byte          ' Create a 4 element array
MyArray[0] = "A"                ' Fill the array with ASCII
MyArray[1] = "B"
MyArray[2] = "C"
MyArray[3] = "D"
HRsout Str MyArray              ' Display the string
```

The code above will display the whole of the array, because the array was declared with only four elements, and each element was filled with an ASCII character i.e. "ABCD".

To specify a fixed-width format for the **Str** modifier, use the form **Str** MyArray\n; where MyArray is the byte array and n is the number of characters to display, or transmit. Changing the **Print** line in the examples above to:

```
HRsout Str MyArray \ 2
```

would display "AB" on the serial terminal.

Str is not only used as a modifier, it is also a command, and is used for initially filling an array with data. The above examples may be re-written as:

```
Dim MyArray[5] as Byte        ' Create a 5 element array
Str MyArray = "ABCD", 0       ' Fill array with ASCII, and null terminate it
HRsout Str MyArray            ' Display the string
```


Strings may also be copied into other strings:

```
Dim String1[5] as Byte          ' Create a 5 element array
Dim String2[5] as Byte          ' Create another 5 element array
Str String1 = "ABCD", 0         ' Fill array with ASCII, and null terminate it
Str String2 = "EFGH", 0         ' Fill array with ASCII, null terminate it
Str String1 = Str String2       ' Copy String2 into String1
HRsout Str String1              ' Display the string
```

The above example will display "EFGH", because String1 has been overwritten by String2.

Using the **Str** command with **Busout**, **Hbusout**, **Shout**, and **Owrite** differs from using it with commands **Serout**, **Print**, **HRsout**, and **Rsout** in that, the latter commands are used more for dealing with text, or ASCII data, therefore these are null terminated.

The **Hbusout**, **Busout**, **Shout**, and **Owrite** commands are not commonly used for sending ASCII data, and are more inclined to send standard 8-bit bytes. Thus, a null terminator would cut short a string of byte data, if one of the values happened to be a 0. So these commands will output data until the length of the array is reached, or a fixed length terminator is used i.e. MyArray\n.

Creating and using Strings

The syntax to create a string is :

Dim String Name **as String** * String Length

String Name can be any valid variable name. See **Dim** .
String Length can be any value up to 255, allowing up to 255 characters to be stored.

The line of code below will create a String named ST that can hold 20 characters:

```
Dim ST as String * 20
```

Two or more strings can be concatenated (linked together) by using the plus (+) operator:

```
' Create three strings capable of holding 20 characters
  Dim DestString as String * 20
  Dim SourceString1 as String * 20
  Dim SourceString2 as String * 20

  SourceString1 = "HELLO "     ' Load String SourceString1 with the text HELLO
' Load String SourceString2 with the text WORLD
  SourceString2 = "WORLD"
' Add both Source Strings together. Place result into String DestString
  DestString = SourceString1 + SourceString2
```

The String DestString now contains the text "HELLO WORLD", and can be transmitted serially or displayed on an serial terminal:

```
HRsout DestString
```

The Destination String itself can be added to if it is placed as one of the variables in the addition expression. For example, the above code could be written as:

```
' Create a String capable of holding 20 characters
  Dim DestString as String * 20
' Create another String capable of holding 20 characters
  Dim SourceString as String * 20

  DestString = "HELLO "        ' Pre-load String DestString with the text HELLO
  SourceString = "WORLD"       ' Load String SourceString with the text WORLD
' Concatenate DestString with SourceString
  DestString = DestString + SourceString
  HRsout DestString  , 13      ' Display the result which is "HELLO WORLD"
```

Note that Strings cannot be subtracted, multiplied or divided, and cannot be used as part of a regular expression otherwise a syntax error will be produced.

It's not only other strings that can be added to a string, the functions **Cstr**, **Estr**, **Mid$**, **Left$**, **Right$**, **Str$**, **ToUpper**, and **ToLower** can also be used as one of variables to concatenate.

A few examples of using these functions are shown below:

Cstr Example

```
' Use Cstr function to place a code memory string into a RAM String variable

   Dim DestString as String * 20     ' Create String of 20 characters
   Dim SourceString as String * 20   ' Create another String
   SourceString = "HELLO "           ' Load the string with characters
   DestString = SourceString + Cstr CodeStr ' Concatenate the string
   HRsout DestString, 13             ' Display the result which is "HELLO WORLD"
   Stop
CodeStr:
   Cdata "WORLD" ,0
```

The above example is really only for demonstration because if a LABEL name is placed as one of the parameters in a string concatenation, an automatic (more efficient) **Cstr** operation will be carried out. Therefore the above example should be written as:

More efficient Example of above code

```
' Place a code memory string into String variable more efficiently than Cstr

' Create a String capable of holding 20 characters
   Dim DestString as String * 20
   Dim SourceString as String * 20     ' Create another String
   SourceString = "HELLO "             ' Load the string with characters
   DestString = SourceString + CodeStr ' Concatenate the string
   HRsout DestString, 13               ' Display the result which is "HELLO WORLD"
   Stop
CodeStr:
   Cdata "WORLD",0
```

A null terminated string of characters held in Data (on-board eeprom) memory can also be loaded or concatenated to a string by using the Estr function:

Estr Example

```
' Use the Estr function in order to place a
' Data memory string into a String variable
' Remember to place Edata before the main code
' so it's recognised as a constant value

   Dim DestString as String * 20     ' Create a String for 20 characters
   Dim SourceString as String * 20   ' Create another String

DataStr Edata "WORLD",0              ' Create a string in Data memory
   SourceString = "HELLO "           ' Load the string with characters
   DestString = SourceString + Estr DataStr ' Concatenate the strings
   HRsout DestString, 13             ' Display the result which is "HELLO WORLD"
```


Converting an integer or floating point value into a string is accomplished by using the **Str$** function:

Str$ Example

```
' Use the Str$ function in order to concatenate an integer value into a
' String variable

   Dim DestString as String * 30     ' Create a String capable for 30 chars
   Dim SourceString as String * 20   ' Create another String
   Dim WordVar1 as Word              ' Create a Word variable

   WordVar1 = 1234                   ' Load the Word variable with a value
   SourceString = "Value = "         ' Load the string with characters
   DestString = SourceString + Str$(Dec WordVar1)  ' Concatenate the string
   HRsout DestString, 13             ' Display the result which is "Value = 1234"
```

Left$ Example

```
' Copy 5 characters from the left of SourceString
' and add to a quoted character string

   Dim SourceString as String * 20   ' Create a String
   Dim DestString as String * 20     ' Create another String

   SourceString = "HELLO WORLD"      ' Load the source string with characters
   DestString = Left$(SourceString, 5) + " WORLD"
   HRsout DestString, 13             ' Display the result which is "HELLO WORLD"
```

Right$ Example

```
' Copy 5 characters from the right of SourceString
' and add to a quoted character string

   Dim SourceString as String * 20   ' Create a String
   Dim DestString as String * 20     ' Create another String

   SourceString = "HELLO WORLD"      ' Load the source string with characters
   DestString = "HELLO " + Right$(SourceString, 5)
   HRsout DestString, 13             ' Display the result which is "HELLO WORLD"
```

Mid$ Example

```
' Copy 5 characters from position 4 of SourceString
' and add to quoted character strings

   Dim SourceString as String * 20   ' Create a String
   Dim DestString as String * 20     ' Create another String

   SourceString = "HELLO WORLD"      ' Load the source string with characters
   DestString = "HEL" + Mid$(SourceString, 4, 5) + "RLD"
   HRsout DestString, 13             ' Display the result which is "HELLO WORLD"
```


Converting a string into uppercase or lowercase is accomplished by the functions **ToUpper** and **ToLower**:

ToUpper Example

```
' Convert the characters in SourceString to UPPER case

  Dim SourceString as String * 20     ' Create a String
  Dim DestString as String * 20       ' Create another String

  SourceString = "hello world"        ' Load source with lowercase characters
  DestString = ToUpper SourceString
  HRsout DestString, 13               ' Display the result which is "HELLO WORLD"
```

ToLower Example

```
' Convert the characters in SourceString to lower case

  Dim SourceString as String * 20     ' Create a String
  Dim DestString as String * 20       ' Create another String

  SourceString = "HELLO WORLD"        ' Load the string with uppercase characters
  DestString = ToLower SourceString
  HRsout DestString, 13               ' Display the result which is "hello world"
```

Loading a String Indirectly

If the Source String is a Byte, Word, Byte Array, Word Array or Float variable, the value contained within the variable is used as a pointer to the start of the Source String's address in RAM.

Example

```
' Copy SourceString into DestString using a pointer to SourceString

  Dim SourceString as String * 20     ' Create a String
  Dim DestString as String * 20       ' Create another String
' Create a Word variable to hold the address of SourceString
  Dim StringAddr as Word

  SourceString = "HELLO WORLD"        ' Load the source string with characters
' Locate the start address of SourceString in RAM
  StringAddr = VarPtr SourceString
  DestString = StringAddr             ' Source string into the destination string
  HRsout DestString, 13               ' Display the result, which will be "HELLO"
```


Slicing a String

Each position within the string can be accessed the same as a Byte Array by using square braces:

```
Dim SourceString as String * 20   ' Create a String

SourceString[0] = "H"      ' Place letter "H" as first character in the string
SourceString[1] = "E"      ' Place the letter "E" as the second character
SourceString[2] = "L"      ' Place the letter "L" as the third character
SourceString[3] = "L"      ' Place the letter "L" as the fourth character
SourceString[4] = "O"      ' Place the letter "O" as the fifth character
SourceString[5] = 0        ' Add a null to terminate the string

HRsout SourceString, 13    ' Display the string, which will be "HELLO"
```

The example above demonstrates the ability to place individual characters anywhere in the string. Of course, you wouldn't use the code above in an actual BASIC program.

A string can also be read character by character by using the same method as shown above:

```
Dim SourceString as String * 20     ' Create a String
Dim Var1 as Byte

SourceString = "HELLO"          ' Load the source string with characters
' Copy character 1 from the source string and place it into Var1
Var1 = SourceString[1]
HRsout Var1, 13                 ' Display character extracted from string ("E")
```

When using the above method of reading and writing to a string variable, the first character in the string is referenced at 0 onwards, just like a Byte Array.

The example below shows a more practical String slicing demonstration.

```
' Display a string's text by examining each character individually
Dim SourceString as String * 20   ' Create a String
Dim Charpos as Byte               ' Holds the position within the string

SourceString = "HELLO WORLD"      ' Load the source string with characters
Charpos = 0                       ' Start at position 0 within the string
Repeat                            ' Create a loop
  HRsout SourceString[Charpos]    ' Display character extracted from string
  Inc Charpos                     ' Move to next position within string
Until Charpos = Len SourceString  ' Keep looping until end of string found
Hrsout, 13
```


Notes

A word of caution regarding Strings: If you're familiar with interpreted BASICs and have used their String variables, you may have run into the "subscript out of range" error. This error occurs when the amount of characters placed in the string exceeds its maximum size.

For example, in the examples above, most of the strings are capable of holding 20 characters. If your program exceeds this range by trying to place 21 characters into a string only created for 20 characters, the compiler will not respond with an error message. Instead, it will access the next RAM location past the end of the String.

If you are not careful about this, it can cause all sorts of subtle anomalies, as previously loaded variables are overwritten. It's up to the programmer (you!) to help prevent this from happening by ensuring that the String in question is large enough to accommodate all the characters required, but not too large that it uses up too much precious RAM.

The compiler will help by giving a reminder message when appropriate, but this can be ignored if you are confident that the String is large enough.

See also : **Creating and using Virtual Strings with Cdata**
Creating and using Virtual Strings with Edata
Cdata, Len, Left$, Mid$, Right$
String Comparisons, Str$, ToLower, ToUpper, VarPtr.

Creating and using Virtual Strings with Cdata

Although writing to code memory too many times is unhealthy for the microcontroller, reading this memory is both fast, and harmless. Which offers an excellent form of data storage and retrieval, the **Cdata** command proves this, as it uses the mechanism of reading and storing in the microcontroller's flash memory.

Combining the unique features of the 'self modifying PICmicros ' with a string format, the compiler is capable of reducing the overhead of printing, or transmitting large amounts of text data. The **Cstr** modifier may be used in commands that deal with text processing i.e. **Print**, **Serout**, **HRsout**, and **Rsout** .

The **Cstr** modifier is used in conjunction with the **Cdata** command. The **Cdata** command is used for initially creating the string of characters:

```
String1: Cdata "HELLO WORLD", 0
```

The above line of code will create, in flash memory, the values that make up the ASCII text "HELLO WORLD", at address String1. Note the null terminator after the ASCII text.

null terminated means that a zero (null) is placed at the end of the string of ASCII characters to signal that the string has finished.

To display, or transmit this string of characters, the following command structure could be used:

```
HRsout Cstr String1
```

The label that declared the address where the list of **Cdata** values resided, now becomes the string's name. In a large program with lots of text formatting, this type of structure can save quite literally hundreds of bytes of valuable code space.

Try both these small programs, and you'll see that using **Cstr** saves a few bytes of code:

First the standard way of displaying text:

```
HRsout "HELLO WORLD\r"
HRsout "HOW ARE YOU?\r"
HRsout "I AM FINE!\r"
```

Now using the **Cstr** modifier:

```
HRsout Cstr TEXT1
HRsout Cstr TEXT2
HRsout Cstr TEXT3
Stop

TEXT1: Cdata "HELLO WORLD\r", 0
TEXT2: Cdata "HOW ARE YOU?\r", 0
TEXT3: Cdata "I AM FINE!\r", 0
```


Again, note the null terminators after the ASCII text in the **Cdata** commands. Without these, the micro-controller will continue to transmit data in an endless loop.

The term 'virtual string' relates to the fact that a string formed from the **Cdata** command cannot (rather should not) be written too, but only read from.

Not only label names can be used with the **Cstr** modifier, constants, variables and expressions can also be used that will hold the address of the **Cdata** 's label (a pointer). For example, the program below uses a Word size variable to hold 2 pointers (address of a label, variable or array) to 2 individual null terminated text strings formed by **Cdata** .

```
   Dim Address as Word          ' Pointer variable

   Address = String1            ' Point address to string 1
   HRsout Cstr Address          ' Display string 1
   Address = String2            ' Point Address to string 2
   HRsout Cstr Address, 13      ' Display string 2
   Stop

' Create the text to display
String1:
   Cdata "HELLO ", 0
String2:
   Cdata "WORLD", 0
```


Creating and using Virtual Strings with Edata

Combining the eeprom memory of the microcontroller with a string format, the compiler is capable of reducing the overhead of printing, or transmitting large amounts of text data using a memory resource that is very often left unused and ignored. The **Estr** modifier may be used in commands that deal with text processing i.e. **Print**, **Serout**, **HRsout**, and **Rsout** and String handling etc.

The **Estr** modifier is used in conjunction with the **Edata** command, which is used to initially create the string of characters:

```
String1 Edata "Hello World\r", 0
```

The above line of code will create, in eeprom memory, the values that make up the ASCII text "Hello World", at address String1 in Data memory. Note the null terminator after the ASCII text.

To display, or transmit this string of characters, the following command structure could be used:

```
HRsout Estr String1
```

The identifier that declared the address where the list of **Edata** values resided, now becomes the string's name. In a large program with lots of text formatting, this type of structure can save many bytes of valuable code space.

Try both these small programs, and you'll see that using **Estr** saves code space:

First the standard way of displaying text:

```
HRsout "Hello World\r"
HRsout "How are you?\r"
HRsout "I am fine!\r"
```

Now using the Estr modifier:

```
Text1 Edata "Hello World\r", 0
Text2 Edata "How are you?\r", 0
Text3 Edata "I am fine!\r", 0

HRsout Estr Text1
HRsout Estr Text2
HRsout Estr Text3
```

Again, note the null terminators after the ASCII text in the **Edata** commands. Without these, the micro-controller will continue to transmit data in an endless loop.

The term 'virtual string' relates to the fact that a string formed from the **Edata** command cannot (rather should not) be written to often, but can be read as many times as wished without causing harm to the device.

Not only identifiers can be used with the **Estr** modifier, constants, variables and expressions can also be used that will hold the address of the **Edata**'s identifier (a pointer). For example, the program below uses a Byte size variable to hold 2 pointers (address of a variable or array) to 2 individual null terminated text strings formed by **Edata** .

```
  Dim Address as Word             ' Pointer variable
' Create the text to display in eeprom memory
String1 Edata "Hello ", 0
String2 Edata "World", 0

  Address = String1               ' Point address to string 1
  HRsout Estr Address             ' Display string 1
  Address = String2               ' Point Address to string 2
  HRsout Estr Address, 13         ' Display string 2
```

Notes

Note that the identifying text *must* be located on the same line as the **Edata** directive or a syntax error will be produced. It must also not contain a postfix colon as does a line label or it will be treat as a line label. Think of it as an alias name to a constant.

Any **Edata** directives *must* be placed at the head of the BASIC program as is done with Symbols, so that the name is recognised by the rest of the program as it is parsed. There is no need to jump over **Edata** directives as you have to with **Cdata** or **Cdata**, because they do not occupy code memory, but reside in high Data memory.

String Comparisons

Just like any other variable type, String variables can be used within comparisons such as **If**-**Then**, **Repeat**-Until, and **While**-**Wend** . In fact, it's an essential element of any programming language. However, there are a few rules to obey because of the microcontroller's architecture.

Equal (=) or Not Equal (<>) comparisons are the only type that apply to Strings, because one String can only ever be equal or not equal to another String. It would be unusual (unless your using the C language) to compare if one String was greater or less than another.

So a valid comparison could look something like the lines of code below:

```
If String1 = String2 Then HRsout "EQUAL\r" : Else : HRsout "NOT EQUAL\r"
```

or

```
If String1 <> String2 Then HRsout "NOT EQUAL\r" : Else : HRsout "EQUAL\r"
```

But as you've found out if you read the Creating Strings section, there is more than one type of String in a microcontroller. There is a String variable, a code memory string, and a quoted character string .

Note that pointers to String variables are not allowed in comparisons, and a syntax error will be produced if attempted.

Starting with the simplest of string comparisons, where one string variable is compared to another string variable. The line of code would look similar to either of the two lines above.

Example 1

```
' Simple string variable comparison

' Create a String capable of holding 20 characters
  Dim String1 as String * 20
  Dim String2 as String * 20' Create another String

  String1 = "EGGS"              ' Pre-load String String1 with the text EGGS
  String2 = "BACON"             ' Load String String2 with the text BACON

  If String1 = String2 Then     ' Is String1 equal to String2 ?
    HRsout "EQUAL\r"            ' Yes. So display EQUAL on the serial terminal
  Else                          ' Otherwise…
    HRsout "NOT EQUAL\r"        ' Display not EQUAL on the serial terminal
  EndIf

  String2 = "EGGS"              ' Now make the strings the same as each other
  If String1 = String2 Then     ' Is String1 equal to String2 ?
    HRsout "EQUAL\r"            ' Yes. So display EQUAL on the serial terminal
  Else                          ' Otherwise…
    HRsout "NOT EQUAL\r"        ' Display not EQUAL on the serial terminal
  EndIf
```

The example above will display not Equal on the serial terminal because String1 contains the text "EGGS" while String2 contains the text "BACON", so they are clearly not equal.

The second line of the serial terminal will show Equal because String2 is then loaded with the text "EGGS" which is the same **as String**1, therefore the comparison is equal.

A similar example to the previous one uses a quoted character string instead of one of the String variables.

Example 2

```
' String variable to Quoted character string comparison

  Dim String1 as String * 20' Create a String of 20 characters

  String1 = "EGGS"              ' Pre-load String String1 with the text EGGS

  If String1 = "BACON" Then     ' Is String1 equal to "BACON" ?
    HRsout "EQUAL\r"            ' Yes. So display EQUAL on the serial terminal
  Else                          ' Otherwise…
    HRsout "NOT EQUAL\r"        ' Display NOT EQUAL on the serial terminal
  EndIf

  If String1 = "EGGS" Then      ' Is String1 equal to "EGGS" ?
    HRsout "EQUAL\r"            ' Yes. So display EQUAL on the serial terminal
  Else                          ' Otherwise…
    HRsout "NOT EQUAL\r"        ' Display NOT EQUAL on the serial terminal
  EndIf
```

The example above produces exactly the same results as example1 because the first comparison is clearly not equal, while the second comparison is equal.

Example 3

```
' Use a string comparison in a Repeat-Until loop

  Dim SourceString as String * 20   ' Create a String
  Dim DestString as String * 20     ' Create another String
  Dim Charpos as Byte               ' Character position within the strings

  Clear DestString                  ' Fill DestString with NULLs
  SourceString = "HELLO"            ' Load String SourceString with the text HELLO

  Repeat                            ' Create a loop
     ' Copy SourceString into DestString one character at a time
    DestString[Charpos] = SourceString[Charpos]
    Inc Charpos                     ' Move to the next character in the strings
  Until DestString = "HELLO"        ' Stop when DestString equal to text "HELLO"
  HRsout DestString, 13             ' Display DestString
```


Example 4

```
' Compare a string variable to a string held in code memory

   Dim String1 as String * 20   ' Create a String capable of 20 characters

   String1 = "BACON"            ' Pre-load String String1 with the text BACON
   If CodeString= "BACON" Then  ' Is CodeString equal to "BACON" ?
     HRsout "EQUAL\r"           ' Yes. So display EQUAL on the serial terminal
   Else                         ' Otherwise…
     HRsout "NOT EQUAL\r"       ' Display NOT EQUAL on the serial terminal
   EndIf

   String1 = "EGGS"             ' Pre-load String String1 with the text EGGS
   If String1 = CodeString Then ' Is String1 equal to CodeString ?
     HRsout "EQUAL\r"           ' Yes. So display EQUAL on the serial terminal
   Else                         ' Otherwise…
     HRsout "NOT EQUAL\r"       ' Display NOT EQUAL on the serial terminal
   EndIf
   Stop

CodeString:
Cdata "EGGS", 0
```

Example 5

```
' String comparisons using Select-Case

   Dim String1 as String * 20   ' Create a String capable of 20 characters

   String1 = "EGGS"             ' Pre-load String String1 with the text EGGS
   Select String1               ' Start comparing the string
     Case "EGGS"                ' Is String1 equal to EGGS?
       HRsout "FOUND EGGS\r"
     Case "BACON"               ' Is String1 equal to BACON?
       HRsout "FOUND BACON\r"
     Case "COFFEE"              ' Is String1 equal to COFFEE?
       HRsout "FOUND COFFEE\r"
     Case Else                  ' Default to...
       HRsout "NO MATCH\r"      ' Displaying no match
  EndSelect
  Stop
```

See also : **Creating and using Strings**
Creating and using Virtual Strings with Cdata
Cdata, If-Then-Else-EndIf, Repeat-Until
Select-Case, While-Wend.

Relational Operators

Relational operators are used to compare two values. The result can be used to make a decision regarding program flow.

The list below shows the valid relational operators accepted by the compiler:

Operator	Relation	Expression Type
=	Equality	X = Y
==	Equality	X == Y (Same as above Equality)
<>	Inequality	X <> Y
!=	Inequality	X != Y (Same as above Inequality)
<	Less than	X < Y
>	Greater than	X > Y
<=	Less than or Equal to	X <= Y
>=	Greater than or Equal to	X >= Y

See also : **If**-**Then**-**Else**-**EndIf**, **Repeat**-**Until**, **Select**-**Case**, **While**-**Wend**.

Boolean Logic Operators

The **If**-**Then**-**Else**-**EndIf**, **While**-**Wend**, and **Repeat**-**Until** conditions now support the logical operators not, and, or, and xor. The not operator inverts the outcome of a condition, changing false to true, and true to false. The following two **If**-**Then** conditions are equivalent:

```
If Var1 <> 100 Then NotEqual      ' GoTo notEqual if Var1 is not 100
If not Var1 = 100 Then NotEqual   ' GoTo notEqual if Var1 is not 100
```

The operators **and**, **or**, and **xor** join the results of two conditions to produce a single true/false result. and or work the same as they do in everyday speech. Run the example below once with and (as shown) and again, substituting **or** for **and**:

```
   Dim Var1 as Byte
   Dim Var2 as Byte

   Var1 = 5
   Var2 = 9
   If Var1 = 5 and Var2 = 10 Then Res_True
   Stop
Res_True:
   HRsout "Result is True.\r"
```

The condition "Var1 = 5 and Var2 = 10" is not true. Although Var1 is 5, Var2 is not 10. and works just as it does in plain English, both conditions must be true for the statement to be true. or also works in a familiar way; if one or the other or both conditions are true, then the statement is true. xor (short for exclusive-or) may not be familiar, but it does have an English counterpart: If one condition or the other (but not both) is true, then the statement is true.

Parenthesis (or rather the lack of it!).

Every compiler has it's quirky rules, and the Proton Amicus18 compiler is no exception. One of its quirks means that parenthesis is not supported in a Boolean condition, or indeed with any of the **If**-**Then**-**Else**-**EndIf**, **While**-**Wend**, and **Repeat**-**Until** conditions. Parenthesis in an expression within a condition is allowed however. So, for example, the expression:

```
If (Var1 + 3) = 10 Then do something.      Is allowed.
```

but:

```
If( (Var1 + 3) = 10) Then do something.    Is not allowed.
```

The Boolean operands do have a precedence in a condition. The and operand has the highest priority, then the or, then the xor. This means that a condition such as:

```
If Var1 = 2 and Var2 = 3 or Var3 = 4 Then do something
```

Will compare Var1 and Var2 to see if the and condition is true. It will then see if the or condition is true, based on the result of the and condition.

Then operand always required.

The Proton Amicus18 compiler relies heavily on the **Then** part. Therefore, if the **Then** part of a condition is left out of the code listing, a Syntax Error will be produced.

Math Operators

The Proton Amicus18 compiler performs all math operations in full hierarchal order. Which means that there is precedence to the operators. For example, multiplies and divides are performed before adds and subtracts. To ensure the operations are carried out in the correct order use parenthesis to group the operations:

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

All math operations are signed or unsigned depending on the variable type used, and performed with 16, or 32-bit precision, again, depending on the variable types and constant values used in the expression.

The operators supported are:

- Addition '+' — Adds variables and/or constants.
- Subtraction '-' — Subtracts variables and/or constants.
- Multiply '*' — Multiplies variables and/or constants.
- Multiply High '**' — Returns the high 16 bits of the 16-bit multiply result.
- Multiply Middle '*/' — Returns the middle 16 bits of the 16-bit multiply result.
- Divide '/' — Divides variables and/or constants.
- Modulus '//' — Returns the remainder after dividing one value by another.
- Bitwise and '&' — Returns the logical AND of two values.
- Bitwise or '|' — Returns the logical OR of two values.
- Bitwise xor '^' — Returns the logical XOR of two values.
- Bitwise Shift Left '<<' — Shifts the bits of a value left a specified number of places.
- Bitwise Shift Right '>>' — Shifts the bits of a value right a specified number of places.
- Bitwise Complement '~' — Reverses the bits in a variable.
- **Abs** — Returns the absolute value of a number.
- **Acos** — Returns the Arc Cosine of a value in Radians.
- **Asin** — Returns the Arc Sine of a value in Radians.
- **Atan** — Returns the Arc Tangent of a value in Radians.
- **Cos** — Returns the Cosine of a value in Radians.
- **Dcd** — 2 n -power decoder of a four-bit value.
- **Dig** — Returns the specified decimal digit of a positive value.
- **Exp** — Deduce the exponential function of a value.
- **iSqr** — Returns the Integer Square Root of a Value
- **Log** — Returns the Natural Log of a value.
- **Log10** — Returns the Log of a value.
- **Max** — Returns the maximum of two numbers.
- **Min** — Returns the minimum of two numbers.
- **Ncd** — Priority encoder of a 16-bit value.
- **Pow** — Computes a Variable to the power of another.
- **Rev** — Reverses the order of the lowest bits in a value.
- **Sin** — Returns the Sine of a value in radians.
- **Sqr** — Returns the floating point Square Root of a value.
- **Tan** — Returns the Tangent of a value in Radians.
- **Div32** — 15-bit x 31 bit divide. (For PBP compatibility only)

Add '+'.

Syntax
Assignment Variable = Variable + Variable

Overview
Adds variables and/or constants, returning an 8, 16, 32-bit or floating point result.

Operators
- ***Assignment Variable*** can be any valid variable type.
- ***Variable*** can be a constant, variable or expression.

Addition works exactly as you would expect with signed and unsigned integers as well as floating point.

```
  Dim Value1 as Word
  Dim Value2 as Word
  Value1 = 1575
  Value2 = 976
  Value1 = Value1 + Value2      ' Add the numbers
  HRsout Dec Value1             ' Display the result

' 32-bit addition
  Dim Value1 as Word
  Dim Value2 as Dword
  Value1 = 1575
  Value2 = 9763647
  Value2 = Value2 + Value1      ' Add the numbers
  HRsout Dec Value1, 13         ' Display the result
```

Subtract '-'.

Syntax
Assignment Variable = Variable - Variable

Overview
Subtracts variables and/or constants, returning an 8, 16, 32-bit or floating point result.

Operators
- ***Assignment Variable*** can be any valid variable type.
- ***Variable*** can be a constant, variable or expression.

Subtract works exactly as you would expect with signed and unsigned integers as well as floating point.

```
  Dim Value1 as Word
  Dim Value2 as Word
  Value1 = 1000
  Value2 = 999
  Value1 = Value1 - Value2      ' Subtract the numbers
  HRsout Dec Value1, 13         ' Display the result
```



```
' 32-bit subtraction
  Dim Value1 as Word
  Dim Value2 as Dword
  Value1 = 1575
  Value2 = 9763647
  Value2 = Value2 - Value1      ' Subtract the numbers
  HRsout Dec Value1             ' Display the result

' 32-bit signed subtraction
  Dim Value1 as Dword
  Dim Value2 as Dword
  Value1 = 1575
  Value2 = 9763647
  Value1 = Value1 - Value2      ' Subtract the numbers
  HRsout Sdec Value1, 13        ' Display the result
```

Multiply '*'.

Syntax
Assignment Variable = Variable * Variable

Overview
Multiplies variables and/or constants, returning an 8, 16, 32-bit or floating point result.

Operators
- ***Assignment Variable*** can be any valid variable type.
- ***Variable*** can be a constant, variable or expression.

Multiply works exactly as you would expect with signed or unsigned integers from -2147483648 to +2147483647 as well as floating point. If the result of multiplication is larger than 2147483647 when using 32-bit variables, the excess bit will be lost.

```
  Dim Value1 as Word
  Dim Value2 as Word
  Value1 = 1000
  Value2 = 19
  Value1 = Value1 * Value2      ' Multiply Value1 by Value2.
  HRsout Dec Value1, 13         ' Display the result

' 32-bit multiplication
  Dim Value1 as Word
  Dim Value2 as Dword
  Value1 = 100
  Value2 = 10000
  Value2 = Value2 * Value1      ' Multiply the numbers.
  HRsout Dec Value1, 13         ' Display the result
```


Multiply High '**'.

Syntax
Assignment Variable = Variable ** Variable

Overview
Multiplies 8 or 16-bit variables and/or constants, returning the high 16 bits of the result.

Operators
- ***Assignment Variable*** can be any valid variable type.
- ***Variable*** can be a constant, variable or expression.

When multiplying two 16-bit values, the result can be as large as 32 bits. Since the largest variable supported by the compiler is 16-bits, the highest 16 bits of a 32-bit multiplication result are normally lost. The ** (double-star) operand produces these upper 16 bits.

For example, suppose 65000 ($FDE8) is multiplied by itself. The result is 4,225,000,000 or $FBD46240. The * (star, or normal multiplication) instruction would return the lower 16 bits, $6240. The ** instruction returns $FBD4.

```
Dim Value1 as Word
Dim Value2 as Word
Value1 = $FDE8
Value2 = Value1 ** Value1    ' Multiply $FDE8 by itself, return high 16 bits
HRsout Hex Value2, 13        ' Display the result.
```

Notes.
This operand enables compatibility with BASIC STAMP code, and melab's compiler code, but is rather obsolete considering the 32-bit capabilities of the Proton Amicus18 compiler.

Multiply Middle '*/'.

Syntax
Assignment Variable = Variable */ Variable

Overview
Multiplies variables and/or constants, returning the middle 16 bits of the 32-bit result.

Operators
- ***Assignment Variable*** can be any valid variable type.
- ***Variable*** can be a constant, variable or expression.

The Multiply Middle operator (*/) has the effect of multiplying a value by a whole number and a fraction. The whole number is the upper byte of the multiplier (0 to 255 whole units) and the fraction is the lower byte of the multiplier (0 to 255 units of 1/256 each). The */ operand allows a workaround for the compiler's integer-only math.

Suppose we are required to multiply a value by 1.5. The whole number, and therefore the upper byte of the multiplier, would be 1, and the lower byte (fractional part) would be 128, since 128/256 = 0.5. It may be clearer to express the */ multiplier in **Hex** as $0180, since hex keeps the contents of the upper and lower bytes separate. Here's an example:


```
Dim Value1 as Word
Value1 = 100
Value1 = Value1 */ $0180   ' Multiply by 1.5 (1 + (128 / 256))
HRsout Dec Value1, 13      ' Display result (150).
```

To calculate constants for use with the */ instruction, put the whole number portion in the upper byte, then use the following formula for the value of the lower byte:

int(fraction * 256)

For example, take PiI(3.14159). The upper byte would be $03 (the whole number), and the lower would be int(0.14159 * 256) = 36 ($24). So the constant Pi for use with */ would be $0324. This isn't a perfect match for Pi, but the error is only about 0.1%.

Notes.
This operand enables compatibility with BASIC STAMP code, and melab's compiler code, but is rather obsolete considering the 32-bit capabilities of the Proton Amicus18 compiler.

Divide '/'.

Syntax
Assignment Variable = Variable / Variable

Overview
Divides variables and/or constants, returning an 8, 16, 32-bit or floating point result.

Operators
- ***Assignment Variable*** can be any valid variable type.
- ***Variable*** can be a constant, variable or expression.

The Divide operator (/) works exactly as you would expect with signed or unsigned integers from -2147483648 to +2147483647 as well as floating point.

```
Dim Value1 as Word
Dim Value2 as Word
Value1 = 1000
Value2 = 5
Value1 = Value1 / Value2     ' Divide the numbers.
HRsout Dec Value1, 13        ' Display the result (200).

' 32-bit division
Dim Value1 as Word
Dim Value2 as Dword
Value1 = 100
Value2 = 10000
Value2 = Value2 / Value1     ' Divide the numbers.
HRsout Dec Value1, 13        ' Display the result
```


Modulus '//'.

Syntax
Assignment Variable = Variable // Variable

Overview
Return the remainder left after dividing one value by another.

Operators
- ***Assignment Variable*** can be any valid variable type.
- ***Variable*** can be a constant, variable or expression.

Some division problems don't have a whole-number result; they return a whole number and a fraction. For example, 1000/6 = 166.667. Integer math doesn't allow the fractional portion of the result, so 1000/6 = 166. However, 166 is an approximate answer, because 166*6 = 996. The division operation left a remainder of 4. The // returns the remainder of a given division operation. Numbers that divide evenly, such as 1000/5, produce a remainder of 0:

```
  Dim Value1 as Word
  Dim Value2 as Word
  Value1 = 1000
  Value2 = 6
  Value1 = Value1 // Value2     ' Get remainder of Value1 / Value2.
  HRsout Dec Value1, 13         ' Display the result (4).

' 32-bit modulus
  Dim Value1 as Word
  Dim Value2 as Dword
  Value1 = 100
  Value2 = 99999
  Value2 = Value2 // Value1     ' mod the numbers.
  HRsout Dec Value1, 13         ' Display the result
```

The modulus operator does not operate with floating point values or variables.

Logical and '&'.

The AND operator (&) returns the bitwise and of two values. Each bit of the values is subject to the following logic:

```
0 and 0 = 0
0 and 1 = 0
1 and 0 = 0
1 and 1 = 1
```

The result returned by & will contain 1s in only those bit positions in which both input values contain 1s:

```
Dim Value1 as Byte
Dim Value2 as Byte
Dim Result as Byte
Value1 = %00001111
Value2 = %10101101
Result = Value1 & Value2
HRsout Bin Result, 13               ' Display and result (%00001101)
```

or

```
HRsout Bin (%00001111 & %10101101), 13 ' Display and result (%00001101)
```

Bitwise operations are not permissible with floating point values or variables.

Logical or '|'.

The OR operator (|) returns the bitwise or of two values. Each bit of the values is subject to the following logic:

```
0 or 0 = 0
0 or 1 = 1
1 or 0 = 1
1 or 1 = 1
```

The result returned by | will contain 1s in any bit positions in which one or the other (or both) input values contain 1s:

```
Dim Value1 as Byte
Dim Value2 as Byte
Dim Result as Byte
Value1 = %00001111
Value2 = %10101001
Result = Value1 | Value2
HRsout Bin Result, 13               ' Display or result (%10101111)
```

or

```
HRsout Bin (%00001111 | %10101001), 13 ' Display or result (%10101111)
```

Bitwise operations are not permissible with floating point values or variables.

Logical xor '^'.

The XOR operator (^) returns the bitwise xor of two values. Each bit of the values is subject to the following logic:

```
0 xor 0 = 0
0 xor 1 = 1
1 xor 0 = 1
1 xor 1 = 0
```

The result returned by ^ will contain 1s in any bit positions in which one or the other (but not both) input values contain 1s:

```
Dim Value1 as Byte
Dim Value2 as Byte
Dim Result as Byte
Value1 = %00001111
Value2 = %10101001
Result = Value1 ^ Value2
HRsout Bin Result, 13                ' Display xor result (%10100110)
```

or

```
HRsout Bin (%00001111 ^ %10101001), 13 ' Display xor result (%10100110)
```

Bitwise operations are not permissible with floating point values or variables.

BitWise Shift Left '<<'.

Shifts the bits of a value to the left a specified number of places. Bits shifted off the left end of a number are lost; bits shifted into the right end of the number are 0s. Shifting the bits of a value left n number of times also has the effect of multiplying that number by two to the nth power.

For example 100 << 3 (shift the bits of the decimal number 100 left three places) is equivalent to 100 * 23.

```
Dim Value1 as Word
Dim Loop as Byte
Value1 = %1111111111111111
For Loop = 1 to 16              ' Repeat with b0 = 1 to 16.
  HRsout Bin Value1 << Loop     ' Shift Value1 left Loop places.
Next
```

Bitwise operations are not permissible with floating point values or variables.

BitWise Shift Right '>>'.

Shifts the bits of a variable to the right a specified number of places. Bits shifted off the right end of a number are lost; bits shifted into the left end of the number are 0s. Shifting the bits of a value right n number of times also has the effect of dividing that number by two to the nth power.

For example 100 >> 3 (shift the bits of the decimal number 100 right three places) is equivalent to 100 / 23.

```
Dim Value1 as Word
Dim Loop as Byte
Value1 = %1111111111111111
For Loop = 1 to 16              ' Repeat with b0 = 1 to 16.
   HRsout Bin Value1 >> Loop    ' Shift Value1 right Loop places.
Next
```

BitWise Complement '~'

The Complement operator (~) inverts the bits of a value. Each bit that contains a 1 is changed to 0 and each bit containing 0 is changed to 1. This process is also known as a "bitwise not".

```
Dim Value1 as Word
Dim Value2 as Word
Value2 = %1111000011110000
Value1 = ~Value2                ' Complement Value2.
HRsout Bin16 Value1, 13         ' Display the result
```

Complementing can be carried out with all variable types except Floats. Attempting to complement a floating point variable will produce a syntax error.

Abs

Syntax
Assignment Variable = **Abs** Variable

Overview
Return the absolute value of a constant, variable or expression.

Operators
- ***Assignment Variable*** can be any valid variable type.
- ***Variable*** can be a constant, variable or expression.

32-bit Example
```
Dim DwordVar1 as Dword        ' Create a Dword variable
Dim DwordVar2 as Dword        ' Create a Dword variable

DwordVar1 = -1234567          ' Load DwordVar1 with value -1234567
DwordVar2 = Abs DwordVar1     ' Extract the absolute value from DwordVar1
HRsout Dec DwordVar2, 13      ' Display the result, which is 1234567
```

Floating Point example
```
Dim FloatVar1 as Float        ' Create a Float variable
Dim FloatVar2 as Float        ' Create a Float variable

FloatVar1 = -1234567          ' Load FloatVar1 with value -1234567.123
FloatVar2 = Abs FloatVar 1    ' Extract the absolute value from Floatvar1
HRsout Dec FloatVar2, 13      ' Display the result, which is 1234567.123
```


Acos

Syntax
Assignment Variable = **Acos** Variable

Overview
Deduce the Arc Cosine of a value

Operators
- ***Assignment Variable*** can be any valid variable type.
- ***Variable*** can be a constant, variable or expression that requires the Arc Cosine (Inverse Cosine) extracted. The value expected and returned by the floating point **Acos** is in Radians. The value must be in the range of -1 to +1

Example
```
Dim Floatin as Float        ' Holds the value to Acos
Dim Floatout as Float       ' Holds the result of the Acos
Floatin = 0.8               ' Load the variable
Floatout = Acos Floatin     ' Extract the Acos of the value
HRsout Dec Floatout, 13     ' Display the result
```

Notes
Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

Asin

Syntax
Assignment Variable = **Asin** Variable

Overview
Deduce the Arc Sine of a value

Operators
- ***Assignment Variable*** can be any valid variable type.
- ***Variable*** can be a constant, variable or expression that requires the Arc Sine (Inverse Sine) extracted. The value expected and returned by **Asin** is in Radians. The value must be in the range of -1 to +1

Example
```
Dim Floatin as Float        ' Holds the value to Asin
Dim Floatout as Float       ' Holds the result of the Asin
Floatin = 0.8               ' Load the variable
Floatout = Asin Floatin     ' Extract the Asin of the value
HRsout Dec Floatout, 13     ' Display the result
```

Notes
Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

Atan

Syntax
Assignment Variable = **Atan** Variable

Overview
Deduce the Arc Tangent of a value

Operators
- ***Assignment Variable*** can be any valid variable type.
- ***Variable*** can be a constant, variable or expression that requires the Arc Tangent (Inverse Tangent) extracted. The value expected and returned by the floating point **Atan** is in Radians.

Example
```
Dim Floatin as Float        ' Holds the value to Atan
Dim Floatout as Float       ' Holds the result of the Atan
Floatin = 1                 ' Load the variable
Floatout = Atan Floatin     ' Extract the Atan of the value
HRsout Dec Floatout, 13     ' Display the result
```

Notes
Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

Cos

Syntax
Assignment Variable = **Cos** Variable

Overview
Deduce the Cosine of a value

Operators
- ***Assignment Variable*** can be any valid variable type.
- ***Variable*** can be a constant, variable or expression that requires the Cosine extracted. The value expected and returned by **Cos** is in Radians.

Example
```
Dim Floatin as Float        ' Holds the value to Cos with
Dim Floatout as Float       ' Holds the result of the Cos
Floatin = 123               ' Load the variable
Floatout = Cos Floatin      ' Extract the Cos of the value
HRsout Dec Floatout, 13     ' Display the result
```

Notes
Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

Dcd

2 n -power decoder of a four-bit value. **Dcd** accepts a value from 0 to 15, and returns a 16-bit number with that bit number set to 1. For example:

```
WordVar1= Dcd 12                ' Set bit 12.
HRsout Bin16 WordVar1, 13       ' Display result (%0001000000000000)
```

Dcd does not (as yet) support Dword, or Float type variables. Therefore the highest value obtainable is 65535.

Dig (BASIC Stamp version)

In this form, the **Dig** operator is compatible with the BASIC STAMP, and the melab's PicBASIC Pro compiler. **Dig** returns the specified decimal digit of a 16-bit positive value. Digits are numbered from 0 (the rightmost digit) to 4 (the leftmost digit of a 16- bit number; 0 to 65535). Example:

```
WordVar1= 9742
HRsout WordVar1 Dig 2           ' Display digit 2 (7)
For Loop = 0 to 4
   HRsout WordVar1 Dig Loop     ' Display digits 0 through 4 of 9742.
Next
```

Note

Dig does not support Float type variables.

Exp

Syntax
Assignment Variable = **Exp** Variable

Overview
Deduce the exponential function of a value. This is e to the power of value where e is the base of natural logarithms. **Exp** 1 is 2.7182818.

Operators
- ***Assignment Variable*** can be any valid variable type.
- ***Variable*** can be a constant, variable or expression.

Example
```
Dim Floatin as Float        ' Holds the value to Exp with
Dim Floatout as Float       ' Holds the result of the Exp
Floatin = 1                 ' Load the variable
Floatout = Exp Floatin      ' Extract the Exp of the value
HRsout Dec Floatout, 13     ' Display the result
```

Notes
Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

ISqr

Syntax
Assignment Variable = **ISqr** Variable

Overview
Deduce the Integer Square Root of a value.

Operators
- ***Assignment Variable*** can be any valid variable type.
- ***Variable*** can be a constant, variable or expression that requires the square root extracted.

Example
```
Dim Wordin as Word          ' Holds the value to Sqr
Dim Wordout as Word         ' Holds the result of the Sqr
Wordin = 600                ' Load the variable
Wordout = ISqr Wordin       ' Extract the Sqr of the integer value
HRsout Dec Wordout, 13      ' Display the result
```

See **Sqr** for a floating point version of **ISqr**.

Log

Syntax
Assignment Variable = **Log** Variable

Overview
Deduce the Natural Logarithm a value

Operators
- ***Assignment Variable*** can be any valid variable type.
- ***Variable*** can be a constant, variable or expression that requires the natural logarithm extracted.

Example
```
Dim Floatin as Float        ' Holds the value to Log with
Dim Floatout as Float       ' Holds the result of the Log
Floatin = 1                 ' Load the variable
Floatout = Log Floatin      ' Extract the Log of the value
HRsout Dec Floatout, 13     ' Display the result
```

Notes
Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

Log10

Syntax
Assignment Variable = **Log10** Variable

Overview
Deduce the Logarithm a value

Operators
Assignment Variable can be any valid variable type.
Variable can be a constant, variable or expression that requires the logarithm extracted.

Example
```
Dim Floatin as Float        ' Holds the value to Log10 with
Dim Floatout as Float       ' Holds the result of the Log10
Floatin = 1                 ' Load the variable
Floatout = Log10 Floatin    ' Extract the Log10 of the value
HRsout Dec Floatout, 13     ' Display the result
```

Notes
Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

Max

Returns the maximum of two numbers. Its use is to limit numbers to a specific value. Its syntax is:

' Set Var2 to the larger of Var1 and 100 (Var2 will be between 100 and 255)
Var2 = Var1 **Max** 100

Max does not (as yet) support Dword, or Float type variables. Therefore the highest value obtainable is 65535.

Min

Returns the minimum of two numbers. Its use is to limit numbers to a specific value. Its syntax is:

' Set Var2 to the smaller of Var1 and 100 (Var2 cannot be greater than 100)
Var2 = Var1 **Min** 100

Min does not (as yet) support Dword, or Float type variables. Therefore the highest value obtainable is 65535.

Ncd

Priority encoder of a 16-bit value. **Ncd** takes a 16-bit value, finds the highest bit containing a 1 and returns the bit position plus one (1 through 16). If no bit is set, the input value is 0. **Ncd** returns 0. **Ncd** is a fast way to get an answer to the question "what is the largest power of two that this value is greater than or equal to?" The answer that **Ncd** returns will be that power, plus one. Example:

```
WordVar1= %00001101          ' Highest bit set is bit 3.
Hrsout Dec Ncd WordVar1, 13  ' Display the Ncd of WordVar1(4).
```

Ncd does not (as yet) support Dword, or Float type variables.

Pow

Syntax
Assignment Variable = **Pow** Variable, PowerOfVariable

Overview
Computes Variable to the power of ***PowerOfVariable***.

Operators
- ***Assignment Variable*** can be any valid variable type.
- ***Variable*** can be a constant, variable or expression.
- ***PowerOfVariable*** can be a constant, variable or expression.

Example
```
Dim PowOf as Float
Dim Floatin as Float            ' Holds the value to Pow with
Dim Floatout as Float           ' Holds the result of the Pow
PowOf= 10
Floatin = 2                     ' Load the variable
Floatout = Pow Floatin,PowOf    ' Extract the Pow of the value
HRsout Dec Floatout, 13         ' Display the result
```

Notes
Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

Rev

Reverses the order of the lowest bits in a value. The number of bits to be reversed is from 1 to 32. Its syntax is:

```
   Var1 = %10101100 Rev 4  ' Sets Var1 to %10100011
```

or

```
   Dim DwordVar as Dword
' Sets DwordVar to %10101010000000001111111110100011
   DwordVar = %10101010000000001111111110101100 Rev 4
```


Sin

Syntax
Assignment Variable = **Sin** Variable

Overview
Deduce the Sine of a value

Operators
- ***Assignment Variable*** can be any valid variable type.
- ***Variable*** can be a constant, variable or expression that requires the sine extracted. The value expected and returned by **Sin** is in Radians.

Example
```
Dim Floatin as Float        ' Holds the value to Sin
Dim Floatout as Float       ' Holds the result of the Sin
Floatin = 123               ' Load the variable
Floatout = Sin Floatin      ' Extract the Sin of the value
HRsout Dec Floatout, 13     ' Display the result
```

Notes
Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

Sqr

Syntax
Assignment Variable = **Sqr** Variable

Overview
Deduce the Square Root of a value. Computes using floating point.

Operators
- ***Assignment Variable*** can be any valid variable type.
- ***Variable*** can be a constant, variable or expression that requires the square root extracted.

Example
```
Dim Floatin as Float        ' Holds the value to Sqr
Dim Floatout as Float       ' Holds the result of the Sqr
Floatin = 600               ' Load the variable
Floatout = Sqr Floatin      ' Extract the Sqr of the value
HRsout Dec Floatout, 13     ' Display the result
```

Notes
Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

See **Isqr** for an integer version of **Sqr**.

Tan

Syntax
Assignment Variable = **Tan** Variable

Overview
Deduce the Tangent of a value

Operators
- ***Assignment Variable*** can be any valid variable type.
- ***Variable*** can be a constant, variable or expression that requires the tangent extracted. The value expected and returned by the floating point **Tan** is in radians.

Example
```
Dim Floatin as Float        ' Holds the value to Tan
Dim Floatout as Float       ' Holds the result of the Tan
Floatin = 1                 ' Load the variable
Floatout = Tan Floatin      ' Extract the Tan of the value
HRsout Dec Floatout, 13     ' Display the result
```

Notes
Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

Div32

In order to make the Proton Amicus18 compiler more compatible with code produced for the melab's PicBASIC Pro compiler, the **Div32** operator has been added. The melab's compiler's multiply operand operates as a 16-bit x 16-bit multiply, thus producing a 32-bit result. However, since the compiler only supports a maximum variable size of 16 bits (Word), access to the result had to happen in 2 stages:

```
Var = Var1 * Var2  ' Returns the lower 16 bits of the multiply
```

while...

```
Var = Var1 ** Var2 ' Returns the upper 16 bits of the multiply
```

There was no way to access the 32-bit result as a valid single value.

In many cases it is desirable to be able to divide the entire 32-bit result of the multiply by a 16-bit number for averaging, or scaling. **Div32** is actually limited to dividing a 31-bit unsigned integer (0 - 2147483647) by a 15-bit unsigned integer (0 - 32767). This ought to be sufficient in most situations.

Because the melab's PICBASIC Pro compiler only allowed a maximum variable size of 16 bits (0 - 65535), **Div32** relies on the fact that a multiply was performed just prior to the **Div32** command, and that the internal compiler variables still contain the 32-bit result of the multiply. No other operation may occur between the multiply and the **Div32** or the internal variables may be altered, thus destroying the 32-bit multiplication result.

The following example demonstrates the operation of **Div32**:

```
Dim WordVar1 as Word
Dim WordVar2 as Word
Dim WordVar3 as Word
Dim Fake as Word              ' Must be a Word type variable for result

WordVar2 = 300
WordVar3 = 1000

Fake = WordVar2 * WordVar3    ' Operators ** or */ could also be used instead
WordVar1= Div32 100
Hrsout Dec WordVar1, 13
```

The above program assigns WordVar2 the value 300 and WordVar3 the value 1000. When multiplied together, the result is 300000. However, this number exceeds the 16-bit word size of a variable (65535). Therefore, the dummy variable, Fake, contains only the lower 16 bits of the result. **Div32** uses the compiler's internal (System) variables as the operands.

Notes.

This operand enables a certain compatibility with melab's compiler code, but is very much obsolete considering the 32-bit, and floating point capabilities of the Proton Amicus18 compiler.

Commands and Directives

Adin Read the on-board Analogue to Digital Converter (ADC).
Asm-EndAsm Insert assembly language code section.
Box Draw a square on a graphic LCD.
Branch Computed GoTo (equiv. to On..GoTo).
Break Exit a loop prematurely.
Bstart Send a Start condition to the I^2C bus.
Bstop Send a Stop condition to the I^2C bus.
Brestart Send a Restart condition to the I^2C bus.
BusAck Send an Acknowledge condition to the I^2C bus.
Busin Read bytes from an I^2C device.
Busout Write bytes to an I^2C device.
Button Detect and debounce a key press.
Call Call an assembly language subroutine.
Cdata Define initial contents in memory.
Circle Draw a circle on a graphic LCD.
Clear Place a variable or bit in a low state, or clear all RAM area.
ClearBit Clear a bit of a port or variable, using a variable index.
Cls Clear the LCD.
Config_Start Set or Reset programming fuse configurations.
Counter Count the number of pulses occurring on a pin.
Cread Read data from code memory.
Cursor Position the cursor on the LCD.
Cwrite Write data to code memory.
Dec Decrement a variable.
Declare Adjust library routine parameters.
DelayMs Delay (1 millisecond resolution).
DelayUs Delay (1 microsecond resolution).
Dig Return the value of a decimal digit.
Dim Create a variable.
Disable Disable software interrupts previously Enabled.
DTMFout Produce a DTMF Touch Tone note.
Edata Define initial contents of on-board eeprom.
Enable Enable software interrupts previously Disabled.
End Stop execution of the BASIC program (Same as Stop).
Eread Read a value from on-board eeprom.
Ewrite Write a value to on-board eeprom.
For...to...Next...Step Repeatedly execute statements.
FreqOut Generate one or two tones, of differing or the same frequencies.
GetBit Examine a bit of a port or variable, using a variable index.
GoSub Call a BASIC subroutine at a specified label.
GoTo Continue execution at a specified label.
HbStart Send a Start condition to the I^2C bus using the MSSP peripheral.
HbStop Send a Stop condition to the I^2C bus using the MSSP peripheral.
HbRestart Send a Restart condition to the I^2C bus using the MSSP peripheral.
HbusAck Send an Ack condition to the I^2C bus using the MSSP peripheral.
Hbusin Read from an I^2C device using the MSSP peripheral.
Hbusout Write to an I^2C device using the MSSP peripheral.
High Make a pin or port high.
Hpwm Generate a Pwm signal using the CCP peripheral.
HRsin Receive data from the serial port using the USART peripheral.
HRsout Transmit data from the serial port using the USART peripheral.
Hserin Receive data from the serial port using the USART peripheral.
Hserout Transmit data from the serial port using the USART peripheral.
I2Cin Read bytes from an I^2C device with user definable SDA\SCL lines.

I2Cout Write bytes to an I^2C device with user definable SDA\SCL lines.
If..Then..ElseIf..Else..EndIf Conditionally execute statements.
Inc Increment a variable.
Include Load a BASIC file into the source code.
Inkey Scan a keypad.
Input Make pin an input.
LCDread Read a single byte from a Graphic LCD.
LCDwrite Write bytes to a Graphic LCD.
Left$ Extract n amount of characters the left of a String.
Cdata Place information into code memory. For access by Lread.
Line Draw a line in any direction on a graphic LCD.
LineTo Draw a straight line in any direction on a graphic LCD, starting from the previous Line command's end position.
LoadBit Set or Clear a bit of a port or variable, using a variable index.
LookDown Search a constant lookdown table for a value.
LookDownL Search constant or variable lookdown table for a value.
LookUp Fetch a constant value from a lookup table.
LookUpL Fetch a constant or variable value from lookup table.
Low Make a pin or port low.
Lread Read a value from an Cdata table and place into Variable.
Lread8, **Lread16**, **Lread32** Read a single or multi-byte value from a Cdata table with more efficiency than Lread**.**
Mid$ Extract n amount of characters from a String beginning at n characters from the left.
On Interrupt Execute a subroutine using a Software interrupt (*Not Recommended*).
On_Hardware_Interrupt Execute an Assembler subroutine on a Hardware interrupt.
On_Low_Interrupt Execute an Assembler subroutine when a Low Priority Hardware interrupt.
On GoSub Call a Subroutine based on an Index value.
On GoTo Jump to an address in code memory based on an Index value.
Output Make a pin an output.
Oread Receive data from a device using the Dallas 1-wire protocol.
Owrite Send data to a device using the Dallas 1-wire protocol.
Org Set Program Origin.
Pixel Read a single pixel from a Graphic LCD.
Plot Set a single pixel on a Graphic LCD.
Pop Pull a single variable or multiple variables from a software stack.
Pot Read a potentiometer on specified pin.
Print Display characters on an LCD.
PulsIn Measure the pulse width on a pin.
PulseOut Generate a pulse to a pin.
Push Place a single variable or multiple variables onto a software stack.
Pwm Output a pulse width modulated pulse train to pin.
Random Generate a pseudo-random number.
RC5in Receive and decode Philips Infrared RC5 packets.
RCin Measure a pulse width on a pin.
Repeat...Until Execute a block of instructions until a condition is true.
Resume Re-enable software interrupts and return.
Return Continue at the statement following the last GoSub.
Right$ Extract n amount of characters from the right of a String.
Rsin Asynchronous serial input from a fixed pin and baud rate.
Rsout Asynchronous serial output to a fixed pin and baud rate.
Seed Seed the random number generator, to obtain a more random result.
Select..Case..EndSelect Conditionally run blocks of code.
Serin Receive asynchronous serial data (i.e. RS232 data).
Serout Transmit asynchronous serial data (i.e. RS232 data).
Servo Control a servo motor.

Set	Place a variable or bit in a high state.
SetBit	Set a bit of a port or variable, using a variable index.
Shin	Synchronous serial input.
Shout	Synchronous serial output.
Sleep	Power down the processor for a period of time.
Snooze	Power down the processor for short period of time.
SonyIn	Receive Sony SIRC (Sony Infrared Remote Control) data from a predetermined pin.
Sound	Generate a tone or white-noise on a specified pin.
Sound2	Generate 2 tones from 2 separate pins.
Stop	Stop program execution.
Str	Load a Byte array with values.
Strn	Create a null terminated Byte array.
Str$	Convert the contents of a variable to a null terminated String.
Swap	Exchange the values of two variables.
Symbol	Create an alias to a constant, port, pin, or register.
Toggle	Reverse the state of a port's bit.
ToLower	Convert the characters in a String to lower case.
ToUpper	Convert the characters in a String to UPPER case.
Toshiba_Command	Send a command to a Toshiba T6963 graphic LCD.
Toshiba_UDG	Create User Defined Graphics for Toshiba T6963 graphic LCD.
UnPlot	Clear a single pixel on a Graphic LCD.
Val	Convert a null terminated String to an integer value.
VarPtr	Locate the address of a variable.
While…Wend	Execute statements while condition is true.
Xin	Receive data using the X10 protocol.
Xout	Transmit data using the X10 protocol.

Adin

Syntax
Variable = **Adin** channel number

Overview
Read the value from the on-board Analogue to Digital Converter.

Operators
- ***Variable*** is a user defined variable.
- ***Channel number*** can be a constant or a variable expression.

Example
```
' Read ADC from AN0 and display the value on the serial terminal.

  Declare Adin_Res = 10      ' 10-bit result required
  Declare Adin_Tad = FRC     ' RC OSC chosen
  Declare Adin_Stime = 50    ' Allow 50us sample time

  Dim Var1 as Word

  TRISA = %00000001          ' Configure AN0 (PortA.0) as an input
  ADCON1 = %10000000         ' Set analogue input on PortA.0
  While 1 = 1                ' Create an endless loop
    Var1 = Adin 0            ' Place the conversion into variable Var1
    Hrsout Dec Var1,13       ' Display the value on the serial terminal
    DelayMs 500              ' A delay between samples
  Wend                       ' Close the loop
```

Adin Declares
There are three **Declare** directives for use with **Adin**. These are:

Declare Adin_Res 8 or 10.
Sets the number of bits in the result.

If this **Declare** is not used, then the default is a resolution of 10-bits.
Using the above **Declare** allows an 8-bit result to be obtained.

Declare Adin_Tad 2_FOSC, 8_FOSC, 32_FOSC, 64_FOSC, or FRC.
Sets the ADC's clock source.

There are five options for the clock source used by the ADC. 2_FOSC, 8_FOSC, 32_FOSC, and 64_FOSC are ratios of the external oscillator, while FRC is the internal RC oscillator. Instead of using the prede-fined names for the clock source, values from 0 to 4 may be used.

Care must be used when issuing this **Declare**, as the wrong type of clock source may result in poor resolution, or no conversion at all. If in doubt use FRC which will produce a slight reduction in resolution and conversion speed, but is guaranteed to work first time, every time. FRC is the default setting if the **Declare** is not issued in the BASIC listing.

Declare Adin_Stime 0 to 65535 microseconds (us).
Allows the internal capacitors to fully charge before a sample is taken. This may be a value from 0 to 65535 microseconds (us).

A value too small may result in a reduction of resolution. While too large a value will result in poor conversion speeds without any extra resolution being attained.

A typical value for **Adin**_Stime is 50 to 100. This allows adequate charge time without loosing too much conversion speed. But experimentation will produce the right value for your particular requirement. The default value if the **Declare** is not used in the BASIC listing is 50.

Notes
Before the **Adin** command may be used, the appropriate registers must be manipulated in order to configure the ADC peripheral. The ADC macros can be used for setting up the device's ADC. These are loaded by using an include statement:

```
   Include "ADC.inc"     ' Load the ADC macros into the program
'
' Open the ADC  :
'                   Fosc / 32
'                   Right justified for 10-bit operation
'                   Tad value of 2
'                   Vref+ at Vcc : Vref- at Gnd
'                   Make AN0 an analogue input
'
   OpenADC(ADC_FOSC_32 & ADC_RIGHT_JUST & ADC_2_TAD, ADC_REF_VDD_VSS, ADC_1ANA)
```

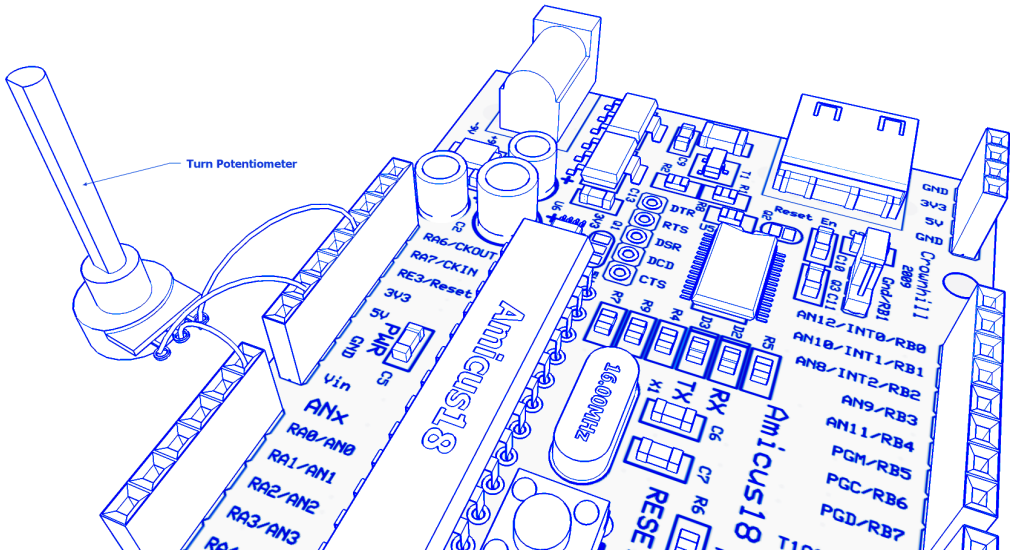
Further information concerning the ADC macros can be found at the back of this document.

If multiple conversions are being implemented, then a small delay should be used after the **Adin** command. This allows the ADC's internal capacitors to discharge fully:

```
Again:
   Var1 = Adin 0        ' Place the conversion into variable Var1
   DelayUs 2            ' Wait for 2 microseconds
   GoTo Again           ' Read the ADC forever
```

The layout below shows a typical setup for a simple ADC test using a potentiometer.

The inside pin of the 100KΩ potentiometer is connected to Gnd, the middle pin is connected to AN0, and the outside pin is connected to 3V3. Use the example from the previous page. Turning the potentiometer clockwise will increase the voltage seen by the ADC from 0 to 3.3 Volts. The value displayed will range from 0 to 1023 (10-bits).



See also : **ADC macros, Rcin**, **Pot**.

Asm..EndAsm

Syntax

Asm
assembler mnemonics
EndAsm

or

@ assembler mnemonic

Overview

Incorporate in-line assembler in the BASIC code. The mnemonics are passed directly to the assembler without the compiler interfering in any way. This allows a great deal of flexibility that cannot always be achieved using BASIC commands alone.

When the **Asm** directive is found within the BASIC program, the RAM banks are Reset before the assembler code is operated upon. The same happens when the **EndAsm** directive is found, in that the RAM banks are Reset upon leaving the assembly code. However, this may not always be required and can waste precious code memory. Placing a dash after **Asm** or **EndAsm** will remove the RAM Reset mnemonics.

```
Asm-
EndAsm-
```

Only remove the RAM resets if you are confident enough to do so, as the microcontroller has fragmented RAM.

The compiler also allows assembler mnemonics to be used within the BASIC program without wrapping them in **Asm-EndAsm**, however, the constants, labels, and variables used must be valid BASIC types.

Box

Syntax
Box Set_Clear, Xpos Start, Ypos Start, Size

Overview
Draw a square on a graphic LCD.

Operators
- ***Set_Clear*** may be a constant or variable that determines if the square will set or clear the pixels. A value of 1 will set the pixels and draw a square, while a value of 0 will clear any pixels and erase a square.
- ***Xpos*** Start may be a constant or variable that holds the X position for the centre of the square. Can be a value from 0 to 127.
- ***Ypos*** Start may be a constant or variable that holds the Y position for the centre of the square. Can be a value from 0 to 63.
- ***Size*** may be a constant or variable that holds the Size of the square (in pixels). Can be a value from 0 to 255.

Example
```
' Draw square at position 63,32 with size of 20 pixels on Samsung KS0108 LCD

  Dim Xpos as Byte
  Dim Ypos as Byte
  Dim Size as Byte
  Dim SetClr as Byte

  DelayMs 20            ' Wait for things to stabilise
  Cls                   ' Clear the LCD
  Xpos = 63
  Ypos = 32
  Size = 20
  SetClr = 1
  Box SetClr, Xpos, Ypos, Radius
```

Notes
Because of the aspect ratio of the pixels on the Samsung graphic LCD (approx 1.5 times higher than wide) the square will appear elongated.

See Also : **Circle**, **Line**, **LineTo**, **Plot**, **UnPlot**.

Branch

Syntax
Branch Index, [Label1 {,...Labeln }]

Overview
Cause the program to jump to different locations based on a variable index. On a microcontroller device with only one page of memory.

Operators
- ***Index*** is a constant, variable, or expression, that specifies the address to branch to.
- ***Label1,...Labeln*** are valid labels that specify where to branch to. A maximum of 256 labels may be placed between the square brackets.

Example
```
   Dim Index as Byte
Start:
   Index = 2           ' Assign Index a value of 2
   Branch Index,[Lab_0, Lab_1, Lab_2] ' Jump to Lab_2 because Index = 2
Lab_0:
   Index = 2           ' Index now equals 2
   GoTo Start
Lab_1:
   Index = 0           ' Index now equals 0
   GoTo Start
Lab_2:
   Index = 1           ' Index now equals 1
   GoTo Start
```

The above example we first assign the index variable a value of 2, then we define our labels. Since the first position is considered 0 and the variable index equals 2 the **Branch** command will cause the program to jump to the third label in the brackets [Lab_2].

Notes
Branch operates the same as **On** x **GoTo**. It's useful when you want to organise a structure such as:

```
   If Var1 = 0 Then GoTo Lab_0  ' Var1 =0: go to label "Lab_0"
   If Var1 = 1 Then GoTo Lab_1  ' Var1 =1: go to label "Lab_1"
   If Var1 = 2 Then GoTo Lab_2  ' Var1 =2: go to label "Lab_2"
```

You can use **Branch** to organise this into a single statement:

```
   Branch Var1, [Lab_0, Lab_1, Lab_2]
```

This works exactly the same as the above **If**...**Then** example. If the value is not in range (in this case if Var1 is greater than 2), **Branch** does nothing. The program continues with the next instruction..

See also : **On GoTo**

Break

Syntax
Break

Overview
Exit a **For**...**Next**, **While**...**Wend** or **Repeat**...**Until** loop prematurely.

Example 1
```
' Break out of a For-Next loop when the count reaches 10
  Dim Var1 as Byte
  For Var1 = 0 to 39          ' Create a loop of 40 revolutions
    HRsout Dec Var1,13        ' Display the revolutions on the serial terminal
    If Var1 = 10 Then Break   ' Break out of the loop when Var1 = 10
    DelayMs 200               ' Delay so we can see what's happening
  Next                        ' Close the For-Next loop
  HRsout "Exited At ", Dec Var1, 13   ' Display value when loop was broke
```

Example 2
```
' Break out of a Repeat-Until loop when the count reaches 10
  Dim Var1 as Byte
  Var1 = 0
  Repeat                      ' Create a loop
    HRsout Dec Var1, 13       ' Display the revolutions on the serial terminal
    If Var1 = 10 Then Break   ' Break out of the loop when Var1 = 10
    DelayMs 200               ' Delay so we can see what's happening
    Inc Var1
  Until Var1 > 39             ' Close the loop after 40 revolutions
  HRsout "Exited At ", Dec Var1, 13   ' Display value when loop was broke
```

Example 3
```
' Break out of a While-Wend loop when the count reaches 10
  Dim Var1 as Byte
  Var1 = 0
  While Var1 < 40             ' Create a loop of 40 revolutions
    HRsout Dec Var1, 13       ' Display the revolutions on the serial terminal
    If Var1 = 10 Then Break   ' Break out of the loop when Var1 = 10
    DelayMs 200               ' Delay so we can see what's happening
    Inc Var1
  Wend                        ' Close the loop
  HRsout "Exited At ", Dec Var1, 13   ' Display value when loop was broke
```

Notes
The **Break** command is similar to a **GoTo** but operates internally. When the **Break** command is encountered, the compiler will force a jump to the loop's internal exit label.

If the **Break** command is used outside of a **For**-**Next**, **Repeat**-**Until** or **While**-**Wend** loop, an error will be produced.

See also: **For**...**Next**, **While**...**Wend**, **Repeat**...**Until**.

Bstart

Syntax
Bstart

Overview
Send a Start condition to the I²C bus.

Notes
Because of the subtleties involved in interfacing to some I²C devices, the compiler's standard **Busin**, and **Busout** commands were found lacking somewhat. Therefore, individual pieces of the I²C protocol may be used in association with the new structure of **Busin**, and **Busout**. See relevant sections for more information.

Example
```
' Interface to a 24LC256 serial eeprom
  Dim Loop as Byte
  Dim Array[10] as Byte
' Transmit bytes to the I2C bus
  Bstart                ' Send a Start condition
  Busout %10100000      ' Target an eeprom, and send a WRITE command
  Busout 0              ' Send the High Byte of the address
  Busout 0              ' Send the Low Byte of the address
  For Loop = 48 to 57   ' Create a loop containing ASCII 0 to 9
    Busout Loop         ' Send the value of Loop to the eeprom
  Next                  ' Close the loop
  Bstop                 ' Send a Stop condition
  DelayMs 10            ' Wait for the data to be entered into eeprom matrix
' Receive bytes from the I2C bus
  Bstart                ' Send a Start condition
  Busout %10100000      ' Target an eeprom, and send a WRITE command
  Busout 0              ' Send the High Byte of the address
  Busout 0              ' Send the Low Byte of the address
  Brestart              ' Send a Restart condition
  Busout %10100001      ' Target an eeprom, and send a Read command
  For Loop = 0 to 9     ' Create a loop
    Array[Loop] = Busin     ' Load an array with bytes received
    If Loop = 9 Then Bstop : Else : BusAck ' ACK or Stop ?
  Next                  ' Close the loop
  HRsout Str Array, 13      ' Display the Array as a String
```

See also: **Bstop**, **Brestart**, **BusAck**, **Busin**, **Busout**, **HbStart**, **HbRestart**, **HbusAck**, **Hbusin**, **Hbusout**.

Bstop

Syntax
Bstop

Overview
Send a Stop condition to the I^2C bus.

Brestart

Syntax
Brestart

Overview
Send a Restart condition to the I^2C bus.

BusAck

Syntax
BusAck

Overview
Send an Acknowledge condition to the I^2C bus.

BusNack

Syntax
BusNack

Overview
Send a Not Acknowledge condition to the I^2C bus.

See also: **Bstop**, **Bstart**, **Brestart**, **Busin**, **Busout**, **HbStart**, **HbRestart**, **HbusAck**, **HbusNack**, **Hbusin**, **Hbusout**.

Busin

Syntax
Variable = **Busin** Control, { Address }

or

Variable = **Busin**

or

Busin Control, { Address }, [Variable {, Variable…}]

or

Busin Variable

Overview
Receives a value from the I^2C bus, and places it into variable/s. If versions two or four (see above) are used, then No Acknowledge, or Stop is sent after the data. Versions one and three first send the control and optional address out of the clock pin (SCL), and data pin (SDA).

Operators
- ***Variable*** is a user defined variable or constant.
- ***Control*** may be a constant value or a Byte sized variable expression.
- ***Address*** may be a constant value or a variable expression.

The four variations of the **Busin** command may be used within the same program. The *second* and *fourth* types are useful for simply receiving a single byte from the bus, and must be used in conjunction with one of the low level commands. i.e. **Bstart**, **Brestart**, **BusAck**, or **Bstop**. The *first*, and *third* types may be used to receive several values and designate each to a separate variable, or variable type.

The **Busin** command operates as an I^2C master, using the microcontroller's MSSP module, and may be used to interface with any device that complies with the 2-wire I^2C protocol.

The most significant 7-bits of control byte contain the control code and the slave address of the device being interfaced with. Bit-0 is the flag that indicates whether a read or write command is being implemented.

For example, if we were interfacing to an external eeprom such as the 24LC256, the control code would be %10100001 or $A1. The most significant 4-bits (1010) are the eeprom's unique slave address. Bits 1 to 3 reflect the three address pins of the eeprom. And bit-0 is set to signify that we wish to read from the eeprom. Note that this bit is automatically set by the **Busin** command, regardless of its initial setting.

Example

```
' Receive a byte from the I2C bus and place it into variable Var1.

   Dim Var1 as Byte                ' We'll only read 8-bits
   Dim Address as Word             ' 16-bit address required
   Symbol Control %10100001        ' Target an eeprom

   Address = 20                    ' Read the value at address 20
   Var1 = Busin Control, Address   ' Read the byte from the eeprom
```

or

```
   Busin Control, Address, [Var1] ' Read the byte from the eeprom
```

Address, is an optional parameter that may be an 8-bit or 16-bit value. If a variable is used in this position, the size of address is dictated by the size of the variable used (Byte or Word). In the case of the previous eeprom interfacing, the 24LC256 eeprom requires a 16-bit address. While the smaller types require an 8-bit address. Make sure you assign the right size address for the device interfaced with, or you may not achieve the results you intended.

The value received from the bus depends on the size of the variables used, except for variation three, which only receives a Byte (8-bits). For example:

```
   Dim WordVar as Word                  ' Create a Word size variable
   WordVar = Busin Control, Address
```

Will receive a 16-bit value from the bus. While:

```
   Dim Var1 as Byte                     ' Create a Byte size variable
   Var1 = Busin Control, Address
```

Will receive an 8-bit value from the bus.

Using the third variation of the **Busin** command allows differing variable assignments. For example:

```
   Dim Var1 as Byte
   Dim WordVar as Word
   Busin Control, Address, [Var1, WordVar]
```

Will receive two values from the bus, the first being an 8-bit value dictated by the size of variable Var1 which has been declared as a byte. And a 16-bit value, this time dictated by the size of the variable WordVar which has been declared as a word. Of course, Bit type variables may also be used, but in most cases these are not of any practical use as they still take up a byte within the eeprom.

The *second* and *fourth* variations allow all the subtleties of the I^2C protocol to be exploited, as each operation may be broken down into its constituent parts. It is advisable to refer to the datasheet of the device being interfaced to fully understand its requirements. See section on **Bstart**, **Brestart**, **BusAck**, or **Bstop**, for example code.

Declares

See **Busout** for declare explanations.

Notes

When the **Busout** command is used, the appropriate SDA and SCL Port and Pin are automatically setup as inputs, and outputs.

Because the I²C protocol calls for an open-collector interface, pull-up resistors are required on both the SDA and SCL lines. Values of 1KΩ to 4.7KΩ will suffice.

You may imagine that it's limiting having a fixed set of pins for the I²C interface, but you must remember that several different devices may be attached to a single bus, each having a unique slave address. Which means there is usually no need to use up more than two pins on the microcontroller in order to interface to many devices.

Str modifier with Busin

Using the **Str** modifier allows variations three and four of the **Busin** command to transfer the bytes received from the I²C bus directly into a byte array. If the amount of received characters is not enough to fill the entire array, then a formatter may be placed after the array's name, which will only receive characters until the specified length is reached. An example of each is shown below:

```
Dim Array[10] as Byte           ' Define an array of 10 bytes
Dim Address as Word             ' Create a word sized variable

Address = 0
Busin %10100000, Address, [Str Array] ' Load data into all the array
' Load data into only the first 5 elements of the array
Busin %10100000, Address, [Str Array\5]
Bstart                          ' Send a Start condition
Busout %10100000                ' Target an eeprom, and send a WRITE command
Busout 0                        ' Send the HighByte of the address
Busout 0                        ' Send the LowByte of the address
Brestart                        ' Send a Restart condition
Busout %10100001                ' Target an eeprom, and send a Read command
Busin Str Array                 ' Load all the array with bytes received
Bstop                           ' Send a Stop condition
```

An alternative ending to the above example is:

```
Busin Str Array\5       ' Load data into only the first 5 elements of the array
Bstop                   ' Send a Stop condition
```

See also : **BusAck**, **BusNack**, **Bstart**, **Brestart**, **Bstop**, **Busout**, **HbStart**, **HbRestart**, **HbusAck**, **HbusNack**, **Hbusin**, **Hbusout**.

Busout

Syntax
Busout Control, { Address }, [Variable {, Variable…}]

or

Busout Variable

Overview
Transmit a value to the I²C bus, by first sending the control and optional address out of the clock pin (SCL), and data pin (SDA). Or alternatively, if only one operator is included after the **Busout** command, a single value will be transmitted, along with an ACK reception.

Operators
- ***Variable*** is a user defined variable or constant.
- ***Control*** may be a constant value or a Byte sized variable expression.
- ***Address*** may be a constant, variable, or expression.

The **Busout** command operates as an I²C master and may be used to interface with any device that complies with the 2-wire I²C protocol.

The most significant 7-bits of control byte contain the control code and the slave address of the device being interfaced with. Bit-0 is the flag that indicates whether a read or write command is being implemented.

For example, if we were interfacing to an external eeprom such as the 24LC256, the control code would be %10100000 or $A0. The most significant 4-bits (1010) are the eeprom's unique slave address. Bits 1 to 3 reflect the three address pins of the eeprom. And Bit-0 is clear to signify that we wish to write to the eeprom. Note that this bit is automatically cleared by the **Busout** command, regardless of its initial value.

Example
```
' Send a byte to the I2C bus
  Dim Var1 as Byte                  ' We'll only read 8-bits
  Dim Address as Word               ' 16-bit address required
  Symbol Control = %10100000        ' Target an eeprom
  Address = 20                      ' Write to address 20
  Var1 = 200                        ' The value place into address 20
  Busout Control, Address, [Var1]   ' Send the byte to the eeprom
  DelayMs 10                        ' Allow time for allocation of byte
```

Address, is an optional parameter that may be an 8-bit or 16-bit value. If a variable is used in this position, the size of address is dictated by the size of the variable used (Byte or Word). In the case of the above eeprom interfacing, the 24LC256 eeprom requires a 16-bit address. While the smaller types require an 8-bit address. Make sure you assign the right size address for the device interfaced with, or you may not achieve the results you intended.

The value sent to the bus depends on the size of the variables used. For example:

```
Dim WordVar as Word              ' Create a Word size variable
Busout Control, Address, [WordVar]
```

Will send a 16-bit value to the bus. While:

```
Dim Var1 as Byte                 ' Create a Byte size variable
Busout Control, Address, [Var1]
```

Will send an 8-bit value to the bus.

Using more than one variable within the brackets allows differing variable sizes to be sent. For example:

```
Dim ByteVar as Byte
Dim WordVar as Word
Busout Control, Address, [Var1, WordVar]
```

Will send two values to the bus, the first being an 8-bit value dictated by the size of variable ByteVar which has been declared as a byte. And a 16-bit value, this time dictated by the size of the variable WordVar which has been declared as a word. Of course, Bit type variables may also be used, but in most cases these are not of any practical use as they still take up a byte within the eeprom.

A string of characters can also be transmitted, by enclosing them in quotes:

```
Busout Control, Address, ["Hello World", ByteVar, WordVar]
```

Using the second variation of the **Busout** command, necessitates using the low level commands i.e. **Bstart**, **Brestart**, **BusAck**, or **Bstop**.

Using the **Busout** command with only one value after it, sends a byte of data to the I^2C bus, and returns holding the Acknowledge reception. This acknowledge indicates whether the data has been received by the slave device.

The ACK reception is returned in the microcontroller's CARRY flag, which is STATUSbits_C, and also System variable PP4.0. A value of zero indicates that the data was received correctly, while a one indicates that the data was not received, or that the slave device has sent a NACK return. You must read and understand the datasheet for the device being interfacing to, before the ACK return can be used successfully. An code snippet is shown below:

```
' Transmit a byte to a 24LC256 I2C eeprom
  Dim PP4 as Byte System     ' Bring the system variable into the BASIC program

  Bstart                     ' Send a Start condition
  Busout %10100000           ' Target an eeprom, and send a WRITE command
  Busout 0                   ' Send the High Byte of the address
  Busout 0                   ' Send the Low Byte of the address
  Busout "A"                 ' Send the value 65 to the bus
  If PP4.0 = 1 Then GoTo Not_Received ' Has ACK been received OK ?
  Bstop                      ' Send a Stop condition
  DelayMs 10                 ' Wait for the data to be entered into eeprom
  Stop
Not_Received:                ' Come here if the byte was not received correctly
  Hrsout "A"
```


Str modifier with Busout.
The **Str** modifier is used for transmitting a string of bytes from a byte array variable. A string is a set of bytes sized values that are arranged or accessed in a certain order. The values 1, 2, 3 would be stored in a string with the value 1 first, followed by 2 then followed by the value 3. A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string 1,2,3 would be stored in a byte array containing three bytes (elements).

Below is an example that sends four bytes from an array:

```
Dim MyArray[10] as Byte          ' Create a 10-byte array.
MyArray[0] = "A"                 ' Load the first 4 bytes of the array
MyArray[1] = "B"                 ' With the data to send
MyArray[2] = "C"
MyArray[3] = "D"
Busout %10100000, Address, [Str MyArray\4] ' Send 4-byte string.
```

Note that we use the optional \n argument of **Str**. If we didn't specify this, the program would try to keep sending characters until all 10 bytes of the array were transmitted. Since we do not wish all 10 bytes to be transmitted, we chose to tell it explicitly to only send the first 4 bytes.

The above example may also be written as:

```
Dim MyArray[10] as Byte        ' Create a 10-byte array
Str MyArray = "ABCD"           ' Load the first 4 bytes of the array
Bstart                         ' Send a Start condition
Busout %10100000               ' Target an eeprom, and send a WRITE command
Busout 0                       ' Send the High Byte of the address
Busout 0                       ' Send the Low Byte of the address
Busout Str MyArray\4           ' Send 4-byte string.
Bstop                          ' Send a Stop condition
```

The above example, has exactly the same function as the previous one. The only differences are that the string is now constructed using the **Str** as a command instead of a modifier, and the low-level HBUS commands have been used.

Declares
There are three **Declare** directives for use with **Busout**.
These are:

Declare SDA_Pin Port . Pin
Declares the port and pin used for the data line (SDA). This may be any valid port on the microcontroller. If this declare is not issued in the BASIC program, then the default Port and Pin is PortA.0

Declare SCL_Pin Port . Pin
Declares the port and pin used for the clock line (SCL). This may be any valid port on the microcontroller. If this declare is not issued in the BASIC program, then the default Port and Pin is PortA.1

These declares, as is the case with all the Declares, may only be issued once in any single program, as they setup the I^2C library code at design time.

Declare Slow_Bus On - Off or 1 - 0
Slows the bus speed when using an oscillator higher than 4MHz.

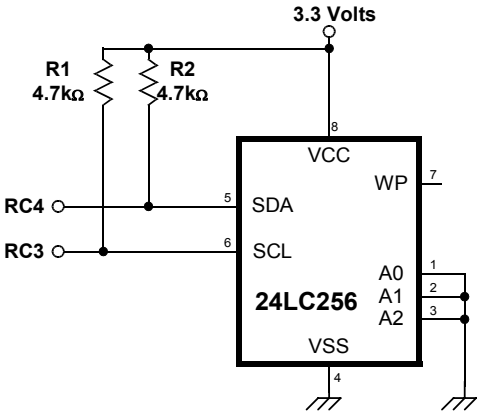
The standard speed for the I²C bus is 100KHz. Some devices use a higher bus speed of 400KHz. If you use an 8MHz or higher oscillator, the bus speed may exceed the devices specs, which will result in intermittent transactions, or in some cases, no transactions at all. Therefore, use this **Declare** if you are not sure of the device's spec. The datasheet for the device used will inform you of its bus speed.

Notes
When the **Busout** command is used, the appropriate SDA and SCL Port and Pin are automatically setup as inputs, and outputs.

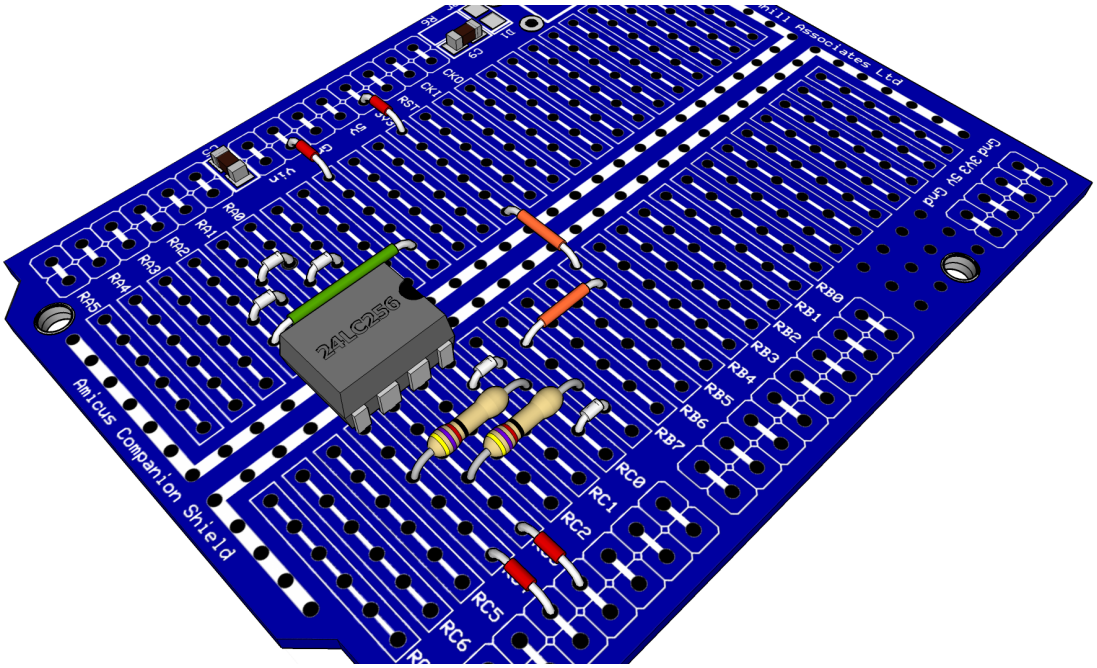
Because the I²C protocol calls for an open-collector interface, pull-up resistors are required on both the SDA and SCL lines. Values of 1KΩ to 4.7KΩ will suffice.

You may imagine that it's limiting having a fixed set of pins for the I²C interface, but you must remember that several different devices may be attached to a single bus, each having a unique slave address. Which means there is usually no need to use up more than two pins on the microcontroller, in order to interface to many devices.

A typical use for the I²C commands is for interfacing with serial eeproms. Shown right are the connections to the I²C bus of a 24LC256 serial eeprom.



And here's what it would look like built on the Amicus Companion shield:



See also : **BusAck**, **BusNack**, **Bstart**, **Brestart**, **Bstop**, **Busin**, **HbStart**, **HbRestart**, **HbusAck**, **HbusNack**, **Hbusin**, **Hbusout**.

Button

Syntax
Button Pin, DownState, Delay, Rate, Workspace, TargetState, Label

Overview
Debounce button input, perform auto-repeat, and branch to address if button is in target state.
Button circuits may be active-low or active-high.

Operators
- ***Pin*** is a Port.Bit, constant, or variable (0 - 15), that specifies the I/O pin to use. This pin will automatically be set to input.
- ***DownState*** is a variable, constant, or expression (0 or 1) that specifies which logical state occurs when the button is pressed.
- ***Delay*** is a variable, constant, or expression (0 - 255) that specifies how long the button must be pressed before auto-repeat starts. The delay is measured in cycles of the Button routine. Delay has two special settings: 0 and 255. If Delay is 0, Button performs no debounce or auto-repeat. If Delay is 255, Button performs debounce, but no auto-repeat.
- ***Rate*** is a variable, constant, or expression (0 – 255) that specifies the number of cycles between auto-repeats. The rate is expressed in cycles of the Button routine.
- ***Workspace*** is a byte variable used by Button for workspace. It must be cleared to 0 before being used by Button for the first time and should not be adjusted outside of the Button command.
- ***TargetState*** is a variable, constant, or expression (0 or 1) that specifies which state the button should be in for a branch to occur. (0 = not pressed, 1 = pressed).
- ***Label*** is a label that specifies where to branch if the button is in the target state.

Example
```
   Dim BtnVar as Byte          ' Workspace for Button instruction.
Loop:
' Go to NoPress unless BtnVar = 0
   Button 0, 0, 255, 250, BtnVar, 0, NoPress
   HRsout "Button Pressed\r"
NoPress:
   GoTo Loop
```

Notes
When a button is pressed, the contacts make or break a connection. A short (1 to 20ms) burst of noise occurs as the contacts scrape and bounce against each other. Button's debounce feature prevents this noise from being interpreted as more than one switch action.

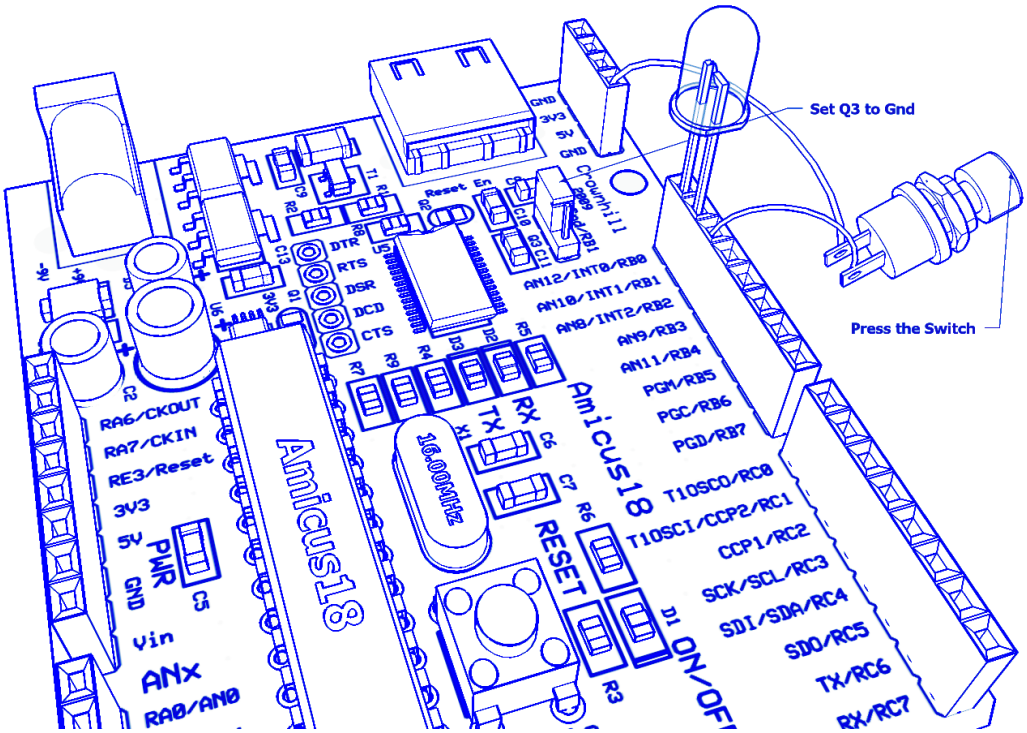
Button also reacts to a button press the way a computer keyboard does to a key press. When a key is pressed, a character immediately appears on the screen. If the key is held down, there's a delay, then a rapid stream of characters appears on the screen. Button's auto-repeat function can be set up to work much the same way.

Button is designed for use inside a program loop. Each time through the loop, Button checks the state of the specified pin. When it first matches DownState, the switch is debounced. Then, as dictated by TargetState, it either branches to address (TargetState = 1) or doesn't (TargetState = 0).

If the switch stays in *DownState*, **Button** counts the number of program loops that execute. When this count equals *Delay*, **Button** once again triggers the action specified by *TargetState* and address. Thereafter, if the switch remains in *DownState*, **Button** waits *Rate* number of cycles between actions. The *Workspace* variable is used by **Button** to keep track of how many cycles have occurred since the pin switched to *TargetState* or since the last auto-repeat.

Button does not stop program execution. In order for its delay and auto repeat functions to work properly, **Button** must be executed from within a program loop.

A suitable layout for use with **Button** is shown below.



The suitable program to allow the layout above to operate is shown below:

```
   Dim BtnVar as Byte          ' Workspace for Button instruction.
   Symbol LED = RB1            ' Alias name LED to RB0 (PortB.0)
   While 1 = 1                 ' Create an endless loop
' Go to NoPress unless BtnVar = 0.
     Button 0, 0, 255, 250, BtnVar, 0, NoPress
     High LED                  ' Illuminate the LED
     DelayMs 1000              ' Wait for 1 second
NoPress:
     Low LED                   ' Extinguish the LED if no button pressed
   Wend                        ' Do it forever
```

When the button is pressed, the LED will illuminate for 1 second.

Don't forget to move jumper Q3 to the *Gnd* position, so that the RB1 line becomes Ground.

Call

Syntax
Call Label

Overview
Execute the assembly language subroutine named label.

Operators
- ***Label*** must be a valid label name.

Example
```
' Call an assembler routine
  Call Asm_Sub

  Asm
     Asm_Sub
     {mnemonics}
     Return
  EndAsm
```

Notes
The **GoSub** command is usually used to execute a BASIC subroutine. However, if your subroutine happens to be written in assembler, the **Call** command should be used. The main difference between **GoSub** and **Call** is that when **Call** is used, the label's existence is not checked until assembly time. Using **Call**, a label in an assembly language section can be accessed that would otherwise be inaccessible to **GoSub**. This also means that any errors produced will be assembler types.

The **Call** command adds Bank switching instructions prior to actually calling the subroutine, however, if **Call** is used in an all assembler environment, the extra mnemonics preceding the command can interfere with carefully sculptured code such **as Bit** tests etc. By wrapping the subroutine's name in parenthesis, the Bank instruction is suppressed, and the **Call** command becomes the **Call** mnemonic.

```
  Call (Subroutine_Name)
```

See also : **GoSub**, **GoTo.**

Cdata

Syntax
Cdata { alphanumeric data }

Overview
Place information directly into memory for access by **Cread** and **Cwrite**.

Operators
- alphanumeric data can be any value, alphabetic character, or string enclosed in quotes (") or numeric data without quotes.

The **Cread**, **Lread**, **Lread8**, **Lread16**, **Lread32** and **Cwrite** commands can use a label address as a location variable. For example:

Example
```
  Dim Dbyte as Byte
  Dim Loop as Byte
  For Loop = 0 to 9                ' Create a loop of 10
    Dbyte = Cread CodeString + Loop   ' Read memory location CodeString+ Loop
    HRsout Dbyte                   ' Display the value read
  Next
  HRsout 13                        ' Terminate the line
  For Loop = 0 to 9                ' Create a loop of 10
    Dbyte = Lread CodeString + Loop   ' Read memory location CodeString + Loop
    HRsout Dbyte                   ' Display the value read
  Next
  Stop
CodeString:
  Cdata "Hello World"              ' Create a string of text in code memory
```

The program above reads and displays 10 values from the address located by the Label accompanying the **Cdata** command. Resulting in "Hello Worl" being displayed.

Using the read8 command is even easier:

```
  Dim Dbyte as Byte
  Dim Loop as Byte
  For Loop = 0 to 9                ' Create a loop of 10
    Dbyte = Lread8 CodeString[Loop]   ' Read memory location CodeString + Loop
    HRsout Dbyte                   ' Display the value read
  Next
  HRsout 13                        ' Terminate the line
  Stop
CodeString:
  Cdata "Hello World"              ' Create a string of text in code memory
```


Formatting a Cdata table.
Sometimes it is necessary to create a data table with a known format for its values. For example all values will occupy 4 bytes of data space even though the value itself would only occupy 1 or 2 bytes.

```
Cdata 100000, 10000, 1000, 100, 10, 1
```

The above line of code would produce an uneven code space usage, as each value requires a different amount of code space to hold the values. 100000 would require 4 bytes of code space, 10000 and 1000 would require 2 bytes, but 100, 10, and 1 would only require 1 byte.

Reading these values using **Cread** or **Lread** would cause problems because there is no way of knowing the amount of bytes to read in order to increment to the next valid value.

The answer is to use formatters to ensure that a value occupies a predetermined amount of bytes. These are:

Byte
Word
Dword
Float

Placing one of these formatters before the value in question will force a given length.

```
Cdata  Dword 100000, Dword 10000, Dword 1000 ,_
       Dword 100, Dword 10, Dword 1
```

Byte will force the value to occupy one byte of code space, regardless of it's value. Any values above 255 will be truncated to the least significant byte.

Word will force the value to occupy 2 bytes of code space, regardless of its value. Any values above 65535 will be truncated to the two least significant bytes. Any value below 255 will be padded to bring the memory count to 2 bytes.

Dword will force the value to occupy 4 bytes of code space, regardless of its value. Any value below 65535 will be padded to bring the memory count to 4 bytes. The line of code shown above uses the Dword formatter to ensure all the values in the **Cdata** table occupy 4 bytes of code space.

Float will force a value to its floating point equivalent, which always takes up 4 bytes of code space.

If all the values in an **Cdata** table are required to occupy the same amount of bytes, then a single formatter will ensure that this happens.

```
   Cdata as Dword 100000, 10000, 1000, 100, 10, 1
```

The above line has the same effect as the formatter previous example using separate Dword formatters, in that all values will occupy 4 bytes, regardless of their value. All four formatters can be used with the as keyword.

The example below illustrates the formatters in use.

```
' Convert a Dword value into a string array using only BASIC commands
' Similar principle to the Str$ command

   Dim P10 as Dword              ' Power of 10 variable
   Dim Cnt as Byte
   Dim J as Byte
   Dim Value as Dword            ' Value to convert
   Dim String1 as String * 11    ' Holds the converted value
   Dim Ptr as Byte               ' Pointer within the Byte array

   Clear                         ' Clear all RAM before we start
   Value = 1234576               ' Value to convert
   GoSub DwordToStr              ' Convert Value to string
   HRsout String1, 13            ' Display the result
   Stop
' Convert a Dword value into a string array
' Value to convert is placed in 'Value'
' Byte array 'String1' is built up with the ASCII equivalent
DwordToStr:
   Ptr = 0
   J = 0
   Repeat
      P10 = Cread DwordTbl + (J * 4)
      Cnt = 0
      While Value >= P10
         Value = Value - P10
         Inc Cnt
      Wend
      If Cnt <> 0 Then
         String1[Ptr] = Cnt + "0"
         Inc Ptr
      EndIf
      Inc J
   Until J > 8

   String1[Ptr] = Value + "0"
   Inc Ptr
   String1[Ptr] = 0              ' Add the null to terminate the string
   Return

' Cdata table is formatted for all 32 bit values.
' Which means each value will require 4 bytes of code space
DwordTbl:
   Cdata as Dword 1000000000, 100000000, 10000000, 1000000, 100000,_
                  10000, 1000, 100, 10
```


Label names as pointers.
If a label's name is used in the list of values in a **Cdata** table, the labels address will be used. This is useful for accessing other tables of data using their address from a lookup table. See example below.

```
' Display text from two Cdata tables
' Based on their address located in a separate table

  Dim Address as Word
  Dim Loop as Word
  Dim DataByte as Byte

  Address = Cread AddrTable       ' Locate the address of the first string
  While 1 = 1                     ' Create an infinite loop
    DataByte = Cread Address      ' Read each character from the Cdata string
    If DataByte = 0 Then Break    ' Exit if null found
    HRsout DataByte               ' Display the character
    Inc Address                   ' Next character
  Wend                            ' Close the loop
  Hrsout 13                       ' Move to line 2 of the serial terminal

  Address = Cread AddrTable + 2   ' Locate the address of the second string
  While 1 = 1                     ' Create an infinite loop
    DataByte = Cread Address      ' Read each character from the Cdata string
    If DataByte = 0 Then Break    ' Exit if null found
    Print DataByte                ' Display the character
    Inc Address                   ' Next character
  Wend                            ' Close the loop
  Stop

AddrTable:                        ' Table of address's
  Cdata Word String1, Word String2
String1:
  Cdata "HELLO", 0
String2:
  Cdata "WORLD", 0
```

See also : **Cread, Cwrite, Cdata, Lread, Lread8, Lread16, Lread32.**

Cerase

Syntax
Cerase Address

Overview
Erase code memory in blocks of 64 bytes.

Operators
- ***Address*** is a constant, variable, label, or expression that represents a valid location within the microcontroller's code (flash) memory.

Example
```
' Write to a code memory location within the microcontroller

  Dim Var1 as Byte
  Dim WordVar as Word
  Var1 = 234
  WordVar = 1043
'
' Erase a 64 byte block of code memory
'
  Cerase WriteHere
'
' Write to code memory, as a block of 32 bytes
'
  Cwrite WriteHere, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10,_
                     11, 12, 13, 14, 15, 16, 17, 18, 19, 20,_
                     21, 22, 23, 24, 25, 26, 27, 28, 29, Var1, WordVar]
  Stop
'
' This is where the data will be written too
' It must be placed on a 64 byte boundary
'
  Org $0200           ' Place the label WriteHere at address $0200 (512)
                      ' Make sure this will not overwrite your existing code
WriteHere:
```

Notes
Code memory must be erased using the **Cerase** command before writing to it. Code memory can only be erased in blocks of 64 bytes and on a 64 byte boundary.

Code memory can only be written in blocks of 32 bytes. Any less will not be written.

See section 6.0 of the PIC18F25K20 data sheet for more details concerning code (flash) memory reading, writing and erasing.

See also : **Cdata**, **Cread, Cwrite, Lread, Lread8, Lread16, Lread32**.

Circle

Syntax
Circle Set_Clear, Xpos, Ypos, Radius

Overview
Draw a circle on a graphic LCD.

Operators
- ***Set_Clear*** may be a constant or variable that determines if the circle will set or clear the pixels. A value of 1 will set the pixels and draw a circle, while a value of 0 will clear any pixels and erase a circle.
- ***Xpos*** may be a constant or variable that holds the X position for the centre of the circle. Can be a value from 0 to the X resolution of the display.
- ***Ypos*** may be a constant or variable that holds the Y position for the centre of the circle. Can be a value from 0 to the Y resolution of the display.
- ***Radius*** may be a constant or variable that holds the Radius of the circle. Can be a value from 0 to 255.

Example
```
' Draw circle at pos63,32 with a radius of 20 pixels on a Samsung KS0108 LCD

  Dim Xpos as Byte
  Dim Ypos as Byte
  Dim Radius as Byte
  Dim SetClr as Byte

  DelayMs 200           ' Wait for things to stabilise
  Cls                   ' Clear the LCD
  Xpos = 63
  Ypos = 32
  Radius = 20
  SetClr = 1
  Circle SetClr, Xpos, Ypos, Radius
```

Notes
Because of the aspect ratio of the pixels on the samsung graphic LCD (approx 1.5 times higher than wide) the circle will appear elongated.

See Also : **Box**, **Line**, **Pixel**, **Plot**, **UnPlot**.

Clear

Syntax
Clear Variable or Variable.Bit

or

Clear

Overview
Place a variable or bit in a low state. For a variable, this means filling it with 0's. For a bit this means setting it to 0.

Clear has another purpose. If no variable is present after the command, all RAM area on the microcontroller is cleared.

Operators
- ***Variable*** can be any variable or register.
- ***Variable.Bit*** can be any variable and bit combination.

Example
```
Clear               ' Clear ALL RAM area
Clear Var1.3        ' Clear bit 3 of Var1
Clear Var1          ' Load Var1 with the value of 0
Clear STATUS.0      ' Clear the carry flag
Clear Array         ' Clear all of an Array variable. i.e. Reset to zero's
Clear String1       ' Clear all of a String variable. i.e. Reset to zero's
```

Notes
There is a major difference between the **Clear** and **Low** commands. **Clear** does not alter the Tris register if a Port is targeted.

See Also : **Set, Low, High**

ClearBit

Syntax
ClearBit Variable, Index

Overview
Clear a bit of a variable or register using a variable index to the bit of interest.

Operators
- ***Variable*** is a user defined variable, of type Byte, Word, or Dword.
- ***Index*** is a constant, variable, or expression that points to the bit within Variable that requires clearing.

Example
```
' Clear then Set each bit of variable ExVar
  Dim ExVar as Byte
  Dim Index as Byte
  ExVar = %11111111
  For Index = 0 to 7            ' Create a loop for 8 bits
    ClearBit ExVar,Index        ' Clear each bit of ExVar
    HRsout Bin8 ExVar,13        ' Display the binary result
    DelayMs 100                 ' Slow things down to see what's happening
  Next                          ' Close the loop
  HRsout 13
 For Index = 7 to 0 Step -1     ' Create a loop for 8 bits
    SetBit ExVar,Index          ' Set each bit of ExVar
    HRsout Bin8 ExVar, 13       ' Display the binary result
    DelayMs 100                 ' Slow things down to see what's happening
  Next                          ' Close the loop
  Hrsout 13
```

Notes
There are many ways to clear a bit within a variable, however, each method requires a certain amount of manipulation, either with rotates, or alternatively, the use of indirect addressing using the FSRx, and INDFx registers. Each method has its merits, but requires a certain amount of knowledge to accomplish the task correctly. The **ClearBit** command makes this task extremely simple using a register rotate method, however, this is not necessarily the quickest method, or the smallest, but it is the easiest. For speed and size optimisation, there is no shortcut to experience.

To Clear a known constant bit of a variable or register, then access the bit directly using Port.n.

```
  PortA.1 = 0
```
or
```
  Var1.4 = 0
```

If a Port is targeted by **ClearBit**, the Tris register is not affected.

See also : **GetBit**, **LoadBit**, **SetBit**.

Cls

Syntax
Cls

Or if using a Toshiba T6963 graphic LCD

Cls Text
Cls Graphic

Overview
Clears the alphanumeric or graphic LCD and places the cursor at the home position i.e. line 1, position 1 (line 0, position 0 for graphic LCDs).

Toshiba graphic LCDs based upon the T6963 chipset have separate RAM for text and graphics. Issuing the word Text after the **Cls** command will only clear the Text RAM, while issuing the word Graphic after the **Cls** command will only clear the Graphic RAM. Issuing the **Cls** command on its own will clear both areas of RAM.

Example 1
```
' Clear an alphanumeric or Samsung KS0108 graphic LCD
  Cls                 ' Clear the LCD
  Print "HELLO"       ' Display the word "HELLO" on the LCD
  Cursor 2, 1         ' Move the cursor to line 2, position 1
  Print "WORLD"       ' Display the word "WORLD" on the LCD
  Stop
```

In the above example, the LCD is cleared using the **Cls** command, which also places the cursor at the home position i.e. line 1, position 1. Next, the word HELLO is displayed in the top left corner. The cursor is then moved to line 2 position 1, and the word WORLD is displayed.

Example 2
```
' Clear a Toshiba T6963 graphic LCD.
  Cls                 ' Clear all RAM within the LCD
  Print "HELLO"       ' Display the word "HELLO" on the LCD
  Line 1,0,0,63,63    ' Draw a line on the LCD
  DelayMs 1000        ' Wait for 1 second
  Cls Text            ' Clear only the text RAM, leaving the line displayed
  DelayMs 1000        ' Wait for 1 second
  Cls Graphic         ' Now clear the line from the display
  Stop
```

Notes
The **Cls** command will also initialise any of the above LCDs. (set the ports to inputs/outputs etc), however, this is most important to Toshiba graphic LCDs, and the **Cls** command should always be placed at the head of the BASIC program, prior to issuing any command that interfaces with the LCD. i.e. **Print**, **Plot** etc.

See also : **Cursor, Print, Toshiba_Command.**

Config_Start – Config_End

Syntax
Config_Start
{ configuration fuse settings }
Config_End

Overview
Enable or Disable particular fuse settings on the microcontroller.

Operators
Refer to the PIC18F25K20 data sheet (section 23) for details concerning fuse settings.

Example
```
' Disable Watchdog timer and specify an HS oscillator etc
Config_Start
  DEBUG = Off     ' Background debugger disabled; RB6/RB7 configured as I/O
  XINST = Off     ' Instruction set extension mode disabled
  STVREN = Off    ' Reset on stack overflow/underflow disabled
  WDTEN = Off     ' WatchDog Timer disabled (control is placed on SWDTEN bit)
  FCMEN = Off     ' Fail-Safe Clock Monitor disabled
  FOSC = HSPLL    ' HS oscillator, PLL enabled and under software control
  IESO = Off      ' Two-Speed Start-up disabled
  WDTPS = 128     ' Watchdog oscillator prescaler 1:128
  BOREN = Off     ' Brown-out Reset disabled in hardware and software
  BORV = 18       ' VBOR set to 1.8 V nominal
  MCLRE = On      ' MCLR pin enabled, RE3 input pin disabled
  HFOFST = Off    ' The clock is held off until the HF-INTOSC is stable.
  LPT1OSC = On    ' Timer1 operates in standard power mode
  PBADEN = Off    ' PORTB<4:0> pins are configured as digital I/O on Reset
  CCP2MX = PORTC  ' CCP2 input/output is multiplexed with RC1
  LVP = Off       ' Single-Supply ICSP disabled
  CP0 = Off       ' Block 0 (000800-001FFFh) not code-protected
  CP1 = Off       ' Block 1 (002000-003FFFh) not code-protected
  CPB = Off       ' Boot block (000000-0007FFh) not code-protected
  CPD = Off       ' Data EEPROM not code-protected
  WRT0 = Off      ' Block 0 (000800-001FFFh) not write-protected
  WRT1 = Off      ' Block 1 (002000-003FFFh) not write-protected
  WRTB = Off      ' Boot block (000000-0007FFh) not write-protected
  WRTC = Off      ' Config registers (300000-3000FFh) not write-protected
  WRTD = Off      ' Data EEPROM not write-protected
  EBTR0 = Off     ' Block 0 (000800-001FFFh) not protected from table reads
                  ' executed in other blocks
  EBTR1 = Off     ' Block 1 (002000-003FFFh) not protected from table reads
                  ' executed in other blocks
  EBTRB = Off     ' Boot block (000000-0007FFh) not protected from reads
                  ' executed in other blocks
Config_End
```

The configs shown are the defaults used within the Amicus18 microcontroller.

Any errors in the fuse setting texts will result in Assembler errors being produced.

Notes
Always apply all the fuse settings, even if they are being disabled.

It's important to remember that custom fuse configurations will not take effect when using the boot-loader, as this defaults to the above settings. In order to use custom fuse settings, a device programmer such as the PICkit2 will be required.

Below are all the fuse setting that are allowed within the **Config_Start** and **Config_End** block.

Oscillator Selection bits:

FOSC = LP	LP oscillator
FOSC = XT	XT oscillator
FOSC = HS	HS oscillator
FOSC = RC	External RC oscillator, CLKOUT function on RA6
FOSC = EC	EC oscillator, CLKOUT function on RA6
FOSC = ECIO6	EC oscillator, port function on RA6
FOSC = HSPLL	HS oscillator, PLL enabled (Clock Frequency = 4 x FOSC1)
FOSC = RCIO6	External RC oscillator, port function on RA6
FOSC = INTIO67	Internal oscillator block, port function on RA6 and RA7
FOSC = INTIO7	Internal oscillator block, CLKOUT function on RA6, port function on RA7

Fail-Safe Clock Monitor Enable bit:

FCMEN = OFF	Fail-Safe Clock Monitor disabled
FCMEN = ON	Fail-Safe Clock Monitor enabled

Internal/External Oscillator Switchover bit:

IESO = OFF	Oscillator Switchover mode disabled
IESO = ON	Oscillator Switchover mode enabled

Power-up Timer Enable bit:

PWRT = ON	PWRT enabled
PWRT = OFF	PWRT disabled

Brown-out Reset Enable bits:

BOREN = OFF	Brown-out Reset disabled in hardware and software
BOREN = ON	Brown-out Reset enabled and controlled by software (SBOREN is enabled)
BOREN = NOSLP	Brown-out Reset enabled in hardware only and disabled in Sleep mode (SBOREN is disabled)
BOREN = SBORDIS	Brown-out Reset enabled in hardware only (SBOREN is disabled)

Brown Out Voltage:

BORV = 30	VBOR set to 3.0 V nominal
BORV = 27	VBOR set to 2.7 V nominal
BORV = 22	VBOR set to 2.2 V nominal
BORV = 18	VBOR set to 1.8 V nominal

Watchdog Timer Enable bit:

WDTEN = OFF	WDT is controlled by SWDTEN bit of the WDTCON register
WDTEN = ON	WDT is always enabled. SWDTEN bit has no effect.

Watchdog Timer Postscale Select bits:

WDTPS = 1	1:1
WDTPS = 2	1:2
WDTPS = 4	1:4
WDTPS = 8	1:8
WDTPS = 16	1:16
WDTPS = 32	1:32
WDTPS = 64	1:64
WDTPS = 128	1:128
WDTPS = 256	1:256
WDTPS = 512	1:512
WDTPS = 1024	1:1024
WDTPS = 2048	1:2048
WDTPS = 4096	1:4096
WDTPS = 8192	1:8192
WDTPS = 16384	1:16384
WDTPS = 32768	1:32768

MCLR Pin Enable bit:

MCLRE = OFF	RE3 input pin enabled; MCLR disabled
MCLRE = ON	MCLR pin enabled, RE3 input pin disabled

HF-INTOSC Fast Startup:

HFOFST = OFF	The system clock is held off until the HF-INTOSC is stable.
HFOFST = ON	HF-INTOSC starts clocking the CPU without waiting for the oscillator to stablize.

Low-Power Timer1 Oscillator Enable bit:

LPT1OSC = OFF	Disabled, T1 operates in standard power mode.
LPT1OSC = ON	Timer1 configured for low-power operation

PORTB A/D Enable bit:

PBADEN = OFF	PORTB<4:0> pins are configured as digital I/O on Reset
PBADEN = ON	PORTB<4:0> pins are configured as analog input channels on Reset

CCP2 Mux bit:

CCP2MX = PORTBE	CCP2 input/output is multiplexed with RB3
CCP2MX = PORTC	CCP2 input/output is multiplexed with RC1

Stack Full/Underflow Reset Enable bit:

STVREN = OFF	Stack full/underflow will not cause Reset
STVREN = ON	Stack full/underflow will cause Reset

Single-Supply ICSP Enable bit:

LVP = OFF	Single-Supply ICSP disabled
LVP = ON	Single-Supply ICSP enabled

Extended Instruction Set Enable bit:

XINST = OFF	Instruction set extension and Indexed Addressing mode disabled (Legacy mode)
XINST = ON	Instruction set extension and Indexed Addressing mode enabled

Background Debugger Enable bit:

DEBUG = ON	Background debugger enabled, RB6 and RB7 are dedicated to In-Circuit Debug
DEBUG = OFF	Background debugger disabled, RB6 and RB7 configured as general purpose I/O pins

Code Protection Block 0:

CP0 = ON	Block 0 (000800-001FFFh) code-protected
CP0 = OFF	Block 0 (000800-001FFFh) not code-protected

Code Protection Block 1:

CP1 = ON	Block 1 (002000-003FFFh) code-protected
CP1 = OFF	Block 1 (002000-003FFFh) not code-protected

Code Protection Block 2:

CP2 = ON	Block 2 (004000-005FFFh) code-protected
CP2 = OFF	Block 2 (004000-005FFFh) not code-protected

Code Protection Block 3:

CP3 = ON	Block 3 (006000-007FFFh) code-protected
CP3 = OFF	Block 3 (006000-007FFFh) not code-protected

Boot Block Code Protection bit:

CPB = ON	Boot block (000000-0007FFh) code-protected
CPB = OFF	Boot block (000000-0007FFh) not code-protected

Data EEPROM Code Protection bit:

CPD = ON	Data EEPROM code-protected
CPD = OFF	Data EEPROM not code-protected

Write Protection Block 0:

WRT0 = ON	Block 0 (000800-001FFFh) write-protected
WRT0 = OFF	Block 0 (000800-001FFFh) not write-protected

Write Protection Block 1:

WRT1 = ON	Block 1 (002000-003FFFh) write-protected
WRT1 = OFF	Block 1 (002000-003FFFh) not write-protected

Write Protection Block 2:

WRT2 = ON	Block 2 (004000-005FFFh) write-protected
WRT2 = OFF	Block 2 (004000-005FFFh) not write-protected

Write Protection Block 3:

WRT3 = ON	Block 3 (006000-007FFFh) write-protected
WRT3 = OFF	Block 3 (006000-007FFFh) not write-protected

Boot Block Write Protection bit:

WRTB = ON	Boot block (000000-0007FFh) write-protected
WRTB = OFF	Boot block (000000-0007FFh) not write-protected

Configuration Register Write Protection bit:

WRTC = ON	Configuration registers (300000-3000FFh) write-protected
WRTC = OFF	Configuration registers (300000-3000FFh) not write-protected

Data EEPROM Write Protection bit:

WRTD = ON	Data EEPROM write-protected
WRTD = OFF	Data EEPROM not write-protected

Table Read Protection Block 0:

EBTR0 = ON	Block 0 (000800-001FFFh) protected from table reads executed in other blocks
EBTR0 = OFF	Block 0 (000800-001FFFh) not protected from table reads executed in other blocks

Table Read Protection Block 1:

EBTR1 = ON	Block 1 (002000-003FFFh) protected from table reads executed in other blocks
EBTR1 = OFF	Block 1 (002000-003FFFh) not protected from table reads executed in other blocks

Table Read Protection Block 2:

EBTR2 = ON	Block 2 (004000-005FFFh) protected from table reads executed in other blocks
EBTR2 = OFF	Block 2 (004000-005FFFh) not protected from table reads executed in other blocks

Table Read Protection Block 3:

EBTR3 = ON	Block 3 (006000-007FFFh) protected from table reads executed in other blocks
EBTR3 = OFF	Block 3 (006000-007FFFh) not protected from table reads executed in other blocks

Boot Block Table Read Protection bit:

EBTRB = ON	Boot block (000000-0007FFh) protected from table reads executed in other blocks
EBTRB = OFF	Boot block (000000-0007FFh) not protected from table reads executed in other blocks

Counter

Syntax
Variable = **Counter** Pin, Period

Overview
Count the number of pulses that appear on pin during period, and store the result in variable.

Operators
- ***Variable*** is a user-defined variable.
- ***Pin*** is a Port.Pin constant declaration i.e. PortA.0.
- ***Period*** may be a constant, variable, or expression.

Example
```
' Count the pulses that occur on PortA.0 within a 100ms period
' and displays the results.

   Dim WordVar as Word           ' Create a word size variable
   Symbol Pin = PA0              ' Assign the input pin to bit-0 of PortA
Loop:
   WordVar = Counter Pin, 100' Variable WordVar now contains the Count
   HRsout Dec WordVar,13         ' Display decimal result on serial terminal
   DelayMs 300
  GoTo Loop                      ' Do it forever
```

Notes
The resolution of period is in milliseconds (ms). It obtains its scaling from the oscillator declaration, **Declare** Xtal.

Counter checks the state of the pin in a concise loop, and counts the rising edge of a transition (low to high).

With a 4MHz oscillator, the pin is checked every 20us, and every 1.28us with a 64MHz oscillator. From this we can determine that the highest frequency of pulses that may be counted is:

25KHz using a 4MHz oscillator.
1.28MHz using the default 64MHz oscillator.

See also : **PulsIn, Rcin.**

Cread

Syntax
Variable = **Cread** Address

Overview
Read data from anywhere in code memory.

Operators
- ***Variable*** is a user defined variable, of type Byte, Word, Dword, or Float.
- ***Address*** is a constant, variable, label, or expression that represents any valid address within the Amicus18 microcontroller.

Example
```
' Read code memory locations within the Amicus18 microcontroller

   Dim Var1 as Byte
   Dim WordVar as Word
   Dim Address as Word
   Address = 1000                ' Address now holds the base address
   Var1 = Cread 1000             ' Read 8-bit data at address 1000 into Var1
   WordVar = Cread Address + 10  ' Read 16-bit data at address 1000 + 10
   WordVar = Cread Label + 10    ' Read 16-bit data at Label + 10
   Stop
Label: Cdata "Hello World"
```

Note. **Cread** and **Lread** are interchangeable.

See also : **Cdata, Cwrite, Cdata, Lread, Lread8, Lread16, Lread32.**

Cursor

Syntax
Cursor Line, Position

Overview
Move the cursor position on an Alphanumeric or Graphic LCD to a specified line (ypos) and position (xpos).

Operators
- ***Line*** is a constant, variable, or expression that corresponds to the line (Ypos) number from 1 to maximum lines (0 to maximum lines if using a graphic LCD).
- ***Position*** is a constant, variable, or expression that moves the position within the position (Xpos) chosen, from 1 to maximum position (0 to maximum position if using a graphic LCD).

Example 1
```
  Dim bLine as Byte
  Dim bXpos as Byte
  bLine = 2
  bXpos = 1
  Cls                   ' Clear the LCD
  Print "HELLO"          ' Display the word "HELLO" on the LCD
  Cursor bLine, bXpos   ' Move the cursor to line 2, position 1
  Print "WORLD"          ' Display the word "WORLD" on the LCD
```

In the above example, the LCD is cleared using the **Cls** command, which also places the cursor at the home position i.e. line 1, position 1. Next, the word HELLO is displayed in the top left corner. The cursor is then moved to line 2 position 1, and the word WORLD is displayed.

Example 2
```
  Dim Xpos as Byte
  Dim Ypos as Byte
Again:
  Ypos = 1                ' Start on line 1
  For Xpos = 1 to 16      ' Create a loop of 16
    Cls                   ' Clear the LCD
    Cursor Ypos, Xpos     ' Move the cursor to position Ypos,Xpos
    Print "*"             ' Display the character
    DelayMs 100
  Next
  Ypos = 2                ' Move to line 2
  For Xpos = 16 to 1 Step -1 ' Create another loop, this time reverse
    Cls                   ' Clear the LCD
    Cursor Ypos, Xpos     ' Move the cursor to position Ypos,Xpos
    Print "*"             ' Display the character
    DelayMs 100
  Next
  GoTo Again              ' Repeat forever
```

Example 2 displays an asterisk character moving around the perimeter of a 2-line by 16 character LCD.

See also : **Cls**, **Print**

Cwrite

Syntax
Cwrite Address, [Variable {, Variable…}]

Overview
Write data to code (flash) memory.

Operators
- ***Variable*** can be 32 constants, variables, or expressions.
- ***Address*** is a constant, variable, label, or expression that represents any valid location within the microcontroller's code (flash) memory.

Example
```
' Write to a code memory location within the microcontroller

   Dim Var1 as Byte
   Dim WordVar as Word
   Var1 = 234
   WordVar = 1043
'
' Erase a 64 byte block of code (flash) memory
'
   Cerase WriteHere
'
' Write to code memory, as a block of 32 bytes
'
   Cwrite WriteHere, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10,_
                      11, 12, 13, 14, 15, 16, 17, 18, 19, 20,_
                      21, 22, 23, 24, 25, 26, 27, 28, 29, Var1, WordVar]
   Stop
'
' This is where the data will be written too
' It must be placed on a 64 byte boundary
'
   Org $0200           ' Place the label WriteHere at address $0200 (512)
                       ' Make sure this will not overwrite your existing code
WriteHere:
```

Notes
Code memory can only be written in blocks of 32 bytes. Any less will not be written to memory.

Code memory must be erased using the **Cerase** command before writing to it. Code memory can only be erased in blocks of 64 and on a 64 byte boundary.

See section 6.0 of the PIC18F25K20 data sheet for more details concerning code (flash) memory reading, writing and erasing.

See also : **Cerase, Cdata**, **Cread, Lread, Lread8, Lread16, Lread32**.

Dec

Syntax
Dec Variable

Overview
Decrement a variable i.e. Var1 = Var1 - 1

Operators
- ***Variable*** is a user defined variable

Example
```
Dim Var1 as Byte

Var1 = 11
Repeat
   Dec Var1
   Hrsout Dec Var1, 13
   DelayMs 200
Until Var1 = 0
```

The above example shows the equivalent to the **For**-**Next** loop:

```
For Var1 = 10 to 0 Step -1 : Next
```

See also : **Inc**.

Declare

Syntax
[**Declare**] code modifying directive = modifying value

Overview
Adjust certain aspects of the produced code, i.e. Crystal frequency, LCD port and pins, serial baud rate etc.

Operators
- ***code modifying*** directive is a set of pre-defined words. See list below.

modifying value is the value that corresponds to the command. See list below.

The **Declare** directive is an indispensable part of the compiler. It moulds the library subroutines, and passes essential user information to them.

Misc Declares.

Declare WatchDog = On or Off, or True or False, or 1, 0
The WatchDog **Declare** directive enables or disables the watchdog timer. It also sets the microcontroller's Config fuses for no watchdog. In addition, it removes any ClrWdt mnemonics from the assembled code, thus producing slightly smaller programs. The default for the compiler is WatchDog Off, therefore, if the watchdog timer is required, then this **Declare** will need to be invoked.

The WatchDog **Declare** can be issued multiple times within the BASIC code, enabling and disabling the watchdog timer as and when required.

Declare FSR_Context_Save = On or Off, or True or False, or 1, 0
When using Hardware interrupts, it is not always necessary to save the FSR0 register. So in order to save code space and time spent within the interrupt handler, the FSR_Context_Save **Declare** can enable or disable the auto Context saving and restoring of the FSR register.

If String variables are used in the BASIC program, the FSR1L/H register pair will also be saved/restored. And FSR2L/H registers will be saved/restored if a stack is implemented.

Declare PLL_Req = On or Off, or True or False, or 1, 0
The Amicus18 microcontroller has a built in PLL (Phase Locked Loop) that can multiply the oscillator by a factor of 4. This is set by the fuses at programming time, and the PLL_Req **Declare** enables or disables the PLL fuse. Using the PLL fuse allows a 1:1 ratio of instructions to clock cycles instead of the normal 4:1 ratio. It can be used with Xtal settings from 4 to 20MHz. Note that the compiler will automatically set it's frequency to a multiple of 4 if the PLL_Req **Declare** is used to enable the PLL fuse. For example, if a 16MHz Xtal setting is declared, and the PLL_Req **Declare** is used in the BASIC program, the compiler will automatically set itself up as using a 64MHz oscillator. i.e. 4 * 16. Thus keeping the timings for library functions correct.

Declare Warnings = On or Off, or True or False, or 1, 0
The Warnings **Declare** directive enables or disables the compiler's warning messages. This can have disastrous results if a warning is missed or ignored, so use this directive sparingly, and at your own peril.

The Warnings **Declare** can be issued multiple times within the BASIC code, enabling and disabling the warning messages at key points in the code as and when required.

Declare Reminders = On or Off, or True or False, or 1, 0
The Reminders **Declare** directive enables or disables the compiler's reminder messages. The compiler issues a reminder for a reason, so use this directive sparingly, and at your own peril.

The Reminders **Declare** can be issued multiple times within the BASIC code, enabling and disabling the warning messages at key points in the code as and when required.

Declare Label_Bank_Resets = On or Off, or True or False, or 1, 0
The compiler has very intuitive RAM bank handling, however, if you think that an anomaly is occurring due to misplaced or mishandled RAM bank settings, you can issue this **Declare** and it will Reset the RAM bank on every BASIC label, which will force the compiler to re-calculate its bank settings. If nothing else, it will reassure you that bank handling is not the cause of the problem, and you can get on with finding the cause of the programming problem.

Using this **Declare** will increase the size of the code produced, as it will place BCF mnemonics MOVLB mnemonics within the ASM code produced.

The Label_Bank_Resets **Declare** can be issued multiple times within the BASIC code, enabling and disabling the bank resets at key points in the code as and when required. See **Line** LABELS for more information.

Declare Float_Display_Type = Fast or Standard
By default, the compiler uses a relatively small routine for converting floating point values to decimal, ready for **Rsout**, **Print**, **Str$** etc. However, because of its size, it is only capable of converting relatively small values. i.e. approx 6 digits of accuracy. In order to produce a more accurate result, the compiler needs to use a larger routine. This is implemented by using the above **Declare**.

Using the Fast model for the above **Declare** will trigger the compiler into using the more accurate floating point to decimal routine. Note that even though the routine is larger than the standard converter, it actually operates much faster.

The compiler defaults to Standard if the **Declare** is not issued in the BASIC program.

Declare Create_Coff = On or Off, or True or False or 1, 0
When the Create_Coff **Declare** is set to On, the compiler produces a cof file (Common Object File). This is used for simulating the BASIC code within the MPLAB^{tm} IDE environment or the ISIS simulator.

Declare ICD_Req = On or Off, or True or False, or 1, 0
When the ICD_Req **Declare** is set to On, the compiler configures itself so that the Microchip ICD2^{tm} In-Circuit-Debugger, or PICkit2^{tm} can be used. The ICD2^{tm} and PICkit2^{tm} are very invasive to the program, in so much that they require certain RAM areas for itself.

Whenever ICD2^{tm} or PICkit2^{tm} compatibility is enabled, the compiler will automatically deduct the reserved RAM from the available RAM within the Amicus18 microcontroller, therefore you must take this into account when declaring variables. Remember, there aren't as many variables available with the ICD enabled.

If the ICD is enabled along with hardware interrupts, the compiler will also reserve the RAM required for context saving and restoring. This also will be reflected in the amount of RAM available within the Amicus18 microcontroller.

Declare Amicus18_Start_Address = Start Address
If using a bootloader that resides in low code memory, as opposed to the Amicus18's bootloader which resides in high code memory, the microcontroller's interrupt vectors require redirecting, as well as the compiler's library subroutines require moving up. This can be accomplished using the **Amicus18_Start_Address Declare**.

Start Address is the location where the bootloader ends, or it's length. For example:

```
Declare Amicus18_Start_Address = $0800
```


Trigonometry Declares.

The compiler defaults to using floating point trigonometry functions **Sin** and **Cos**, as well as **Sqr** . However, if only the BASIC Stamp compatible integer functions are required, they can be enabled by the following three declares. Note that by enabling the integer type function, the floating point function will be disabled permanently within the BASIC code. As with most of the declares, only one of any type is recognised per program.

Declare Stamp_Compatible_Cos = On or Off, or True or False, or 1, 0
Enable/Disable floating point **Cos** function in favour of the BASIC Stamp compatible integer **Cos** function.

Declare Stamp_Compatible_Sin = On or Off, or True or False, or 1, 0
Enable/Disable floating point **Sin** function in favour of the BASIC Stamp compatible integer **Sin** function.

Declare Stamp_Compatible_Sqr = On or Off, or True or False, or 1, 0
Enable/Disable floating point **Sqr** (square root) function in favour of the BASIC Stamp compatible integer **Sqr** function.

Adin Declares.

Declare Adin_Res 8, 10.
Sets the number of bits in the result.

If this **Declare** is not used, then the default is 10 bits. Using the above **Declare** allows an 8-bit result to be obtained from the 10-bit Amicus18 microcontroller.

Declare Adin_Tad 2_FOSC, 8_FOSC, 32_FOSC, 64_FOSC , or FRC.
Sets the ADC's clock source.

The Amicus18's microcontroller has five options for the clock source used by the ADC; 2_FOSC, 8_FOSC, 32_FOSC, and 64_FOSC, are ratios of the external oscillator, while FRC is the internal RC oscillator.

Care must be used when issuing this **Declare**, as the wrong type of clock source may result in poor resolution, or no conversion at all. If in doubt use FRC which will produce a slight reduction in resolution and conversion speed, but is guaranteed to work first time, every time. FRC is the default setting if the **Declare** is not issued in the BASIC listing.

Declare Adin_Stime 0 to 65535 microseconds (us).
Allows the internal capacitors to fully charge before a sample is taken. This may be a value from 0 to 65535 microseconds (us).

A value too small may result in a reduction of resolution. While too large a value will result in poor conversion speeds without any extra resolution being attained.

A typical value for Adin_Stime is 50 to 100. This allows adequate charge time without loosing too much conversion speed.

But experimentation will produce the right value for your particular requirement. The default value if the **Declare** is not used in the BASIC listing is 50.

Busin - Busout Declares.

Declare SDA_Pin Port . Pin
Declares the port and pin used for the data line (SDA). This may be any valid port on the Amicus18 microcontroller. If this declare is not issued in the BASIC program, then the default Port and Pin is RA0 (PortA.0)

Declare SCL_Pin Port . Pin
Declares the port and pin used for the clock line (SCL). This may be any valid port on the Amicus18 microcontroller. If this declare is not issued in the BASIC program, then the default Port and Pin is RA1 (PortA.1)

Declare Slow_Bus On - Off or 1 - 0
Slows the bus speed when using an oscillator higher than 4MHz.

The standard speed for the I^2C bus is 100KHz. Some devices use a higher bus speed of 400KHz. If you use an 8MHz or higher oscillator, the bus speed may exceed the devices specs, which will result in intermittent writes or reads, or in some cases, none at all. Therefore, use this **Declare** if you are not sure of the device's spec. The datasheet for the device used will inform you of its bus speed.

Declare Bus_SCL On - Off, 1 - 0 or True - False
Eliminates the necessity for a pull-up resistor on the SCL line.

The I^2C protocol dictates that a pull-up resistor is required on both the SCL and SDA lines, however, this is not always possible due to circuit restrictions etc, so once the Bus_SCL On **Declare** is issued at the top of the program, the resistor on the SCL line can be omitted from the circuit. The default for the compiler if the Bus_SCL **Declare** is not issued, is that a pull-up resistor is required.

Hbusin - Hbusout Declare.

Declare Hbus_Bitrate Constant 100, 400, 1000 etc.
The standard speed for the I^2C bus is 100KHz. Some devices use a higher bus speed of 400KHz. The above **Declare** allows the I^2C bus speed to be increased or decreased. Use this **Declare** with caution, as too high a bit rate may exceed the device's specs, which will result in intermittent transactions, or in some cases, no transactions at all. The datasheet for the device used will inform you of its bus speed. The default bit rate is the standard 100KHz.

Hserin, Hserout, HRsin and HRsout Declares.

Declare Hserial_Baud Constant value
Sets the BAUD rate that will be used to receive a value serially. The baud rate is calculated using the Xtal frequency declared in the program. The default baud rate if the **Declare** is not included in the program listing is 2400 baud.

Declare Hserial_RCSTA Constant value (0 to 255)
Hserial_RCSTA, sets the respective hardware register RCSTA, to the value in the **Declare**. See the PIC18F25K20 data sheet for the device used for more information regarding this register.

Declare Hserial_TXSTA Constant value (0 to 255)
Hserial_TXSTA, sets the respective hardware register, TXSTA, to the value in the **Declare**. See the PIC18F25K20 data sheet for the device used for more information regarding this register. The TXSTA register BRGH bit (bit 2) controls the high speed mode for the baud rate generator. Certain baud rates at certain oscillator speeds require this bit to be set to operate properly. To do this, set Hserial_TXSTA to a value of $24 instead of the default $20. Refer to the PIC18F25K20 data sheet for the hardware serial port baud rate tables and additional information.

Declare Hserial_Parity Odd or Even
Enables/Disables parity on the serial port. For **HRsin**, **HRsout**, **Hserin** and **Hserout**. The default serial data format is 8N1, 8 data bits, no parity bit and 1 stop bit. 7E1 (7 data bits, even parity, 1 stop bit) or 7O1 (7data bits, odd parity, 1 stop bit) may be enabled using the Hserial_Parity declare.

```
Declare Hserial_Parity = Even     ' Use if even parity desired
Declare Hserial_Parity = Odd      ' Use if odd parity desired
```

Declare Hserial_Clear On or Off
Clear the overflow error bit before commencing a read.

Because the hardware serial port only has a 2-byte input buffer, it can easily overflow is characters are not read from it often enough. When this occurs, the USART stops accepting any new characters, and requires resetting. This overflow error can be Reset by strobing the CREN bit within the RCSTA register. Example:

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

or

```
Clear RCSTA.4
Set RCSTA.4
```

Alternatively, the Hserial_Clear declare can be used to automatically clear this error, even if no error occurred. However, the program will not know if an error occurred while reading, therefore some characters may be lost.

```
Declare Hserial_Clear = On
```


Hpwm Declares.

The Amicus18 microcontroller has an alternate pin for CCP2, as used by **Hpwm**. The following Declare allows the use of different pin:

```
Declare CCP2_Pin PortB.3 ' Select Hpwm port and bit for CCP2 module (ch 2)
```

Alphanumeric (Hitachi) LCD Print Declares.

Declare LCD_DTPin Port . Pin
Assigns the Port and Pins that the LCD's DT lines will attach to.

The LCD may be connected to the Amicus18 using either a 4-bit bus or an 8-bit bus. If an 8-bit bus is used, all 8 bits must be on one port. If a 4-bit bus is used, it must be connected to either the bottom 4 or top 4 bits of one port. For example:

```
Declare LCD_DTPin PortB.4    ' Used for 4-line interface.
Declare LCD_DTPin PortB.0    ' Used for 8-line interface.
```

In the above examples, PortB is only a personal preference. The LCD's DT lines can be attached to any valid port on the Amicus18 hardware. If the **Declare** is not used in the program, then the default Port and Pin is RB4 (PortB.4), which assumes a 4-line interface.

Declare LCD_ENPin Port . Pin
Assigns the Port and Pin that the LCD's EN line will attach to. This also assigns the graphic LCD's EN pin, however, the default value remains the same as for the alphanumeric type, so this will require changing.

If the **Declare** is not used in the program, then the default Port and Pin is RB3 (PortB.3).

Declare LCD_RSPin Port . Pin
Assigns the Port and Pins that the LCD's RS line will attach to. This also assigns the graphic LCD's RS pin, however, the default value remains the same as for the alphanumeric type, so this will require changing.

If the **Declare** is not used in the program, then the default Port and Pin is RB2 (PortB.2).

Declare LCD_Interface 4 or 8
Inform the compiler as to whether a 4-line or 8-line interface is required by the LCD.

If the **Declare** is not used in the program, then the default interface is a 4-line type.

Declare LCD_Lines 1, 2, or 4
Inform the compiler as to how many lines the LCD has.

LCD's come in a range of sizes, the most popular being the 2 line by 16 character types. However, there are 4-line types as well. Simply place the number of lines that the particular LCD has into the declare.

If the **Declare** is not used in the program, then the default number of lines is 2.

Declare LCD_CommandUS 1 to 65535
Time to wait (in microseconds) between commands sent to the LCD.

If the **Declare** is not used in the program, then the default delay is 2000us (2ms).

Declare LCD_DataUs 1 to 255
Time to wait (in microseconds) between data sent to the LCD.

If the **Declare** is not used in the program, then the default delay is 50us.

Graphic LCD Declares.

Declare LCD_Type 0 or 1 or 2, Alpha or Graphic or Samsung or Toshiba
Inform the compiler as to the type of LCD that the **Print** command will output to. If Graphic, Samsung or 1 is chosen then any output by the **Print** command will be directed to a graphic LCD based on the Samsung KS0108 chipset. A value of 2, or the text Toshiba, will direct the output to a graphic LCD based on the Toshiba T6963 chipset. A value of 0 or Alpha, or if the **Declare** is not issued, will target the standard Hitachi alphanumeric LCD type

Targeting the graphic LCD will also enable commands such as **Plot**, **UnPlot**, **LCDread**, **LCDwrite**, **Pixel**, **Box**, **Circle** and **Line**.

Samsung KS0108 Graphic LCD specific Declares.

Declare LCD_DTPort Port
Assign the port that will output the 8-bit data to the graphic LCD.

If the **Declare** is not used, then the default port is PortB.

Declare LCD_RWPin Port . Pin
Assigns the Port and Pin that the graphic LCD's RW line will attach to.

If the **Declare** is not used in the program, then the default Port and Pin is RC0 (PortC.0).

Declare LCD_CS1Pin Port . Pin
Assigns the Port and Pin that the graphic LCD's CS1 line will attach to.

If the **Declare** is not used in the program, then the default Port and Pin is RC1 (PortC.1).

Declare LCD_CS2Pin Port . Pin
Assigns the Port and Pin that the graphic LCD's CS2 line will attach to.

If the **Declare** is not used in the program, then the default Port and Pin is RC2 (PortC.2).

Declare Internal_Font On - Off, 1 or 0
The graphic LCD's that are compatible with Proton Amicus18 are non-intelligent types, therefore, a separate character set is required. This may be in one of two places, either externally, in an I^2C eeprom, or internally in a **Cdata** table.

If an external font is chosen, the I^2C eeprom must be connected to the specified SDA and SCL pins (as dictated by **Declare** SDA_Pin and **Declare** SCL_Pin).

The **Cdata** table that contains the font must have a label, named Font: preceding it. For example:

```
Font:   Cdata $7E, $11, $11, $11, $7E, $0   ' Chr "A"
        Cdata $7F, $49, $49, $49, $36, $0   ' Chr "B"
        { rest of font table }
```


The font table may be anywhere in memory, however, it is best placed after the main program code.

If the **Declare** is omitted from the program, then an external font is the default setting.

Declare Font_Addr 0 to 7
Set the slave address for the I^2C eeprom that contains the font.

When an external source for the font is chosen, it may be on any one of 8 eeproms attached to the I^2C bus. So as not to interfere with any other eeproms attached, the slave address of the eeprom carrying the font code may be chosen.

If the **Declare** is omitted from the program, then address 0 is the default slave address of the font eeprom.

Declare GLCD_CS_Invert On - Off, 1 or 0
Some graphic LCD types have inverters on their CS lines. Which means that the LCD displays left hand data on the right side, and vice-versa. The GLCD_CS_Invert **Declare**, adjusts the library LCD handling library subroutines to take this into account.

Declare GLCD_Strobe_Delay 0 to 65535 us (microseconds).
Create a delay of n microseconds between strobing the EN line of the graphic LCD. This can help noisy, or badly decoupled circuits overcome random bits appearing on the LCD. The default if the **Declare** is not used in the BASIC program is a delay of 0.

Toshiba T6963 Graphic LCD specific Declares.

Declare LCD_DTPort Port
Assign the port that will output the 8-bit data to the graphic LCD.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_WRPin Port . Pin
Assigns the Port and Pin that the graphic LCD's WR line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_RDPin Port . Pin
Assigns the Port and Pin that the graphic LCD's RD line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_CEPin Port . Pin
Assigns the Port and Pin that the graphic LCD's CE line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_CDPin Port . Pin
Assigns the Port and Pin that the graphic LCD's CD line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_RSTPin Port . Pin
Assigns the Port and Pin that the graphic LCD's RST line will attach to.

The LCD's RST (Reset) **Declare** is optional and if omitted from the BASIC code the compiler will not manipulate it. However, if not used as part of the interface, you must set the LCD's RST pin high for normal operation.

Declare LCD_X_Res 0 to 255
LCD displays using the T6963 chipset come in varied screen sizes (resolutions). The compiler must know how many horizontal pixels the display consists of before it can build its library subroutines.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_Y_Res 0 to 255
LCD displays using the T6963 chipset come in varied screen sizes (resolutions). The compiler must know how many vertical pixels the display consists of before it can build its library subroutines.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_Font_Width 6 or 8
The Toshiba T6963 graphic LCDs have two internal font sizes, 6 pixels wide by eight high, or 8 pixels wide by 8 high. The particular font size is chosen by the LCD's FS pin. Leaving the FS pin floating or bringing it high will choose the 6 pixel font, while pulling the FS pin low will choose the 8 pixel font. The compiler must know what size font is required so that it can calculate screen and RAM boundaries.

Note that the compiler does not control the FS pin and it is down to the circuit layout whether or not it is pulled high or low. There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_RAM_Size 1024 to 65535
Toshiba graphic LCDs contain internal RAM used for Text, Graphic or Character Generation. The amount of RAM is usually dictated by the display's resolution. The larger the display, the more RAM is normally present. Standard displays with a resolution of 128x64 typically contain 4096 bytes of RAM, while larger types such as 240x64 or 190x128 typically contain 8192 bytes or RAM. The display's datasheet will inform you of the amount of RAM present.

If this **Declare** is not issued within the BASIC program, the default setting is 8192 bytes.

Declare LCD_Text_Pages 1 to n
As mentioned above, Toshiba graphic LCDs contain RAM that is set aside for text, graphics or characters generation. In normal use, only one page of text is all that is required, however, the compiler can rearrange its library subroutines to allow several pages of text that is continuous. The amount of pages obtainable is directly proportional to the RAM available within the LCD itself. Larger displays require more RAM per page, therefore always limit the amount of pages to only the amount actually required or unexpected results may be observed as text, graphic and character generator RAM areas merge.

This **Declare** is purely optional and is usually not required. The default is 3 text pages if this **Declare** is not issued within the BASIC program.

Declare LCD_Graphic_Pages 1 to n
Just as with text, the Toshiba graphic LCDs contain RAM that is set aside for graphics. In normal use, only one page of graphics is all that is required, however, the compiler can re-arrange its library subroutines to allow several pages of graphics that is continuous. The amount of pages obtainable is directly proportional to the RAM available within the LCD itself. Larger displays require more RAM per page, therefore always limit the amount of pages to only the amount actually required or unexpected results may be observed as text, graphic and character generator RAM areas merge.

This **Declare** is purely optional and is usually not required. The default is 1 graphics page if this **Declare** is not issued within the BASIC program.

Declare LCD_Text_Home_Address 0 to n
The RAM within a Toshiba graphic LCD is split into three distinct uses, text, graphics and character generation. Each area of RAM must not overlap or corruption will appear on the display as one uses the other's assigned space. The compiler's library subroutines calculate each area of RAM based upon where the text RAM starts. Normally the text RAM starts at address 0, however, there may be occasions when it needs to be set a little higher in RAM. The order of RAM is; Text, Graphic, then Character Generation.

This **Declare** is purely optional and is usually not required. The default is the text RAM staring at address 0 if this **Declare** is not issued within the BASIC program.

Keypad Declare.

Declare Keypad_Port Port
Assigns the Port that the keypad is attached to.

The keypad routine requires pull-up resistors, therefore, the best Port for this device is PortB which comes equipped with internal pull-ups. If the **Declare** is not used in the program, then PortB is the default Port.

Rsin - Rsout Declares.

Declare Rsout_Pin Port . Pin
Assigns the Port and Pin that will be used to output serial data from the **Rsout** command. This may be any valid port on the microcontroller.

If the **Declare** is not used in the program, then the default Port and Pin is RC6 (PortC.6).

Declare Rsin_Pin Port . Pin
Assigns the Port and Pin that will be used to input serial data by the **Rsin** command. This may be any valid port on the microcontroller.

If the **Declare** is not used in the program, then the default Port and Pin is RC7 (PortC.7).

Declare Rsout_Mode Inverted, True or 1, 0
Sets the serial mode for the data transmitted by **Rsout**. This may be inverted or true. Alternatively, a value of 1 may be substituted to represent inverted, and 0 for true.

If the **Declare** is not used in the program, then the default mode is true.

Declare Rsin_Mode Inverted, True or 1, 0
Sets the serial mode for the data received by **Rsin**. This may be inverted or true. Alternatively, a value of 1 may be substituted to represent inverted, and 0 for true.

If the **Declare** is not used in the program, then the default mode is true.

Declare Serial_Baud 0 to 65535 bps (baud)
Informs the **Rsin** and **Rsout** routines as to what baud rate to receive and transmit data.

Virtually any baud rate may be transmitted and received (within reason), but there are standard bauds, namely:

300, 600, 1200, 2400, 4800, 9600, and 19200.

When using a 4MHz crystal, the highest baud rate that is reliably achievable is 9600. However, an increase in the oscillator speed allows higher baud rates to be achieved, including 38400 baud.

If the **Declare** is not used in the program, then the default baud is 9600.

Declare Rsout_Pace 0 to 65535 microseconds (us)
Implements a delay between characters transmitted by the **Rsout** command.

On occasion, the characters transmitted serially are in a stream that is too fast for the receiver to catch, this results in missed characters. To alleviate this, a delay may be implemented between each individual character transmitted by **Rsout**.

If the **Declare** is not used in the program, then the default is no delay between characters.

Declare Rsin_Timeout 0 to 65535 milliseconds (ms)
Sets the time, in ms, that **Rsin** will wait for a start bit to occur.

Rsin waits in a tight loop for the presence of a start bit. If no timeout parameter is issued, then it will wait forever.

The **Rsin** command has the option of jumping out of the loop if no start bit is detected within the time allocated by timeout.

If the **Declare** is not used in the program, then the default timeout value is 10000ms which is 10 seconds.

Serin - Serout Declare.

If communications are with existing software or hardware, its speed and mode will determine the choice of baud rate and mode. In general, 7-bit/even-parity (7E) mode is used for text, and 8-bit/no-parity (8N) for byte-oriented data. Note: the most common mode is 8-bit/no-parity, even when the data transmitted is just text. Most devices that use a 7-bit data mode do so in order to take advantage of the parity feature. Parity can detect some communication errors, but to use it you lose one data bit. This means that incoming data bytes transferred in 7E (even-parity) mode can only represent values from 0 to 127, rather than the 0 to 255 of 8N (no-parity) mode.

The compiler's serial commands **Serin** and **Serout** have the option of still using a parity bit with 4 to 8 data bits. This is through the use of a **Declare**:

With parity disabled (the default setting):

```
Declare Serial_Data 4 ' Set Serin and Serout data bits to 4
Declare Serial_Data 5 ' Set Serin and Serout data bits to 5
Declare Serial_Data 6 ' Set Serin and Serout data bits to 6
Declare Serial_Data 7 ' Set Serin and Serout data bits to 7
Declare Serial_Data 8 ' Set Serin and Serout data bits to 8 (default)
```

With parity enabled:

```
Declare Serial_Data 5 ' Set Serin and Serout data bits to 4
Declare Serial_Data 6 ' Set Serin and Serout data bits to 5

Declare Serial_Data 7 ' Set Serin and Serout data bits to 6
Declare Serial_Data 8 ' Set Serin and Serout data bits to 7 (default)
Declare Serial_Data 9 ' Set Serin and Serout data bits to 8
```

Serial_Data data bits may range from 4 bits to 8 (the default if no **Declare** is issued). Enabling parity uses one of the number of bits specified.

Declaring Serial_Data as 9 allows 8 bits to be read and written along with a 9th parity bit.

Parity is a simple error-checking feature. When a serial sender is set for even parity (the mode the compiler supports) it counts the number of 1s in an outgoing byte and uses the parity bit to make that number even. For example, if it is sending the 7-bit value: %0011010, it sets the parity bit to 1 in order to make an even number of 1s (four).

The receiver also counts the data bits to calculate what the parity bit should be. If it matches the parity bit received, the serial receiver assumes that the data was received correctly. Of course, this is not necessarily true, since two incorrectly received bits could make parity seem correct when the data was wrong, or the parity bit itself could be bad when the rest of the data was correct.

Many systems that work exclusively with text use 7-bit/ even-parity mode. For example, to receive one data byte through bit-0 of PortA at 9600 baud, 7E, inverted:

Shin - Shout Declare.

Declare Shift_DelayUs 0 - 65535 microseconds (us)
Extend the active state of the shift clock.

The clock used by **Shin** and **Shout** runs at approximately 45KHz dependent on the oscillator. The active state is held for a minimum of 2 microseconds. By placing this declare in the program, the active state of the clock is extended by an additional number of microseconds up to 65535 (65.535 milliseconds) to slow down the clock rate.

If the **Declare** is not used in the program, then the default is no clock delay.

Crystal Frequency Declare.

Declare Xtal 3, 4, 7, 8, 10, 12, 14, 16, 19, 20, 22, 24, 25, 29, 32, 33, 40, 48, 64, 80, 96

Inform the compiler as to what frequency crystal is being used. Oscillator values 80 and 96 are over-clocking modes produced by crystal frequencies of 20MHz and 24MHz.

Some commands are very dependant on the oscillator frequency, **Rsin**, **Rsout**, **DelayMs**, and **DelayUs** being just a few. In order for the compiler to adjust the correct timing for these commands, it must know what frequency crystal is being used.

The Xtal frequencies 3, 7, 14, 19 and 22 are for 3.58MHz, 7.2MHz, 14.32MHz, 19.66MHz, 22.1184MHz and 29.2MHz respectively.

If the **Declare** is not used in the program, then the default frequency is 64MHz.

DelayCs

Syntax
DelayCs Length

Overview
Delay execution for an amount of instruction cycles.

Operators
- ***Length*** can only be a constant with a value from 1 to 1000.

Example
```
DelayCs 100             ' Delay for 100 cycles
```

Notes
DelayCs is oscillator independent, as long as you inform the compiler of the crystal frequency to use, using the **Declare** directive.

The length of a given instruction cycle is determined by the oscillator frequency. For example, running the microcontroller at it's defualt speed of 64MHz will result in an instruction cycle of 62.5ns (nano seconds).

See also : **DelayUs**, **DelayMs**, **Sleep**, **Snooze**.

DelayMs

Syntax
DelayMs Length

Overview
Delay execution for length x milliseconds (ms). Delays may be up to 65535ms (65.535 seconds) long.

Operators
- ***Length*** can be a constant, variable, or expression.

Example
```
Dim ByteVar as Byte
Dim WordVar as Word
ByteVar = 50
WordVa1= 1000
DelayMs 100             ' Delay for 100ms
DelayMs ByteVar         ' Delay for 50ms
DelayMs WordVa1         ' Delay for 1000ms
DelayMs WordVar + 10    ' Delay for 1010ms
```

Notes
DelayMs is oscillator independent, as long as you inform the compiler of the crystal frequency to use, using the **Declare** directive.

See also : **DelayUs**, **DelayCs**, **Sleep**, **Snooze**.

DelayUs

Syntax
DelayUs Length

Overview
Delay execution for length x microseconds (us). Delays may be up to 65535us (65.535 milliseconds) long.

Operators
- Length can be a constant, variable, or expression.

Example
```
Dim ByteVar as Byte
Dim WordVar as Word
ByteVar = 50
WordVar1= 1000
DelayUs 1                ' Delay for 1us
DelayUs 100              ' Delay for 100us
DelayUs ByteVar          ' Delay for 50us
DelayUs WordVar          ' Delay for 1000us
DelayUs WordVar + 10     ' Delay for 1010us
```

Notes
DelayUs is oscillator independent, as long as you inform the compiler of the crystal frequency to use, using the **Xtal** declare.

If a constant is used as length, then delays down to 1us can be achieved, however, if a variable is used as length, then there's a minimum delay time depending on the frequency of the crystal used:

Crystal Freq	Minimum Delay
4MHz	24us
8MHz	12us
10MHz	8us
16MHz	5us
20MHz	2us
24MHz	2us
25MHz	2us
32MHz	2us
33MHz	2us
40MHz	2us
48MHz	2us
64MHz onwards	2us

See also : **Declare**, **DelayMs**, **DelayCs**, **Sleep**, **Snooze**

Dig

Syntax
Variable = **Dig** Value, Digit number

Overview
Returns the value of a decimal digit.

Operators
- ***Value*** is a constant, 8-bit, 16-bit, 32-bit variable or expression, from which the digit number is to be extracted.
- ***Digit*** number is a constant, variable, or expression, that represents the digit to extract from value. (0 - 4 with 0 being the rightmost digit).

Example
```
Dim Var1 as Byte
Dim Var2 as Byte
Var1 = 124
Var2 = Dig Var1, 1      ' Extract the second digit's value
Hrsout Dec Var2, 13     ' Display the value, which is 2
```


Dim

Syntax
Dim Variable as Size

Overview
All user-defined variables must be declared using the **Dim** statement.

Operators
- ***Variable*** can be any alphanumeric character or string.
- ***Size*** is the physical size of the variable, it may be Bit, Byte, Word, Dword, Float, or String.

Example
```
' Create different sized variables
  Dim ByteVar as Byte             ' Create an 8-bit Byte sized variable
  Dim WordVar as Word             ' Create a 16-bit Word sized variable
  Dim DWordVar as Dword           ' Create a signed 32-bit Dword sized variable
  Dim BitVar as Bit               ' Create a 1-bit Bit sized variable
  Dim FloatVar as Float           ' Create a 32-bit floating point variable
  Dim StringVar as String * 20    ' Create a 20 character string variable
```

Notes
Any variable that is declared without the 'as' text after it, will assume an 8-bit Byte type.

Dim should be placed near the beginning of the program. Any references to variables not declared or before they are declared may, in some cases, produce errors.

Variable names, as in the case or labels, may freely mix numeric content and underscores.

```
  Dim MyVar as Byte
```
or
```
  Dim MY_Var as Word
```
or
```
  Dim My_Var2 as Bit
```

Variable names may start with an underscore, but must not start with a number. They can be no more than 32 characters long. Any characters after this limit will be ignored.

```
  Dim 2MyVar  is not allowed.
```

Variable names are case insensitive, which means that the variable:

```
  Dim myvar as Byte
```

Is the same as…

```
  Dim MYVar as Byte
```

Dim can also be used to create Alias's to other variables:

```
  Dim Var1 as Byte         ' Create a Byte sized variable
  Dim Var_Bit as Var1.1    ' Var_Bit now represents Bit-1 of Var1
```

Alias's, as in the case of constants, do not require any RAM space, because they point to a variable, or part of a variable that has already been declared.

RAM space required.
Each type of variable requires differing amounts of RAM memory for its allocation. The list below illustrates this.

- String Requires the specified length of characters + 1.
- Float Requires 4 bytes of RAM.
- Dword Requires 4 bytes of RAM.
- Word Requires 2 bytes of RAM.
- Byte Requires 1 byte of RAM.
- Bit Requires 1 byte of RAM for every 8 Bit variables used.

Each type of variable may hold a different minimum and maximum value.

String type variables can hold a maximum of 255 characters.

Float type variables may theoretically hold a value from -1e37 to +1e38, but because of the 32-bit architecture of the compiler, a maximum and minimum value should be thought of as -2147483646.999 to +2147483646.999 making this the most accurate of the variable family types. However, more so than Dword types, this comes at a price as Float calculations and comparisons will use more code space within the microcontroller. Use this type of variable sparingly, and only when strictly necessary. Smaller floating point values offer more accuracy.

Dword type variables may hold a value from -2147483648 to +2147483647 making this one of the largest of the variable family types. This comes at a price however, as Dword calculations and comparisons will use more code space within the microcontroller. Use this type of variable sparingly, and only when necessary.

Word type variables may hold a value from 0 to 65535, which is usually large enough for most applications. It still uses more memory, but not nearly as much as a Dword type.

Byte type variables may hold a value for 0 to 255, and are the usual work horses of most programs. Code produced for Byte sized variables is very low compared to Word, or Dword types, and should be chosen if the program requires faster, or more efficient operation.

Bit type variables may hold a 0 or a 1. These are created 8 at a time, therefore declaring a single Bit type variable in a program will not save RAM space, but it will save code space, as Bit type variables produce the most efficient use of code for comparisons etc.

There are modifiers that may also be used with variables. These are **HighByte**, **LowByte**, **Byte0**, **Byte1**, **Byte2**, and **Byte3**.

Byte2, and **Byte3** may only be used in conjunction with a 32-bit Dword type variable.

HighByte and **Byte1** are one and the same thing, when used with a Word type variable, they refer to the High byte of a Word type variable:

```
  Dim WordVar as Word          ' Create a Word sized variable
  Dim WordVar_Hi as WordVar.HighByte
' WordVar_Hi now represents the High Byte of variable WordVar
```

Variable WordVar_Hi is now accessed as a Byte sized type, but any reference to it actually alters the high byte of WordVar.

However, if **Byte1** is used in conjunction with a Dword type variable, it will extract the second byte. **HighByte** will still extract the high byte of the variable, as will **Byte3**.

The same is true of **LowByte** and **Byte0**, but they refer to the Low Byte of a Word type variable:

```
  Dim WordVar as Word                ' Create a Word sized variable
  Dim WordVar_Lo as WordVar.LowByte
' WordVar_Lo now represents the Low Byte of variable WordVar
```

Variable WordVar_Lo is now accessed as a Byte sized type, but any reference to it actually alters the low byte of WordVar.

The modifier **Byte2** will extract the 3rd byte from a 32-bit Dword type variable, as an alias. Likewise **Byte3** will extract the high byte of a 32-bit variable.

RAM space for variables is allocated within the Amicus18 microcontroller in the order that they are placed in the BASIC code. For example:

```
  Dim Var1 as Byte
  Dim Var2 as Byte
```

Places Var1 first, then Var2:

```
  Var1 equ n
  Var2 equ n
```

The position of the variable within is usually of little importance if BASIC code is used, however, if assembler routines are being implemented, always assign any variables used within them first.

Problems may also arise if a Word, or Dword variable crosses a Bank boundary. If this happens, a warning message will be displayed in the error window. Most of the time, this will not cause any problems, however, to err on the side of caution, try and ensure that Word, or Dword type variables are fully inside a Bank. This is easily accomplished by placing a dummy Byte variable before the offending Word, or Dword type variable, or relocating the offending variable within the list of **Dim** statements.

See Also : **Aliases, Declaring Arrays, Constants, Floating Point Math, Symbol, Creating and using Strings .**

DTMFout

Syntax
DTMFout Pin, { OnTime }, { OffTime, } [Tone {, Tone…}]

Overview
Produce a DTMF Touch Tone sequence on Pin.

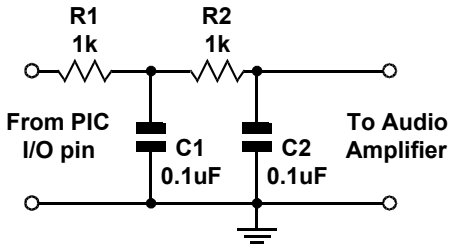
Operators
- ***Pin*** is a Port.Bit constant that specifies the I/O pin to use. This pin will be set to output during generation of tones and set to input after the command is finished.
- ***OnTime*** is an optional variable, constant, or expression (0 - 65535) specifying the duration, in ms, of the tone. If the OnTime parameter is not used, then the default time is 200ms
- ***OffTime*** is an optional variable, constant, or expression (0 - 65535) specifying the length of silent delay, in ms, after a tone (or between tones, if multiple tones are specified). If the OffTime parameter is not used, then the default time is 50ms
- ***Tone*** may be a variable, constant, or expression (0 - 15) specifying the DTMF tone to generate. Tones 0 through 11 correspond to the standard layout of the telephone keypad, while 12 through 15 are the fourth-column tones used by phone test equipment and in some radio applications.

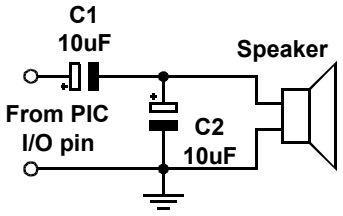
Example
```
DTMFout PortA.0, [ 7, 4, 9, 9, 9, 0 ]    ' Call Crownhill.
```

If the microcontroller was connected to the phone line correctly, the above command would dial 749-990. If you wanted to slow down the dialling in order to break through a noisy phone line or radio link, you could use the optional OnTime and OffTime values:

```
' Set the OnTime to 500ms and OffTime to 100ms
DTMFout PortA.0, 500, 100, [7, 4, 9, 9, 9, 0] ' Call Crownhill Slowly.
```

Notes
DTMF tones are used to dial a telephone, or remotely control pieces of radio equipment. The Amicus18 microcontroller can generate these tones digitally using the **DTMFout** command. The circuits illustrate how to connect a speaker or audio amplifier to hear the tones produced.





The microcontroller is a digital device, however, DTMF tones are analogue waveforms, consisting of a mixture of two sine waves at different audio frequencies. So how can a digital device generate an analogue output? The microcontroller creates and mixes two sine waves mathematically, then uses the resulting stream of numbers to control the duty cycle of an extremely fast pulse-width modulation (Pwm) routine. Therefore, what's actually being produced from the I/O pin is a rapid stream of pulses. The purpose of the filtering arrangements illustrated above is to smooth out the high-frequency Pwm, leaving behind only the lower frequency audio. You should keep this in mind if you wish to interface the microcontroller's DTMF output to radios and other equipment that could be adversely affected by the presence of high-frequency noise on the input. Make sure to filter the DTMF output scrupulously. The circuits above are only a foundation; you may want to use an active low-pass filter with a cut-off frequency of approximately 2KHz.

Edata

Syntax
Edata Constant1 { ,...Constantn etc }

Overview
Places constants or strings directly into the on-board eeprom memory of the Amicus18 microcontroller

Operators
- ***Constant1,Constantn*** are values that will be stored in the on-board eeprom. When using an **Edata** statement, all the values specified will be placed in the eeprom starting at location 0. The **Edata** statement does not allow you to specify an eeprom address other than the beginning location at 0. To specify a location to write or read data from the eeprom other than 0 refer to the **Eread**, **Ewrite** commands.

Example
```
' Stores the values 1000,20,255,15, and the ASCII values for
' H','e','l','l','o' in the eeprom starting at memory position 0.

  Edata 1000, 20, $FF, %00001111, "Hello"
```

Notes
16-bit, 32-bit and floating point values may also be placed into eeprom memory. These are placed LSB first (lowest significant byte). For example, if 1000 is placed into an **Edata** statement, then the order is:

```
  Edata 1000
```

In eeprom it looks like 232, 03

Alias's to constants may also be used in an **Edata** statement:

```
  Symbol Alias = 200

  Edata Alias, 120, 254, "Hello World"
```

Addressing an Edata table.
Eeprom data starts at address 0 and works up towards the maximum amount that the microcontroller will allow. However, it is rarely the case that the information stored in eeprom memory is one continuous piece of data. Eeprom memory is normally used for storage of several values or strings of text, so a method of accessing each piece of data is essential. Consider the following piece of code:

```
  Edata "HELLO"
  Edata "WORLD"
```

Now we know that eeprom memory starts at 0, so the text "HELLO" must be located at address 0, and we also know that the text "HELLO" is built from 5 characters with each character occupying a byte of eeprom memory, so the text "WORLD" must start at address 5 and also contains 5 characters, so the next available piece of eeprom memory is located at address 10. To access the two separate text strings we would need to keep a record of the start and end address's of each character placed in the tables.

Counting the amount of eeprom memory used by each piece of data is acceptable if only a few **Edata** tables are used in the program, but it can become tedious if multiple values and strings are needing to be stored, and can lead to program glitches if the count is wrong.

Placing an identifying name before the **Edata** table will allow the compiler to do the byte counting for you. The compiler will store the eeprom address associated with the table in the identifying name as a constant value. For example:

```
Hello_Text  Edata "HELLO"
World_Text    Edata "WORLD"
```

The name Hello_Text is now recognised as a constant with the value of 0, referring to address 0 that the text string "HELLO" starts at. The World_Text is a constant holding the value 5, which refers to the address that the text string "WORLD" starts at.

Note that the identifying text *must* be located on the same line as the **Edata** directive or a syntax error will be produced. It must also not contain a postfix colon as does a line label or it will be treat as a line label. Think of it as an alias name to a constant.

Any **Edata** directives *must* be placed at the head of the BASIC program as is done with Symbols, so that the name is recognised by the rest of the program as it is parsed. There is no need to jump over **Edata** directives as you have to with **Cdata** or **Cdata**, because they do not occupy code memory, but reside in high Data memory.

The example program below illustrates the use of eeprom addressing.

```
' Display two text strings held in eeprom memory

Dim Char as Byte                ' Holds the character read from eeprom
Dim Charpos as Byte             ' Holds the address within eeprom memory

' Create a string of text in eeprom memory. null terminated
HELLO  Edata "HELLO ",0
' Create another string of text in eeprom memory. null terminated
WORLD  Edata "WORLD",0

   Charpos = HELLO              ' Point Charpos to the start of text "HELLO"
   GoSub DisplayText            ' Display the text "HELLO"
   Charpos = WORLD              ' Point Charpos to the start of text "WORLD"
   GoSub DisplayText            ' Display the text "WORLD"
   Stop                         ' We're all done

' Subroutine to read and display the text held at the address in Charpos
DisplayText:
   While 1 = 1                  ' Create an infinite loop
      Char = Eread Charpos      ' Read the eeprom data
      If Char = 0 Then Break    ' Exit when null found
      HRsout Char               ' Display the character
      Inc Charpos               ' Move up to the next address
   Wend                         ' Close the loop
   Hrsout 13                    ' New Line
   Return                       ' Exit the subroutine
```


Formatting an Edata table.
Sometimes it is necessary to create a data table with a known format for its values. For example all values will occupy 4 bytes of data space even though the value itself would only occupy 1 or 2 bytes.

```
Edata 100000, 10000, 1000, 100, 10, 1
```

The above line of code would produce an uneven data space usage, as each value requires a different amount of data space to hold the values. 100000 would require 4 bytes of eeprom space, 10000 and 1000 would require 2 bytes, but 100, 10, and 1 would only require 1 byte.

Reading these values using **Eread** would cause problems because there is no way of knowing the amount of bytes to read in order to increment to the next valid value.

The answer is to use formatters to ensure that a value occupies a predetermined amount of bytes.

These are:

Byte
Word
Dword
Float

Placing one of these formatters before the value in question will force a given length.

```
Edata  Dword 100000, Dword 10000 , Dword 1000, Dword 100, Dword 10, Dword 1
```

Byte will force the value to occupy one byte of eeprom space, regardless of it's value. Any values above 255 will be truncated to the least significant byte.

Word will force the value to occupy 2 bytes of eeprom space, regardless of its value. Any values above 65535 will be truncated to the two least significant bytes. Any value below 255 will be padded to bring the memory count to 2 bytes.

Dword will force the value to occupy 4 bytes of eeprom space, regardless of its value. Any value below 65535 will be padded to bring the memory count to 4 bytes. The line of code shown above uses the Dword formatter to ensure all the values in the **Edata** table occupy 4 bytes of eeprom space.

Float will force a value to its floating point equivalent, which always takes up 4 bytes of eeprom space.

If all the values in an **Edata** table are required to occupy the same amount of bytes, then a single formatter will ensure that this happens.

```
Edata as Dword 100000, 10000, 1000, 100, 10, 1
```

The above line has the same effect as the formatter previous example using separate Dword formatters, in that all values will occupy 4 bytes, regardless of their value. All four formatters can be used with the as keyword.

The example below illustrates the formatters in use.

```
' Convert a Dword value into a string
' Using only BASIC commands
' Similar principle to the Str$ command

   Dim P10 as Dword              ' Power of 10 variable
   Dim Cnt as Byte
   Dim J as Byte

   Dim Value as Dword            ' Value to convert
   Dim StringVar as String * 11 ' Holds the converted value
   Dim Ptr as Byte               ' Pointer within the Byte array

   Clear                         ' Clear all RAM before we start
   Value = 1234576               ' Value to convert
   GoSub DwordToStr              ' Convert Value to string
   HRsout StringVar              ' Display the result
   Stop
'
' Convert a Dword value into a string
' Value to convert is placed in 'Value'
' String StringVar is built up with the ASCII equivalent
'
DwordToStr:
   Ptr = 0
   J = 0
   Repeat
      P10 = Eread J * 4
      Cnt = 0
      While Value >= P10
         Value = Value - P10
         Inc Cnt
      Wend
      If Cnt <> 0 Then
         StringVar[Ptr] = Cnt + "0"
         Inc Ptr
      EndIf
      Inc J
   Until J > 8
   StringVar[Ptr] = Value + "0"
   Inc Ptr
   StringVar[Ptr] = 0            ' Add the null to terminate the string
   Return

' Edata table is formatted for all 32 bit values.
' Which means each value will require 4 bytes of eeprom space
Edata as Dword 1000000000, 100000000, 10000000, 1000000,_
                100000, 10000, 1000, 100, 10
```


Label names as pointers in an Edata table.
If a label's name is used in the list of values in an **Edata** table, the labels address will be used. This is useful for accessing other tables of data using their address from a lookup table. See example below.

```
' Display text from two Cdata tables Based on their address
' located in a separate table

  Dim Address as Word
  Dim DataByte as Byte

  Address = Eread 0                 ' Locate the address of the first string
  While 1 = 1                       ' Create an infinite loop
    DataByte = Cread Address        ' Read each character from the Cdata string
    If DataByte = 0 Then Break      ' Exit if null found
    HRsout DataByte                 ' Display the character
    Inc Address                     ' Next character
  Wend                              ' Close the loop
  HRsout 13                         ' Next line of the serial terminal
  Address = Eread 2                 ' Locate the address of the second string
  While 1 = 1                       ' Create an infinite loop
    DataByte = Cread Address        ' Read each character from the Cdata string
    If DataByte = 0 Then Break      ' Exit if null found
    HRsout DataByte                 ' Display the character
    Inc Address                     ' Next character
  Wend                              ' Close the loop
  HRsout 13                         ' Next line of the serial terminal
  Stop

' Table of address's located in eeprom memory
  Edata as Word String1, String2
String1:
  Cdata "HELLO",0
String2:
  Cdata "WORLD",0
```

See also : **Eread**, **Ewrite**.

End

Syntax
End

Overview
End stops the microcontroller process by placing it into a continuous loop. The port pins remain the same and the device is placed in low power mode.

See also : **Stop**, **Sleep**, **Snooze**.

Eread

Syntax
Variable = **Eread** Address

Overview
Read information from the on-board eeprom of the Amicus18 microcontroller.

Operators
- ***Variable*** is a user defined variable.
- ***Address*** is a constant, variable, or expression, that contains the address of interest within eeprom memory.

Example
```
Dim Var1 as Byte
Dim WordVar1 as Word
Dim DWordVar1 as Dword

Edata 10, 354, 123456789      ' Place some data into the eeprom
Var1 = Eread 0                ' Read the 8-bit value from address 0
WordVar1= Eread 1             ' Read the 16-bit value from address 1
DWordVar1 = Eread 3           ' Read the 32-bit value from address 3
```

Notes
If a Float, or Dword type variable is used as the assignment variable, then 4-bytes will be read from the eeprom. Similarly, if a Word type variable is used as the assignment variable, then a 16-bit value (2-bytes)will be read from eeprom, and if a Byte type variable is used, then 8-bits will be read. To read an 8-bit value while using a Word sized variable, use the **LowByte** modifier:

```
WordVar1.LowByte = Eread 0    ' Read an 8-bit value
WordVar1.HighByte = 0         ' Clear the high byte of WordVar
```

If a 16-bit (Word) size value is read from the eeprom, the address must be incremented by two for the next read. Also, if a Float or Dword type variable is read, then the address must be incremented by 4.

The Amicus18 microcontroller has 256 bytes of eeprom data.

Eeprom memory is non-volatile, and is an excellent place for storage of long-term information, or tables of values.

Reading data with the **Eread** command is almost instantaneous, but writing data to the eeprom can take a few milliseconds per byte.

See also : **Edata**, **Ewrite**

Ewrite

Syntax
Ewrite Address, [Variable {, Variable…etc }]

Overview
Write information to the on-board eeprom of the Amicus18 microcontroller.

Operators
- ***Address*** is a constant, variable, or expression, that contains the address of interest within eeprom memory.
- ***Variable*** is a user defined variable.

Example
```
Dim Var1 as Byte
Dim WordVar1 as Word
Dim Address as Byte
Var1 = 200
WordVar1= 2456
Address = 0                         ' Point to address 0 within the eeprom
Ewrite Address, [WordVar, Var1]     ' Write a 16-bit then an 8-bit value
```

Notes
If a Dword type variable is used, then a 32-bit value (4-bytes) will be written to the eeprom. Similarly, if a Word type variable is used, then a 16-bit value (2-bytes) will be written to eeprom, and if a Byte type variable is used, then 8-bits will be written. To write an 8-bit value while using a Word sized variable, use the **LowByte** modifier:

```
Ewrite Address, [ WordVar.LowByte, Var1 ]
```

If a 16-bit (Word) size value is written to the eeprom, the address must be incremented by two before the next write:

```
For Address = 0 to 64 Step 2
  Ewrite Address, [WordVar]
Next
```

The Amicus18 microcontroller has 256 bytes of eeprom data.

Eeprom memory is non-volatile, and is an excellent place for storage of long-term information, or tables of values.

Writing data with the **Ewrite** command can take a few milliseconds per byte, but reading data from the eeprom is almost instantaneous,.

See also : **Edata**, **Eread**

For...Next...Step

Syntax
For Variable = StartCount **to** EndCount [**Step** { StepVal }]
{code body}
Next

Overview
The **For**...**Next** loop is used to execute a statement, or series of statements a predetermined amount of times.

Operators
- ***Variable*** refers to an index variable used for the sake of the loop. This index variable can itself be used in the code body but beware of altering its value within the loop as this can cause many problems.
- ***StartCount*** is the start number of the loop, which will initially be assigned to the variable. This does not have to be an actual number - it could be the contents of another variable.
- ***EndCount*** is the number on which the loop will finish. This does not have to be an actual number, it could be the contents of another variable, or an expression.
- ***StepVal*** is an optional constant or variable by which the variable increases or decreases with each trip through the **For**-**Next** loop. If startcount is larger than endcount, then a minus sign must precede stepval.

Example 1
```
' Display in decimal, all the values of WordVar within an upward loop
  Dim WordVar as Word
  For WordVar = 0 to 2000 Step 2      ' Perform an upward loop
    HRsout Dec WordVar, 13            ' Display the value of WordVar
  Next                                ' Close the loop
```

Example 2
```
' Display in decimal, all the values of WordVar within a downward loop
  Dim WordVar as Word
  For WordVar = 2000 to 0 Step -2     ' Perform a downward loop
    HRsout Dec WordVar, 13            ' Display the value of WordVar
  Next                                ' Close the loop
```

Example 3
```
' Display in decimal, all the values of DWordVar within a downward loop
  Dim DWordVar as Dword
  For DWordVar = 200000 to 0 Step -200  ' Perform a downward loop
    HRsout Dec DwordVar, 13             ' Display the value of DWordVar
  Next                                  ' Close the loop
```

Example 4
```
' Display all WordVar1 using a expressions as parts of the For-Next construct
  Dim WordVar1 as Word
  Dim WordVar2 as Word
  WordVar2 = 1000
  For WordVar1= WordVar2 + 10 to WordVar2 + 1000   ' Perform a loop
    HRsout Dec WordVar1, 13             ' Display the value of WordVar1
  Next                                  ' Close the loop
```

Notes
You may have noticed from the above examples, that no variable is present after the **Next** command. A variable after **Next** is purely optional.

For-**Next** loops may be nested as deeply as the memory on the Amicus18 microcontroller will allow. To break out of a loop you may use the **GoTo** command without any ill effects, but it recommended that the **Break** command is used instead:

```
   For Var1 = 0 to 20                  ' Create a loop of 21
    If Var1 = 10 Then GoTo BreakOut    ' Break out of loop when Var1 is 10
   Next                                ' Close the loop
BreakOut:
   Stop
```

See also : **While**...**Wend**, **Repeat**...**Until**.

FreqOut

Syntax
FreqOut Pin, Period, Freq1 {, Freq2}

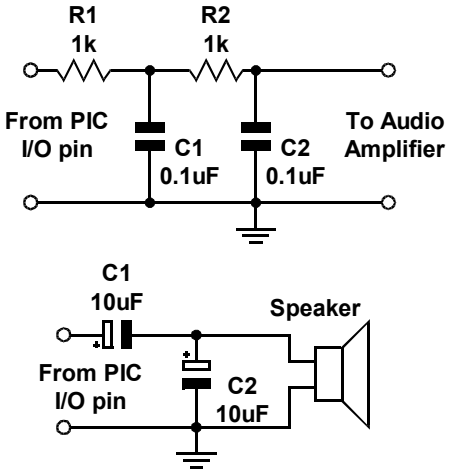
Overview
Generate one or two sine-wave tones, of differing or the same frequencies, for a specified period.

Operators
- ***Pin*** is a Port-Bit combination that specifies which I/O pin to use.
- ***Period*** may be a variable, constant, or expression (0 - 65535) specifying the amount of time to generate the tone(s).
- ***Freq1*** may be a variable, constant, or expression (0 - 32767) specifying frequency of the first tone.
- ***Freq2*** may be a variable, constant, or expression (0 - 32767) specifying frequency of the second tone. When specified, two frequencies will be mixed together on the same I/O pin.

Example
```
' Generate a 2500Hz (2.5KHz) tone for 1 second (1000 ms) on bit 0 of PortA.
  FreqOut PortA.0, 1000, 2500

' Play two tones at once for 1000ms. One at 2.5KHz, the other at 3KHz.
  FreqOut PortA.0, 1000, 2500, 3000
```

Notes
FreqOut generates one or two sine waves using a pulse-width modulation algorithm. **FreqOut** will work with a 4MHz crystal, however, it is best used with higher frequency crystals, and operates best with a frequency of 20MHz or above. The raw output from **FreqOut** requires filtering, to eliminate most of the switching noise. The circuits shown below will filter the signal in order to play the tones through a speaker or audio amplifier.



The two circuits shown above, work by filtering out the high-frequency Pwm used to generate the sine waves. **FreqOut** works over a very wide range of frequencies (0 to 32767KHz) so at the upper end of its range, the Pwm filters will also filter out most of the desired frequency. You may need to reduce the values of the parallel capacitors shown in the circuit, or to create an active filter for your application.

Example 2

```
' Play a tune using FreqOut to generate the notes

   Dim Loop as Byte       ' Counter for notes.
   Dim Freq1 as Word      ' Frequency1.
   Dim Freq2 as Word      ' Frequency2
   Symbol C = 2092        ' C note
   Symbol D = 2348        ' D note
   Symbol E = 2636        ' E note
   Symbol G = 3136        ' G note
   Symbol R = 0           ' Silent pause.
   Symbol Pin = PortA.0   ' Sound output pin
   All_Digital= True      ' Set all ports to digital mode
   Loop = 0
   Repeat                 ' Create a loop for 29 notes within the LookUpL table.
      Freq1 = LookUpL Loop,[E,D,C,D,E,E,E,R,D,D,D,R,_
                            E,G,G,R,E,D,C,D,E,E,E,E,D,D,E,D,C]
      If Freq1 = 0 Then
         Freq2 = 0
      Else
         Freq2 = Freq1 - 8
      EndIf
      FreqOut Pin, 225, Freq1, Freq2
      Inc Loop
   Until Loop > 28
   Stop
```

See also : **DTMFout**, **Sound**, **Sound2**.

GetBit

Syntax
Variable = **GetBit** Variable, Index

Overview
Examine a bit of a variable, or register.

Operators
- ***Variable*** is a user defined variable, of type Byte, Word, or Dword.
- ***Index*** is a constant, variable, or expression that points to the bit within Variable that requires examining.

Example
```
' Examine and display each bit of variable ExVar
  Dim ExVar as Byte
  Dim Index as Byte
  Dim Var1 as Byte

  ExVar = %10110111
Again:
  HRsout Bin8 ExVar, 13          ' Display the original variable
  For Index = 7 to 0 Step -1     ' Create a loop for 8 bits
    Var1 = GetBit ExVar,Index    ' Examine each bit of ExVar
    HRsout Dec1 Var1             ' Display the binary result
    DelayMs 100                  ' Slow things down to see what's happening
  Next                           ' Close the loop
  Hrsout 13
  GoTo Again                     ' Do it forever
```

See also : **ClearBit**, **LoadBit**, **SetBit**.

GoSub

Syntax
GoSub Label

or

GoSub Label [Variable, {Variable, Variable... etc}], Receipt Variable

Overview
GoSub jumps the program to a defined label and continues execution from there. Once the program hits a **Return** command the program returns to the instruction following the **GoSub** that called it and continues execution from that point.

Parameters can be pushed onto a software stack before the call is made, and a variable can be popped from the stack before continuing execution of the next commands.

Operators
- ***Label*** is a user-defined label placed at the beginning of a line which must have a colon ':' directly after it.
- ***Variable*** is a user defined variable of type Bit, Byte, Byte Array, Word, Word Array, Dword, Float, or String, or Constant value, that will be pushed onto the stack before the call to a subroutine is performed.
- ***Receipt*** Variable is a user defined variable of type Bit, Byte, Byte Array, Word, Word Array, Dword, Float, or String, that will hold a value popped from the stack after the subroutine has returned.

Example 1
```
' Implement a standard subroutine call
   GoTo Start        ' Jump over the subroutines
SubA:

   subroutine A code
   ……
   ……

   Return

SubB:

   subroutine B code
   ……
   ……

   Return

' Actual start of the main program
Start:
   GoSub SubA
   GoSub SubB
   Stop
```


Example 2

```
' Call a subroutine with parameters
   Declare Stack_Size = 20       ' Create a stack capable of holding 20 bytes

   Dim WordVar1 as Word          ' Create a Word variable
   Dim WordVar2 as Word          ' Create another Word variable
   Dim Receipt as Word           ' Create a variable to hold result

   WordVar1 = 1234               ' Load the Word variable with a value
   WordVar2 = 567                ' Load the other Word variable with a value
' Call the subroutine and return a value
   GoSub AddThem [WordVar1, WordVar2], Receipt
   HRsout Dec Receipt,13         ' Display the result as decimal
   Stop

' Subroutine starts here. Add two parameters passed and return the result
AddThem:
   Dim AddWordVar1 as Word       ' Create two uniquely named variables
   Dim AddWordVar2 as Word

   Pop AddWordVar2               ' Pop the last variable pushed
   Pop AddWordVar1               ' Pop the first variable pushed
   AddWordVar1 = AddWordVar1 + AddWordVar2  ' Add the values together
   Return AddWordVar1            ' Return the result of the addition
```

In reality, what's happening with the **GoSub** in the above program is simple, if we break it into its constituent events:

```
   Push WordVar1
   Push WordVar2
   GoSub AddThem
   Pop Receipt
```

Notes

Only one parameter can be returned from the subroutine, any others will be ignored.

If a parameter is to be returned from a subroutine but no parameters passed to the subroutine, simply issue a pair of empty square braces:

```
   GoSub LABEL [ ], Receipt
```

The same rules apply for the parameters as they do for **Push**, which is after all, what is happening.

Proton Amicus18 compiler allows any amount of Gosubs in a program, but the Amicus18 microcontroller only has a 28-level return address stack which allows up to 28 consecutive Gosubs to occur.

A subroutine must always end with a **Return** command.

What is a Stack?

All microprocessors and most microcontrollers have access to a Stack, which is an area of RAM allocated for temporary data storage. But this is sadly lacking on a microcontroller device. However, the Amicus18 microcontroller has low-level mnemonics that allow a Stack to be created and used very efficiently.

A stack is first created in high memory by issuing the **Stack_Size Declare**.

```
Declare Stack_Size = 40
```

The above line of code will reserve 40 bytes at the top of RAM that cannot be touched by any BASIC command, other than **Push** and **Pop**. This means that it is a safe place for temporary variable storage.

Taking the above line of code as an example, we can examine what happens when a variable is pushed on to the 40 byte stack, and then popped off again.

First the RAM is allocated. The Amicus18 microcontroller (PIC18F25K20) has 1536 bytes of RAM that stretches linearly from address 0 to 1535. Reserving a stack of 40 bytes will reduce the top of memory so that the compiler will only see 1495 bytes (1535 - 40). This will ensure that it will not inadvertently try and use it for normal variable storage.

Pushing.
When a Word variable is pushed onto the stack, the memory map would look like the diagram below:

```
Top of Memory  |..............Empty RAM.............. | Address 1535
                             ~             ~
                             ~             ~
               |..............Empty RAM.............. | Address 1502
               |..............Empty RAM.............. | Address 1501
               |  Low Byte address of Word variable   | Address 1496
Start of Stack |  High Byte address of Word variable  | Address 1495
```

The high byte of the variable is first pushed on to the stack, then the low byte. And as you can see, the stack grows in an upward direction whenever a **Push** is implemented, which means it shrinks back down whenever a **Pop** is implemented.

If we were to **Push** a Dword variable on to the stack as well as the Word variable, the stack memory would look like:

```
Top of Memory  |...........Empty RAM..................|
Address 1535             ~             ~
                       ~             ~
               |...........Empty RAM..................|
Address 1502   |...........Empty RAM..................| Address 1501
               |  Low Byte address of Dword variable  | Address 1500
               |  Mid1 Byte address of Dword variable | Address 1499
               |  Mid2 Byte address of Dword variable | Address 1498
               |  High Byte address of Dword variable | Address 1497
               |  Low Byte address of Word variable   | Address 1496
Start of Stack |  High Byte address of Word variable  | Address 1495
```


Popping.

When using the **Pop** command, the same variable type that was pushed last must be popped first, or the stack will become out of phase and any variables that are subsequently popped will contain invalid data. For example, using the above analogy, we need to **Pop** a Dword variable first. The Dword variable will be popped Low Byte first, then Mid1 Byte, then Mid2 Byte, then lastly the High Byte. This will ensure that the same value pushed will be reconstructed correctly when placed into its recipient variable. After the **Pop**, the stack memory map will look like:

```
Top of Memory  |.............Empty RAM.............. | Address 1535
                            ~            ~
                            ~            ~
               |.............Empty RAM.............. | Address 1502
               |.............Empty RAM.............. | Address 1501
               |  Low Byte address of Word variable  | Address 1496
Start of Stack |  High Byte address of Word variable | Address 1495
```

If a Word variable was then popped, the stack will be empty, however, what if we popped a Byte variable instead? the stack would contain the remnants of the Word variable previously pushed. Now what if we popped a Dword variable instead of the required Word variable? the stack would underflow by two bytes and corrupt any variables using those address's . The compiler cannot warn you of this occurring, so it is up to you, the programmer, to ensure that proper stack management is carried out. The same is true if the stack overflows. i.e. goes beyond the top of RAM. The compiler cannot give a warning.

Technical Details of Stack implementation.

The stack implemented by the compiler is known as an Incrementing Last-In First-Out Stack. Incrementing because it grows upwards in memory. Last-In First-Out because the last variable pushed, will be the first variable popped.

The stack is not circular in operation, so that a stack overflow will rollover into the microcontroller's hardware register, and an underflow will simply overwrite RAM immediately below the Start of Stack memory. If a circular operating stack is required, it will need to be coded in the main BASIC program, by examination and manipulation of the stack pointer (see below).

Indirect register pair FSR2L and FSR2H are used as a 16-bit stack pointer, and are incremented for every Byte pushed, and decremented for every Byte popped. Therefore checking the FSR2 registers in the BASIC program will give an indication of the stack's condition if required. This also means that the BASIC program cannot use the FSR2 register pair as part of its code, unless for manipulating the stack. Note that none of the compiler's commands, other than **Push** and **Pop**, use FSR2.

Whenever a variable is popped from the stack, the stack's memory is not actually cleared, only the stack pointer is moved. Therefore, the above diagrams are not quite true when they show empty RAM, but unless you have use of the remnants of the variable, it should be considered as empty, and will be overwritten by the next **Push** command.

See also : **Call**, **GoTo**, **Push**, **Pop**.

GoTo

Syntax
GoTo Label

Overview
Jump to a defined label and continue execution from there.

Operators
- ***Label*** is a user-defined label placed at the beginning of a line which must have a colon ':' directly after it.

Example
```
   If Var1 = 3 Then GoTo Jumpover

   code here executed only if Var1<>3
   ……
   ……

JumpOver:
   {continue code execution}
```

In this example, if Var1=3 then the program jumps over all the code below it until it reaches the label JumpOver where program execution continues as normal.

See also : **Call**, **GoSub**.

HbStart

Syntax
HbStart

Overview
Send a Start condition to the I²C bus using the microcontroller's MSSP module.

Notes
Because of the subtleties involved in interfacing to some I²C devices, the compiler's standard **Hbusin**, and **Hbusout** commands were found lacking. Therefore, individual pieces of the I²C protocol may be used in association with the new structure of **Hbusin**, and **Hbusout**. See relevant sections for more information.

Example
```
' Interface to a 24LC256 serial eeprom
  Dim Loop as Byte
  Dim Array[10] as Byte
' Transmit bytes to the I2C bus
  HbStart                       ' Send a Start condition
  Hbusout %10100000             ' Target an eeprom, and send a WRITE command
  Hbusout 0                     ' Send the High Byte of the address
  Hbusout 0                     ' Send the Low Byte of the address
  For Loop = 48 to 57           ' Create a loop containing ASCII 0 to 9
    Hbusout Loop                ' Send the value of Loop to the eeprom
  Next                          ' Close the loop
  HbStop                        ' Send a Stop condition
  DelayMs 10                    ' Wait for data to be entered into eeprom matrix
' Receive bytes from the I2C bus
  HbStart                       ' Send a Start condition
  Hbusout %10100000             ' Target an eeprom, and send a WRITE command
  Hbusout 0                     ' Send the High Byte of the address
  Hbusout 0                     ' Send the Low Byte of the address
  HbRestart                     ' Send a Restart condition
  Hbusout %10100001             ' Target an eeprom, and send a Read command
  For Loop = 0 to 9             ' Create a loop
    Array[Loop] = Hbusin        ' Load an array with bytes received
    If Loop = 9 Then HbStop : Else : HbusAck ' ACK or Stop ?
  Next                          ' Close the loop
  HRsout Str Array, 13          ' Display the Array as a String
```

See also : **HbusAck**, **HbRestart**, **HbStop**, **Hbusin**, **Hbusout**.

HbStop

Syntax
HbStop

Overview
Send a Stop condition to the I^2C bus using the microcontroller's MSSP module.

HbRestart

Syntax
HbRestart

Overview
Send a Restart condition to the I^2C bus using the microcontroller's MSSP module.

HbusAck

Syntax
HbusAck

Overview
Send an Acknowledge condition to the I^2C bus using the microcontroller's MSSP module.

HbusNack

Syntax
HbusNack

Overview
Send a Not Acknowledge condition to the I^2C bus using the microcontroller's MSSP module.

See also : **HbStart**, **HbRestart, HbStop, Hbusin**, **Hbusout**.

Hbusin

Syntax
Variable = **Hbusin** Control, { Address }

or

Variable = **Hbusin**

or

Hbusin Control, { Address }, [Variable {, Variable…}]

or

Hbusin Variable

Overview
Receives a value from the I^2C bus using the MSSP module, and places it into variable/s. If structures *two* or *four* (see above) are used, then *no* Acknowledge, or Stop is sent after the data. Structures *one* and *three* first send the control and optional address out of the clock pin (SCL), and data pin (SDA).

Operators
- ***Variable*** is a user defined variable or constant.
- ***Control*** may be a constant value or a Byte sized variable expression.
- ***Address*** may be a constant value or a variable expression.

The four variations of the **Hbusin** command may be used in the same BASIC program. The *second* and *fourth* types are useful for simply receiving a single byte from the bus, and must be used in conjunction with one of the low level commands. i.e. **HbStart**, **HbRestart**, **HbusAck**, or HbStop. The *first*, and *third* types may be used to receive several values and designate each to a separate variable, or variable type.

The **Hbusin** command operates as an I^2C master, using the microcontroller's MSSP module, and may be used to interface with any device that complies with the 2-wire I^2C protocol.

The most significant 7-bits of control byte contain the control code and the slave address of the device being interfaced with. Bit-0 is the flag that indicates whether a read or write command is being implemented.

For example, if we were interfacing to an external eeprom such as the 24LC256, the control code would be %10100001 or $A1. The most significant 4-bits (1010) are the eeprom's unique slave address. Bits 2 to 3 reflect the three address pins of the eeprom. And bit-0 is set to signify that we wish to read from the eeprom. Note that this bit is automatically set by the **Hbusin** command, regardless of its initial setting.

Example

```
' Receive a byte from the I2C bus and place it into variable Var1.

   Dim Var1 as Byte                ' We'll only read 8-bits
   Dim Address as Word             ' 16-bit address required
   Symbol Control %10100001        ' Target an eeprom
   Address = 20                    ' Read the value at address 20
   Var1 = Hbusin Control, Address ' Read the byte from the eeprom
```

or

```
   Hbusin Control, Address, [Var1]   ' Read the byte from the eeprom
```

Address, is an optional parameter that may be an 8-bit or 16-bit value. If a variable is used in this position, the size of address is dictated by the size of the variable used (Byte or Word). In the case of the previous eeprom interfacing, the 24LC256 eeprom requires a 16-bit address. While the smaller types require an 8-bit address. Make sure you assign the right size address for the device interfaced with, or you may not achieve the results you intended.

The value received from the bus depends on the size of the variables used, except for variation three, which only receives a Byte (8-bits). For example:

```
   Dim WordVar as Word           ' Create a Word size variable
   WordVar = Hbusin Control, Address
```

Will receive a 16-bit value from the bus. While:

```
   Dim Var1 as Byte              ' Create a Byte size variable
   Var1 = Hbusin Control, Address
```

Will receive an 8-bit value from the bus.

Using the *third* variation of the **Hbusin** command allows differing variable assignments. For example:

```
   Dim Var1 as Byte
   Dim WordVar as Word
   Hbusin Control, Address, [Var1, WordVar]
```

Will receive two values from the bus, the first being an 8-bit value dictated by the size of variable Var1 which has been declared as a byte. And a 16-bit value, this time dictated by the size of the variable WordVar which has been declared as a word. Of course, Bit type variables may also be used, but in most cases these are not of any practical use as they still take up a byte within the eeprom.

The *second* and *fourth* variations allow all the subtleties of the I^2C protocol to be exploited, as each operation may be broken down into its constituent parts. It is advisable to refer to the datasheet of the device being interfaced to fully understand its requirements. See section on **HbStart**, **HbRestart**, **HbusAck**, or **HbStop**, for example code.

Hbusin Declare

Declare Hbus_Bitrate *Constant* 100, 400, 1000

The standard speed for the I²C bus is 100KHz. Some devices use a higher bus speed of 400KHz. The above **Declare** allows the I²C bus speed to be increased or decreased. Use this **Declare** with caution, as too high a bit rate may exceed the device's specs, which will result in intermittent transactions, or in some cases, no transactions at all. The datasheet for the device used will inform you of its bus speed. The default bit rate is the standard 100KHz.

Notes

When the **Hbusin** command is used, the appropriate SDA and SCL Port and Pin are automatically setup as inputs. The SDA, and SCL lines are predetermined as hardware pins on the microcontroller, where the SCL pin is PortC.3, and SDA is PortC.4. Therefore, there is no need to pre-declare these.

Because the I²C protocol calls for an open-collector interface, pull-up resistors are required on both the SDA and SCL lines. Values of 1KΩ to 4.7KΩ will suffice.

Str modifier with Hbusin

Using the **Str** modifier allows variations *three* and *four* of the **Hbusin** command to transfer the bytes received from the I²C bus directly into a byte array. If the amount of received characters is not enough to fill the entire array, then a formatter may be placed after the array's name, which will only receive characters until the specified length is reached. An example of each is shown below:

```
  Dim Array[10] as Byte       ' Define an array of 10 bytes
  Dim Address as Byte         ' Create a word sized variable

  Hbusin %10100000, Address, [Str Array]   ' Load data into all the array
' Load data into only the first 5 elements of the array
  Hbusin %10100000, Address, [Str Array\5]
  HbStart                     ' Send a Start condition
  Hbusout %10100000           ' Target an eeprom, and send a WRITE command
  Hbusout 0                   ' Send the HighByte of the address
  Hbusout 0                   ' Send the LowByte of the address
  HbRestart                   ' Send a Restart condition
  Hbusout %10100001           ' Target an eeprom, and send a Read command
  Hbusin Str Array            ' Load all the array with bytes received
  HbStop                      ' Send a Stop condition
```

An alternative ending to the above example is:

```
  Hbusin Str Array\5          ' Load data into the first 5 elements of the array
  HbStop                      ' Send a Stop condition
```

See also : **HbusAck, HbusNack, HbRestart, HbStop, HbStart, Hbusout**.

Hbusout

Syntax
Hbusout Control, { Address }, [Variable {, Variable…}]

or

Hbusout Variable

Overview
Transmit a value to the I²C bus using the microcontroller's MSSP module, by first sending the control and optional address out of the clock pin (SCL), and data pin (SDA). Or alternatively, if only one operator is included after the **Hbusout** command, a single value will be transmitted, along with an ACK reception.

Operators
- ***Variable*** is a user defined variable or constant.
- ***Control*** may be a constant value or a Byte sized variable expression.
- ***Address*** may be a constant, variable, or expression.

The **Hbusout** command operates as an I²C master and may be used to interface with any device that complies with the 2-wire I²C protocol.

The most significant 7-bits of control byte contain the control code and the slave address of the device being interfaced with. Bit-0 is the flag that indicates whether a read or write command is being implemented.

For example, if we were interfacing to an external eeprom such as the 24LC256, the control code would be %10100000 or $A0. The most significant 4-bits (1010) are the eeprom's unique slave address. Bits 2 to 3 reflect the three address pins of the eeprom. And Bit-0 is clear to signify that we wish to write to the eeprom. Note that this bit is automatically cleared by the **Hbusout** command, regardless of its initial value.

Example
```
' Send a byte to the I2C bus.

  Dim Var1 as Byte              ' We'll only read 8-bits
  Dim Address as Word           ' 16-bit address required
  Symbol Control = %10100000    ' Target an eeprom
  Address = 20                  ' Write to address 20
  Var1 = 200                    ' The value place into address 20
  Hbusout Control, Address, [ Var1 ]  ' Send the byte to the eeprom
  DelayMs 10                    ' Allow time for allocation of byte
```

Address, is an optional parameter that may be an 8-bit or 16-bit value. If a variable is used in this position, the size of address is dictated by the size of the variable used (Byte or Word). In the case of the above eeprom interfacing, the 24LC256 eeprom requires a 16-bit address. While the smaller types require an 8-bit address. Make sure you assign the right size address for the device interfaced with, or you may not achieve the results you intended.

The value sent to the bus depends on the size of the variables used. For example:

```
Dim WordVar as Word          ' Create a Word size variable
Hbusout Control, Address, [WordVar]
```

Will send a 16-bit value to the bus. While:

```
Dim Var1 as Byte             ' Create a Byte size variable
Hbusout Control, Address, [Var1]
```

Will send an 8-bit value to the bus.

Using more than one variable within the brackets allows differing variable sizes to be sent. For example:

```
Dim Var1 as Byte
Dim WordVar as Word
Hbusout Control, Address, [Var1, WordVar]
```

Will send two values to the bus, the first being an 8-bit value dictated by the size of variable Var1 which has been declared as a byte. And a 16-bit value, this time dictated by the size of the variable WordVar which has been declared as a word. Of course, Bit type variables may also be used, but in most cases these are not of any practical use as they still take up a byte within the eeprom.

A string of characters can also be transmitted, by enclosing them in quotes:

```
Hbusout Control, Address, ["Hello World", Var1, WordVar]
```

Using the second variation of the **Hbusout** command, necessitates using the low level commands i.e. **HbStart**, **HbRestart**, **HbusAck**, **HbusNack**, or **HbStop**.

Using the **Hbusout** command with only one value after it, sends a byte of data to the I^2C bus, and returns holding the Acknowledge reception. This acknowledge indicates whether the data has been received by the slave device.

The ACK reception is returned in the microcontroller's CARRY flag, which is STATUS.0, and also System variable PP4.0. A value of zero indicates that the data was received correctly, while a one indicates that the data was not received, or that the slave device has sent a NACK return. You must read and understand the datasheet for the device being interfacing to, before the ACK return can be used successfully. An code snippet is shown below:

```
' Transmit a byte to a 24LC256 serial eeprom
  HbStart                ' Send a Start condition
  Hbusout %10100000      ' Target an eeprom, and send a WRITE command
  Hbusout 0              ' Send the HighByte of the address
  Hbusout 0              ' Send the LowByte of the address
  Hbusout "A"            ' Send the value 65 to the bus
  If STATUSbits_C = 1 Then GoTo Not_Received ' Has ACK been received OK ?
  HbStop                 ' Send a Stop condition
  DelayMs 10             ' Wait for the data to be entered into eeprom matrix
```


Str modifier with Hbusout.
The **Str** modifier is used for transmitting a string of bytes from a byte array variable. A string is a set of bytes sized values that are arranged or accessed in a certain order. The values 1, 2, 3 would be stored in a string with the value 1 first, followed by 2 then followed by the value 3. A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string 1,2,3 would be stored in a byte array containing three bytes (elements).

Below is an example that sends four bytes from an array:

```
Dim MyArray[10] as Byte    ' Create a 10-byte array
MyArray[0] = "A"           ' Load the first 4 bytes of the array
MyArray[1] = "B"           ' With the data to send
MyArray[2] = "C"
MyArray[3] = "D"
Hbusout %10100000, Address, [Str MyArray \4]    ' Send 4-byte string.
```

Note that we use the optional \n argument of **Str**. If we didn't specify this, the program would try to keep sending characters until all 10 bytes of the array were transmitted. Since we do not wish all 10 bytes to be transmitted, we chose to tell it explicitly to only send the first 4 bytes.

The above example may also be written as:

```
Dim MyArray[10] as Byte      ' Create a 10-byte array
Str MyArray = "ABCD"         ' Load the first 4 bytes of the array
HbStart                      ' Send a Start condition
Hbusout %10100000            ' Target an eeprom, and send a WRITE command
Hbusout 0                    ' Send the High Byte of the address
Hbusout 0                    ' Send the Low Byte of the address
Hbusout Str MyArray\4        ' Send 4-byte string.
HbStop                       ' Send a Stop condition
```

The above example, has exactly the same function as the previous one. The only differences are that the string is now constructed using the **Str** as a command instead of a modifier, and the low-level Hbus commands have been used.

Notes
When the **Hbusout** command is used, the appropriate SDA and SCL Port and Pin are automatically setup as inputs. The SDA, and SCL lines are predetermined as hardware pins on the microcontroller, where the SCL pin is RC3 (PortC.3), and SDA is RC4 (PortC.4). Therefore, there is no need to pre-declare these.

See also : **HbusAck, HbusNack, HbRestart, HbStop, Hbusin, HbStart**.

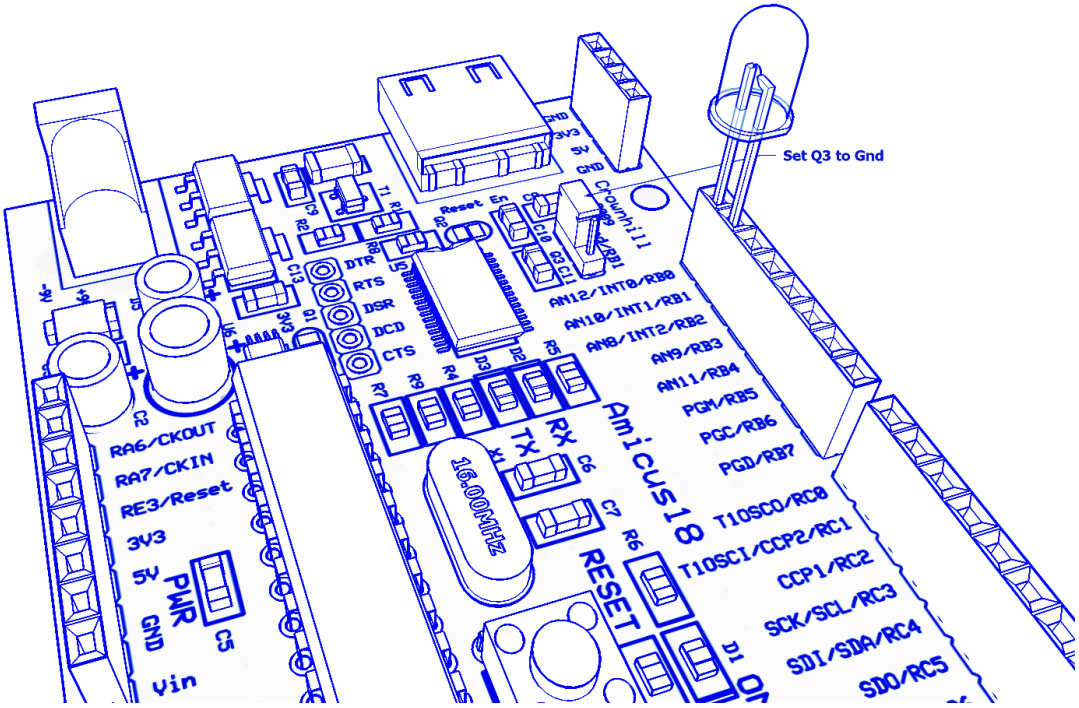
High

Syntax
High Port or Port.Bit

Overview
Place a Port or bit in a high state. For a Port, this means filling it with 1's. For a bit this means setting it to 1.

Operators
- Port can be any valid port.
- Port.Bit can be any valid port and bit combination, i.e. RA1, RB4, PortA.1 etc…

Example
```
Symbol LED = RB0
While 1 = 1          ' Create an infinite loop
  High LED           ' Bring the LED pin high
  DelayMs 500        ' Wait 500ms
  Low LED            ' Pull the LED pin low
  DelayMs 500        ' Wait 500ms
Wend                 ' Close the loop
```



See also : **Clear, Dim, Low, Set, Symbol.**

Hpwm

Syntax
Hpwm Channel, Dutycycle, Frequency

Overview
Output a pulse width modulated pulse train using the CCP module's PWM hardware. The PWM pulses produced can run continuously in the background while the program is executing other instructions.

Operators
- ***Channel*** is a constant value that specifies which hardware Pwm channel to use. The Amicus18 microcotroller has 2 statndard PWM channels, but the Frequency must be the same on both channels. The microcontroller has fixed pins for PWM, Channel 1 is CCP1 which is pin RC2 (PortC.2). Channel 2 is CCP2 which is pin RC1 (PortC.1).
- ***Dutycycle*** is a variable, constant (0-255), or expression that specifies the on/off (high/low) ratio of the signal. It ranges from 0 to 255, where 0 is off (low all the time) and 255 is on (high) all the time. A value of 127 gives a 50% duty cycle (square wave).
- ***Frequency*** is a variable, constant (0-32767), or expression that specifies the desired frequency of the **Pwm** signal. Not all frequencies are available at all oscillator settings. The highest frequency at any oscillator speed is 32767Hz. The lowest usable **Hpwm** Frequency at each oscillator setting is shown in the table below:

Xtal frequency	Lowest useable Pwm frequency
4MHz	244Hz
8MHz	488Hz
10MHz	610Hz
12MHz	732Hz
16MHz	976Hz
20MHz	1221Hz
24MHz	1465Hz
33MHz	2015Hz
40MHz	2442Hz
48MHz	2930Hz
64MHz (default)	3906Hz

Example
```
Hpwm 1,127,1000        ' Send a 50% duty cycle Pwm signal at 1KHz
DelayMs 500
Hpwm 1,64,2000         ' Send a 25% duty cycle Pwm signal at 2KHz
```

See also : **Pwm**, **PulseOut**, **Servo**.

HRsin

Syntax
Variable = **HRsin**, { Timeout, Timeout Label }

or

HRsin { Timeout, Timeout Label }, { Parity Error Label }, Modifiers, Variable {, Variable... }

Overview
Receive one or more values from the microcontroller's USART.

Operators
- ***Timeout*** is an optional value for the length of time the **HRsin** command will wait before jumping to label Timeout Label. Timeout is specified in 1 millisecond units.
- ***Timeout*** Label is an optional valid BASIC label where **HRsin** will jump to in the event that a character has not been received within the time specified by Timeout.
- ***Parity Error Label*** is an optional valid BASIC label where **HRsin** will jump to in the event that a Parity error is received. Parity is set using Declares. Parity Error detecting is not supported in the inline version of **HRsin** (first syntax example above).
- ***Modifier*** is one of the many formatting modifiers, explained below.
- ***Variable*** is a variable, that will be loaded by **HRsin**.

Example
```
' Receive values serially and timeout if no reception after 1 second

   Declare Hserial_Baud = 115200  ' Set baud rate to 115200
   Dim Var1 as Byte

Loop:
   Var1 = HRsin, {1000, Timeout}  ' Receive a byte serially into Var1
   HRsout Dec Var1                ' Display the byte received
   GoTo Loop                      ' Loop forever
Timeout:
   HRsout "Timed Out",13          ' Display an error if HRsin timed out
   Stop
```

HRsin Modifiers.
As we already know, **HRsin** will wait for and receive a single byte of data, and store it in a variable . If the Amicus18 board was connected to a PC running a terminal program and the user pressed the "A" key on the keyboard, after the **HRsin** command executed, the variable would contain 65, which is the ASCII code for the letter "A"

What would happen if the user pressed the "1" key? The result would be that the variable would contain the value 49 (the ASCII code for the character "1"). This is an important point to remember: every time you press a character on the keyboard, the computer receives the ASCII value of that character. It is up to the receiving side to interpret the values as necessary.

In this case, perhaps we actually wanted the variable to end up with the value 1, rather than the ASCII code 49.

The **HRsin** command provides a modifier, called the decimal modifier, which will interpret this for us. Look at the following code:

```
Dim SerData as Byte
HRsin Dec SerData
```

Notice the decimal modifier in the **HRsin** command that appears just to the left of the SerData variable. This tells **HRsin** to convert incoming text representing decimal numbers into true decimal form and store the result in SerData. If the user running the terminal software pressed the "1", "2" and then "3" keys followed by a space or other non-numeric text, the value 123 will be stored in the variable SerData, allowing the rest of the program to perform any numeric operation on the variable.

Without the decimal modifier, however, you would have been forced to receive each character ("1", "2" and "3") separately, and then would still have to do some manual conversion to arrive at the number 123 (one hundred twenty three) before you can do the desired calculations on it.

The decimal modifier is designed to seek out text that represents decimal numbers. The characters that represent decimal numbers are the characters "0" through "9". Once the **HRsin** command is asked to use the decimal modifier for a particular variable, it monitors the incoming serial data, looking for the first decimal character. Once it finds the first decimal character, it will continue looking for more (accumulating the entire multi-digit number) until is finds a non-decimal numeric character. Remember that it will not finish until it finds at least one decimal character followed by at least one non-decimal character.

To illustrate this further, examine the following examples (assuming we're using the same code example as above):

Serial input: "ABC"
Result: The program halts at the **HRsin** command, continuously waiting for decimal text.

Serial input: "123" (with no characters following it)
Result: The program halts at the **HRsin** command. It recognises the characters "1", "2" and "3" as the number one hundred twenty three, but since no characters follow the "3", it waits continuously, since there's no way to tell whether 123 is the entire number or not.

Serial input: "123" (followed by a space character)
Result: Similar to the above example, except once the space character is received, the program knows the entire number is 123, and stores this value in SerData. The **HRsin** command then ends, allowing the next line of code to run.

Serial input: "123A"
Result: Same as the example above. The "A" character, just like the space character, is the first non-decimal text after the number 123, indicating to the program that it has received the entire number.

Serial input: "ABCD123EFGH"
Result: Similar to examples 3 and 4 above. The characters "ABCD" are ignored (since they're not decimal text), the characters "123" are evaluated to be the number 123 and the following character, "E", indicates to the program that it has received the entire number.

The final result of the **Dec** modifier is limited to 16 bits (up to the value 65535). If a value larger than this is received by the decimal modifier, the end result will be incorrect because the result rolled-over the maximum 16-bit value. Therefore, **HRsin** modifiers may not (at this time) be used to load Dword (32-bit) variables.

The decimal modifier is only one of a family of conversion modifiers available with **HRsin** See below for a list of available conversion modifiers. All of the conversion modifiers work similar to the decimal modifier (as described above). The modifiers receive bytes of data, waiting for the first byte that falls within the range of characters they accept (e.g., "0" or "1" for binary, "0" to "9" for decimal, "0" to "9" and "A" to "F" for hex. Once they receive a numeric character, they keep accepting input until a non-numeric character arrives, or in the case of the fixed length modifiers, the maximum specified number of digits arrives.

While very effective at filtering and converting input text, the modifiers aren't completely foolproof. As mentioned before, many conversion modifiers will keep accepting text until the first non-numeric text arrives, even if the resulting value exceeds the size of the variable. After **HRsin**, a Byte variable will contain the lowest 8 bits of the value entered and a Word (16-bits) would contain the lowest 16 bits. You can control this to some degree by using a modifier that specifies the number of digits, such as Dec2, which would accept values only in the range of 0 to 99.

Conversion Modifier	Type of Number Numeric	Characters Accepted
Dec{1..10}	Decimal, optionally limited to 1 - 10 digits	0 through 9
Hex{1..8}	Hexadecimal, optionally limited to 1 - 8 digits	0 through 9, A through F
Bin{1..32}	Binary, optionally limited to 1 - 32 digits	0, 1

A variable preceded by **Bin** will receive the ASCII representation of its binary value.
For example, if **Bin** Var1 is specified and "1000" is received, Var1 will be set to 8.

A variable preceded by **Dec** will receive the ASCII representation of its decimal value.
For example, if **Dec** Var1 is specified and "123" is received, Var1 will be set to 123.

A variable preceded by **Hex** will receive the ASCII representation of its hexadecimal value.
For example, if **Hex** Var1 is specified and "FE" is received, Var1 will be set to 254.

SKIP followed by a count will skip that many characters in the input stream.
For example, SKIP 4 will skip 4 characters.

The **HRsin** command can be configured to wait for a specified sequence of characters before it retrieves any additional input. For example, suppose a device attached to the microcontroller is known to send many different sequences of data, but the only data you wish to observe happens to appear right after the unique characters, "XYZ". A modifier named Wait can be used for this purpose:

```
HRsin Wait("XYZ"), SerData
```


The above code waits for the characters "X", "Y" and "Z" to be received, in that order, then it receives the next data byte and places it into variable SerData.

Str modifier.
The **HRsin** command also has a modifier for handling a string of characters, named **Str**.

The **Str** modifier is used for receiving a string of characters into a byte array variable.

A string is a set of characters that are arranged or accessed in a certain order. The characters "ABC" would be stored in a string with the "A" first, followed by the "B" then followed by the "C". A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string "ABC" would be stored in a byte array containing three bytes (elements).

Below is an example that receives ten bytes and stores them in the 10-byte array, SerString:

```
Dim SerString[10] as Byte       ' Create a 10-byte array.
HRsin Str SerString             ' Fill the array with received data.
HRsout 13, Str SerString        ' Display the string.
```

If the amount of received characters is not enough to fill the entire array, then a formatter may be placed after the array's name, which will only receive characters until the specified length is reached. For example:

```
Dim SerString[10] as Byte       ' Create a 10-byte array.
HRsin Str SerString\5           ' Fill the first 5-bytes of the array
HRsout 13, Str SerString\5      ' Display the 5-character string.
```

The example above illustrates how to fill only the first n bytes of an array, and then how to display only the first n bytes of the array. n refers to the value placed after the backslash.

Because of its complexity, serial communication can be rather difficult to work with at times. Using the guidelines below when developing a project using the **HRsin** and **HRsout** commands may help to eliminate some obvious errors:

Always build your project in steps.
Start with small, manageable pieces of code, (that deal with serial communication) and test them, one individually.
Add more and more small pieces, testing them each time, as you go.
Never write a large portion of code that works with serial communication without testing its smallest workable pieces first.

Pay attention to timing.
Be careful to calculate and overestimate the amount of time, operations should take within the micro-controller for a given oscillator frequency. Misunderstanding the timing constraints is the source of most problems with code that communicate serially. If the serial communication in your project is bi-directional, the above statement is even more critical.

Pay attention to wiring.
Take extra time to study and verify serial communication wiring diagrams. A mistake in wiring can cause strange problems in communication, or no communication at all. Make sure to connect the ground pins (Vss) between the devices that are communicating serially.

Because of additional overheads in the microcontroller, and the fact that the **HRsin** command only offers a 2 level receive buffer for serial communication, received data may sometimes be missed or garbled. If this occurs, try lowering the baud rate, or increasing the crystal frequency. Using simple variables (not arrays) will also increase the chance that the microcontroller will receive the data properly.

Declares
There are five **Declare** directives for use with **HRsin**. These are:

Declare Hserial_Baud Constant value
Sets the BAUD rate that will be used to receive a value serially. The baud rate is calculated using the Xtal frequency declared in the program. The default baud rate if the **Declare** is not included in the program listing is 9600 baud.

Declare Hserial_Parity Odd or Even
Enables/Disables parity on the serial port. For both **HRsin** and **HRsout** The default serial data format is 8N1, 8 data bits, no parity bit and 1 stop bit. 7E1 (7 data bits, even parity, 1 stop bit) or 7O1 (7data bits, odd parity, 1 stop bit) may be enabled using the Hserial_Parity declare.

```
Declare Hserial_Parity = Even     ' Use if even parity desired
Declare Hserial_Parity = Odd      ' Use if odd parity desired
```

Declare Hserial_Clear On or Off
Clear the overflow error bit before commencing a read.

Because the hardware serial port only has a 2-byte input buffer, it can easily overflow is characters are not read from it often enough. When this occurs, the USART stops accepting any new characters, and requires resetting. This overflow error can be Reset by strobing the CREN bit within the RCSTA register. Example:

```
RCSTA.4 = 0
RCSTA.4 = 1
```
or
```
Clear RCSTA.4
Set RCSTA.4
```

Alternatively, the Hserial_Clear declare can be used to automatically clear this error, even if no error occurred. However, the program will not know if an error occurred while reading, therefore some characters may be lost.

```
Declare Hserial_Clear = On
```

See also : **Declare**, **Rsin**, **Rsout**, **Serin**, **Serout**, **HRsout**, **Hserin**, **Hserout**.

HRsout

Syntax
HRsout Item {, Item... }

Overview
Transmit one or more Items from the microcontroller's USART.

Operators
- Item may be a constant, variable, expression, or string.

There are no operators as such, instead there are modifiers. For example, if the text Dec precedes an ***Item***, the ASCII representation for each digit is transmitted.

The modifiers are listed below:

Modifier	Operation
At ypos,xpos	Position the cursor on a serial LCD
Cls	Clear a serial LCD (also creates a 30ms delay)
Bin{1..32}	Send binary digits
Dec{1..10}	Send decimal digits
Hex{1..8}	Send hexadecimal digits
Sbin{1..32}	Send signed binary digits
Sdec{1..10}	Send signed decimal digits
Shex{1..8}	Send signed hexadecimal digits
Ibin{1..32}	Send binary digits with a preceding '%' identifier
Idec{1..10}	Send decimal digits with a preceding '#' identifier
Ihex{1..8}	Send hexadecimal digits with a preceding '$' identifier
ISbin{1..32}	Send signed binary digits with a preceding '%' identifier
ISdec{1..10}	Send signed decimal digits with a preceding '#' identifier
IShex{1..8}	Send signed hexadecimal digits with a preceding '$' identifier
Rep c\n	Send character c repeated n times
Str array\n	Send all or part of an array
Cstr cdata	Send string data defined in a **Cdata** statement.

The numbers after the **Bin**, **Dec**, and **Hex** modifiers are optional. If they are omitted, then the default is all the digits that make up the value will be displayed.

If a floating point variable is to be displayed, then the digits after the **Dec** modifier determine how many remainder digits are send. i.e. numbers after the decimal point.

```
Dim FloatVar as Float
FloatVar = 3.145
HRsout Dec2 FloatVar, 13     ' Send 2 values after the decimal point
```

The above program will send 3.14

If the digit after the **Dec** modifier is omitted, then 3 values will be displayed after the decimal point.


```
Dim FloatVar as Float
FloatVar = 3.1456
HRsout Dec FloatVar, 13    ' Send 3 values after the decimal point
```

The above program will send 3.145

There is no need to use the **Sdec** modifier for signed floating point values, as the compiler's **Dec** modifier will automatically display a minus result:

```
Dim FloatVar as Float
FloatVar = -3.1456
HRsout Dec FloatVar, 13    ' Send 3 values after the decimal point
```

The above program will send -3.145

Hex or **Bin** modifiers cannot be used with floating point values or variables.

Example 1

```
Dim Var1 as Byte
Dim WordVar as Word
Dim DwordVar as Dword

HRsout "Hello World", 13                ' Display the text "Hello World"
HRsout "Var1= ", Dec Var1, 13           ' Display the decimal value of Var1
HRsout "Var1= ", Hex Var1, 13           ' Display the hexadecimal value of Var1
HRsout "Var1= ", Bin Var1, 13           ' Display the binary value of Var1
HRsout "DwordVar= ", Hex6 DwordVar, 13  ' Display 6 hex chars of Dword
```

Example 2

```
' Display a negative value on the serial LCD.
Symbol Negative = -200
HRsout At 1, 1, Sdec Negative
```

Combining code memory reading with a string format, the compiler is capable of reducing the overhead of printing, or transmitting large amounts of text data.

The **Cstr** modifier may be used in commands that deal with text processing i.e. **Serout**, **Hserout**, and **Print** etc.

The **Cstr** modifier is used in conjunction with the **Cdata** command. The **Cdata** command is used for initially creating the string of characters:

```
String1: Cdata "Hello World", 0
```


The previous line of code will create, in flash memory, the values that make up the ASCII text "HELLO WORLD", at address String1. Note the null terminator after the ASCII text.

null terminated means that a zero (null) is placed at the end of the string of ASCII characters to signal that the string has finished.

To display, or transmit this string of characters, the following command structure could be used:

```
HRsout Cstr String1
```

The label that declared the address where the list of **Cdata** values resided, now becomes the string's name. In a large program with lots of text formatting, this type of structure can save quite literally hundreds of bytes of valuable code space.

Try both these small programs, and you'll see that using **Cstr** saves a few bytes of code:

First the standard way of displaying text:

```
HRsout "Hello World",13
HRsout "How are you?",13
HRsout "I am fine!",13
```

Now using the **Cstr** modifier:

```
HRsout Cstr Text1
HRsout Cstr Text2
HRsout Cstr Tetx3

Text1: Cdata "Hello World", 13, 0
Text 2: Cdata "How are you?", 13, 0
Text 3: Cdata "I am fine!", 13, 0
```

Again, note the null terminators after the ASCII text in the **Cdata** commands. Without these, the microcontroller will continue to transmit data in an endless loop.

The term 'virtual string' relates to the fact that a string formed from the **Cdata** command cannot be written too, but only read from.

The **Str** modifier is used for sending a string of bytes from a byte array variable. A string is a set of bytes sized values that are arranged or accessed in a certain order. The values 1, 2, 3 would be stored in a string with the value 1 first, followed by 2 then followed by the value 3. A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string 1,2,3 would be stored in a byte array containing three bytes (elements).

Below is an example that displays four bytes (from a byte array):

```
Dim MyArray[10] as Byte      ' Create a 10-byte array.
MyArray[0] = "H"             ' Load the first 5 bytes of the array
MyArray[1] = "E"             ' With the data to send
MyArray[2] = "L"
MyArray[3] = "L"
MyArray[4] = "O"
HRsout Str MyArray\5, 13     ' Display a 5-byte string.
```


Note that we use the optional \n argument of **Str**. If we didn't specify this, the microcontroller would try to keep sending characters until all 10 bytes of the array were transmitted. Since we do not wish all 10 bytes to be transmitted, we chose to tell it explicitly to only send the first 5 bytes.

The previous example may also be written as:

```
Dim MyArray[10] as Byte       ' Create a 10-byte array.
Str MyArray = "Hello"         ' Load the first 5 bytes of the array
HRsout Str MyArray\5, 13      ' Send 5-byte string.
```

The above example, has exactly the same function as the previous one. The only difference is that the string is now constructed using **Str** as a command instead of a modifier.

Declares
There are four **Declare** directives for use with **HRsout**. These are:

Declare Hserial_Baud Constant value
Sets the BAUD rate that will be used to transmit a value serially. The baud rate is calculated using the Xtal frequency declared in the program. The default baud rate if the **Declare** is not included in the program listing is 9600 baud.

Declare Hserial_Parity Odd or Even
Enables/Disables parity on the serial port. For both **HRsout** and **HRsin** The default serial data format is 8N1, 8 data bits, no parity bit and 1 stop bit. 7E1 (7 data bits, even parity, 1 stop bit) or 7O1 (7data bits, odd parity, 1 stop bit) may be enabled using the Hserial_Parity declare.

```
Declare Hserial_Parity = Even     ' Use if even parity desired
Declare Hserial_Parity = Odd      ' Use if odd parity desired
```

See also : **Declare**, **Rsin**, **Rsout**, **Serin**, **Serout**, **HRsin**, **Hserin**, **Hserout**.

Hserin

Syntax
Hserin Timeout, Timeout Label, Parity Error Label, [Modifiers, Variable {, Variable... }]

Overview
Receive one or more values from the microcontroller's USART. (For Compatibility with the melabs compiler)

Operators
- ***Timeout*** is an optional value for the length of time the **Hserin** command will wait before jumping to label Timeout Label. Timeout is specified in 1 millisecond units.
- ***Timeout Label*** is an optional valid BASIC label where **Hserin** will jump to in the event that a character has not been received within the time specified by Timeout.
- ***Parity Error Label*** is an optional valid BASIC label where **Hserin** will jump to in the event that a Parity error is received. Parity is set using Declares. Parity Error detecting is not supported in the inline version of **Hserin** (first syntax example above).
- ***Modifier*** is one of the many formatting modifiers, explained below.
- ***Variable*** is a variable, that will be loaded by **Hserin**.

Example
```
' Receive values serially and timeout if no reception after 1 second
   Declare Hserial_Baud = 115200  ' Set baud rate to 115200
   Declare Hserial_Clear = On     ' Clear the buffer before receiving
   Dim Var1 as Byte

Loop:
   Hserin 1000, Timeout, [Var1]   ' Receive a byte serially into Var1
   HRsout Dec Var1, 13            ' Display the byte received
   GoTo Loop                      ' Loop forever
Timeout:
   HRsout "\rTimed Out\r"         ' Display an error if Hserin timed out
   Stop
```

Hserin Modifiers.
As we already know, **Hserin** will wait for and receive a single byte of data, and store it in a variable . If the Amicus18 board was connected to a PC running a terminal program and the user pressed the "A" key on the keyboard, after the **Hserin** command executed, the variable would contain 65, which is the ASCII code for the letter "A"

What would happen if the user pressed the "1" key? The result would be that the variable would contain the value 49 (the ASCII code for the character "1"). This is an important point to remember: every time you press a character on the keyboard, the computer receives the ASCII value of that character. It is up to the receiving side to interpret the values as necessary. In this case, perhaps we actually wanted the variable to end up with the value 1, rather than the ASCII code 49.

The **Hserin** command provides a modifier, called the decimal modifier, which will interpret this for us. Look at the following code:

```
   Dim SerData as Byte
   Hserin [Dec SerData]
```

Notice the decimal modifier in the **Hserin** command that appears just to the left of the SerData variable. This tells **Hserin** to convert incoming text representing decimal numbers into true decimal form

and store the result in SerData. If the user running the terminal software pressed the "1", "2" and then "3" keys followed by a space or other non-numeric text, the value 123 will be stored in the variable SerData, allowing the rest of the program to perform any numeric operation on the variable.

Without the decimal modifier, however, you would have been forced to receive each character ("1", "2" and "3") separately, and then would still have to do some manual conversion to arrive at the number 123 (one hundred twenty three) before you can do the desired calculations on it.

The decimal modifier is designed to seek out text that represents decimal numbers. The characters that represent decimal numbers are the characters "0" through "9". Once the **Hserin** command is asked to use the decimal modifier for a particular variable, it monitors the incoming serial data, looking for the first decimal character. Once it finds the first decimal character, it will continue looking for more (accumulating the entire multi-digit number) until is finds a non-decimal numeric character. Remember that it will not finish until it finds at least one decimal character followed by at least one non-decimal character.

To illustrate this further, examine the following examples (assuming we're using the same code example as above):

Serial input: "ABC"
Result: The program halts at the **Hserin** command, continuously waiting for decimal text.

Serial input: "123" (with no characters following it)
Result: The program halts at the **Hserin** command. It recognises the characters "1", "2" and "3" as the number one hundred twenty three, but since no characters follow the "3", it waits continuously, since there's no way to tell whether 123 is the entire number or not.

Serial input: "123" (followed by a space character)
Result: Similar to the above example, except once the space character is received, the program knows the entire number is 123, and stores this value in SerData. The **Hserin** command then ends, allowing the next line of code to run.

Serial input: "123A"
Result: Same as the example above. The "A" character, just like the space character, is the first non-decimal text after the number 123, indicating to the program that it has received the entire number.

Serial input: "ABCD123EFGH"
Result: Similar to examples 3 and 4 above. The characters "ABCD" are ignored (since they're not decimal text), the characters "123" are evaluated to be the number 123 and the following character, "E", indicates to the program that it has received the entire number.

The final result of the **Dec** modifier is limited to 16 bits (up to the value 65535). If a value larger than this is received by the decimal modifier, the end result will be incorrect because the result rolled-over the maximum 16-bit value. Therefore, **Hserin** modifiers may not (at this time) be used to load Dword (32-bit) variables.

The decimal modifier is only one of a family of conversion modifiers available with **Hserin** See below for a list of available conversion modifiers. All of the conversion modifiers work similar to the decimal modifier (as described above). The modifiers receive bytes of data, waiting for the first byte that falls within the range of characters they accept (e.g., "0" or "1" for binary, "0" to "9" for decimal, "0" to "9" and "A" to "F" for hex. Once they receive a numeric character, they keep accepting input until a non-numeric character arrives, or in the case of the fixed length modifiers, the maximum specified number of digits arrives.

While very effective at filtering and converting input text, the modifiers aren't completely foolproof. As mentioned before, many conversion modifiers will keep accepting text until the first non-numeric text arrives, even if the resulting value exceeds the size of the variable. After **Hserin**, a Byte variable will contain the lowest 8 bits of the value entered and a Word (16-bits) would contain the lowest 16 bits. You can control this to some degree by using a modifier that specifies the number of digits, such as Dec2, which would accept values only in the range of 0 to 99.

Conversion Modifier	Type of Number Numeric	Characters Accepted
Dec{1..10}	Decimal, optionally limited to 1 - 10 digits	0 through 9
Hex{1..8}	Hexadecimal, optionally limited to 1 - 8 digits	0 through 9, A through F
Bin{1..32}	Binary, optionally limited to 1 - 32 digits	0, 1

A variable preceded by **Bin** will receive the ASCII representation of its binary value.
For example, if **Bin** Var1 is specified and "1000" is received, Var1 will be set to 8.

A variable preceded by **Dec** will receive the ASCII representation of its decimal value.
For example, if **Dec** Var1 is specified and "123" is received, Var1 will be set to 123.

A variable preceded by **Hex** will receive the ASCII representation of its hexadecimal value.
For example, if **Hex** Var1 is specified and "FE" is received, Var1 will be set to 254.

SKIP followed by a count will skip that many characters in the input stream.
For example, SKIP 4 will skip 4 characters.

The **Hserin** command can be configured to wait for a specified sequence of characters before it retrieves any additional input. For example, suppose a device attached to the Amicus18 board is known to send many different sequences of data, but the only data you wish to observe happens to appear right after the unique characters, "XYZ". A modifier named Wait can be used for this purpose:

```
Hserin [Wait("XYZ"), SerData]
```

The above code waits for the characters "X", "Y" and "Z" to be received, in that order, then it receives the next data byte and places it into variable SerData.

Str modifier.
The **Hserin** command also has a modifier for handling a string of characters, named **Str**.

The **Str** modifier is used for receiving a string of characters into a byte array variable.

A string is a set of characters that are arranged or accessed in a certain order. The characters "ABC" would be stored in a string with the "A" first, followed by the "B" then followed by the "C". A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string "ABC" would be stored in a byte array containing three bytes (elements).

Below is an example that receives ten bytes and stores them in the 10-byte array, SerString:

```
Dim SerString[10] as Byte       ' Create a 10-byte array.
Hserin [Str SerString]          ' Fill the array with received data.
Print Str SerString             ' Display the string.
```


If the amount of received characters is not enough to fill the entire array, then a formatter may be placed after the array's name, which will only receive characters until the specified length is reached. For example:

```
Dim SerString[10] as Byte       ' Create a 10-byte array.
Hserin [Str SerString\5]        ' Fill the first 5-bytes of the array
Print Str SerString\5           ' Display the 5-character string.
```

The example above illustrates how to fill only the first n bytes of an array, and then how to display only the first n bytes of the array. n refers to the value placed after the backslash.

Because of its complexity, serial communication can be rather difficult to work with at times. Using the guidelines below when developing a project using the **Hserin** and **Hserout** commands may help to eliminate some obvious errors:

Always build your project in steps.
Start with small, manageable pieces of code, (that deal with serial communication) and test them, one individually.

Add more and more small pieces, testing them each time, as you go.

Never write a large portion of code that works with serial communication without testing its smallest workable pieces first.

Pay attention to timing.
Be careful to calculate and overestimate the amount of time, operations should take within the microcontroller for a given oscillator frequency. Misunderstanding the timing constraints is the source of most problems with code that communicate serially. If the serial communication in your project is bi-directional, the above statement is even more critical.

Pay attention to wiring.
Take extra time to study and verify serial communication wiring diagrams. A mistake in wiring can cause strange problems in communication, or no communication at all. Make sure to connect the ground pins (Vss) between the devices that are communicating serially.

Verify port setting on the PC and in the **Hserin** / **Hserout** commands.
Unmatched settings on the sender and receiver side will cause garbled data transfers or no data transfers. This is never more critical than when a line transceiver is used(i.e. MAX232). Always remember that a line transceiver inverts the serial polarity.
If the serial data received is unreadable, it is most likely caused by a baud rate setting error, or a polarity error.

If receiving data from another device that is not a microcontroller, try to use baud rates of 9600 and below, or alternatively, use a higher frequency crystal.

Because of additional overheads in the microcontroller, and the fact that the **Hserin** command offers a 2 level hardware receive buffer for serial communication, received data may sometimes be missed or garbled. If this occurs, try lowering the baud rate.

Declares
There are five **Declare** directives for use with **Hserin** . These are:

Declare Hserial_Baud Constant value
Sets the BAUD rate that will be used to receive a value serially. The baud rate is calculated using the Xtal frequency declared in the program. The default baud rate if the **Declare** is not included in the program listing is 9600 baud.

Declare Hserial_Parity Odd or Even
Enables/Disables parity on the serial port. For both **Hserin** and **HRsout** The default serial data format is 8N1, 8 data bits, no parity bit and 1 stop bit. 7E1 (7 data bits, even parity, 1 stop bit) or 7O1 (7data bits, odd parity, 1 stop bit) may be enabled using the Hserial_Parity declare.

```
Declare Hserial_Parity = Even     ' Use if even parity desired
Declare Hserial_Parity = Odd      ' Use if odd parity desired
```

Declare Hserial_Clear On or Off
Clear the overflow error bit before commencing a read.

Because the hardware serial port only has a 2-byte input buffer, it can easily overflow is characters are not read from it often enough. When this occurs, the USART stops accepting any new characters, and requires resetting. This overflow error can be Reset by strobing the CREN bit within the RCSTA register.

Example:

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

or

```
Clear RCSTA.4
Set RCSTA.4
```

Alternatively, the Hserial_Clear declare can be used to automatically clear this error, even if no error occurred. However, the program will not know if an error occurred while reading, therefore some characters may be lost.

```
Declare Hserial_Clear = On
```

See also : **Declare**, **Hserout**, **HRsin**, **HRsout**, **Rsin**, **Rsout**, **Serin**, **Serout**.

Hserout

Syntax
Hserout [Item {, Item... }]

Overview
Transmit one or more Items from the microcontroller's USART

Operators
- Item may be a constant, variable, expression, or string.

There are no operators as such, instead there are modifiers. For example, if the text **Dec** precedes an item, the ASCII representation for each digit is transmitted.

The modifiers are listed below:

Modifier	Operation
At ypos,xpos	Position the cursor on a serial LCD
Cls	Clear a serial LCD (also creates a 30ms delay)
Bin{1..32}	Send binary digits
Dec{1..10}	Send decimal digits
Hex{1..8}	Send hexadecimal digits
Sbin{1..32}	Send signed binary digits
Sdec{1..10}	Send signed decimal digits
Shex{1..8}	Send signed hexadecimal digits
Ibin{1..32}	Send binary digits with a preceding '%' identifier
Idec{1..10}	Send decimal digits with a preceding '#' identifier
Ihex{1..8}	Send hexadecimal digits with a preceding '$' identifier
ISbin{1..32}	Send signed binary digits with a preceding '%' identifier
ISdec{1..10}	Send signed decimal digits with a preceding '#' identifier
IShex{1..8}	Send signed hexadecimal digits with a preceding '$' identifier
Rep c\n	Send character c repeated n times
Str array\n	Send all or part of an array
Cstr cdata	Send string data defined in a **Cdata** statement.

The numbers after the **Bin**, **Dec**, and **Hex** modifiers are optional. If they are omitted, then the default is all the digits that make up the value will be displayed.

If a floating point variable is to be displayed, then the digits after the **Dec** modifier determine how many remainder digits are send. i.e. numbers after the decimal point.

```
Dim FloatVar as Float
FloatVar = 3.145
Hserout [Dec2 FloatVar,13]   ' Send 2 values after the decimal point
```

The above program will transmit the ASCII text 3.14

If the digit after the **Dec** modifier is omitted, then 3 values will be displayed after the decimal point.

```
Dim FloatVar as Float
FloatVar = 3.1456
Hserout [Dec FloatVar,13]    ' Send 3 values after the decimal point
```

The above program will send 3.145

There is no need to use the **Sdec** modifier for signed floating point values, as the compiler's **Dec** modifier will automatically display a minus result:

```
Dim FloatVar as Float
FloatVar = -3.1456
Hserout [Dec FloatVar, 13]   ' Send 3 values after the decimal point
```

The above program will send -3.145

Hex or **Bin** modifiers cannot be used with floating point values or variables.

The Xpos and Ypos values in the **At** modifier both start at 1. For example, to place the text "HELLO WORLD" on line 1, position 1, the code would be:

```
Hserout [At 1, 1, "HELLO WORLD"]
```

Example 1

```
Dim Var1 as Byte
Dim WordVar as Word
Dim DwordVar as Dword

Hserout ["Hello World", 13]       ' Display the text "Hello World"
Hserout ["Var1= ", Dec Var1, 13]  ' Display the decimal value of Var1
Hserout ["Var1= ", Hex Var1, 13]  ' Display the hexadecimal value of Var1
Hserout ["Var1= ", Bin Var1, 13]  ' Display the binary value of Var1
' Display 6 hex characters of a Dword type variable
Hserout ["DwordVar= ", Hex6 DwordVar, 13]
```

Example 2

```
' Display a negative value on the serial terminal.
Symbol Negative = -200
Hserout [At 1, 1, Sdec Negative]
```

Example 3

```
' Display a negative value on a serial terminal with a preceding identifier.
Hserout [At 1, 1, IShex -$1234]
```

Example 3 will produce the text "$-1234" on the serial terminal.

Combining the code memory reading capabilities of the microcontroller with a string format, the compiler is capable of reducing the overhead of printing, or transmitting large amounts of text data. The **Cstr** modifier may be used in commands that deal with text processing i.e. **Serout**, **HRsout**, and **Print** etc.

The **Cstr** modifier is used in conjunction with the **Cdata** command. The **Cdata** command is used for initially creating the string of characters:

```
String1: Cdata "HELLO WORLD", 0
```

The above line of case will create, in flash memory, the values that make up the ASCII text "HELLO WORLD", at address String1. Note the null terminator after the ASCII text.

null terminated means that a zero (null) is placed at the end of the string of ASCII characters to signal that the string has finished.

To display, or transmit this string of characters, the following command structure could be used:

```
  Hserout [Cstr String1]
```

The label that declared the address where the list of **Cdata** values resided, now becomes the string's name. In a large program with lots of text formatting, this type of structure can save quite literally hundreds of bytes of valuable code space.

Try both these small programs, and you'll see that using **Cstr** saves a few bytes of code:

First the standard way of displaying text:

```
  Hserout ["HELLO WORLD",13]
  Hserout ["HOW ARE YOU?",13]
  Hserout ["I AM FINE!",13]
```

Now using the **Cstr** modifier:

```
  Hserout [Cstr TEXT1]
  Hserout [Cstr TEXT2]
  Hserout [Cstr TEXT3]
  Stop

TEXT1: Cdata "HELLO WORLD", 13, 0
TEXT2: Cdata "HOW ARE YOU?", 13, 0
TEXT3: Cdata "I AM FINE!", 13, 0
```

Again, note the null terminators after the ASCII text in the **Cdata** commands. Without these, the microcontroller will continue to transmit data in an endless loop.

The term 'virtual string' relates to the fact that a string formed from the **Cdata** command cannot be written too, but only read from.

The **Str** modifier is used for sending a string of bytes from a byte array variable. A string is a set of bytes sized values that are arranged or accessed in a certain order. The values 1, 2, 3 would be stored in a string with the value 1 first, followed by 2 then followed by the value 3. A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string 1,2,3 would be stored in a byte array containing three bytes (elements).

Below is an example that displays four bytes (from a byte array):

```
Dim MyArray[10] as Byte       ' Create a 10-byte array.
MyArray[0] = "H"              ' Load the first 5 bytes of the array
MyArray[1] = "E"              ' With the data to send
MyArray[2] = "L"
MyArray[3] = "L"
MyArray[4] = "O"
Hserout [Str MyArray\5]       ' Display a 5-byte string.
```

Note that we use the optional \n argument of **Str**. If we didn't specify this, the microcontroller would try to keep sending characters until all 10 bytes of the array were transmitted. Since we do not wish all 10 bytes to be transmitted, we chose to tell it explicitly to only send the first 5 bytes.

The above example may also be written as:

```
Dim MyArray[10] as Byte       ' Create a 10-byte array.
Str MyArray = "HELLO"         ' Load the first 5 bytes of the array
Hserout [Str MyArray\5]       ' Send 5-byte string.
```

The above example, has exactly the same function as the previous one. The only difference is that the string is now constructed using **Str** as a command instead of a modifier.

Declares
There are four **Declare** directives for use with **Hserout**. These are:

Declare Hserial_Baud Constant value
Sets the BAUD rate that will be used to transmit a value serially. The baud rate is calculated using the Xtal frequency declared in the program. The default baud rate if the **Declare** is not included in the program listing is 9600 baud.

Declare Hserial_Parity Odd or Even
Enables/Disables parity on the serial port. For both **Hserout** and **Hserin** The default serial data format is 8N1, 8 data bits, no parity bit and 1 stop bit. 7E1 (7 data bits, even parity, 1 stop bit) or 7O1 (7data bits, odd parity, 1 stop bit) may be enabled using the Hserial_Parity declare.

```
Declare Hserial_Parity = Even     ' Use if even parity desired
Declare Hserial_Parity = Odd      ' Use if odd parity desired
```

See also : **Declare**, **Rsin**, **Rsout**, **Serin**, **Serout**, **Hserin**.

I2Cin

Syntax
I2Cin Dpin, Cpin, Control, { Address }, [Variable {, Variable…}]

Overview
Receives a value from the I²C bus, and places it into variable/s.

Operators
- ***Dpin*** is a Port.Pin constant that specifies the I/O pin that will be connected to the I²C device's data line (SDA). This pin's I/O direction will be changed to input and will remain in that state after the instruction is completed.
- ***Cpin*** is a Port.Pin constant that specifies the I/O pin that will be connected to the I²C device's clock line (SCL). This pin's I/O direction will be changed to output.
- ***Variable*** is a user defined variable of type bit, byte, byte array, word, word array, dword, or float.
- ***Control*** is a constant value or a byte sized variable expression.
- ***Address*** is an optional constant value or a variable expression.

The **I2Cin** command operates as an I²C master, and may be used to interface with any device that complies with the 2-wire I²C protocol. The most significant 7-bits of control byte contain the control code and the slave address of the device being interfaced with. Bit-0 is the flag that indicates whether a read or write command is being implemented.

For example, if we were interfacing to an external eeprom such as the 24LC256, the control code would be %10100001 or $A1. The most significant 4-bits (1010) are the eeprom's unique slave address. Bits 1 to 3 reflect the three address pins of the eeprom. And bit-0 is set to signify that we wish to read from the eeprom. Note that this bit is automatically set by the **I2Cin** command, regardless of its initial setting.

Example
```
' Receive a byte from the I2C bus and place it into variable Var1.
  Dim Var1 as Byte                ' We'll only read 8-bits
  Dim Address as Word             ' 16-bit address required
  Symbol Control %10100001        ' Target an eeprom
  Symbol SDA = PortC.3            ' Alias the SDA (Data) line
  Symbol SCL = PortC.4            ' Alias the SSL (Clock) line
  Address = 20                    ' Read the value at address 20
  I2Cin SDA, SCL, Control, Address, [Var1] ' Read the byte from the eeprom
```

Address is an optional parameter that may be an 8-bit or 16-bit value. If a variable is used in this position, the size of address is dictated by the size of the variable used (byte or word). In the case of the previous eeprom interfacing, the 24LC256 eeprom requires a 16-bit address. While the smaller types require an 8-bit address. Make sure you assign the right size address for the device interfaced with, or you may not achieve the results you intended.

The **I2Cin** command allows differing variable assignments. For example:

```
Dim Var1 as Byte
Dim WordVar as Word
I2Cin SDA, SCL, Control, Address, [Var1, WordVar]
```

The above example will receive two values from the bus, the first being an 8-bit value dictated by the size of variable Var1 which has been declared as a byte. And a 16-bit value, this time dictated by the size of the variable WordVar which has been declared as a word. Of course, bit type variables may also be used, but in most cases these are not of any practical use as they still take up a byte within the eeprom.

Declares
See **I2Cout** for declare explanations.

Notes
When the **I2Cin** command is used, the appropriate SDA and SCL Port and Pin are automatically setup as inputs, and outputs. Because the I^2C protocol calls for an open-collector interface, pull-up resistors are required on both the SDA and SCL lines. Values of 4.7KΩ to 10KΩ will suffice.

Str modifier with I2Cin
Using the **Str** modifier allows the **I2Cin** command to transfer the bytes received from the I^2C bus directly into a byte array. If the amount of received characters is not enough to fill the entire array, then a formatter may be placed after the array's name, which will only receive characters until the specified length is reached. An example of each is shown below:

```
  Dim Array[10] as Byte              ' Define an array of 10 bytes
  Dim Address as Byte                ' Create a word sized variable
' Load data into all the array
  I2Cin SDA, SCL, %10100000, Address, [Str Array]
' Load data into only the first 5 elements of the array
  I2Cin SDA, SCL, %10100000, Address, [Str Array\5]
```

See Also: **BusAck, Bstart, Brestart, Bstop, Busout, HbStart, HbRestart, HbusAck, Hbusin, Hbusout, I2Cout**

I2Cout

Syntax
I2Cout Control, { Address }, [OutputData]

Overview
Transmit a value to the I²C bus, by first sending the control and optional address out of the clock pin (SCL), and data pin (SDA).

Operators
- ***Dpin*** is a Port.Pin constant that specifies the I/O pin that will be connected to the I²C device's data line (SDA). This pin's I/O direction will be changed to input and will remain in that state after the instruction is completed.
- ***Cpin*** is a Port.Pin constant that specifies the I/O pin that will be connected to the I²C device's clock line (SCL). This pin's I/O direction will be changed to output.
- ***Control*** is a constant value or a byte sized variable expression.
- ***Address*** is an optional constant, variable, or expression.
- ***OutputData*** is a list of variables, constants, expressions and modifiers that informs **I2Cout** how to format outgoing data. **I2Cout** can transmit individual or repeating bytes, convert values into decimal, hex or binary text representations, or transmit strings of bytes from variable arrays.

These actions can be combined in any order in the OutputData list.

The **I2Cout** command operates as an I²C master and may be used to interface with any device that complies with the 2-wire I²C protocol. The most significant 7-bits of control byte contain the control code and the slave address of the device being interfaced with. Bit-0 is the flag that indicates whether a read or write command is being implemented.

For example, if we were interfacing to an external eeprom such as the 24LC256, the control code would be %10100000 or $A0. The most significant 4-bits (1010) are the eeprom's unique slave address. Bits 1 to 3 reflect the three address pins of the eeprom. And Bit-0 is clear to signify that we wish to write to the eeprom. Note that this bit is automatically cleared by the **I2Cout** command, regardless of its initial value.

Example
```
' Send a byte to the I2C bus.
  Dim Var1 as Byte                ' We'll only read 8-bits
  Dim Address as Word             ' 16-bit address required
  Symbol Control = %10100000      ' Target an eeprom
  Symbol SDA = PortC.3            ' Alias the SDA (Data) line
  Symbol SCL = PortC.4            ' Alias the SSL (Clock) line
  Address = 20                    ' Write to address 20
  Var1 = 200                      ' The value place into address 20
  I2Cout SDA, SCL, Control, Address, [Var1] ' Send the byte to the eeprom
  DelayMs 5                       ' Allow time for allocation of byte
```

Address is an optional parameter that may be an 8-bit or 16-bit value. If a variable is used in this position, the size of address is dictated by the size of the variable used (byte or word). In the case of the above eeprom interfacing, the 24LC256 eeprom requires a 16-bit address. While the smaller types require an 8-bit address. Make sure you assign the right size address for the device interfaced with, or you may not achieve the results you intended.

The value sent to the bus depends on the size of the variables used. For example:

```
Dim WordVar as Word             ' Create a Word size variable
I2Cout SDA, SCL, Control, Address, [WordVar]
```

Will send a 16-bit value to the bus. While:

```
Dim Var1 as Byte                ' Create a Byte size variable
I2Cout SDA, SCL, Control, Address, [Var1]
```

Will send an 8-bit value to the bus. Using more than one variable within the brackets allows differing variable sizes to be sent. For example:

```
Dim Var1 as Byte
Dim WordVar as Word
I2Cout SDA, SCL, Control, Address, [Var1, WordVar]
```

Will send two values to the bus, the first being an 8-bit value dictated by the size of variable Var1 which has been declared as a byte. And a 16-bit value, this time dictated by the size of the variable WordVar which has been declared as a word. Of course, bit type variables may also be used, but in most cases these are not of any practical use as they still take up a byte within the eeprom.

A string of characters can also be transmitted, by enclosing them in quotes:

```
I2Cout SDA, SCL, Control, Address, ["Hello World", Var1, WordVar]
```

Str modifier with I2Cout

The **Str** modifier is used for transmitting a string of bytes from a byte array variable. A string is a set of bytes sized values that are arranged or accessed in a certain order. The values 1, 2, 3 would be stored in a string with the value 1 first, followed by 2 then followed by the value 3. A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string 1,2,3 would be stored in a byte array containing three bytes (elements). Below is an example that sends four bytes from an array:

```
Dim MyArray[10] as Byte       ' Create a 10-byte array.
MyArray[0] = "A"              ' Load the first 4 bytes of the array
MyArray[1] = "B"              ' With the data to send
MyArray[2] = "C"
MyArray[3] = "D"
' Send a 4-byte string
I2Cout SDA, SCL, %10100000, Address, [Str MyArray\4]
```

Note that we use the optional \n argument of **Str**. If we didn't specify this, the program would try to keep sending characters until all 10 bytes of the array were transmitted. Since we do not wish all 10 bytes to be transmitted, we chose to tell it explicitly to only send the first 4 bytes.

Declares
There are two **Declare** directives for use with **I2Cout**. These are:

Declare I2C_Slow_Bus On - Off or 1 – 0
Slows the bus speed when using an oscillator higher than 4MHz. The standard speed for the I^2C bus is 100KHz. Some devices use a higher bus speed of 400KHz. If you use an 8MHz or higher oscillator, the bus speed may exceed the devices specs, which will result in intermittent transactions, or in some cases, no transactions at all. Therefore, use this **Declare** if you are not sure of the device's spec. The data-sheet for the device used will inform you of its bus speed.

Declare I2C_Bus_SCL On - Off, 1 - 0 or True – False
Eliminates the necessity for a pull-up resistor on the SCL line.

The I^2C protocol dictates that a pull-up resistor is required on both the SCL and SDA lines, however, this is not always possible due to circuit restrictions etc, so once the I2C_Bus_SCL On **Declare** is issued at the top of the program, the resistor on the SCL line can be omitted from the circuit. The default for the compiler if the I2C_Bus_SCL **Declare** is not issued, is that a pull-up resistor is required.

Notes
When the **I2Cout** command is used, the appropriate SDA and SCL Port and Pin are automatically setup as inputs, and outputs. Because the I^2C protocol calls for an open-collector interface, pull-up resistors are required on both the SDA and SCL lines. Values of 4.7KΩ to 10KΩ will suffice.

You may imagine that it's limiting having a fixed set of pins for the I^2C interface, but you must remember that several different devices may be attached to a single bus, each having a unique slave address. Which means there is usually no need to use up more than two pins on the microcontroller in order to interface to many devices.

See Also: **BusAck**, **Bstart**, **Brestart**, **Bstop**, **Busin**, **HbStart**, **HbRestart**, **HbusAck**, **Hbusin**, **Hbusout**, **I2Cin**

If..Then..ElseIf..Else..EndIf

Syntax
If Comparison **Then** Instruction : { Instruction }

Or, you can use the single line form syntax:

If Comparison **Then** Instruction : { Instruction } : **ElseIf** Comparison **Then** Instruction : **Else** Instruction

Or, you can use the block form syntax:

If Comparison **Then**
Instruction(s)...

ElseIf Comparison **Then**
Instruction(s)...

ElseIf Comparison **Then**
Instruction(s)...

Else
Instruction(s)...

EndIf

The curly braces signify optional conditions.

Overview
Evaluates the comparison and, if it fulfils the criteria, executes expression. If comparison is not fulfilled the instruction is ignored, unless an **Else** directive is used, in which case the code after it is implemented until the **EndIf** is found.

When all the instruction are on the same line as the **If**-**Then** statement, all the instructions on the line are carried out if the condition is fulfilled.

Operators
- ***Comparison*** is composed of variables, numbers and comparators.
- ***Instruction*** is the statement to be executed should the comparison fulfil the **If** criteria

Example 1
```
Symbol LED = RB4
Var1 = 3
Low LED
If Var1 > 4 Then High LED : DelayMs 500 : Low LED
```

In the above example, Var1 is not greater than 4 so the If criteria isn't fulfilled. Consequently, the **High** LED statement is never executed leaving the state of port pin PortB.4 low. However, if we change the value of variable Var1 to 5, then the LED will turn on for 500ms then off, because Var1 is now greater than 4, so fulfils the comparison criteria.

A second form of If, evaluates the expression and if it is true then the first block of instructions is executed. If it is false then the second block (after the **Else**) is executed.

The program continues after the **EndIf** instruction.

The **Else** is optional. If it is missed out then if the expression is false the program continues after the **EndIf** line.

Example 2

```
If X & 1 = 0 Then
   A = 0
   B = 1
Else
   A = 1
EndIf
If Z = 1 Then
   A = 0
   B = 0
EndIf
```

Example 3

```
If X = 10 Then
   High LED1
ElseIf X = 20 Then
   High LED2
Else
   High LED3
EndIf
```

A forth form of **If**, allows the **Else** or **ElseIf** to be placed on the same line as the **If**:

```
If X = 10 Then High LED1 : ElseIf X = 20 Then High LED2 : Else : High LED3
```

Notice that there is no **EndIf** instruction. The comparison is automatically terminated by the end of line condition. So in the above example, if X is equal to 10 then LED1 will illuminate, if X equals 20 then LED will illuminate, otherwise, LED3 will illuminate.

The **If** statement allows any type of variable, register or constant to be compared. A common use for this is checking a Port bit:

```
If RA0 = 1 Then High LED : Else : Low LED
```

Any commands on the same line after **Then** will only be executed if the comparison if fulfilled:

```
If Var1 = 1 Then High LED : DelayMs 500 : Low LED
```

Notes

A **GoTo** command is optional after the **Then**:

```
If RB0 = 1 Then LABEL
```

Then operand always required.

The compiler relies heavily on the **Then** part. Therefore, if the **Then** part of a construct is left out of the code listing, a Syntax Error will be produced.

See also : **Boolean Logic Operators, Select..Case..EndSelect.**

Include

Syntax
Include "Filename"

Overview
Include another file at the current point in the compilation. All the lines in the new file are compiled as if they were in the current file at the point of the Include directive.

A Common use for the include command is shown in the example below. Here a small master document is used to include a number of smaller library files which are all compiled together to make the overall program.

Operators
- ***Filename*** is any valid Proton Amicus18 file.

Example
```
' Main Program Includes sub files
   Include "StartCode.bas"
   Include "MainCode.bas"
   Include "EndCode.bas"
```

Notes
The file to be included into the BASIC listing may be in one of three places on the hard drive if a specific path is not chosen.

- 1… Within the BASIC program's directory.
- 2… Within the compiler's current directory.
- 3… Within the compiler's Includes folder of the compiler's current directory.

The list above also shows the order in which they are searched for.

Using Include files to tidy up your code.
Placing the include file at the beginning of the program allows all of the variables used by the routines held within it to be pre-declared. This makes for a tidier program, as a long list of variables is not present in the main program.

There are some considerations that must be taken into account when writing code for an include file, these are:

- **1). Always jump over the subroutines.**

When the include file is placed at the top of the program this is the first place that the compiler starts, therefore, it will run the subroutine/s first and the **Return** command will be pointing to a random place within the code. To overcome this, place a **GoTo** statement just before the subroutine starts.

For example:

```
   GoTo OverThisSubroutine          ' Jump over the subroutine

' The subroutine is placed here

OverThisSubroutine:                 ' Jump to here first
```


- **2). Variable and Label names should be as meaningful as possible.**

For example. Instead of naming a variable Loop, change it to Isub_Loop. This will help eliminate any possible duplication errors, caused by the main program trying to use the same variable or label name. However, try not to make them too obscure as your code will be harder to read and understand, it might make sense at the time of writing, but come back to it after a few weeks and it will be meaningless.

- **3). Comment, Comment, and Comment some more.**

This cannot be emphasised enough. Always place a plethora of remarks and comments. The purpose of the subroutine/s within the include file should be clearly explained at the top of the program, also, add comments after virtually every command line, and clearly explain the purpose of all variables and constants used. This will allow the subroutine to be used many weeks or months after its conception. A rule of thumb that I use is that I can understand what is going on within the code by reading only the comments to the right of the command lines.

- **4).Change the file extension**

In order not to get an include file confused with the main code file, it is advisable to give the include file the extension of ".inc". For example "MyIncludeFile.inc"

Inc

Syntax
Inc Variable

Overview
Increment a variable i.e. Var1 = Var1 + 1

Operators
- ***Variable*** is a user defined variable

Example
```
Var1 = 1
Repeat
   HRsout Dec Var1, 13
   DelayMs 200
   Inc Var1
Until Var1 > 10
```

The above example shows the equivalent to the **For**-**Next** loop:

```
For Var1 = 1 to 10
  HRsout Dec Var1, 13
   DelayMs 200
Next
```

See also : **Dec**.

Inkey

Syntax
Variable = **Inkey**

Overview
Scan a keypad and place the returned value into variable

Operators
- ***Variable*** is a user defined variable

Example
```
Dim Var1 as Byte
While 1 = 1                 ' Create an infinite loop
  Var1 = Inkey              ' Scan the keypad
  DelayMs 50                ' Debounce by waiting 50ms
  HRsout "Key ", Dec Var1, 13 ' Display result on the serial terminal
  DelayMs 500               ' Wait for half a second
Wend                        ' Close the loop
```

Notes
Inkey will return a value between 0 and 16. If no key is pressed, the value returned is 16.

Using a **LookUp** command, the returned values can be re-arranged to correspond with the legends printed on the keypad:

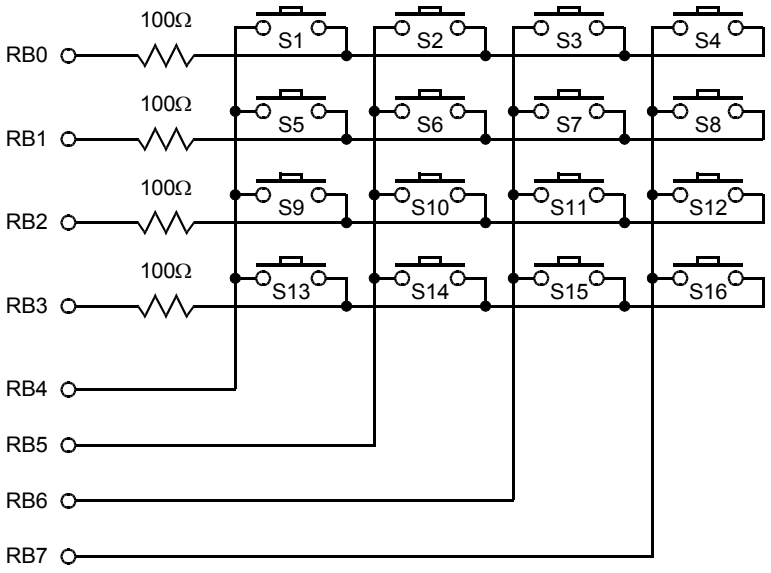
```
Var1 = Inkey
Key = LookUp Var1, [255,1,4,7,"*",2,5,8,0,3,6,9,"#",0,0,0]
```

The above example is only a demonstration, the values inside the **LookUp** command will need to be re-arranged for the type of keypad used, and it's connection configuration.

Declare
Declare Keypad_Port Port
Assigns the Port that the keypad is attached to.

The keypad routine requires pull-up resistors, therefore, the best Port for this device is PortB, which comes equipped with internal pull-ups. If the **Declare** is not used in the program, then PortB is the default Port.

The circuit illustrates a typical connection of a 16-button keypad to the Amicus18 board.



Input

Syntax
Input Port . Pin

Overview
Makes the specified Port or Pin an input.

Operators
- ***Port.Pin*** must be a Port, or Port.Pin constant declaration.

Example
```
Input RA0              ' Make bit-0 of PortA an input
Input PORTA            ' Make all of PortA an input
```

Notes
An Alternative method for making a particular pin an input is by directly modifying the Tris register:

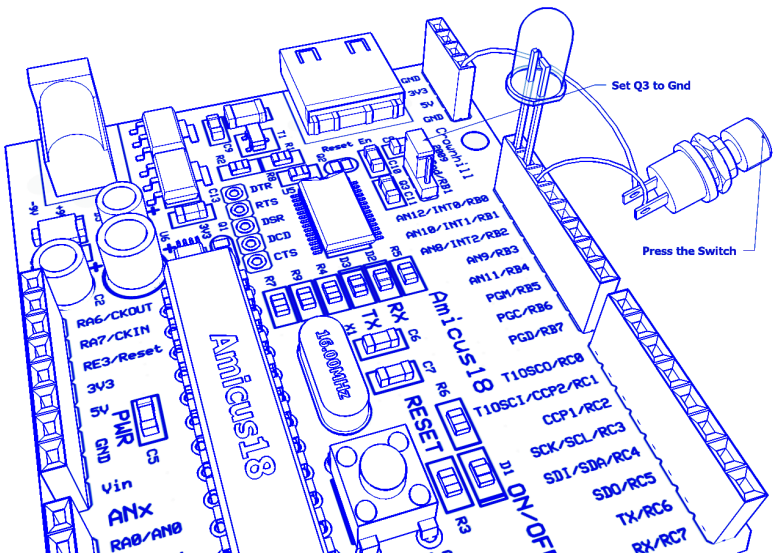
```
TRISB.0 = 1            ' Set PortB, bit-0 to an input
```

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

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

In the above examples, setting a Tris bit to 1 makes the pin an input, and conversely, setting the bit to 0 makes the pin an output.

Example2
```
' Light an LED when a button is pressed
Input RB2              ' Make bit-2 of PortB an input
PortB_Pullups = On     ' Enable the internal pullup resistors on PortB
While 1 = 1            ' Create an infinite loop
   If RB2 = 0 Then     ' Is the button pressed ?
      DelayMs 50       ' Yes. So debounce it
      High RB0         ' Illuminate the LED
   Else                ' Otherwise…
      Low RB0          ' Extinguish the LED
   EndIf
Wend                   ' Do it forever
```



See also : **Output**.

LCDread

Syntax
Variable = **LCDread** Ypos, Xpos

Or

Variable = **LCDread** Text Ypos, Xpos

Overview
Read a byte from a graphic LCD. Can also read Text RAM from a Toshiba T6963 LCD.

Operators
- ***Variable*** is a user defined variable.
- ***Ypos*** :
  With a Samsung KS0108 graphic LCD this may be a constant, variable or expression within the range of 0 to 7 This corresponds to the line number of the LCD, with 0 being the top row.

  With a Toshiba T6963 graphic LCD this may be a constant, variable or expression within the range of 0 to the Y resolution of the display. With 0 being the top line.
- ***Xpos***:
  With a Samsung KS0108 graphic LCD this may be a constant, variable or expression with a value of 0 to 127. This corresponds to the X position of the LCD, with 0 being the far left column.

  With a Toshiba graphic LCD this may be a constant, variable or expression with a value of 0 to the X resolution of the display divided by the font width (LCD_X_Res / LCD_Font_Width). This corresponds to the X position of the LCD, with 0 being the far left column.

Example
```
' Read and display the top row of the Samsung KS0108 graphic LCD
  Declare LCD_Type = Samsung      ' Target a Samsung graphic LCD

  Dim Var1 as Byte
  Dim Xpos as Byte
  Cls                             ' Clear the LCD
  Print "Testing 1 2 3"
  For Xpos = 0 to 127             ' Create a loop of 128
    Var1 = LCDread 0, Xpos        ' Read the LCD's top line
    Print At 1, 0, "Chr= ", Dec Var1,"  "
    DelayMs 100
  Next
  Stop
```

Notes
The graphic LCDs that are compatible with compiler are the Samsung KS0108, and the Toshiba T6963. The Samsung display has a pixel resolution of 64 x 128. The 64 being the Y axis, made up of 8 lines each having 8-bits. The 128 being the X axis, made up of 128 positions. The Toshiba LCDs are available with differing resolutions.

As with **LCDwrite**, the graphic LCD must be targeted using the **LCD_Type Declare** directive before this command may be used.

The Toshiba T6963 graphic LCDs split their graphic and text information within internal RAM. This means that the **LCDread** command can also be used to read the textual information as well as the graphical information present on the LCD. Placing the word TEXT after the **LCDread** command will direct the reading process to Text RAM.

Example

```
' Read text from a Toshiba graphic LCD
  Declare LCD_Type = Toshiba            ' Use a Toshiba T6963 graphic LCD
'
' LCD interface pin assignments
'
  Declare LCD_DTPort = PortB            ' LCD's Data port
  Declare LCD_WRPin = PortA.2           ' LCD's WR line
  Declare LCD_RDPin = PortA.1           ' LCD's RD line
  Declare LCD_CEPin = PortA.0           ' LCD's CE line
  Declare LCD_CDPin = PortA.3           ' LCD's CD line
  Declare LCD_RSTPin = PortA.4          ' LCD's Reset line (Optional)
'
' LCD characteristics
'
  Declare LCD_X_Res = 128               ' LCD's X Resolution
  Declare LCD_Y_Res = 64                ' LCD's Y Resolution
  Declare LCD_Font_Width = 8            ' The width of the LCD's font

  Dim Charpos as Byte                   ' The X position of the read
  Dim Char as Byte                      ' The byte read from the LCD

  DelayMs 200                           ' Wait for the LCD to stabilise
  Cls                                   ' Clear the LCD
  Print At 0,0," This is for Copying" ' Display text on top line of LCD
  For Charpos = 0 to 20                 ' Create a loop of 21 cycles
    Char = LCDread TEXT 0,Charpos       ' Read the top line of the LCD
    Print At 1,Charpos,Char             ' Print the byte read on the second line
    DelayMs 100                         ' A small delay so we can see things happen
  Next                                  ' Close the loop
  Stop
```

See also : **LCDwrite for a description of the screen formats, Pixel, Plot, Toshiba_Command Toshiba_UDG, UnPlot, see Print for LCD connections.**

LCDwrite

Syntax
LCDwrite Ypos, Xpos, [Value ,{ Value etc…}]

Overview
Write a byte to a graphic LCD.

Operators
- ***Ypos*** :
  With a Samsung KS0108 graphic LCD this may be a constant, variable or expression within the range of 0 to 7 This corresponds to the line number of the LCD, with 0 being the top row.

  With a Toshiba T6963 graphic LCD this may be a constant, variable or expression within the range of 0 to the Y resolution of the display. With 0 being the top line.
- ***Xpos***:
  With a Samsung KS0108 graphic LCD this may be a constant, variable or expression with a value of 0 to 127. This corresponds to the X position of the LCD, with 0 being the far left column.
  With a Toshiba graphic LCD this may be a constant, variable or expression with a value of 0 to the X resolution of the display divided by the font width (LCD_X_Res / LCD_Font_Width). This corresponds to the X position of the LCD, with 0 being the far left column.
- ***Value*** may be a constant, variable, or expression, within the range of 0 to 255 (byte).

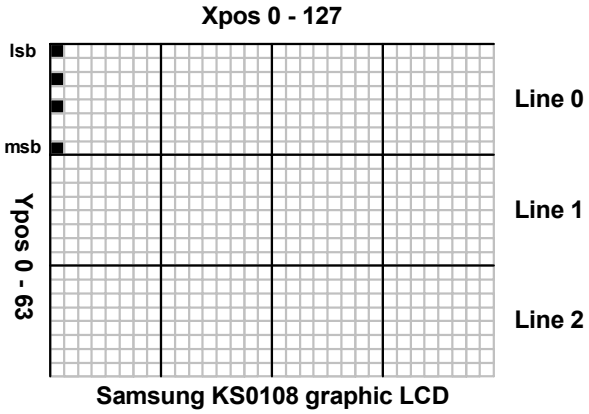
Example 1
```
' Display a line on the top row of a Samsung KS0108 graphic LCD
  Declare LCD_Type = Samsung          ' Target a Samsung graphic LCD
  Dim Xpos as Byte
  Cls                                 ' Clear the LCD
  For Xpos = 0 to 127                 ' Create a loop of 128
    LCDwrite 0, Xpos, [%11111111]     ' Write to the LCD's top line
    DelayMs 100
  Next
  Stop
```

Example 2
```
' Display a line on the top row of a Toshiba 128x64 graphic LCD
  Declare LCD_Type = Toshiba          ' Target a Toshiba graphic LCD
  Dim Xpos as Byte
  Cls                                 ' Clear the LCD
  For Xpos = 0 to 20                  ' Create a loop of 21
    LCDwrite 0, Xpos, [%00111111]     ' Write to the LCD's top line
    DelayMs 100
  Next
  Stop
```

Notes
The graphic LCDs that are compatible with Proton Amicus18 are the Samsung KS0108, and the Toshiba T6963. The Samsung display has a pixel resolution of 64 x 128. The 64 being the Y axis, made up of 8 lines each having 8-bits. The 128 being the X axis, made up of 128 positions. The Toshiba LCDs are available with differing resolutions.

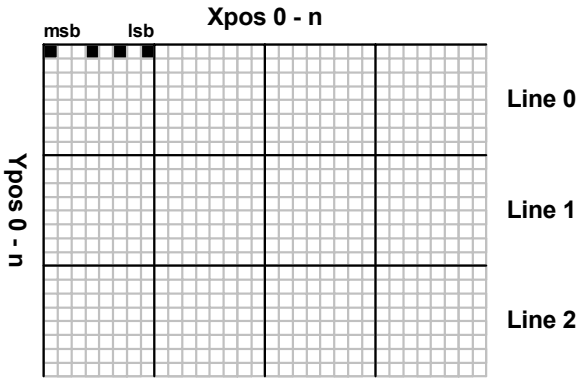
There are important differences between the Samsung and Toshiba screen formats. The diagrams below show these in more detail:

The diagram below illustrates the position of one byte at position 0,0 on a Samsung KS0108 LCD screen. The least significant bit is located at the top. The byte displayed has a value of 149 (10010101).



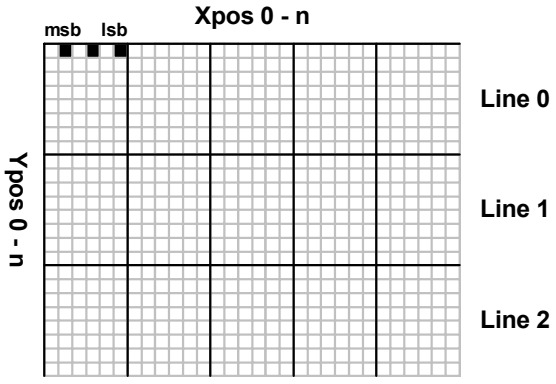
Samsung KS0108 graphic LCD

The diagram below illustrates the position of one byte at position 0,0 on a Toshiba T6963 LCD screen in 8-bit font mode. The least significant bit is located at the right of the screen byte. The byte displayed has a value of 149 (10010101).



Toshiba T6963 LCD. (8-bit Font mode)

The diagram below illustrates the position of one byte at position 0,0 on a Toshiba T6963 LCD screen in 6-bit font mode. The least significant bit is located at the right of the screen byte. The byte displayed still has a value of 149 (10010101), however, only the first 6 bits are displayed (010101) and the other two are discarded.



Toshiba T6963 LCD. (6-bit Font mode)

See also : **LCDread, Plot, Toshiba_Command, Toshiba_UDG, UnPlot.**
See Print for LCD connections.

Len

Syntax
Variable = **Len** (Source String)

Overview
Find the length of a String. (not including the null terminator) .

Operators
- ***Variable*** is a user defined variable of type Bit, Byte, Byte Array, Word, Word Array, Dword, or Float.
- ***Source String*** can be a String variable, or a Quoted String of Characters. The Source String can also be a Byte, Word, Byte Array, Word Array or Float variable, in which case the value contained within the variable is used as a pointer to the start of the Source String's address in RAM. A third possibility for Source String is a Label name, in which case a null terminated Quoted String of Characters is read from a **Cdata** table.

Example 1
```
' Display the length of SourceString
  Dim SourceString as String * 20     ' Create a String capable of 20 characters
  Dim Length as Byte
  SourceString = "HELLO WORLD"        ' Load the source string with characters
  Length = Len SourceString           ' Find the length
  HRsout Dec Length  , 13             ' Display the result, which will be 11
```

Example 2
```
' Display the length of a Quoted Character String
  Dim Length as Byte
  Length = Len "HELLO WORLD"          ' Find the length
  HRsout Dec Length  , 13             ' Display the result, which will be 11
```

Example 3
```
' Display the length of SourceString using a pointer to SourceString
  Dim SourceString as String * 20     ' Create String capable of 20 characters
  Dim Length as Byte
' Create a Word variable to hold the address of SourceString
  Dim StringAddr as Word
  SourceString = "HELLO WORLD"        ' Load the source string with characters
  StringAddr = VarPtr SourceString    ' Locate start address of SourceString
  Length = Len StringAddr             ' Find the length
  HRsout Dec Length, 13               ' Display the result, which will be 11
  Stop
```

Example 4
```
' Display the length of a Cdata string
  Dim Length as Byte
  Length = Len Source                 ' Find the length
  HRsout Dec Length                   ' Display the result, which will be 11
  Stop
' Create a null terminated string of characters in code memory
Source:
  Cdata "HELLO WORLD", 0
```

See also : **Creating and using Strings, Creating and using Virtual Strings with Cdata, Cdata, Left$, Mid$, Right$, Str$, ToLower, ToUpper, VarPtr.**

Left$

Syntax
Destination String = **Left$** (Source String, Amount of characters)

Overview
Extract n amount of characters from the left of a source string and copy them into a destination string.

Operators
- ***Destination String*** can only be a String variable, and should be large enough to hold the correct amount of characters extracted from the Source String.
- ***Source String*** can be a String variable, or a Quoted String of Characters. See below for more variable types that can be used for Source String.
- ***Amount of characters*** can be any valid variable type, expression or constant value, that signifies the amount of characters to extract from the left of the Source String. Values start at 1 for the leftmost part of the string and should not exceed 255 which is the maximum allowable length of a String variable.

Example 1.
```
' Copy 5 characters from the left of SourceString into DestString
  Dim SourceString as String * 20   ' Create a String capable of 20 characters
  Dim DestString as String * 20     ' Create another String for 20 characters

  SourceString = "HELLO WORLD"      ' Load the source string with characters
' Copy 5 characters from the source string into the destination string
  DestString = Left$ (SourceString, 5)
  HRsout DestString, 13             ' Display the result, which will be "HELLO"
```

Example 2.
```
' Copy 5 chars from the left of a Quoted Character String into DestString
  Dim DestString as String * 20     ' Create a String capable of 20 characters

' Copy 5 characters from the quoted string into the destination string
  DestString = Left$ ("HELLO WORLD", 5)
  HRsout DestString, 13             ' Display the result, which will be "HELLO"
```

The Source String can also be a Byte, Word, Byte Array, Word Array or Float variable, in which case the value contained within the variable is used as a pointer to the start of the Source String's address in RAM.

Example 3.
```
' Copy 5 chars from left SourceString into DestString pointer to SourceString

  Dim SourceString as String * 20   ' Create a String capable of 20 characters
  Dim DestString as String * 20     ' Create another String for 20 characters
' Create a Word variable to hold the address of SourceString
  Dim StringAddr as Word

  SourceString = "HELLO WORLD"      ' Load the source string with characters
' Locate the start address of SourceString in RAM
  StringAddr = VarPtr (SourceString)
' Copy 5 characters from the source string into the destination string
  DestString = Left$ (StringAddr, 5)
  HRsout DestString  , 13           ' Display the result, which will be "HELLO"
```


A third possibility for Source String is a Label name, in which case a null terminated Quoted String of Characters is read from a **Cdata** table.

Example 4.

```
' Copy 5 characters from the left of a Cdata table into DestString

   Dim DestString as String * 20     ' Create a String capable of 20 characters

' Copy 5 characters from label Source into the destination string
   DestString = Left$ (Source, 5)
   HRsout DestString  , 13           ' Display the result, which will be "HELLO"
   Stop

' Create a null terminated string of characters in code memory
Source:
   Cdata "HELLO WORLD", 0
```

See also : **Creating and using Strings, Creating and using Virtual Strings with Cdata, Cdata, Len, Mid$, Right$, Str$, ToLower, ToUpper , VarPtr.**

Line

Syntax
Line Set_Clear, Xpos Start, Ypos Start, Xpos End, Ypos End

Overview
Draw a straight line in any direction on a graphic LCD.

Operators
- ***Set_Clear*** may be a constant or variable that determines if the line will set or clear the pixels. A value of 1 will set the pixels and draw a line, while a value of 0 will clear any pixels and erase a line.
- ***Xpos Start*** may be a constant or variable that holds the X position for the start of the line. Can be a value from 0 to 255.
- ***Ypos Start*** may be a constant or variable that holds the Y position for the start of the line. Can be a value from 0 to 255.
- ***Xpos End*** may be a constant or variable that holds the X position for the end of the line. Can be a value from 0 to 255.
- ***Ypos End*** may be a constant or variable that holds the Y position for the end of the line. Can be a value from 0 to 255.

Example
```
' Draw a line from 0,0 to 120,34

  Dim Xpos_Start as Byte
  Dim Xpos_End as Byte
  Dim Ypos_Start as Byte
  Dim Ypos_End as Byte
  Dim SetClr as Byte

  DelayMs 200           ' Wait for things to stabilise
  Cls                   ' Clear the LCD
  Xpos_Start = 0
  Ypos_Start = 0
  Xpos_End = 120
  Ypos_End = 34
  SetClr = 1
  Line SetClr, Xpos_Start, Ypos_Start, Xpos_End, Ypos_End
  Stop
```

See Also : **Box**, **Circle**.

LineTo

Syntax
LineTo Set_Clear, Xpos End, Ypos End

Overview
Draw a straight line in any direction on a graphic LCD, starting from the previous **Line** command's end position.

Operators
- ***Set_Clear*** may be a constant or variable that determines if the line will set or clear the pixels. A value of 1 will set the pixels and draw a line, while a value of 0 will clear any pixels and erase a line.
- ***Xpos End*** may be a constant or variable that holds the X position for the end of the line. Can be a value from 0 to 255.
- ***Ypos End*** may be a constant or variable that holds the Y position for the end of the line. Can be a value from 0 to 255.

Example
```
' Draw a line from 0,0 to 120,34. Then from 120,34 to 0,63

  Dim Xpos_Start as Byte
  Dim Xpos_End as Byte
  Dim Ypos_Start as Byte
  Dim Ypos_End as Byte
  Dim SetClr as Byte

  DelayMs 200           ' Wait for things to stabilise
  Cls                   ' Clear the LCD
  Xpos_Start = 0
  Ypos_Start = 0
  Xpos_End = 120
  Ypos_End = 34
  SetClr = 1
  Line SetClr, Xpos_Start, Ypos_Start, Xpos_End, Ypos_End
  Xpos_End = 0
  Ypos_End = 63
  LineTo SetClr, Xpos_End, Ypos_End
  Stop
```

Notes
The **LineTo** command uses the compiler's internal system variables to obtain the end position of a previous **Line** command. These X and Y coordinates are then used as the starting X and Y coordinates of the **LineTo** command.

See Also : **Line**, **Box**, **Circle**.

LoadBit

Syntax
LoadBit Variable, Index, Value

Overview
Clear, or Set a bit of a variable or register using a variable index to point to the bit of interest.

Operators
- ***Variable*** is a user defined variable, of type Byte, Word, or Dword.
- ***Index*** is a constant, variable, or expression that points to the bit within Variable that requires accessing.
- ***Value*** is a constant, variable, or expression that will be placed into the bit of interest. Values greater than 1 will set the bit.

Example
```
' Copy variable ExVar bit by bit into variable PT_Var
   Dim ExVar as Word
   Dim Index as Byte
   Dim Value as Byte
   Dim PT_Var as Word
Again:
   PT_Var = %0000000000000000
   ExVar = %1011011000110111
   HRsout Bin16 ExVar, 13              ' Display the original variable
   For Index = 0 to 15                 ' Create a loop for 16 bits
      Value = GetBit ExVar, Index      ' Examine each bit of variable ExVar
      LoadBit PT_Var, Index, Value     ' Set or Clear each bit of PT_Var
      HRsout Bin16 PT_Var, 13          ' Display the copied variable
      DelayMs 100                      ' Slow things down to see what's happening
   Next                                ' Close the loop
   Hrsout 13
   GoTo Again                          ' Do it forever
```

Notes
There are many ways to clear or set a bit within a variable, however, each method requires a certain amount of manipulation, either with rotates, or alternatively, the use of indirect addressing using the FSR, and INDF registers. Each method has its merits, but requires a certain amount of knowledge to accomplish the task correctly. The **LoadBit** command makes this task extremely simple by taking advantage of the indirect method using FSR0, and INDF0, however, this is not necessarily the quickest method, or the smallest, but it is the safest. For speed and size optimisation, there is no shortcut to experience.

To Clear a known constant bit of a variable or register, then access the bit directly using Port.n. i.e.

```
RA1 = 0
```

To Set a known constant bit of a variable or register, then access the bit directly using Port.n. i.e.

```
RA1 = 1
```

If a Port is targeted by **LoadBit**, it's Tris register is not affected.

See also : **ClearBit**, **GetBit**, **SetBit**.

LookDown

Syntax
Variable = **LookDown** Index, [Constant {, Constant…etc }]

Overview
Search constants(s) for index value. If index matches one of the constants, then store the matching constant's position (0-N) in variable. If no match is found, then the variable is unaffected.

Operators
- ***Variable*** is a user define variable that holds the result of the search.
- ***Index*** is the variable/constant being sought.
- ***Constant(s),...*** is a list of values. A maximum of 256 values may be placed between the square brackets.

Example
```
Dim Value as Byte
Dim Result as Byte
Value = 177           ' The value to look for in the list
Result = 255          ' Default to value 255
Result = LookDown Value, [75,177,35,1,8,29,245]
HRsout "Value matches ", Dec Result, " in list\r"
```

In the above example, **HRsout** displays, "Value matches 1 in list" because Value (177) matches item 1 of [75,177,35,1,8,29,245]. Note that index numbers count up from 0, not 1; that is in the list [75,177,35,1,8,29,245], 75 is item 0.

If the value is not in the list, then Result is unchanged.

Notes
LookDown is similar to the index of a book. You search for a topic and the index gives you the page number. Lookdown searches for a value in a list, and stores the item number of the first match in a variable.

LookDown also supports text phrases, which are basically lists of byte values, so they are also eligible for Lookdown searches:

```
Dim Value as Byte
Dim Result as Byte
Value = "e"                   ' The value to look for in the list
Result = 255                  ' Default to value 255
Result = LookDown Value, ["Hello World"]
Hrsout Result, 13             ' Display the result
```

In the above example, Result will hold a value of 1, which is the position of character 'e'

See also : **Cdata**, **Cread**, **Edata**, **Eread**, **Cdata**, **LookDownL**, **LookUp**, **LookUpL**, **Lread**.

LookDownL

Syntax
Variable = **LookDownL** Index, {Operator} [Value {, Value…etc }]

Overview
A comparison is made between index and value; if the result is true 0 is written into ***Variable***. If that comparison was false, another comparison is made between ***Value*** and ***Value1***; if the result is true, 1 is written into ***Variable***. This process continues until a true is yielded, at which time the index is written into ***Variable***, or until all entries are exhausted, in which case ***Variable*** returns unchanged.

Operators
- ***Variable*** is a user define variable that holds the result of the search.
- ***Index*** is the variable or constant being sought.
- ***Value(s)*** can be a mixture of constants, quoted character strings, and variables. Expressions may not be used in the Value list, although they may be used as the index value. A maximum of 256 values may be placed between the square brackets.
- ***Operator*** is an optional comparison operator and may be one of the following:

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

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

Example
```
' LookdownL Demo
  Dim ByteVar As Byte
  Dim LookVar As Byte
  Dim Index As Word
'
' Display the contents of the lookup table
'
  HRSOut "10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22 ", 13
'
' Scan the LookdownL table
'
  LookVar = 14
  Index = LookDownL LookVar, < [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22]
  HRSOut "Index: ", Dec Index, 13        ' Display the index value
'
' Find the value based upon a copy of the LookdownL table, but now in a Lookup table
'
  ByteVar = LookUp Index, [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22]
  HRSOut "Result: ", Dec ByteVar, 13     ' Display the value found
```

Notes
Because **LookDownL** is more versatile than the standard **LookDown** command, it generates larger code. Therefore, if the search list is made up only of 8-bit constants and strings, use **LookDown**.

See also : **Cdata**, **Cread**, **Edata**, **Eread**, **Cdata**, **LookDown**, **LookUp**, **LookUpL**, **Lread, Lread8, Lread16, Lread32**.

LookUp

Syntax
Variable = **LookUp** Index, [Constant {, Constant…etc }]

Overview
Look up the value specified by the index and store it in variable. If the index exceeds the highest index value of the items in the list, then variable remains unchanged.

Operators
- ***Variable*** may be a constant, variable, or expression. This is where the retrieved value will be stored.
- ***Index*** may be a constant of variable. This is the item number of the value to be retrieved from the list.
- ***Constant(s)*** may be any 8-bit constant value (0-255). A maximum of 256 values may be placed between the square brackets.

Example
```
' Create an animation of a spinning line on an alphanumeric LCD.
   Dim Index as Byte
   Dim Frame as Byte
   DelayMs 100                ' Wait for the LCD to stabilise
   Cls                        ' Clear the LCD
Rotate:
   For Index = 0 to 3         ' Create a loop of 4
      Frame = LookUp Index, ["|\-/"] ' Table of animation characters
      Print At 1, 1, Frame    ' Display the character
      DelayMs 200             ' So we can see the animation
   Next                       ' Close the loop
   GoTo Rotate                ' Repeat forever
```

Notes
Index starts at value 0. For example, in the **LookUp** command below. If the first value (10) is required, then ***Index*** should be loaded with 0, and 1 for the second value (20) etc.

```
   Var1 = LookUp Index, [10, 20, 30]
```

See also : **Cread**, **Edata**, **Eread**, **Cdata**, **LookDown**, **LookDownL**, **LookUpL**, **Lread**, **Lread8**, **Lread16**, **Lread32**.

LookUpL

Syntax
Variable = **LookUpL** Index, [Value {, Value…etc }]

Overview
Look up the value specified by the index and store it in variable. If the index exceeds the highest index value of the items in the list, then variable remains unchanged. Works exactly the same as **LookUp**, but allows variable types or constants in the list of values.

Operators
- ***Variable*** may be a constant, variable, or expression. This is where the retrieved value will be stored.
- ***Index*** may be a constant of variable. This is the item number of the value to be retrieved from the list.
- ***Value(s)*** can be a mixture of constants, quoted character strings, and variables. A maximum of 256 values may be placed between the square brackets.

Example
```
Dim Var1 as Byte
Dim WordVar as Word
Dim Index as Byte
Dim Assign as Word
Var1 = 10
WordVar = 1234
Index = 0         ' Point to the first value in the list (WordVar)
Assign = LookUpL Index, [WordVar, Var1, 12345]
```

Notes
Expressions may not be used in the Value list, although they may be used as the Index value.

Because **LookUpL** is capable of processing any variable and constant type, the code produced is a lot larger than that of **LookUp**. Therefore, if only 8-bit constants are required in the list, use **LookUp** instead.

See also : **Cread**, **Edata**, **Eread**, **Cdata**, **LookDown**, **LookDownL**, **LookUp**, **Lread, Lread8, Lread16, Lread32**.

Low

Syntax
Low Port or Port.Bit

Overview
Place a Port or bit in a low state. For a port, this means filling it with 0's. For a bit this means setting it to 0.

Operators
- ***Port*** can be any valid port.
- ***Port.Bit*** can be any valid port and bit combination, i.e. PortB.0

Example
```
Symbol LED = RB4
Low LED         ' Make output and Pull down bit-4 of PortB
Low RB0         ' Make output and Pull down bit-0 of PortB
Low PORTBB      ' Make output and Pull down all of PortB
```

See also : **High**, **Symbol**.

Lread

Syntax
Variable = **Lread** Label

Overview
Read a value from a **Cdata** table and place into Variable

Operators
- ***Variable*** is a user defined variable.
- ***Label*** is a label name preceding the **Cdata** statement, or expression containing the Label name.

Example
```
  Dim Char as Byte
  Dim Loop as Byte
  Cls
  For Loop = 0 to 9                ' Create a loop of 10
    Char = Lread LABEL + Loop      ' Read memory location LABEL + Loop
    Hrsout Char                    ' Display the value read
  Next
  Hrsout 13
  Stop

LABEL: Cdata "HELLO WORLD"         ' Create a string of text in code memory
```

The program above reads and displays 10 values from the address located by the LABEL accompanying the **Cdata** command. Resulting in "HELLO WORL" being displayed.

Cdata is not simply used for character storage, it may also hold 8, 16, 32 bit, or floating point values. The example below illustrates this:

```
  Dim ByteVar as Byte
  Dim WordVar as Word
  Dim DwordVar as Dword
  Dim FloatVar as Float
  ByteVar = Lread Bit8_Val         ' Read the 8-bit value
  Hrsout Dec ByteVar, 13
  WordVar= Lread Bit16_Val         ' Read the 16-bit value
  Hrsout Dec WordVar, 13
  DwordVar = Lread Bit32_Val       ' Read the 32-bit value
  Hrsout Dec DwordVar, 13
  Flt1 = Lread Flt_Val             ' Read the floating point value
  Hrsout Dec FloatVar, 13
  Stop

Bit8_Val: Cdata 123
Bit16_Val: Cdata 1234
Bit32_Val: Cdata 123456
Flt_Val: Cdata 123.456
```


Floating point example.

```
' Read floating point data from a table and display the results
  Dim FloatVar as Float          ' Create a Floating point variable
  Dim Fcount as Byte
  Cls                            ' Clear the LCD
  Fcount = 0                     ' Clear the table counter
  Repeat                         ' Create a loop
    FloatVar = Lread FlTable + Fcount' Read the data from the Cdata table
    Print At 1,1, Dec3 Flt       ' Display the data read
    Fcount = Fcount + 2          ' Point to next value, by adding 2 to counter
    DelayMs 1000                 ' Slow things down
  Until FloatVar = 0.005         ' Stop when 0.005 is read
  Stop
FlTable:
  Cdata as Float 3.14, 65535.123, 1234.5678, -1243.456, -3.14, 998999.12,_
                 0.005
```

Notes

Cdata tables should be placed at the end of the BASIC program. If an **Cdata** table is placed at the beginning of the program, then a **GoTo** command must jump over the tables, to the main body of code.

```
  GoTo OverDataTable
  Cdata 1,2,3,4,5,6
OverDataTable:

  { rest of code here}
```

See also : **Cdata**, **Cread**, **Cdata**.

Lread8, Lread16, Lread32

Syntax
Variable = **Lread8** Label [Offset Variable]

or

Variable = **Lread16** Label [Offset Variable]

or

Variable = **Lread32** Label [Offset Variable]

Overview
Read an 8, 16, or 32-bit value from an **Cdata** table using an offset of Offset Variable and place into Variable, with more efficiency than using **Lread**.

Lread8 will access 8-bit values from an **Cdata** table.
Lread16 will access 16-bit values from an **Cdata** table.
Lread32 will access 32-bit values from an **Cdata** table, this also includes floating point values.

Operators
- ***Variable*** is a user defined variable of type Bit, Byte, Byte Array, Word, Word Array, Dword, or Float.
- ***Label*** is a label name preceding the **Cdata** statement of which values will be read from.
- ***Offset*** Variable can be a constant value, variable, or expression that points to the location of interest within the **Cdata** table.

Lread8 Example
```
' Extract the second value from within an 8-bit Cdata table
  Dim Offset as Byte           ' Create a Byte size variable for the offset
  Dim Result as Byte           ' Create a Byte size variable to hold the result

  Offset = 1                   ' Point to the second value in the Cdata table
' Read the 8-bit value pointed to by Offset
  Result = Lread8 ByteTable[Offset]
  HRsout Dec Result  , 13      ' Display the decimal result on the serial terminal
  Stop

' Create a table containing only 8-bit values
ByteTable: Cdata as Byte 100, 200
```

Lread16 Example
```
' Extract the second value from within a 16-bit Cdata table
  Dim Offset as Byte           ' Create a Byte size variable for the offset
  Dim Result as Word           ' Create a Word size variable to hold the result

  Offset = 1                   ' Point to the second value in the Cdata table
' Read the 16-bit value pointed to by Offset
  Result = Lread16 WordTable[Offset]
  HRsout Dec Result  , 13      ' Display the decimal result on the serial terminal
  Stop

' Create a table containing only 16-bit values
WordTable: Cdata as Word 1234, 5678
```


Lread32 Example

```
' Extract the second value from within a 32-bit Cdata table
  Dim Offset as Byte            ' Create a Byte size variable for the offset
  Dim Result as Dword           ' Create a Dword size variable to hold the result

  Offset = 1                    ' Point to the second value in the Cdata table
' Read the 32-bit value pointed to by Offset
  Result = Lread32 DwordTable[Offset]
  HRsout Dec Result, 13         ' Display the decimal result on the serial terminal
  Stop

' Create a table containing only 32-bit values
DwordTable: Cdata as Dword 12340, 56780
```

Notes

Data storage in any program is of paramount importance, and although the standard **Lread** command can access multi-byte values from an **Cdata** table, it was not originally intended as such, and is more suited to accessing character data or single 8-bit values. However, the Lread8, Lread16, and Lread32 commands are specifically written in order to efficiently read data from an **Cdata** table, and use the least amount of code space in doing so, thus increasing the speed of operation. Which means that wherever possible, **Lread** should be replaced by Lread8, Lread16, or Lread32.

See also : **Cdata**, **Cread**, **Cdata**, **Lread**.

Mid$

Syntax
Destination String = **Mid$** (Source String, Position within String, Amount of characters)

Overview
Extract n amount of characters from a source string beginning at n characters from the left, and copy them into a destination string.

Operators
- ***Destination String*** can only be a String variable, and should be large enough to hold the correct amount of characters extracted from the Source String.
- ***Source String*** can be a String variable, or a Quoted String of Characters. See below for more variable types that can be used for Source String.
- ***Position within String*** can be any valid variable type, expression or constant value, that signifies the position within the Source String from which to start extracting characters. Values start at 1 for the leftmost part of the string and should not exceed 255 which is the maximum allowable length of a String variable.
- ***Amount of characters*** can be any valid variable type, expression or constant value, that signifies the amount of characters to extract from the left of the Source String. Values start at 1 and should not exceed 255 which is the maximum allowable length of a String variable.

Example 1
```
' Copy 5 characters from position 4 of SourceString into DestString

  Dim SourceString as String * 20   ' Create a String of 20 characters
  Dim DestString as String * 20     ' Create another String

  SourceString = "HELLO WORLD"      ' Load the source string with characters
' Copy 5 characters from the source string into the destination string
  DestString = Mid$ (SourceString, 4, 5)
  HRsout DestString, 13             ' Display the result, which will be "LO WO"
  Stop
```

Example 2
```
' Copy 5 chars from position 4 of a Quoted Character String into DestString

  Dim DestString as String * 20     ' Create a String of 20 characters

' Copy 5 characters from the quoted string into the destination string
  DestString = Mid$ ("HELLO WORLD", 4, 5)
  HRsout DestString, 13             ' Display the result, which will be "LO WO"
  Stop
```

The Source String can also be a Byte, Word, Byte Array, Word Array or Float variable, in which case the value contained within the variable is used as a pointer to the start of the Source String's address in RAM.

Example 3

```
' Copy 5 chars from position 4 of SourceString to DestString with a pointer
' to SourceString

  Dim SourceString as String * 20   ' Create a String of 20 characters
  Dim DestString as String * 20     ' Create another String
' Create a Word variable to hold the address of SourceString
  Dim StringAddr as Word

  SourceString = "HELLO WORLD"      ' Load the source string with characters
' Locate the start address of SourceString in RAM
  StringAddr = VarPtr (SourceString)
' Copy 5 characters from the source string into the destination string
  DestString = Mid$ (StringAddr, 4, 5)
  HRsout DestString  , 13           ' Display the result, which will be "LO WO"
  Stop
```

A third possibility for Source String is a label name, in which case a null terminated Quoted String of Characters is read from a **Cdata** table.

Example 4

```
' Copy 5 characters from position 4 of a Cdata table into DestString

  Dim DestString as String * 20     ' Create a String of 20 characters

' Copy 5 characters from label Source into the destination string
  DestString = Mid$ (Source, 4, 5)
  HRsout DestString, 13             ' Display the result, which will be "LO WO"
  Stop

' Create a null terminated string of characters in code memory
Source:
  Cdata "HELLO WORLD", 0
```

See also : **Creating and using Strings, Creating and using Virtual Strings with Cdata, Cdata, Len, Left$, Right$, Str$, ToLower, ToUpper, VarPtr.**

On GoTo

Syntax
On Index Variable **GoTo** Label1 {,...Labeln }

Overview
Cause the program to jump to different locations based on a variable index. Exactly the same functionality as **Branch**.

Operators
- ***Index Variable*** is a constant, variable, or expression, that specifies the label to jump to.
- ***Label1...Labeln*** are valid labels that specify where to branch to. A maximum of 256 labels may be placed after the **GoTo**.

Example
```
   Dim Index as Byte
   Index = 2                  ' Assign Index a value of 2
Start:                        ' Jump to label 2 (LABEL_2) because Index = 2
   On Index GoTo LABEL_0, LABEL_1, LABEL_2
LABEL_0:
   Index = 2                  ' Index now equals 2
   HRsout "LABEL 0", 13       ' Display the LABEL name on the serial terminal
   DelayMs 500                ' Wait 500ms
   GoTo Start                 ' Jump back to Start
LABEL_1:
   Index = 0                  ' Index now equals 0
   HRsout "LABEL 1", 13       ' Display the LABEL name on the serial terminal
   DelayMs 500                ' Wait 500ms
   GoTo Start                 ' Jump back to Start
LABEL_2:
   Index = 1                  ' Index now equals 1
   HRsout "LABEL 2", 13       ' Display the LABEL name on the serial terminal
   DelayMs 500                ' Wait 500ms
   GoTo Start                 ' Jump back to Start
```

The above example we first assign the index variable a value of 2, then we define our labels. Since the first position is considered 0 and the variable Index equals 2 the **On GoTo** command will cause the program to jump to the third label in the list, which is LABEL_2.

Notes
On GoTo is useful when you want to organise a structure such as:

```
   If Var1 = 0 Then GoTo LABEL_0  ' Var1 = 0: go to label "LABEL_0"
   If Var1 = 1 Then GoTo LABEL_1  ' Var1 = 1: go to label "LABEL_1"
   If Var1 = 2 Then GoTo LABEL_2  ' Var1 = 2: go to label "LABEL_2"
```

You can use **On GoTo** to organise this into a single statement:

```
   On Var1 GoTo LABEL_0, LABEL_1, LABEL_2
```

This works exactly the same as the above **If**...**Then** example. If the value is not in range (in this case if Var1 is greater than 2), **On GoTo** does nothing. The program continues with the next instruction.

See also : **Branch, On GoSub.**

On GoSub

Syntax
On Index Variable **GoSub** Label1 {,...Labeln }

Overview
Cause the program to **Call** a subroutine based on an index value. A subsequent **Return** will continue the program immediately following the **On GoSub** command.

Operators
- ***Index Variable*** is a constant, variable, or expression, that specifies the label to call.
- ***Label1...Labeln*** are valid labels that specify where to call. A maximum of 256 labels may be placed after the **GoSub**.

Example
```
  Dim Index as Byte

  While 1 = 1                     ' Create an infinite loop
    For Index = 0 to 2            ' Create a loop to call all the labels
' Call the label depending on the value of Index
      On Index GoSub LABEL_0, LABEL_1, LABEL_2
      DelayMs 500                 ' Wait 500ms after the subroutine has returned
    Next
  Wend                            ' Do it forever
LABEL_0:
  HRsout "LABEL 0\r"              ' Display the LABEL name on the serial terminal
  Return
LABEL_1:
  HRsout "LABEL 1\r"              ' Display the LABEL name on the serial terminal
  Return
LABEL_2:
  HRsout "LABEL 2\r"              ' Display the LABEL name on the serial terminal
  Return
```

The above example, a loop is formed that will load the variable Index with values 0 to 2. The **On Go-Sub** command will then use that value to call each subroutine in turn. Each subroutine will **Return** to the **DelayMs** command, ready for the next scan of the loop.

Notes
On GoSub is useful when you want to organise a structure such as:

```
  If Var1 = 0 Then GoSub LABEL_0    ' Var1 = 0: call label "LABEL_0"
  If Var1 = 1 Then GoSub LABEL_1    ' Var1 = 1: call label "LABEL_1"
  If Var1 = 2 Then GoSub LABEL_2    ' Var1 = 2: call label "LABEL_2"
```

You can use **On GoSub** to organise this into a single statement:

```
  On Var1 GoSub LABEL_0, LABEL_1, LABEL_2
```

This works exactly the same as the above **If**...**Then** example. If the value is not in range (in this case if Var1 is greater than 2), **On GoSub** does nothing. The program continues with the next instruction..

See also : **Branch, On GoTo.**

On_Hardware_Interrupt

Syntax
On_Hardware_Interrupt Label

Overview
Jump to a subroutine when a Hardware interrupt occurs

Operators
- ***Label*** is a valid identifier

Typical format of the interrupt handler.
The interrupt handler subroutine must follow a fixed pattern. First, the contents of the STATUS, BSR and WREG registers must be saved. Note that this is done automatically when using high priority interrupts.

Upon exiting the interrupt, a simple **Retfie** fast (Return From Interrupt Fast) mnemonic can be used, or the contexts of FSR0L, FSR0H, FSR1L, FSR1H, FSR2L, and FSR2H can be saved by issuing the **Context Save** and **Context Restore** commands.

The code within the interrupt handler should be as quick and efficient as possible because, while it's processing the code the main program is halted. When using assembler interrupts, care should be taken to ensure that the watchdog timer does not time-out. Placing a **ClrWdt** instruction at regular intervals within the code will prevent this from happening. An alternative approach would be to disable the watchdog timer altogether at programming time. Note that this is the default state of the Amicus18's microcontroller.

An interrupt, by it's very nature, may happen at any time during the operation of the foreground program, therefore it is important that the interrupt subroutine has as little impact on the program as possible.

Context Save
Issuing the **Context Save** directive will save the WREG, BSR, STATUS registers, as well as the FSR0L, FSR0H, FSR1L, and FSR1H register pairs. If strings or a stack are used within the program, the FSR2L, and FSR2H register pair will also be saved.

The **Context Save** directive should be placed at the very start of the interrupt subroutine, *before* any other command.

Context Restore
Issuing the **Context Restore** directive will restore the WREG, BSR, STATUS registers, as well as the FSR0L, FSR0H, FSR1L, and FSR1H register pairs. If strings or a stack are used within the program, the FSR2L, and FSR2H register pair will also be restored. The interrupt will then be terminated with the **Retfie** mnemonic

The **Context Restore** directive should be placed at the very end of the interrupt subroutine, *after* any other command.

A typical example of a hardware interrupt is shown below

```
' Flash an LED with a Timer0 overflow interrupt
' While flashing another LED in the foreground
'
   Include "Timers.inc"          ' Load the Timer macros into the program

   Symbol LED1 = RB0             ' LED attachement pin
   Symbol LED2 = RB2             ' LED attachement pin

   On_Hardware_Interrupt GoTo InterruptHandler ' Point to interrupt handler

   GoTo MainProgram              ' Jump over the interrupt handler and any subroutines
'
'-----------------------------------------------------
' Timer 0 overflow Interrupt Handler
' Flash an LED on PortB.0
InterruptHandler:
   Context Save
' Was it a Timer0 overflow that triggered the interrupt ?
   If INTCONbits_T0IF = 1 Then
      Toggle LED1                ' Yes. So. Toggle the LED
      Clear INTCONbits_T0IF      ' Clear the Timer0 overflow flag
   EndIf
   Context Restore               ' Exit the interrupt, restoring registers

'-----------------------------------------------------
' Main Program loop
MainProgram:
   Low LED1                          ' \ Extinguish both LEDs
   Low LED2                          ' /
'
' Configure Timer0 for:
'                         Clear TMR0L and TMR0H registers
'                         Interrupt on Timer0
'                         16-bit operation
'                         Internal clock source
'                         1:128 Prescaler
'
   OpenTimer0(TIMER_INT_ON & T0_16BIT & T0_SOURCE_INT & T0_PS_1_128)

   INTCONbits_GIE = 1                ' Enable global interrupts
'
' Flash an LED attached to PortB.2 slowly in the foreground
' Display the current value of Timer0 on the serial terminal
'
   While 1 = 1                       ' Create an infinite loop
      Hrsout "Timer1 = " , Dec ReadTimer0(),13 ' Display Timer0 value
      High LED2                      ' Illuminate the LED
      DelayMS 500                    ' Wait for 500 milliseconds
     Low LED2                        ' Extinguish the LED
      DelayMS 500                    ' Wait for 500 milliseconds
   Wend                              ' Do it forever
```


Managed Hardware Interrupts

Because an interrupt can occur at any time in the program, the code within the interrupt handler must be carefully crafted in order not to alter the contents of critical SFRs (Special Function Registers) and the compiler's System variables.

However, this process can be handled automatically, to a certain extent, by the compiler by wrapping the interrupt handler with the directives: **High_Int_Sub_Start** and **High_Int_Sub_End**.

When the compiler sees these directives it automatically saves the context of the compiler's system variables used within the interrupt and also saves the SFRs:

WREG, STATUS, BSR, FSR0L/FSR0H, FSR1L/FSR1H, FSR2L/FSR2H, PRODL/PRODH, TBLPTRL/TBLPTRH, and TABLAT.

The context variables and registers are saved in a section reserved at the top of RAM. This does come at a price of code and RAM size as well as a little speed loss when the interrupt is entered and exited, but the benefits can outway the penalties.

This method allows high level commands to be placed within the interrupt handler. As an example, the below program counts and displays 2 values on the serial terminal. One count is in the interrupt and one in the main program.

```
' Demonstrate the use of context saving of the compiler's System variables
' while inside an interrupt subroutine
'
'  Creates a high priority interrupt that increments on Timer0
'  Within the interrupt a value is displayed and incremented
'  While in the foreground another value is displayed and incremented
'
' Note: It is not recommended to use large (slow) routines in an interrupt,
' but this program serves to demonstrate
' the use of the directives High_Int_Sub_Start and High_Int_Sub_End

  Include "Timers.inc"              ' Load the Timer macros into the program

' Point the High interrupt handler to the subroutine
  On_Hardware_Interrupt GoTo HighInterruptSub
'
' Create some variables
'
  Dim HighCounter as Dword          ' Counter for the interrupt routine
  Dim ForeGroundCounter as Dword    ' Counter for the foreground routine

  GoTo Main                         ' Jump over the interrupt handler
```



```
' High Priority Hardware Interrupt Handler
' Interrupt's on a Timer0 Overflow Display on the serial terminal
' and increment a value
'
' Indicate to context save the System Variables for High Priority interrupt
  High_Int_Sub_Start
HighInterruptSub:
'
' Save the compiler's system variables used in the interrupt routine only
' Also save some SFR's PRODL\H, FSR0L\H, FSR1L\H, FSR2L\H, TBLPTRL\H, TABLAT
' SFR's WREG, STATUS, and BSR automatically saved by shadow registers
  Context Save
' Display the value on the serial terminal
  Hrsout "Interrupt  ", Dec HighCounter, 13
  Inc HighCounter               ' Increment the value
  INTCONbits_T0IF = 0           ' Clear the Timer0 Overflow flag
'
' Restore compiler's system variables used within the interrupt routine only
' and exit the interrupt with "Retfie 1"
' Restore the SFR's PRODL\H, FSR0L\H, FSR1L\H, FSR2L\H, TBLPTRL\H, TABLAT
' SFR's WREG, STATUS, and BSR are automatically restored by shadow registers
  Context Restore
' Indicate that the High Priority Interrupt block has ended
  High_Int_Sub_End
'-----------------------------------------------------------------
' Main Program Loop
' Display a value in foreground while interrupt works in the background
'
Main:
  HighCounter = 0
  ForeGroundCounter = 0
'
' Configure Timer0 for:
'                       Clear TMR0L and TMR0H registers
'                       Interrupt on Timer0 overflow
'                       16-bit operation
'                       Internal clock source
'                       1:128 Prescaler
'
  OpenTimer0(TIMER_INT_ON & T0_16BIT & T0_SOURCE_INT & T0_PS_1_128)

  INTCONbits_GIE = 1            ' Enable global interrupts

  While 1 = 1                   ' Create an infinite loop
    INTCONbits_T0IE = 0         ' Disable the interrupt while transmitting
    ' Display the  value on the serial terminal
    Hrsout "ForeGround ", Dec ForeGroundCounter, 13
    INTCONbits_T0IE = 1         ' Re-Enable the interrupt
    Inc ForeGroundCounter       ' Increment the value
    DelayMs 200
  Wend                          ' Close the loop. i.e. do it forever
```

See also : **On_Low_Interrupt, Software Interrupts in BASIC.**

On_Low_Interrupt

Syntax
On_Low_Interrupt Label

Overview
Jump to a subroutine when a Low Priority Hardware interrupt occurs.

Operators
- ***Label*** is a valid identifier

Example
```
' Use Timer1 and Timer3 to demonstrate the use of interrupt priority.
' Timer1 is configured for high-priority interrupts
' Timer3 is configured for low-priority interrupts.
' By writing to the PortB LEDS, it is shown that a high-priority interrupts
' override low-priority interrupts.
'
' Connect three LEDs to PortB pins 0, 2, and 3
' LEDs 0 and 3 flash in the background using interrupts,
' while the LED connected to PortB.1 flashes slowly in the foreground

   Include "Timers.inc"              ' Load the Timer macros into the program

' Create a Word variable from two hardware registers
   Dim wTimer1 As TMR1L.Word
' Create a Word variable from two hardware registers
   Dim wTimer3 As TMR3L.Word

' Declare interrupt Vectors
' Point to the High priority interrupt subroutine
    On_Hardware_Interrupt GoTo Timer1_ISR
' Point to the Low priority interrupt subroutine
    On_Low_Interrupt GoTo Timer3_ISR

    GoTo Main                        ' Jump over the interrupt subroutines

'-------------------------------------------------------
' High Priority Interrupt on Timer1 overflow
Timer1_ISR:
   Clear PIR1bits_TMR1IF            ' Clear the Timer1 interrupt flag
' Turn off PortB.0 indicating high priority ISR has overridden low priority
   Clear PortB.0
   Set PortB.2              ' Turn on PortB.2 indicating high priority interrupt
' Poll Timer1 interrupt flag to wait for another Timer1 overflow
   While PIR1bits_TMR1IF = 0 : Wend
   Clear PIR1bits_TMR1IF            ' Clear the Timer1 interrupt flag again
   Clear PortB.2            ' Turn off PortB.2 indicating high-priority ISR is over
   Retfie fast              ' Exit interrupt, restoreing WREG, STATUS, and BSR
'-------------------------------------------------------
' Low Priority Interrupt on Timer3 overflow
Timer3_ISR:
   Context Save                  ' Save the contents of WREG, STATUS, and BSR
   Clear PIR2bits_TMR3IF         ' Clear the Timer3 interrupt flag
   wTimer3 = $F000               ' Load Timer3 with the value $F000
   Set PortB.0           ' Turn on PortB.0 indicating low priority ISR is occurring
```



```
' Poll Timer3 interrupt flag to wait for another Timer3 overflow
  While PIR2bits_TMR3IF = 0 : Wend
  wTimer3 = $F000                 ' Load Timer3 with the value $F000 again
  Clear PIR2bits_TMR3IF           ' Clear the Timer3 interrupt flag again
  Clear PortB.0             ' Turn off PortB.0. indicating low-priority ISR is over
' Restore the contents of WREG, STATUS, and BSR, then exit interrupt
  Context Restore
'-------------------------------------------------------
' Main Program Starts Here
Main:
  Low PortB                 ' Setup PortB for outputs
'
' Configure Timer1 for:
'                         Clear TMR1L and TMR1H registers
'                         Interrupt on Timer1 overflow
'                         16-bit read/write mode
'                         Internal clock source
'                         1:8 Prescaler
'
  OpenTimer1(TIMER_INT_ON & T1_16BIT_RW & T1_SOURCE_INT & T1_PS_1_8)
'
' Configure Timer3 for:
'                         Clear TMR3L and TMR3H registers
'                         Interrupt on Timer3 overflow
'                         16-bit read/write mode
'                         Internal clock source
'                         1:8 Prescaler
'
  OpenTimer3(TIMER_INT_ON & T3_16BIT_RW & T3_SOURCE_INT & T3_PS_1_8)

  wTimer3 = $F000           ' Write $F000 to Timer3
'
' Set up priority interrupts.
'
  RCONbits_IPEN = 1         ' Enable priority interrupts
  IPR1bits_TMR1IP = 1       ' Set Timer1 as a high priority interrupt
  IPR2bits_TMR3IP = 0       ' Set Timer3 as a low priority interrupt

  INTCONbits_PEIE = 1       ' Enable peripheral interrupts
  INTCONbits_GIE = 1        ' Enable global interrupts

  While 1 = 1               ' Flash the LED on PortB.3
    High PortB.3
    DelayMS 300
    Low PortB.3
    DelayMS 300
  Wend
```

Typical format of the interrupt handler.
The interrupt handler subroutine must follow a fixed pattern. First, the contents of the STATUS, BSR and WREG registers must be saved. Note that this is *NOT* done automatically when using low priority interrupts.

Upon exiting the interrupt, a simple **Retfie** (Return From Interrupt) mnemonic can be used, or the contexts of FSR0L, FSR0H, FSR1L, FSR1H, FSR2L, and FSR2H can be saved by issuing the **Context Save** and **Context Restore** commands.

The code within the interrupt handler should be as quick and efficient as possible because, while it's processing the code the main program is halted. When using assembler interrupts, care should be taken to ensure that the watchdog timer does not time-out. Placing a **ClrWdt** instruction at regular intervals within the code will prevent this from happening. An alternative approach would be to disable the watchdog timer altogether at programming time. Note that this is the default state of the Amicus18's microcontroller.

An interrupt, by it's very nature, may happen at any time during the operation of the foreground program, therefore it is important that the interrupt subroutine has as little impact on the program as possible.

Context Save

Issuing the **Context Save** directive will save the WREG, BSR, STATUS registers, as well as the FSR0L, FSR0H, FSR1L, and FSR1H register pairs. If strings or a stack are used within the program, the FSR2L, and FSR2H register pair will also be saved.

The **Context Save** directive should be placed at the very start of the interrupt subroutine, *before* any other command.

Context Restore

Issuing the **Context Restore** directive will restore the WREG, BSR, STATUS registers, as well as the FSR0L, FSR0H, FSR1L, and FSR1H register pairs. If strings or a stack are used within the program, the FSR2L, and FSR2H register pair will also be restored. The interrupt will then be terminated with the **Retfie** mnemonic

The **Context Restore** directive should be placed at the very end of the interrupt subroutine, *after* any other command.

Note that **Context Save** and **Context Restore** cannot be used in both high and low priority interrupt routines together.

Managed Low-Priority Hardware Interrupts.

Because an interrupt can occur at any time in the program, the code within the interrupt handler must be carefully crafted in order not to alter the contents of critical SFRs (Special Function Registers) and the compiler's System variables.

However, this process can be handled automatically, to a certain extent, by the compiler by wrapping the interrupt handler with the directives: **Low_Int_Sub_Start** and **Low_Int_Sub_End**.

When the compiler sees these directives it automatically saves the context of the compiler's system variables used within the interrupt and also saves the SFRs:

WREG, STATUS, BSR, FSR0L\FSR0H, FSR1L\FSR1H, FSR2L\ FSR2H, PRODL\PRODH, TBLPTRL\TBLPTRH, and TABLAT.

The context variables and registers are saved in a section reserved at the top of RAM. This does come at a price of code and RAM size as well as a little speed loss when the interrupt is entered and exited, but the benefits can outway the penalties.

This method allows high level commands to be placed within the interrupt handler. As an example, the below program counts and displays 2 values on an alphanumeric LCD. One count is in the interrupt and one in the main program.

```
' Demonstrate the use of context saving of the compiler's System variables
' While inside low and high priority interrupt subroutines
'
' Creates low and high priority interrupts incrementing on TIMER0 and TIMER1
' Within the interrupts a value is displayed and incremented
' In the foreground another value is incremented  and displayed serially
'
' Note: It is not recommended to use large (slow) routines in an interrupt,
' but this program serves to demonstrate
' the use of the directives Low_Int_Sub_Start and Low_Int_Sub_End

  Include "Timers.inc"          ' Load the Timer macros into the program

' Point the High Priority interrupt handler to the subroutine
  On_Hardware_Interrupt GoTo HighInterruptsub
' Point the Low Priority interrupt handler to the subroutine
  On_Low_Interrupt GoTo LowInterruptsub
'
' Create some variables
'
  Dim PortB_HighSave as Byte      ' Space for PortB save in the high interrupt
  Dim PortB_LowSave as Byte       ' Space for PortB save in the low interrupt
  Dim HighCounter as Dword        ' Counter for the high interrupt routine
  Dim LowCounter as Dword         ' Counter for the low interrupt routine
  Dim ForeGroundCounter as Dword ' Counter for the Foreground routine

  GoTo Main                       ' Jump over the interrupt handler subroutines
```



```
'-----------------------------------------------------------------------
' High Priority Hardware Interrupt Handler
' Interrupt's on a Timer1 Overflow Display on the LCD and increment a value
' Indicate to context save the System Variables for High Priority interrupt
  High_Int_Sub_Start
HighInterruptsub:
' Save the compiler's system variables used in the interrupt routine only
' Also save some SFR's PRODL\H, FSR0L\H, FSR1L\H, FSR2L\H, TBLPTRL\H, TABLAT
' SFR's WREG, STATUS, and BSR are automatically saved by shadow registers
  Context Save
' Save the condition of PortB register. Because the Print command uses it
  PortB_HighSave = PortB
' Display the value on line 1 of the LCD
  Print at 1,1, "High Int ", Dec HighCounter
  Inc HighCounter                       ' Increment the value
' Restore the condition of PortB before exiting the interrupt
  PortB = PortB_HighSave
  PIR1bits_TMR1IF = 0                   ' Clear the Timer1 Overflow flag
' Restore compiler's system variables used within the interrupt routine only
' and exit the interrupt with "Retfie 1"
' Also restore SFR's PRODL\H, FSR0L\H, FSR1L\H, FSR2L\H, TBLPTRL\H, TABLAT
' SFR's WREG, STATUS, and BSR are automatically restored by shadow registers
  Context Restore
' Indicate that the High Priority Interrupt block has ended
  High_Int_Sub_End
'-----------------------------------------------------------------------
' Low Priority Hardware Interrupt Handler
' Interrupt's on a TIMER0 Overflow
' Display on the LCD and increment a floating point value
' Indicate to context save the System Variables for a Low Priority interrupt
  Low_Int_Sub_Start
LowInterruptsub:
' Save the compiler's system variables used in the interrupt routine only
' Also save some important SFR's. i.e. WREG, STATUS, BSR, PRODL\H, FSR0L\H,
' FSR1L\H, FSR2L\H, TBLPTRL\H, TABLAT
'
  Context Save
' Disable the Timer 1 High priority interrupt while we use the LCD
  PIE1bits_TMR1IE = 0
' Save the condition of PortB register. Because the Print command uses it
  PortB_LowSave = PortB
' Display the value on line 2 of the LCD
  Print at 2, 1, "Low Int  " , Dec LowCounter, "  "
  Inc LowCounter   ' Increment the  value
' Restore the condition of PortB before exiting the interrupt
  PortB = PortB_LowSave
  PIE1bits_TMR1IE = 1       ' Re-Enable the Timer 1 High priority interrupt
  INTCONbits_TMR0IF = 0     ' Clear the TIMER0 Overflow flag
'
' Restore the compiler's system variables used in the interrupt routine only
' and exit the interrupt ' with "Retfie"
' Also restore the important SFR's. i.e. WREG, STATUS, BSR, PRODL\H,
' FSR0L\H, FSR1L\H, FSR2L\H, TBLPTRL\H, TABLAT
  Context Restore
  Low_Int_Sub_End       ' Indicate Low Priority Interrupt block has ended
```



```
'-----------------------------------------------------------------------
' The Main Program Loop Starts Here
'
Main:
   DelayMs 100                  ' Wait for things to stabilise
   Low PortB                    ' Set PortB to Output Low
   HighCounter = 0
   LowCounter = 0
   ForeGroundCounter = 0
   Cls                          ' Clear the LCD
'
' Configure Timer0 for:
'                        Clear TMR0L and TMR0H registers
'                        Interrupt on Timer0 overflow
'                        16-bit operation
'                        Internal clock source
'                        1:128 Prescaler
'
   OpenTimer0(TIMER_INT_ON & T0_16BIT & T0_SOURCE_INT & T0_PS_1_128)
'
' Configure Timer1 for:
'                         Clear TMR1L and TMR1H registers
'                         Interrupt on Timer1 overflow
'                         16-bit read/write mode
'                         Internal clock source
'                         1:8 Prescaler
'
   OpenTimer1(TIMER_INT_ON & T1_16BIT_RW & T1_SOURCE_INT & T1_PS_1_8)
'
' Setup the High and Low priorities for the interrupts
'
   INTCON2bits_TMR0IP = 0       ' Timer0 Interrupt to Low priority
   IPR1bits_TMR1IP = 1          ' Timer1 Interrupt to High priority
   RCONbits_IPEN = 1            ' Enable priority levels on interrupts
   INTCONbits_GIEL = 1          ' Enable low priority peripheral interrupts
   INTCONbits_GIE = 1           ' Enable global interrupts
'
' Display value in forground while interrupts do their thing in background
'
   While 1 = 1                  ' Create an infinite loop
      ' Display the  value on serial terminal
     HRsout  "Foreground ", Dec ForeGroundCounter, 13
     Inc ForeGroundCounter      ' Increment the value
     DelayMs 200
   Wend                         ' Close the loop. i.e. do it forever
```

See also : **On_Hardware_Interrupt, Software Interrupts in BASIC.**

Output

Syntax
Output Port or Port . Pin

Overview
Makes the specified Port or Port.Pin an output.

Operator
- ***Port or Port.Pin*** must be a Port or Port.Pin constant declaration.

Example
```
Output PortA.0          ' Make bit-0 of PortA an output
Output RB               ' Make all of PortB an output
```

Notes
An Alternative method for making a particular pin an output is by directly modifying the Tris:

```
TrisB.0 = 0             ' Set PortB, bit-0 to an output
```

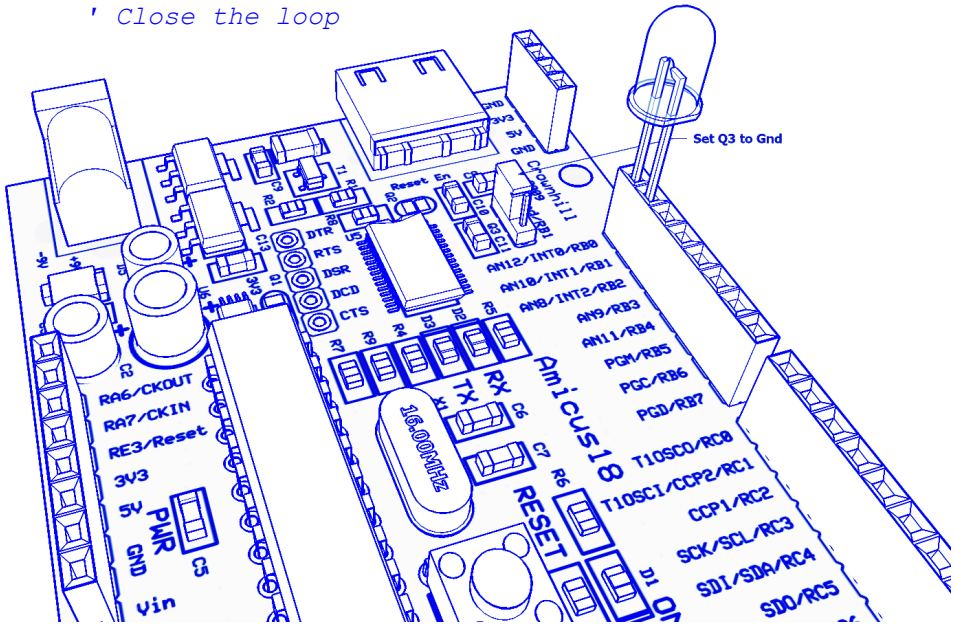
All of the pins on a port may be set to output by setting the whole Tris register at once:

```
TrisB = %00000000       ' Set all of PortB to outputs
```

In the above examples, setting a Tris bit to 0 makes the pin an output, and conversely, setting the bit to 1 makes the pin an input.

The example below will flash an LED connected to RB0 (PortB.0) every second:

```
' Flash an LED every second
Output RB0              ' Make pin RB0 (PortB.0) an output
While 1 = 1             ' Create an infinite loop
   Set RB0              ' Bring RB0 high
   DelayMS 500          ' Wait 500ms
   Clear RB0            ' Pull RB0 low
   DelayMS 500          ' Wait 500ms
Wend                    ' Close the loop
```



See also **Input, Clear, High, Low, Set.**

Org

Syntax
Org Value

Overview
Set the program origin for subsequent code at the address defined in Value

Operator
- ***Value*** can be any constant value within the range of the particular microcontroller's memory.

Example
```
Org 2000                    ' Set the origin to address 2000
Cdata 120, 243, "Hello"     ' Place data starting at address 2000
```

or

```
Symbol Address = 2000

Org Address + 1             ' Set the origin to address 2001
Cdata 120, 243, "Hello"     ' Place data starting at address 2001
```


Oread

Syntax
Oread Pin, Mode, [Inputdata]

Overview
Receive data from a device using the Dallas Semiconductor 1-wire protocol. The 1-wire protocol is a form of asynchronous serial communication developed by Dallas Semiconductor. It requires only one I/O pin which may be shared between multiple 1-wire devices.

Operators
- ***Pin*** is a Port-Bit combination that specifies which I/O pin to use. 1-wire devices require only one I/O pin (normally called DQ) to communicate. This I/O pin will be toggled between output and input mode during the **Oread** command and will be set to input mode by the end of the **Oread** command.
- ***Mode*** is a numeric constant (0 - 7) indicating the mode of data transfer. The Mode argument control's the placement of Reset pulses and detection of presence pulses, as well **as Byte** or bit input. See notes below.
- ***Inputdata*** is a list of variables or arrays to store the incoming data into.

Example
```
Dim Result as Byte
Symbol DQ = PortA.0
Oread DQ, 1, [Result]
```

The above example code will transmit a 'reset' pulse to a 1-wire device (connected to bit 0 of PortA) and will then detect the device's 'presence' pulse and receive one byte and store it in the variable Result.

Notes
The Mode operator is used to control placement of Reset pulses (and detection of presence pulses) and to designate byte or bit input. The table below shows the meaning of each of the 8 possible value combinations for Mode.

Mode Value	Effect
0	No Reset, Byte mode
1	Reset before data, Byte mode
2	Reset after data, Byte mode
3	Reset before and after data, Byte mode
4	No Reset, Bit mode
5	Reset before data, Bit mode
6	Reset after data, Bit mode
7	Reset before and after data, Bit mode

The correct value for Mode depends on the 1-wire device and the portion of the communication that is being dealt with. Consult the data sheet for the device in question to determine the correct value for Mode. In many cases, however, when using the **Oread** command, Mode should be set for either No Reset (to receive data from a transaction already started by an **Owrite**

command) or a Reset after data (to terminate the session after data is received). However, this may vary due to device and application requirements.

When using the Bit (rather than Byte) mode of data transfer, all variables in the InputData argument will only receive one bit. For example, the following code could be used to receive two bits using this mode:

```
Dim BitVar1 as Bit
Dim BitVar2 as Bit
Oread PortA.0, 6, [BitVar1, BitVar2]
```

In the example code shown, a value of 6 was chosen for Mode. This sets Bit transfer and Reset after data mode.

We could also have chosen to make the BitVar1 and BitVar2 variables each a Byte type, however, they would still only have received one bit each in the **Oread** command, due to the Mode that was chosen.

The compiler also has a modifier for handling a string of data, named **Str**.

The **Str** modifier is used for receiving data and placing it directly into a byte array variable.

A string is a set of bytes that are arranged or accessed in a certain order. The values 1, 2, 3 would be stored in a string with the value 1 first, followed by 2 then followed by the value 3. A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string 1 2 3 would be stored in a byte array containing three bytes (elements).

Below is an example that receives ten bytes through a 1-wire interface and stores them in the 10-byte array, MyArray:

```
Dim MyArray[10] as Byte          ' Create a 10-byte array.
Oread DQ, 1, [Str MyArray]
Print Dec Str MyArray            ' Display the values.
```

If the amount of received characters is not enough to fill the entire array, then a formatter may be placed after the array's name, which will only receive characters until the specified length is reached. For example:

```
Dim MyArray[10] as Byte          ' Create a 10-byte array.
Oread DQ, 1, [Str MyArray\5]     ' Fill the first 5-bytes of array with data.
Print Str MyArray\5              ' Display the 5-value string.
```

The example above illustrates how to fill only the first n bytes of an array, and then how to display only the first n bytes of the array. n refers to the value placed after the backslash.

Dallas 1-Wire Protocol.
The 1-wire protocol has a well defined standard for transaction sequences. Every transaction sequence consists of four parts:

- Initialisation.
- ROM Function Command.
- Memory Function Command.
- Transaction / Data.

Additionally, the ROM Function Command and Memory Function Command are always 8 bits wide and are sent least-significant-bit first (LSB).

The Initialisation consists of a Reset pulse (generated by the master) that is followed by a presence pulse (generated by all slave devices).

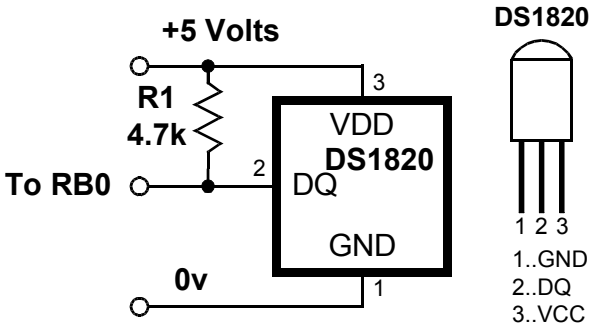
The Reset pulse is controlled by the lowest two bits of the Mode argument in the **Oread** command. It can be made to appear before the ROM Function Command (Mode = 1), after the Transaction / Data portion (Mode = 2), before and after the entire transaction (Mode = 3) or not at all (Mode = 0).

Command	Value	Action
Read ROM	$33	Reads the 64-bit ID of the 1-wire device. This command can only be used if there is a single 1-wire device on the line.
Match ROM	$55	This command, followed by a 64-bit ID, allows the PICmicro to address a specific 1-wire device.
Skip ROM	$CC	Address a 1-wire device without its 64-bit ID. This command can only be used if there is a single 1-wire device on the line.
Search ROM	$F0	Reads the 64-bit IDs of all the 1-wire devices on the line. A process of elimination is used to distinguish each unique device.

Following the Initialisation, comes the ROM Function Command. The ROM Function Command is used to address the desired 1-wire device. The above table shows a few common ROM Function Commands. If only a single 1 wire device is connected, the Match ROM command can be used to address it. If more than one 1-wire device is attached, the microcontroller will ultimately have to address them individually using the Match ROM command.

The third part, the Memory Function Command, allows the microcontroller to address specific memory locations, or features, of the 1-wire device. Refer to the 1-wire device's data sheet for a list of the available Memory Function Commands.

Finally, the Transaction / Data section is used to read or write data to the 1-wire device. The **Oread** command will read data at this point in the transaction. A read is accomplished by generating a brief low-pulse and sampling the line within 15us of the falling edge of the pulse. This is called a 'Read Slot'.



The following program demonstrates interfacing to a Dallas Semiconductor DS1820 1-wire digital thermometer device using the compiler's 1-wire commands, and connections as per the diagram to the right.

The code reads the Counts Remaining and Counts per Degree Centigrade registers within the DS1820 device in order to provide a more accurate temperature (down to 1/10th of a degree).

```
   Symbol DQ = RB0                ' Place the DS1820 on bit-0 of PortB
   Dim Temp as Word               ' Holds the temperature value
   Dim C as Byte                  ' Holds the counts remaining value
   Dim CPerD as Byte              ' Holds the Counts per degree C value
Again:
   Owrite DQ, 1, [$CC, $44]       ' Send Calculate Temperature command
   Repeat
      DelayMs 25                  ' Wait until conversion is complete
      Oread DQ, 4, [C]            ' Keep reading low pulses until
   Until C <> 0                   ' the DS1820 is finished.
   Owrite DQ, 1, [$CC, $BE]       ' Send Read ScratchPad command
   Oread DQ, 2,[Temp.LowByte,Temp.HighByte, C, C, C, C, C, CPerD]
' Calculate the temperature in degrees Centigrade
   Temp = (((Temp >> 1) * 100) - 25) + (((CPerD - C) * 100) / CPerD)
   Hrsout Dec Temp / 100, ".", Dec2 Temp, " Degrees Centigrade\r"
   GoTo Again
```

Note.
The equation used in the program above will not work correctly with negative temperatures. Also note that the 4.7kΩ pull-up resistor (R1) is required for correct operation.

Inline Oread Command.
The standard structure of the **Oread** command is:

```
   Oread Pin, Mode, [Inputdata]
```

However, this did not allow it to be used in conditions such as **If-Then**, **While-Wend** etc. Therefore, there is now an additional structure to the **Oread** command:

```
   Var = Oread Pin, Mode
```

Operands Pin and Mode have not changed their function, but the result from the 1-wire read is now placed directly into the assignment variable.

Oread - Owrite Presence Detection.
Another important feature to both the **Oread** and **Owrite** commands is the ability to jump to a section of the program if a presence is not detected on the 1-wire bus.

```
Owrite Pin, Mode, Label, [Outputdata]

Oread Pin, Mode, Label, [Inputdata]

Var = Oread Pin, Mode, Label
```

The Label parameter is an optional condition, but if used, it must reference a valid BASIC label.

```
' Skip ROM search and do temp conversion
   Owrite  DQ, 1, NoPresence, [$CC, $44]
   While Oread DQ, 4, NoPresence <> 0 : Wend ' Read busy-bit, Still busy..?
' Skip ROM search and read scratchpad memory
   Owrite  DQ, 1, NoPresence, [$CC, $BE]
   Oread  DQ, 2, NoPresence, [Temp.Lowbyte, Temp.Highbyte] ' Read two bytes
   Return

NoPresence:
   Print "No Presence"
   Stop
```

See also : **Owrite**.

Owrite

Syntax
Owrite Pin, Mode, [Outputdata]

Overview
Send data to a device using the Dallas Semiconductor 1-wire protocol. The 1-wire protocol is a form of asynchronous serial communication developed by Dallas Semiconductor. It requires only one I/O pin which may be shared between multiple 1-wire d vices.

Operators
- ***Pin*** is a Port-Bit combination that specifies which I/O pin to use. 1-wire devices require only one I/O pin (normally called DQ) to communicate. This I/O pin will be toggled between output and input mode during the **Owrite** command and will be set to input mode by the end of the **Owrite** command.
- ***Mode*** is a numeric constant (0 - 7) indicating the mode of data transfer. The Mode operator control's the placement of Reset pulses and detection of presence pulses, as well **as Byte** or bit input. See notes below.
- ***Outputdata*** is a list of variables or arrays transmit individual or repeating bytes.

Example
```
Symbol DQ = PortA.0
Owrite DQ, 1, [$4E]
```

The above example will transmit a 'reset' pulse to a 1-wire device (connected to bit 0 of PortA) and will then detect the device's 'presence' pulse and transmit one byte (the value $4E).

Notes
The Mode operator is used to control placement of Reset pulses (and detection of presence pulses) and to designate byte or bit input. The table below shows the meaning of each of the 8 possible value combinations for Mode.

Mode Value	Effect
0	No Reset, Byte mode
1	Reset before data, Byte mode
2	Reset after data, Byte mode
3	Reset before and after data, Byte mode
4	No Reset, Bit mode
5	Reset before data, Bit mode
6	Reset after data, Bit mode
7	Reset before and after data, Bit mode

The correct value for Mode depends on the 1-wire device and the portion of the communication you're dealing with. Consult the data sheet for the device in question to determine the correct value for Mode. In many cases, however, when using the **Owrite** command, Mode should be set for a Reset before data (to initialise the transaction). However, this may vary due to device and application requirements.

When using the Bit (rather than Byte) mode of data transfer, all variables in the InputData argument will only receive one bit. For example, the following code could be used to receive two bits using this mode:

```
Dim BitVar1 as Bit
Dim BitVar2 as Bit
Owrite PortA.0, 6, [BitVar1, BitVar2]
```

In the example code shown, a value of 6 was chosen for Mode. This sets Bit transfer and Reset after data mode. We could also have chosen to make the BitVar1 and BitVar2 variables each a Byte type, however, they would still only use their lowest bit (Bit0) as the value to transmit in the **Owrite** command, due to the Mode value chosen.

The Str Modifier

The **Str** modifier is used for transmitting a string of bytes from a byte array variable. A string is a set of bytes sized values that are arranged or accessed in a certain order. The values 1, 2, 3 would be stored in a string with the value 1 first, followed by 2 then followed by the value 3. A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string 1,2,3 would be stored in a byte array containing three bytes (elements).

Below is an example that sends four bytes (from a byte array) through bit-0 of PortA:

```
Dim MyArray[10] as Byte              ' Create a 10-byte array.
MyArray[0] = $CC                     ' Load the first 4 bytes of the array
MyArray[1] = $44                     ' With the data to send
MyArray[2] = $CC
MyArray[3] = $4E
Owrite PortA.0, 1, [Str MyArray\4]   ' Send 4-byte string.
```

Note that we use the optional \n argument of **Str**. If we didn't specify this, the microcontroller would try to keep sending characters until all 10 bytes of the array were transmitted. Since we do not wish all 10 bytes to be transmitted, we chose to tell it explicitly to only send the first 4 bytes.

The above example may also be written as:

```
Dim MyArray[10] as Byte              ' Create a 10-byte array.
Str MyArray = $CC,$44,$CC,$4E        ' Load the first 4 bytes of the array
Owrite PortA.0, 1, [Str MyArray\4]   ' Send 4-byte string.
```

The above example, has exactly the same function as the previous one. The only difference is that the string is now constructed using the **Str** as a command instead of a modifier.

See also : **Oread for example code, and 1-wire protocol.**

Pixel

Syntax
Variable = **Pixel** Ypos, Xpos

Overview
Read the condition of an individual pixel from a graphic LCD. The returned value will be 1 if the pixel is set, and 0 if the pixel is clear.

Operators
- ***Variable*** is a user defined variable.
- ***Xpos*** can be a constant, variable, or expression, pointing to the X-axis location of the pixel to examine. This must be a value of 0 to the X resolution of the LCD. Where 0 is the far left row of pixels.
- ***Ypos*** can be a constant, variable, or expression, pointing to the Y-axis location of the pixel to examine. This must be a value of 0 to the Y resolution of the LCD. Where 0 is the top column of pixels.

Example
```
' Read a line of pixels from a Samsung KS0108 graphic LCD
  Declare LCD_Type = Graphic      ' Use a Graphic LCD
  Declare Internal_Font = Off     ' Use an external chr set
  Declare Font_Addr = 0           ' Eeprom's address is 0

' Graphic LCD Pin Assignments
  Declare LCD_DTPort = PortB
  Declare LCD_RSPin = PortA.2
  Declare LCD_RWPin = PortA.0
  Declare LCD_ENPin = PortA.5
  Declare LCD_CS1Pin = PortC.0
  Declare LCD_CS2Pin = PortC.2

' Character set eeprom Pin Assignments
  Declare SDA_Pin = PortC.4
  Declare SCL_Pin = PortC.3

  Dim Xpos as Byte
  Dim Ypos as Byte
  Dim Result as Byte

  Cls
  Print At 0, 0, "Testing 1-2-3"
' Read the top row and display the result
  For Xpos = 0 to 127
    Result = Pixel 0, Xpos        ' Read the top row
    Print At 1, 0, Dec Result
    DelayMs 400
  Next
  Stop
```

See also : **LCDread, LCDwrite, Plot, UnPlot. See Print for circuit.**

Plot

Syntax
Plot Ypos, Xpos

Overview
Set an individual pixel on a graphic LCD.

Operators
- ***Xpos*** can be a constant, variable, or expression, pointing to the X-axis location of the pixel to set. This must be a value of 0 to the X resolution of the LCD. Where 0 is the far left row of pixels.
- ***Ypos*** can be a constant, variable, or expression, pointing to the Y-axis location of the pixel to set. This must be a value of 0 to the Y resolution of the LCD. Where 0 is the top column of pixels.

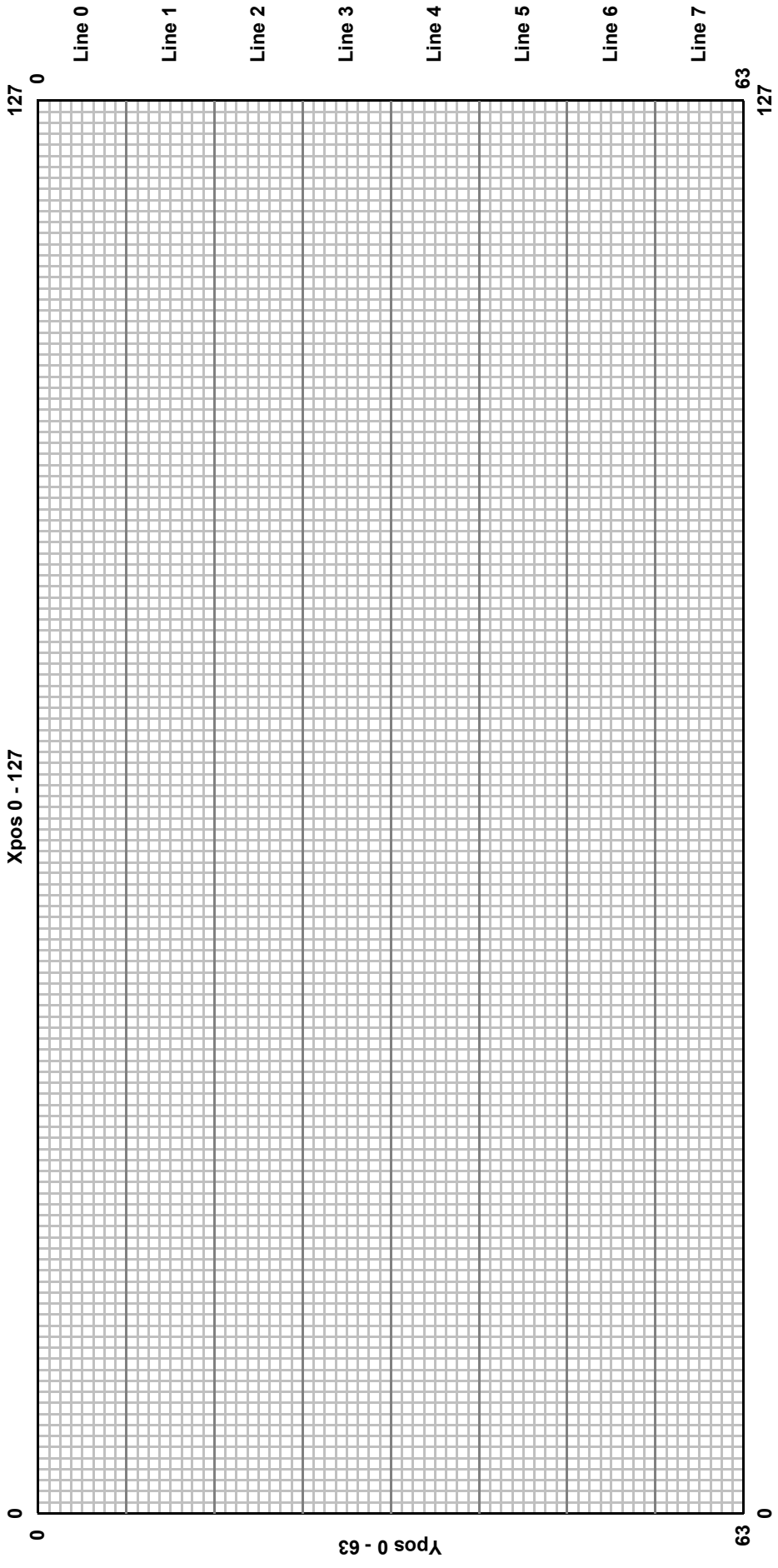
Example
```
  Declare LCD_Type = Graphic      ' Use a Samsung Graphic LCD

' Graphic LCD Pin Assignments
  Declare LCD_DTPort = PortB
  Declare LCD_RSPin = PortA.2
  Declare LCD_RWPin = PortA.0
  Declare LCD_ENPin = PortC.5
  Declare LCD_CS1Pin = PortC.0
  Declare LCD_CS2Pin = PortC.2

  Dim Xpos as Byte
' Draw a line across the LCD
  While 1 = 1                     ' Create an infinite loop
  For Xpos = 0 to 127
    Plot 20, Xpos
    DelayMs 10
  Next
' Now erase the line
  For Xpos = 0 to 127
    UnPlot 20, Xpos
    DelayMs 10
  Next
  Wend
```

See also : **LCDread, LCDwrite, Pixel, UnPlot. See Print for circuit.**

Graphic LCD pixel configuration for a 128x64 resolution display.



Pop

Syntax
Pop Variable, {Variable, Variable etc}

Overview
Pull a single variable or multiple variables from a software stack.
If the **Pop** command is issued without a following variable, it will implement the assembler mnemonic **Pop**, which manipulates the microcontroller's call stack.

Operator
- ***Variable*** is a user defined variable of type Bit, Byte, Byte Array, Word, Word Array, Dword, Float, or String.

The amount of bytes pushed on to the stack varies with the variable type used. The list below shows how many bytes are pushed for a particular variable type, and their order.

- Bit — 1 Byte is popped containing the value of the bit pushed.
- Byte — 1 Byte is popped containing the value of the byte pushed.
- Byte Array — 1 Byte is popped containing the value of the byte pushed.
- Word — 2 Bytes are popped. Low Byte then High Byte containing the value of the word pushed.
- Word Array — 2 Bytes are popped. Low Byte then High Byte containing the value of the word pushed.
- Dword — 4 Bytes are popped. Low Byte, Mid1 Byte, Mid2 Byte then High Byte containing the value of the dword pushed.
- Float — 4 Bytes are popped. Low Byte, Mid1 Byte, Mid2 Byte then High Byte containing the value of the float pushed.
- String — 2 Bytes are popped. Low Byte then High Byte that point to the start address of the string previously pushed.

Example 1
```
' Push two variables on to the stack then retrieve them
  Declare Stack_Size = 20    ' Create a stack capable of holding 20 bytes

  Dim WordVar as Word        ' Create a Word variable
  Dim DwordVar as Dword      ' Create a Dword variable

  WordVar = 1234             ' Load the Word variable with a value
  DwordVar = 567890          ' Load the Dword variable with a value
  Push WordVar, DwordVar     ' Push the Word variable then the Dword variable

  Clear WordVar              ' Clear the Word variable
  Clear DwordVar             ' Clear the Dword variable

  Pop DwordVar, WordVar      ' Pop the Dword variable then the Word variable
  Hrsout Dec WordVar, "  ", Dec DwordVar, 13    ' Display the variables
  Stop
```


Example 2

```
' Push a String on to the stack then retrieve it
  Declare Stack_Size = 10           ' Create a stack capable of holding 10 bytes

  Dim SourceString as String * 20   ' Create a String variable
  Dim DestString as String * 20     ' Create another String variable

  SourceString = "HELLO WORLD"    ' Load the String variable with characters

  Push SourceString                 ' Push the String variable's address

  Pop DestString                    ' Pop the previously pushed String into DestString
  Hrsout DestString, 13             ' Display the string, which will be "HELLO WORLD"
  Stop
```

Example 3

```
' Push a Quoted character string on to the stack then retrieve it
  Declare Stack_Size = 10         ' Create a stack capable of holding 10 bytes

  Dim DestString as String * 20   ' Create a String variable

  Push "HELLO WORLD"           ' Push the Quoted String of Characters on to the stack

  Pop DestString               ' Pop the previously pushed String into DestString
  Hrsout DestString, 13        ' Display the string, which will be "HELLO WORLD"
  Stop
```

See also : **Push, GoSub, Return, See Push for technical details of stack manipulation.**

Pot

Syntax
Variable = **Pot** Pin, Scale

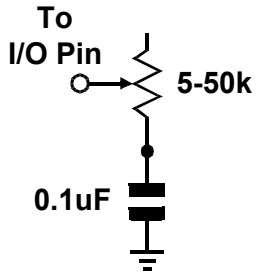
Overview
Read a potentiometer, thermistor, photocell, or other variable resistance.

Operators
- ***Variable*** is a user defined variable.
- ***Pin*** is a Port.Pin constant that specifies the I/O pin to use.
- ***Scale*** is a constant, variable, or expression, used to scale the instruction's internal 16-bit result. The 16- bit reading is multiplied by (scale/ 256), so a scale value of 128 would reduce the range by approximately 50%, a scale of 64 would reduce to 25%, and so on.

Example
```
   Dim Var1 as Byte
Loop:
   Var1 = Pot PortB.0, 100     ' Read potentiometer on pin-0 of PortB.
   Hrsout Dec Var1, 13         ' Transmit the potentiometer reading
   DelayMs 500                 ' Wait to 500ms
   GoTo Loop                   ' Repeat the process.
```

Notes
Internally, the **Pot** instruction calculates a 16-bit value, which is scaled down to an 8-bit value. The amount by which the internal value must be scaled varies with the size of the resistor being used.

The pin specified by **Pot** must be connected to one side of a resistor, whose other side is connected through a capacitor to ground. A resistance measurement is taken by timing how long it takes to discharge the capacitor through the resistor.



The value of scale must be determined by experimentation, however, this is easily accomplished as follows:

Set the device under measure, the pot in this instance, to maximum resistance and read it with scale set to 255. The value returned in Var1 can now be used as scale:

```
   Var1 = Pot PortB.0, 255
```

See also : **Adin**, **RCin**.

Print

Syntax
Print Item {, Item... }

Overview
Send Text to an LCD module using the Hitachi 44780 controller or a graphic LCD based on the Samsung KS0108, or Toshiba T6963 chipsets.

Operators
- ***Item*** may be a constant, variable, expression, modifier, or string.

There are no operators as such, instead there are modifiers. For example, if the word **Dec** precedes an item, the ASCII representation for each digit is sent to the LCD.

The modifiers are listed below:

Modifier	Operation
At ypos,xpos	Position the cursor on a serial LCD
Cls	Clear a serial LCD (also creates a 30ms delay)
Bin{1..32}	Send binary digits
Dec{1..10}	Send decimal digits
Hex{1..8}	Send hexadecimal digits
Sbin{1..32}	Send signed binary digits
Sdec{1..10}	Send signed decimal digits
Shex{1..8}	Send signed hexadecimal digits
Ibin{1..32}	Send binary digits with a preceding '%' identifier
Idec{1..10}	Send decimal digits with a preceding '#' identifier
Ihex{1..8}	Send hexadecimal digits with a preceding '$' identifier
ISbin{1..32}	Send signed binary digits with a preceding '%' identifier
ISdec{1..10}	Send signed decimal digits with a preceding '#' identifier
IShex{1..8}	Send signed hexadecimal digits with a preceding '$' identifier
Rep c\n	Send character c repeated n times
Str array\n	Send all or part of an array
Cstr cdata	Send string data defined in a **Cdata** statement.

The numbers after the **Bin**, **Dec**, and **Hex** modifiers are optional. If they are omitted, then the default is all the digits that make up the value will be displayed.

If a floating point variable is to be displayed, then the digits after the **Dec** modifier determine how many remainder digits are printed. i.e. numbers after the decimal point.

```
Dim FloatVar as Float
FloatVar = 3.145
Print Dec2 FloatVar     ' Display 2 values after the decimal point
```

The above program will display 3.14

If the digit after the **Dec** modifier is omitted, then 3 values will be displayed after the decimal point.


```
Dim FloatVar as Float
FloatVar = 3.1456
Print Dec FloatVar    ' Display 3 values after the decimal point
```

The above program will display 3.145

There is no need to use the **Sdec** modifier for signed floating point values, as the compiler's **Dec** modifier will automatically display a minus result:

```
Dim FloatVar as Float
FloatVar = -3.1456
Print Dec FloatVar    ' Display 3 values after the decimal point
```

The above program will display -3.145

Hex or **Bin** modifiers cannot be used with floating point values or variables.

The Xpos and Ypos values in the **At** modifier both start at 1. For example, to place the text "HELLO WORLD" on line 1, position 1, the code would be:

```
Print At 1, 1, "HELLO WORLD"
```

Example 1

```
Dim Var1 as Byte
Dim WordVar as Word
Dim DwordVar as Dword

Print "Hello World"                ' Display the text "Hello World"
Print "Var1= ", Dec Var1           ' Display the decimal value of Var1
Print "Var1= ", Hex Var1           ' Display the hexadecimal value of Var1
Print "Var1= ", Bin Var1           ' Display the binary value of Var1
Print "DwordVar= ", Hex6 DwordVar  ' Display 6 hex chars of Dword variable
```

Example 2

```
' Display a negative value on the LCD.
Symbol Negative = -200
Print At 1, 1, Sdec Negative
```

Example 3

```
' Display a negative value on the LCD with a preceding identifier.
Print At 1, 1, IShex -$1234
```

Example 3 will produce the text "$-1234" on the LCD.

The microcontroller used on the Amicus18 board has the ability to read and write to its own flash memory. And although writing to this memory too many times is unhealthy for the microcontroller, reading this memory is both fast, and harmless. Which offers a unique form of data storage and retrieval, the **Cdata** command proves this, as it uses the mechanism of reading and storing in the microcontroller's flash memory.

Combining the unique features of the 'self modifying microcontroller's' with a string format, the compiler is capable of reducing the overhead of printing, or transmitting large amounts of text data. The **Cstr** modifier may be used in commands that deal with text processing i.e. **Serout**, **HRsout**, and **RSOUT** etc.

The **Cstr** modifier is used in conjunction with the **Cdata** command. The **Cdata** command is used for initially creating the string of characters:

```
String1: Cdata "Hello World", 0
```

The above line of case will create, in flash memory, the values that make up the ASCII text "HELLO WORLD", at address String1. Note the null terminator after the ASCII text.

null terminated means that a zero (null) is placed at the end of the string of ASCII characters to signal that the string has finished.

To display this string of characters, the following command structure could be used:

```
Print Cstr String1
```

The label that declared the address where the list of **Cdata** values resided, now becomes the string's name. In a large program with lots of text formatting, this type of structure can save quite literally hundreds of bytes of valuable code space.

Try both these small programs, and you'll see that using **Cstr** saves a few bytes of code:

First the standard way of displaying text:

```
Cls
Print "Hello World"
Print "How are you?"
Print "I am fine!"
Stop
```

Now using the **Cstr** modifier:

```
Cls
Print Cstr Text1
Print Cstr Text2
Print Cstr Text3
Stop

Text1: Cdata "Hello World", 0
Text2: Cdata "How are you?", 0
Text3: Cdata "I am fine!", 0
```

Again, note the null terminators after the ASCII text in the **Cdata** commands. Without these, the microcontroller will continue to transmit data in an endless loop.

The term 'virtual string' relates to the fact that a string formed from the **Cdata** command cannot be written too, but only read from.

The **Str** modifier is used for sending a string of bytes from a byte array variable. A string is a set of bytes sized values that are arranged or accessed in a certain order. The values 1, 2, 3 would be stored in a string with the value 1 first, followed by 2 then followed by the value 3. A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string 1,2,3 would be stored in a byte array containing three bytes (elements).

Below is an example that displays four bytes (from a byte array):

```
Dim MyArray[10] as Byte    ' Create a 10-byte array.
MyArray[0] = "H"           ' Load the first 5 bytes of the array
MyArray[1] = "E"           ' With the data to send
MyArray[2] = "L"
MyArray[3] = "L"
MyArray[4] = "O"
Print Str MyArray\5        ' Display a 5-byte string.
```

Note that we use the optional \n argument of **Str**. If we didn't specify this, the microcontroller would try to keep sending characters until all 10 bytes of the array were transmitted. Since we do not wish all 10 bytes to be transmitted, we chose to tell it explicitly to only send the first 5 bytes.

The above example may also be written as:

```
Dim MyArray[10] as Byte      ' Create a 10-byte array.
Str MyArray = "HELLO"        ' Load the first 5 bytes of the array
Print Str MyArray\5          ' Send 5-byte string.
```

The above example, has exactly the same function as the previous one. The only difference is that the string is now constructed using **Str** as a command instead of a modifier.

Declares
There are several Declares for use with an alphanumeric LCD and **Print**:

Declare LCD_Type 0 or 1 or 2, Alpha or Graphic or Samsung or Toshiba
Inform the compiler as to the type of LCD that the **Print** command will output to. If Graphic, Samsung or 1 is chosen then any output by the **Print** command will be directed to a graphic LCD based on the Samsung KS0108 chipset. A value of 2, or the text Toshiba, will direct the output to a graphic LCD based on the Toshiba T6963 chipset. A value of 0 or Alpha, or if the **Declare** is not issued, will target the standard Hitachi alphanumeric LCD type

Targeting the graphic LCD will also enable commands such as **Plot**, **UnPlot**, **LCDread**, **LCDwrite**, **Pixel**, **Box**, **Circle** and **Line**.

Declare LCD_DTPin Port . Pin
Assigns the Port and Pins that the LCD's DT (data) lines will attach to.
The LCD may be connected to the microcontroller using either a 4-bit bus or an 8-bit bus. If an 8-bit bus is used, all 8 bits must be on one port. If a 4-bit bus is used, it must be connected to either the bottom 4 or top 4 bits of one port. For example:

```
Declare LCD_DTPin PortB.4    ' Used for 4-line interface.

Declare LCD_DTPin PortB.0    ' Used for 8-line interface.
```


In the previous examples, PortB is only a personal preference. The LCD's DT lines may be attached to any valid port on the microcontroller. If the **Declare** is not used in the program, then the default Port and Pin is PortB.4, which assumes a 4-line interface.

Declare LCD_ENPin Port . Pin
Assigns the Port and Pin that the LCD's EN line will attach to.

If the **Declare** is not used in the program, then the default Port and Pin is PortB.3.

Declare LCD_RSPin Port . Pin
Assigns the Port and Pins that the LCD's RS line will attach to.

If the **Declare** is not used in the program, then the default Port and Pin is PortB.2.

Declare LCD_Interface 4 or 8
Inform the compiler as to whether a 4-line or 8-line interface is required by the LCD.

If the **Declare** is not used in the program, then the default interface is a 4-line type.

Declare LCD_Lines 1, 2, or 4
Inform the compiler as to how many lines the LCD has.

LCD's come in a range of sizes, the most popular being the 2 line by 16 character types. However, there are 4 line types as well. Simply place the number of lines that the particular LCD has, into the declare.

If the **Declare** is not used in the program, then the default number of lines is 2.

Notes
If no modifier precedes an item in a **Print** command, then the character's value is sent to the LCD. This is useful for sending control codes to the LCD. For example:

```
Print $FE, 128
```

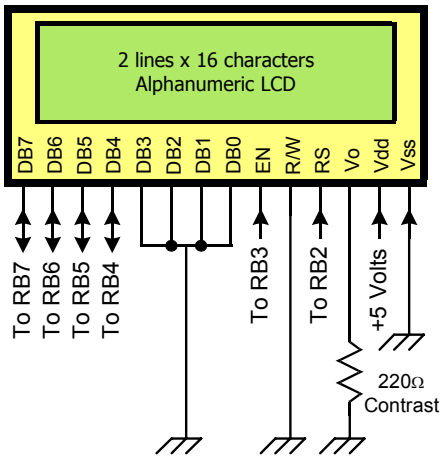
Will move the cursor to line 1, position 1 (Home).

Below is a list of some useful control commands:

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

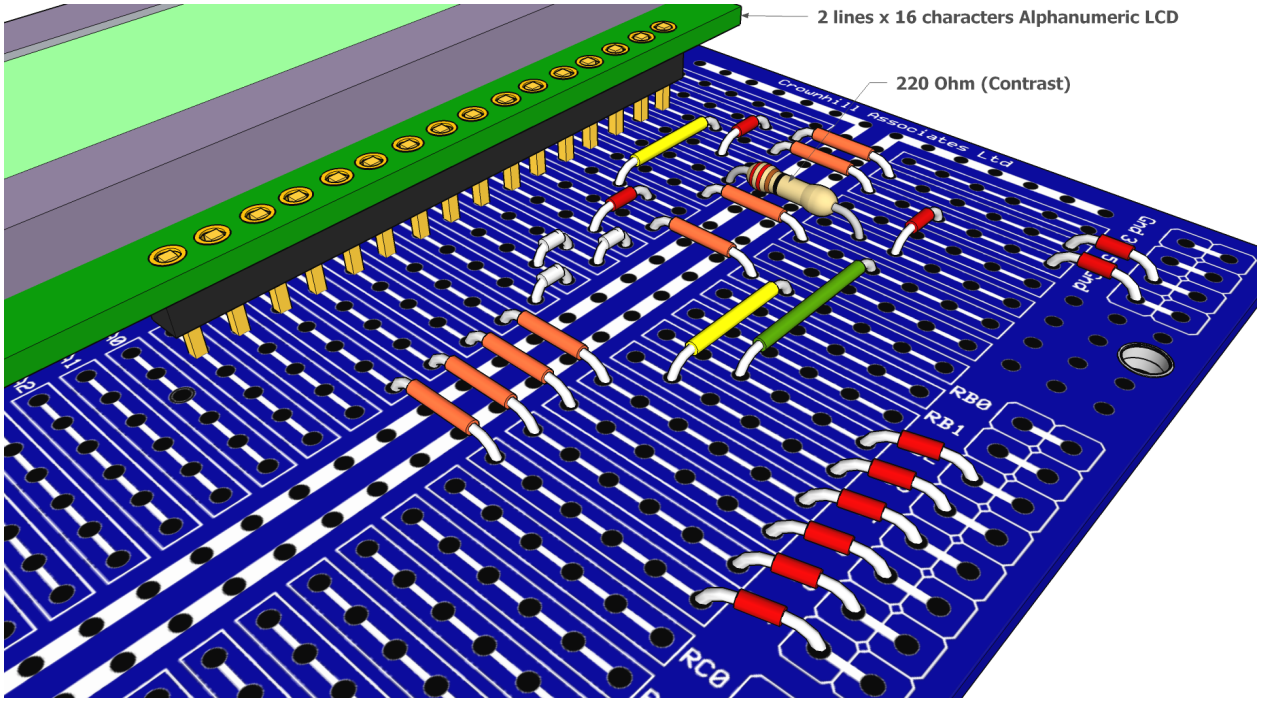
Note that if the command for clearing the LCD is used, then a small delay should follow it:

```
Print $FE, 1 : DelayMs 30
```



The above diagram shows the default connections for an alphanumeric LCD module connected to the Amicus18 board.

And below is the same circuit built on an Amicus18 Companion Shield.



Using a Samsung KS0108 Graphic LCD

Once a Samsung graphic LCD has been chosen using the **Declare LCD_Type** directive, all **Print** outputs will be directed to that LCD.

The standard modifiers used by an alphanumeric LCD may also be used with the graphics LCD. Most of the above modifiers still work in the expected manner, however, the **At** modifier now starts at Ypos 0 and Xpos 0, where values 0,0 will be the top left corner of the LCD.

There are also four new modifiers. These are:

- Font 0 to n — Choose the nth font, if available
- Inverse 0-1 — Invert the characters sent to the LCD
- Or 0-1 — Or the new character with the original
- Xor 0-1 — Xor the new character with the original

Once one of the four new modifiers has been enabled, all future **Print** commands will use that particular feature until the modifier is disabled. For example:

```
' Enable inverted characters from this point
  Print At 0, 0, Inverse 1, "HELLO WORLD"
  Print At 1, 0, "STILL INVERTED"
' Now use normal characters
  Print At 2, 0, Inverse 0, "NORMAL CHARACTERS"
```

If no modifiers are present, then the character's ASCII representation will be displayed:

```
' Print characters A and B
  Print At 0, 0, 65, 66
```

Declares

There are nine declares associated with a Samsung graphic LCD.

Declare LCD_DTPort Port
Assign the port that will output the 8-bit data to the graphic LCD.

If the **Declare** is not used, then the default port is PortD.

Declare LCD_RWPin Port . Pin
Assigns the Port and Pin that the graphic LCD's RW line will attach to.

If the **Declare** is not used in the program, then the default Port and Pin is PortE.0.

Declare LCD_CS1Pin Port . Pin
Assigns the Port and Pin that the graphic LCD's CS1 line will attach to.

If the **Declare** is not used in the program, then the default Port and Pin is PortC.0.

Declare LCD_CS2Pin Port . Pin
Assigns the Port and Pin that the graphic LCD's CS2 line will attach to.

If the **Declare** is not used in the program, then the default Port and Pin is PortC.2.

Note

Along with the new declares, two of the existing LCD declares must also be used. Namely, RS_Pin and EN_Pin.

Declare Internal_Font On - Off, 1 or 0
The graphic LCDs that are compatible with Proton Amicus18 are non-intelligent types, therefore, a separate character set is required. This may be in one of two places, either externally, in an I^2C eeprom, or internally in a **Cdata** table.

If the **Declare** is omitted from the program, then an external font is the default setting.

If an external font is chosen, the I^2C eeprom must be connected to the specified SDA and SCL pins (as dictated by **Declare** SDA and **Declare** SCL).

If an internal font is chosen, it must be on a microcontroller device that has self modifying code features, such as the 16F87X range.

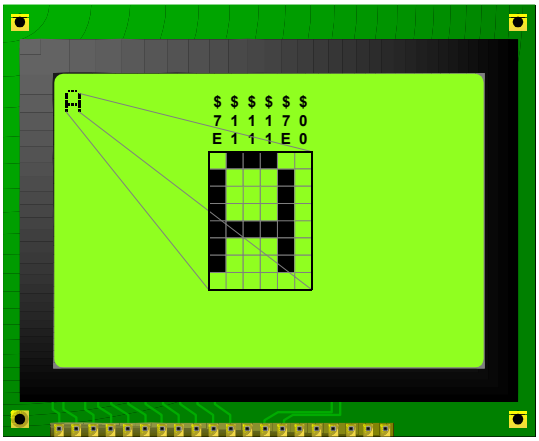
The **Cdata** table that contains the font must have a label, named Font: preceding it. For example:

```
Font: { data for characters 0 to 64 }
   Cdata $7E, $11, $11, $11, $7E, $0   ' Chr 65 "A"
   Cdata $7F, $49, $49, $49, $36, $0   ' Chr 66 "B"
   { rest of font table }
```

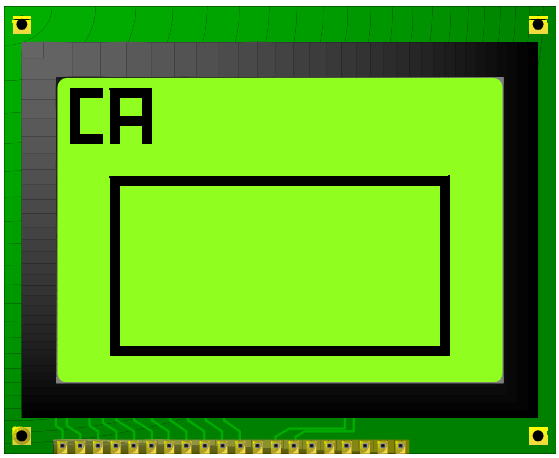
Notice the dash after the font's label, this disables any bank switching code that may otherwise disturb the location in memory of the **Cdata** table.

The font table may be anywhere in memory, however, it is best placed after the main program code.

The font is built up of an 8x6 cell, with only 5 of the 6 rows, and 7 of the 8 columns being used for alphanumeric characters. See the diagram below.



If a graphic character is chosen (chr 0 to 31), the whole of the 8x6 cell is used. In this way, large fonts and graphics may be easily constructed.



The character set itself is 128 characters long (0 -127). Which means that all the ASCII characters are present, including $, %, &, # etc.

Declare Font_Addr 0 to 7
Set the slave address for the I^2C eeprom that contains the font.

When an external source for the font is used, it may be on any one of 8 eeproms attached to the I^2C bus. So as not to interfere with any other eeproms attached, the slave address of the eeprom carrying the font code may be chosen.

If the **Declare** is omitted from the program, then address 0 is the default slave address of the font eeprom.

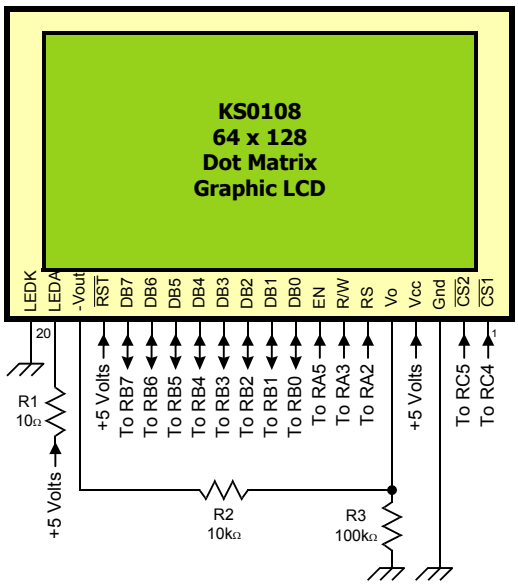
Declare GLCD_CS_Invert On - Off, 1 or 0
Some graphic LCD types have inverters on the CS lines. Which means that the LCD displays left-hand data on the right side, and vice-versa. The GLCD_CS_Invert **Declare**, adjusts the library LCD handling subroutines to take this into account.

Declare GLCD_Strobe_Delay 0 to 65535 microseconds (us).
If a noisy circuit layout is unavoidable when using a graphic LCD, then the above **Declare** may be used. This will create a delay between the Enable line being strobed. This can ease random data being produced on the LCD's screen. See below for more details on circuit layout for graphic LCDs.

If the **Declare** is not used in the program, then no delay is created between strobes, and the LCD is accessed at full efficiency.

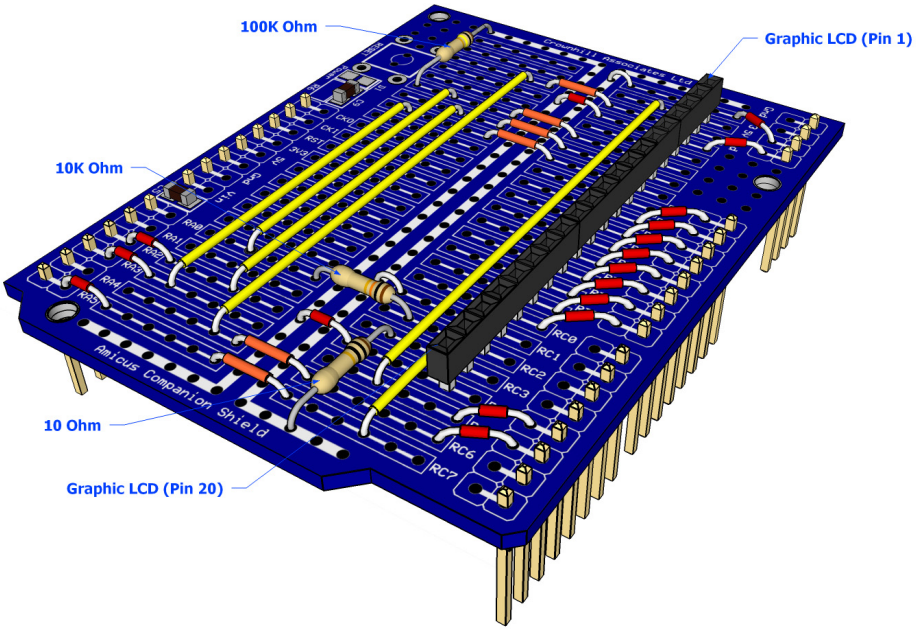
Declare GLCD_Read_Delay 0 to 65535 microseconds (us).
Create a delay of n microseconds between strobing the EN line of the graphic LCD, when reading from the GLCD. This can help noisy, or badly decoupled circuits overcome random bits being examined. The default if the **Declare** is not used in the BASIC program is a delay of 0.

If an internal font is implemented on a Samsung graphic LCD, then only four stack levels are used.

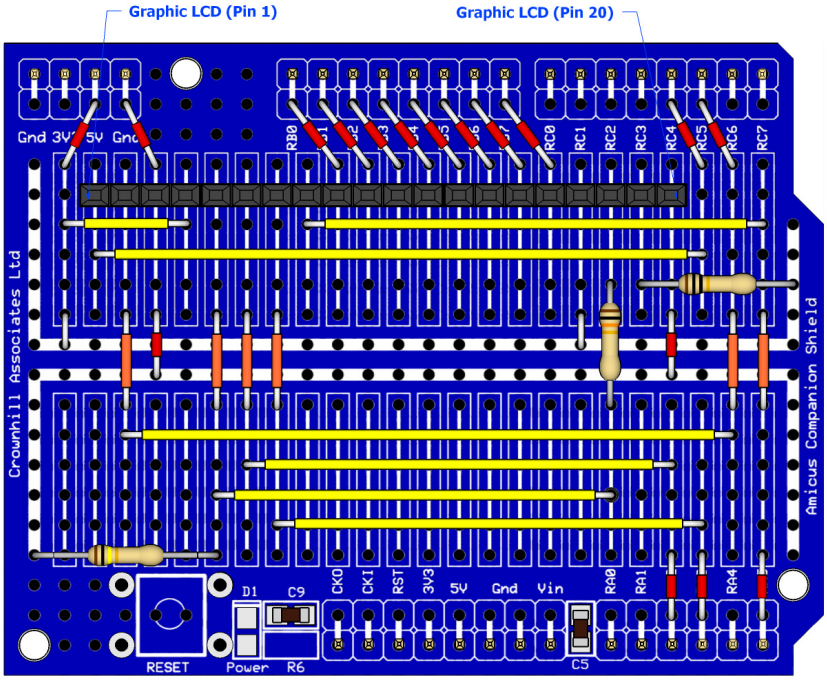


The diagram above shows a typical pin arrangement for a Samsung KS0108 graphic LCD.

The layouts below illustrate how a Samsung graphic LCD can be built on the Amicus companion board.



A top down view of the layout is shown below.



Using a Toshiba T6963 Graphic LCD

Once a Toshiba graphic LCD has been chosen using the **Declare LCD_Type** directive, all **Print** outputs will be directed to that LCD.

The standard modifiers used by an alphanumeric LCD may also be used with the graphics LCD. Most of the modifiers still work in the expected manner, however, the **At** modifier now starts at Ypos 0 and Xpos 0, where values 0,0 correspond to the top left corner of the LCD.

The Samsung modifiers Font, **Inverse**, **or**, and **xor** are not supported because of the method Toshiba LCD's using the T6963 chipset implement text and graphics.

There are several Declares for use with a Toshiba graphic LCD, some optional and some mandatory.

Declare LCD_DTPort Port
Assign the port that will output the 8-bit data to the graphic LCD.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_WRPin Port . Pin
Assigns the Port and Pin that the graphic LCD's WR line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_RDPin Port . Pin
Assigns the Port and Pin that the graphic LCD's RD line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_CEPin Port . Pin
Assigns the Port and Pin that the graphic LCD's CE line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_CDPin Port . Pin
Assigns the Port and Pin that the graphic LCD's CD line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_RSTPin Port . Pin
Assigns the Port and Pin that the graphic LCD's RST line will attach to.

The LCD's RST (Reset) **Declare** is optional and if omitted from the BASIC code the compiler will not manipulate it. However, if not used as part of the interface, you must set the LCD's RST pin high for normal operation.

Declare LCD_X_Res 0 to 255
LCD displays using the T6963 chipset come in varied screen sizes (resolutions). The compiler must know how many horizontal pixels the display consists of before it can build its library subroutines.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_Y_Res 0 to 255
LCD displays using the T6963 chipset come in varied screen sizes (resolutions). The compiler must know how many vertical pixels the display consists of before it can build its library subroutines.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_Font_Width 6 or 8
The Toshiba T6963 graphic LCDs have two internal font sizes, 6 pixels wide by eight high, or 8 pixels wide by 8 high. The particular font size is chosen by the LCD's FS pin. Leaving the FS pin floating or bringing it high will choose the 6 pixel font, while pulling the FS pin low will choose the 8 pixel font. The compiler must know what size font is required so that it can calculate screen and RAM boundaries.

Note that the compiler does not control the FS pin and it is down to the circuit layout whether or not it is pulled high or low. There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_RAM_Size 1024 to 65535
Toshiba graphic LCDs contain internal RAM used for Text, Graphic or Character Generation. The amount of RAM is usually dictated by the display's resolution. The larger the display, the more RAM is normally present. Standard displays with a resolution of 128x64 typically contain 4096 bytes of RAM, while larger types such as 240x64 or 190x128 typically contain 8192 bytes or RAM. The display's datasheet will inform you of the amount of RAM present.

If this **Declare** is not issued within the BASIC program, the default setting is 8192 bytes.

Declare LCD_Text_Pages 1 to n
As mentioned above, Toshiba graphic LCDs contain RAM that is set aside for text, graphics or characters generation. In normal use, only one page of text is all that is required, however, the compiler can re-arrange its library subroutines to allow several pages of text that is continuous. The amount of pages obtainable is directly proportional to the RAM available within the LCD itself. Larger displays require more RAM per page, therefore always limit the amount of pages to only the amount actually required or unexpected results may be observed as text, graphic and character generator RAM areas merge.

This **Declare** is purely optional and is usually not required. The default is 3 text pages if this **Declare** is not issued within the BASIC program.

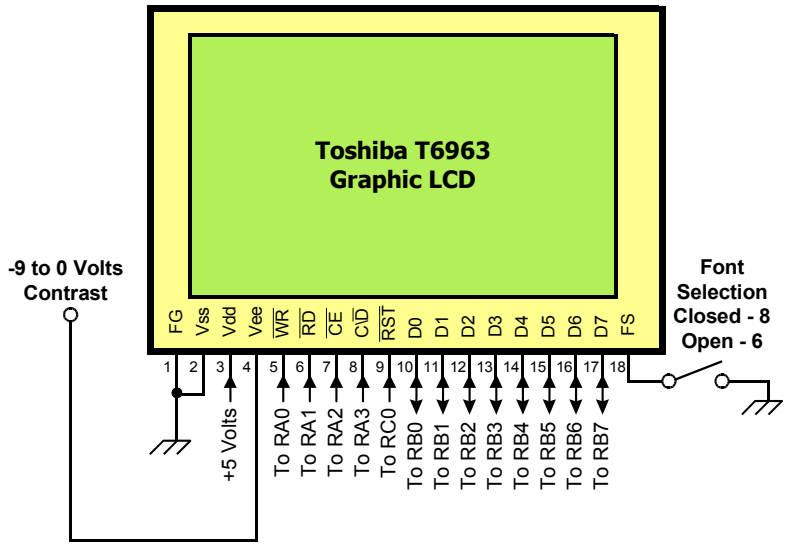
Declare LCD_Graphic_Pages 1 to n
Just as with text, the Toshiba graphic LCDs contain RAM that is set aside for graphics. In normal use, only one page of graphics is all that is required, however, the compiler can re-arrange its library subroutines to allow several pages of graphics that is continuous. The amount of pages obtainable is directly proportional to the RAM available within the LCD itself. Larger displays require more RAM per page, therefore always limit the amount of pages to only the amount actually required or unexpected results may be observed as text, graphic and character generator RAM areas merge.

This **Declare** is purely optional and is usually not required. The default is 1 graphics page if this **Declare** is not issued within the BASIC program.

Declare LCD_Text_Home_Address 0 to n
The RAM within a Toshiba graphic LCD is split into three distinct uses, text, graphics and character generation. Each area of RAM must not overlap or corruption will appear on the display as one uses the other's assigned space. The compiler's library subroutines calculate each area of RAM based upon where the text RAM starts. Normally the text RAM starts at address 0, however, there may be occasions when it needs to be set a little higher in RAM. The order of RAM is; Text, Graphic, then Character Generation.

This **Declare** is purely optional and is usually not required. The default is the text RAM staring at address 0 if this **Declare** is not issued within the BASIC program.

The diagram below shows a typical circuit for an interface with a Toshiba T6963 graphic LCD.



PulsIn

Syntax
Variable = **PulsIn** Pin, State

Overview
Change the specified pin to input and measure an input pulse.

Operators
- ***Variable*** is a user defined variable. This may be a word variable with a range of 1 to 65535, or a byte variable with a range of 1 to 255.
- ***Pin*** is a Port.Pin constant that specifies the I/O pin to use.
- ***State*** is a constant (0 or 1) or name **High** or **Low** that specifies which edge must occur before beginning the measurement.

Example
```
   Dim Var1 as Byte
Loop:
   Var1 = PulsIn PortB.0, 1   ' Measure a pulse on pin 0 of PortB.
   Hrsout Dec Var1, 13        ' Display the reading
   GoTo Loop                  ' Repeat the process.
```

Notes
PulsIn acts as a fast clock that is triggered by a change in state (0 or 1) on the specified pin. When the state on the pin changes to the state specified, the clock starts counting. When the state on the pin changes again, the clock stops. If the state of the pin doesn't change (even if it is already in the state specified in the **PulsIn** instruction), the clock won't trigger. **PulsIn** waits a maximum of 0.65535 seconds for a trigger, then returns with 0 in variable.

The variable can be either a Word or a Byte . If the variable is a word, the value returned by **PulsIn** can range from 1 to 65535 units.

The units are dependant on the frequency of the crystal used. If a 4MHz crystal is used, then each unit is 10us, while a 20MHz or greater crystal produces a unit length of 2us.

If the variable is a byte and the crystal is 4MHz, the value returned can range from 1 to 255 units of 10µs. Internally, **PulsIn** always uses a 16-bit timer. When your program specifies a byte, **PulsIn** stores the lower 8 bits of the internal counter into it. Pulse widths longer than 2550µs will give false, low readings with a byte variable. For example, a 2560µs pulse returns a reading of 256 with a word variable and 0 with a byte variable.

See also : **Counter**, **PulseOut**, **RCin**.

PulseOut

Syntax
PulseOut Pin, Period, { Initial State }

Overview
Generate a pulse on Pin of specified Period. The pulse is generated by toggling the pin twice, thus the initial state of the pin determines the polarity of the pulse. Or alternatively, the initial state may be set by using **High**-**Low** or 1-0 after the Period. Pin is automatically made an output.

Operators
- ***Pin*** is a Port.Pin constant that specifies the I/O pin to use.
- ***Period*** can be a constant of user defined variable. See notes.
- ***State*** is an optional constant (0 or 1) or name **High** or **Low** that specifies the state of the out-going pulse.

Example
```
' Send a high pulse 1ms long to PortB Pin5
  Low PortB.5
  PulseOut PortB.5, 1000

' Send a high pulse 1ms long to PortB Pin5
  PulseOut PortB.5, 1000, High
```

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

See also : **Counter**, **PulsIn**, **RCin**.

Push

Syntax
Push Variable, {Variable, Variable etc}

Overview
Place a single variable or multiple variables onto a software stack.
If the **Push** command is issued without a following variable, it will implement the assembler mnemonic **Push**, which manipulates the microcontroller's call stack.

Operator
- ***Variable*** is a user defined variable of type Bit, Byte, Byte Array, Word, Word Array, Dword, Float, or String, or constant value.

The amount of bytes pushed on to the stack varies with the variable type used. The list below shows how many bytes are pushed for a particular variable type, and their order.

- Bit — 1 Byte is pushed that holds the condition of the bit.
- Byte — 1 Byte is pushed.
- Byte Array — 1 Byte is pushed.
- Word — 2 Bytes are pushed. High Byte then Low Byte.
- Word Array — 2 Bytes are pushed. High Byte then Low Byte.
- Dword — 4 Bytes are pushed. High Byte, Mid2 Byte, Mid1 Byte then Low Byte.
- Float — 4 Bytes are pushed. High Byte, Mid2 Byte, Mid1 Byte then Low Byte.
- String — 2 Bytes are pushed. High Byte then Low Byte that point to the start address of the string in memory.
- Constant — Amount of bytes varies according to the value pushed. High Byte first.

Example 1
```
' Push two variables on to the stack then retrieve them
  Declare Stack_Size = 20    ' Create a small stack capable of holding 20 bytes

  Dim WordVar as Word        ' Create a Word variable
  Dim DwordVar as Dword      ' Create a Dword variable

  WordVar = 1234             ' Load the Word variable with a value
  DwordVar = 567890          ' Load the Dword variable with a value
  Push WordVar, DwordVar     ' Push the Word variable then the Dword variable

  Clear WordVar              ' Clear the Word variable
  Clear DwordVar             ' Clear the Dword variable

  Pop DwordVar, WordVar      ' Pop the Dword variable then the Word variable
  Hrsout Dec WordVar, "  ", Dec DwordVar, 13 ' Display the variables
  Stop
```


Example 2

```
' Push a String on to the stack then retrieve it
  Declare Stack_Size = 10           ' Create a stack capable of holding 10 bytes

  Dim SourceString as String * 20   ' Create a String variable
  Dim DestString as String * 20     ' Create another String variable

  SourceString = "HELLO WORLD"      ' Load the String variable with characters

  Push SourceString                 ' Push the String variable's address

  Pop DestString                ' Pop the previously pushed String into DestString
  Hrsout DestString, 13         ' Display the string, which will be "HELLO WORLD"
  Stop
```

Formatting a Push.

Each variable type, and more so, constant value, will push a different amount of bytes on to the stack. This can be a problem where values are concerned because it will not be known what size variable is required in order to **Pop** the required amount of bytes from the stack. For example, the code below will push a constant value of 200 on to the stack, which requires 1 byte.

```
  Push 200
```

All well and good, but what if the recipient popped variable is of a Word or Dword type.

```
  Pop WordVar
```

Popping from the stack into a Word variable will actually pull 2 bytes from the stack, however, the code above has only pushed on byte, so the stack will become out of phase with the values or variables previously pushed. This is not really a problem where variables are concerned, as each variable has a known byte count and the user knows if a Word is pushed, a Word should be popped.

The answer lies in using a formatter preceding the value or variable pushed, that will force the amount of bytes loaded on to the stack. The formatters are Byte, Word, Dword or Float.

The Byte formatter will force any variable or value following it to push only 1 byte to the stack.

```
  Push Byte 12345
```

The Word formatter will force any variable or value following it to push only 2 bytes to the stack:

```
  Push Word 123
```

The Dword formatter will force any variable or value following it to push only 4 bytes to the stack:

```
  Push Dword 123
```

The Float formatter will force any variable or value following it to push only 4 bytes to the stack, and will convert a constant value into the 4-byte floating point format:

```
  Push Float 123
```

So for the **Push** of 200 code above, you would use:

```
  Push Word 200
```


In order for it to be popped back into a Word variable, because the push would be the high byte of 200, then the low byte.

If using the multiple variable **Push**, each parameter can have a different formatter preceding it.

```
Push Word 200, Dword 1234, Float 1234
```

Note that if a floating point value is pushed, 4 bytes will be placed on the stack because this is a known format.

What is a Stack?
All microprocessors and most microcontrollers have access to a Stack, which is an area of RAM allocated for temporary data storage. But this is sadly lacking on a microcontroller device. However, the 18F devices have an architecture and low-level mnemonics that allow a Stack to be created and used very efficiently.

A stack is first created in high memory by issuing the **Stack_Size Declare**.

```
Declare Stack_Size = 40
```

The above line of code will reserve 40 bytes at the top of RAM that cannot be touched by any BASIC command, other than **Push** and **Pop**. This means that it is a safe place for temporary variable storage.

Taking the above line of code as an example, we can examine what happens when a variable is pushed on to the 40 byte stack, and then popped off again.

First the RAM is allocated. 18F25K20 has 1536 bytes of RAM that stretches linearly from address 0 to 1535. Reserving a stack of 40 bytes will reduce the top of memory so that the compiler will only see 1495 bytes (1535 - 40). This will ensure that it will not inadvertently try and use it for normal variable storage.

Pushing.
When a Word variable is pushed onto the stack, the memory map would look like the diagram below:

Top of Memory	.....Empty RAM.....	Address 1535
	~ ~	
	~ ~	
	.....Empty RAM.....	Address 1502
	.....Empty RAM.....	Address 1501
	Low Byte Address of Word Variable	Address 1496
Start of Stack	High Byte Address of Word Variable	Address 1495

The high byte of the variable is first pushed on to the stack, then the low byte. And as you can see, the stack grows in an upward direction whenever a **Push** is implemented, which means it shrinks back down whenever a **Pop** is implemented.

If we were to **Push** a Dword variable on to the stack as well as the Word variable, the stack memory would look like:

Top of Memory	.....Empty RAM.....	Address 1535
	~ ~	
	~ ~	
	.....Empty RAM.....	Address 1502
	.....Empty RAM.....	Address 1501
	Low Byte Address of Dword Variable	Address 1500
	Mid1 Byte Address of Dword Variable	Address 1499
	Mid2 Byte Address of Dword Variable	Address 1498
	High Byte Address of Dword Variable	Address 1497
	Low Byte Address of Word Variable	Address 1496
Start of Stack	High Byte Address of Word Variable	Address 1495

Popping.
When using the **Pop** command, the same variable type that was pushed last must be popped first, or the stack will become out of phase and any variables that are subsequently popped will contain invalid data. For example, using the above analogy, we need to **Pop** a Dword variable first. The Dword variable will be popped Low Byte first, then Mid1 Byte, then Mid2 Byte, then lastly the High Byte. This will ensure that the same value pushed will be reconstructed correctly when placed into its recipient variable. After the **Pop**, the stack memory map will look like:

Top of Memory	.....Empty RAM.....	Address 1535
	~ ~	
	~ ~	
	.....Empty RAM.....	Address 1502
	.....Empty RAM.....	Address 1501
	Low Byte Address of Word Variable	Address 1496
Start of Stack	High Byte Address of Word Variable	Address 1495

If a Word variable was then popped, the stack will be empty, however, what if we popped a Byte variable instead? the stack would contain the remnants of the Word variable previously pushed. Now what if we popped a Dword variable instead of the required Word variable? the stack would underflow by two bytes and corrupt any variables using those address's . The compiler cannot warn you of this occurring, so it is up to you, the programmer, to ensure that proper stack management is carried out. The same is true if the stack overflows. i.e. goes beyond the top of RAM. The compiler cannot give a warning.

Technical Details of Stack implementation.
The stack implemented by the compiler is known as an Incrementing Last-In First-Out Stack. Incrementing because it grows upwards in memory. Last-In First-Out because the last variable pushed, will be the first variable popped.

The stack is not circular in operation, so that a stack overflow will rollover into the microcontroller's hardware register, and an underflow will simply overwrite RAM immediately below the Start of Stack memory. If a circular operating stack is required, it will need to be coded in the main BASIC program, by examination and manipulation of the stack pointer (see below).

Indirect register pair FSR2L and FSR2H are used as a 16-bit stack pointer, and are incremented for every Byte pushed, and decremented for every Byte popped. Therefore checking the FSR2 registers in the BASIC program will give an indication of the stack's condition if required. This also means that the BASIC program cannot use the FSR2 register pair as part of its code, unless for manipulating the stack. Note that none of the compiler's commands, other than **Push** and **Pop**, use FSR2.

Whenever a variable is popped from the stack, the stack's memory is not actually cleared, only the stack pointer is moved. Therefore, the above diagrams are not quite true when they show empty RAM, but unless you have use of the remnants of the variable, it should be considered as empty, and will be overwritten by the next **Push** command.

See also : **Pop**, **GoSub**, **Return**.

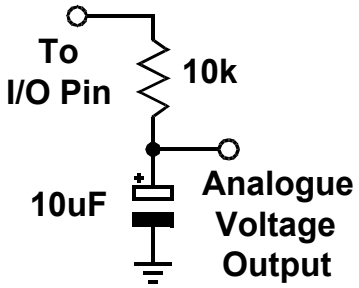
Pwm

Syntax
Pwm Pin, Duty, Cycles

Overview
Output pulse-width-modulation on a pin, then return the pin to input state.

Operators
- ***Pin*** is a Port.Pin constant that specifies the I/O pin to use.
- ***Duty*** is a variable, constant (0-255), or expression, which specifies the analogue level desired (0 to 3.3 volts).
- ***Cycles*** is a variable or constant (0-255) which specifies the number of cycles to output. Larger capacitors require multiple cycles to fully charge. Cycle time is dependant on Xtal frequency. If a 4MHz crystal is used, then cycle takes approx 5 ms. If a 20MHz or greater crystal is used, then ***cycle*** takes approx 1 ms.

Notes
Pwm can be used to generate analogue voltages (0 to 3.3V) through a pin connected to a resistor and capacitor to ground; the resistor-capacitor junction is the analogue output (see circuit). Since the capacitor gradually discharges, **Pwm** should be executed periodically to refresh the analogue voltage.



Pwm emits a burst of 1s and 0s whose ratio is proportional to the duty value you specify. If duty is 0, then the pin is continuously low (0); if duty is 255, then the pin is continuously high. For values in between, the proportion is duty/255. For example, if duty is 100, the ratio of 1s to 0s is 100/255 = 0.392, approximately 39 percent.

When such a burst is used to charge a capacitor arranged, the voltage across the capacitor is equal to:

(duty / 255) * 3.3.

So if duty is 100, the capacitor voltage is

(100 / 255) * 3.3 = 1.29 volts.

This voltage will drop as the capacitor discharges through whatever load it is driving. The rate of discharge is proportional to the current drawn by the load; more current = faster discharge. You can reduce this effect in software by refreshing the capacitor's charge with frequent use of the **Pwm** command, or you can buffer the output using an op-amp to greatly reduce the need for frequent **Pwm** cycles.

See also : **Hpwm, PulseOut, Servo.**

Random

Syntax
Variable = **Random**

or

Random Variable

Overview
Generate a pseudo-randomised value.

Operator
- ***Variable*** is a user defined variable that will hold the pseudo-random value. The pseudo-random algorithm used has a working length of 1 to 65535 (only zero is not produced).

Example
```
' Create and display a pseudo random number
  Dim Rnd as Word

  Seed $0345                ' Create a starting point for the random number generator
  While 1 = 1               ' Create an infinite loop
    Rnd = Random            ' Get a pseudo random value
    Hrsout Dec Rnd, 13      ' Display the result on the serial terminal
    DelayMs 500             ' Delay to the results can be viewed
  Wend                      ' Do it forever
```

See also: **Seed**.

RC5in

Syntax
Variable = **RC5in**

Overview
Receive Philips RC5 infrared data from a predetermined pin. The pin is automatically made an input.

Operator
- ***Variable*** can be a bit, byte, word, dword, or float variable, that will be loaded by **RC5in**. The return data from the **RC5in** command consists of two bytes, the *System* byte containing the type of remote used. i.e. TV, Video etc, and the *Command* byte containing the actual button value. The order of the bytes is *Command* (low byte) then *System* (high byte). If a byte variable is used to receive data from the infrared sensor then only the *Command* byte will be received.

Example
```
' Receive Philips RC5 data from an infrared sensor attached to PortC.0
  RC5in_Pin = RC0                ' Choose the port and pin for the infrared sensor
  Dim RC5_Word as Word           ' Create a Word variable to receive the data
' Alias the Command byte to RC5_Word low byte
  Dim RC5_Command as RC5_Word.Lowbyte
' Alias the Command byte to RC5_Word high byte
  Dim RC5_System as RC5_Word.Highbyte
  While 1 = 1                    ' Create an infinite loop
    Repeat
      RC5_Word = RC5In           ' Receive a signal from the infrared sensor
    Until RC5_Command <> 255     ' Keep looking until a valid header found
     Hrsout "System  ", Dec RC5_System,13     ' Display the System value
     Hrsout "Command ", Dec RC5_Command,13    ' Display the Command value
  Wend
```

There is a single **Declare** for use with **RC5in**:

Declare RC5in_Pin Port . Pin
Assigns the Port and Pin that will be used to input infrared data by the **RC5in** command. This may be any valid port on the microcontroller.

If the **Declare** is not used in the program, then the default Port and Pin is PortB.0.

Notes
The **RC5in** command will return with both *Command* and *System* bytes containing 255 if a valid header was not received. The CARRY (STATUS.0) flag will also be set if an invalid header was received. This is an ideal method of determining if the signal received is of the correct type.

RCin

Syntax
Variable = **RCin** Pin, State

Overview
Count time while pin remains in state, usually used to measure the charge/ discharge time of resistor/capacitor (RC) circuit.

Operators
- ***Pin*** is a Port.Pin constant that specifies the I/O pin to use. This pin will be placed into input mode and left in that state when the instruction finishes.
- ***State*** is a variable or constant (1 or 0) that will end the Rcin period. Text, High or Low may also be used instead of 1 or 0.
- ***Variable*** is a variable in which the time measurement will be stored.

Example
```
Dim Result as Word              ' Word variable to hold result.
High PortB.0                    ' Discharge the capacitor
DelayMs 1                       ' Wait for 1 ms.
Result = RCin PortB.0, High     ' Measure RC charge time.
Hrsout Dec Result, 13           ' Display the value on the serial terminal.
```

Notes
The resolution of **RCin** is dependent upon the oscillator frequency. If a 4MHz oscillator is used, the time in state is returned in 10us increments. If a 20MHz, or greater, oscillator is used, the time in state will have a 2us resolution. Declaring an Xtal value has no effect on **RCin**. The resolution always changes with the actual oscillator speed. If the pin never changes state 0 is returned.

When **RCin** executes, it starts a counter. The counter stops as soon as the specified pin is no longer in State (0 or 1). If pin is not in State when the instruction executes, **RCin** will return 1 in Variable, since the instruction requires one timing cycle to discover this fact. If pin remains in State longer than 65535 timing cycles **RCin** returns 0.

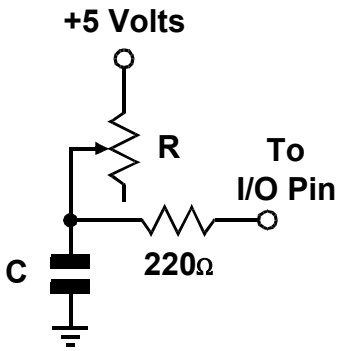


Figure A

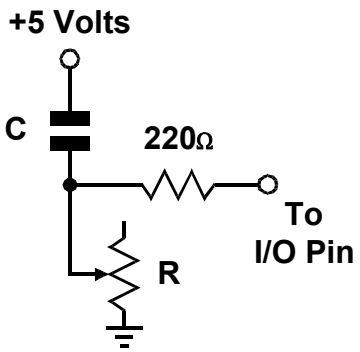


Figure B

The diagrams above show two suitable RC circuits for use with **RCin**. The circuit in figure B is preferred, because the microcontroller's logic threshold is approximately 1.5 volts. This means that the voltage seen by the pin will start at 5V then fall to 1.5V (a span of 3.5V) before **RCin** stops. With the circuit in figure A, the voltage will start at 0V and rise to 1.5V (spanning only 1.5V) before **RCin** stops.

For the same combination of R and C, the circuit shown in figure A will produce a higher result, and therefore more resolution than figure B.

Before **RCin** executes, the capacitor must be put into the state specified in the **RCin** command. For example, with figure B, the capacitor must be discharged until both plates (sides of the capacitor) are at 5V. It may seem strange that discharging the capacitor makes the input high, but you must remember that a capacitor is charged when there is a voltage difference between its plates. When both sides are at +5 Volts, the capacitor is considered discharged. Below is a typical sequence of instructions for the circuit in figure A.

```
Dim Result as Word              ' Word variable to hold result.
High PortB.0                    ' Discharge the capacitor
DelayMs 1                       ' Wait for 1 ms.
Result = RCin PortB.0, High     ' Measure RC charge time.
Hrsout Dec Result, 13           ' Display the value on the serial terminal.
```

Using **RCin** is very straightforward, except for one detail: For a given R and C, what value will **RCin** return? It's actually rather easy to calculate, based on a value called the RC time constant, or tau (τ) for short. Tau represents the time required for a given RC combination to charge or discharge by 63 percent of the total change in voltage that they will undergo. More importantly, the value τ is used in the generalized RC timing calculation. Tau's formula is just R multiplied by C:

τ = R x C

The general RC timing formula uses τ to tell us the time required for an RC circuit to change from one voltage to another:

time = -τ * (ln (Vfinal / Vinitial))

In this formula ln is the natural logarithm. Assume we're interested in a 10kΩ resistor and 0.1µF cap. Calculate τ:

τ = (10 x 103) x (0.1 x 10-6) = 1 x 10-3

The RC time constant is 1 x 10-3 or 1 millisecond. Now calculate the time required for this RC circuit to go from 5V to 1.5V (as in figure B):

Time = -1 x 10-3* (ln(5.0v / 1.5v)) = 1.204 x 10-3

Using a 20MHz crystal, the unit of time is 2µs, that time (1.204 x 10-3) works out to 602 units. With a 10kΩ resistor and 0.1µF capacitor, **RCin** would return a value of approximately 600. Since Vinitial and Vfinal don't change, we can use a simplified rule of thumb to estimate **RCin** results for circuits similar to figure A:

RCin units = 600 x R (in kΩ) x C (in µF)

Another useful rule of thumb can help calculate how long to charge/discharge the capacitor before **RCin**. In the example shown, that's the purpose of the **High** and **DelayMs** commands. A given RC charges or discharges 98 percent of the way in 4 time constants (4 x R x C).

In both circuits, the charge/discharge current passes through a 220Ω series resistor and the capacitor. So if the capacitor were 0.1µF, the minimum charge/discharge time should be:

Charge time = 4 x 220 x (0.1 x 10-6) = 88 x 10-6

So it takes only 88µs for the cap to charge/discharge, which means that the 1ms charge/discharge time of the example is more than adequate.

You may be wondering why the 220Ω resistor is necessary at all. Consider what would happen if resistor R in figure A were a pot, and was adjusted to 0Ω. When the I/O pin went high to discharge the cap, it would see a short direct to ground. The 220Ω series resistor would limit the short circuit current to 5V/220Ω = 23mA and protect the microcontroller from any possible damage.

See also : **Adin**, **Counter**, **Pot**, **PulsIn**.

Repeat...Until

Syntax
Repeat Condition
Instructions
Instructions
Until Condition

or

Repeat { Instructions : } **Until** Condition

Overview
Execute a block of instructions until a condition is true.

Example
```
Var1 = 1
Repeat
  Hrsout Dec Var1, 13
  DelayMs 200
  Inc Var1
Until Var1 > 10
```

or

```
Repeat High LED : Until PortB.0 = 1 ' Wait for a Port change
```

Notes
The **Repeat**-**Until** loop differs from the **While**-**Wend** type in that, the **Repeat** loop will carry out the instructions within the loop at least once, then continuously until the condition is true, but the **While** loop only carries out the instructions if the condition is true.

The **Repeat**-**Until** loop is an ideal replacement to a **For**-**Next** loop, and actually takes less code space, thus performing the loop faster.

Two commands have been added especially for a **Repeat** loop, these are **Inc** and **Dec**.

```
Inc. Increment a variable i.e. Var1 = Var1 + 1

Dec. Decrement a variable i.e. Var1 = Var1 - 1
```

The above example shows the equivalent to the **For**-**Next** loop:

```
For Var1 = 1 to 10 : Next
```

See also : **While**...**Wend**, **For**...**Next**...**Step**.

Return

Syntax
Return

or

Return Variable

Overview
Return from a subroutine.

Operator
- ***Variable*** is an optional user defined variable of type Bit, Byte, Byte Array, Word, Word Array, Dword, Float, or String, or Constant value, that will be pushed onto the stack before the subroutine is exited.

Example
```
' Call a subroutine with parameters
   Declare Stack_Size = 20       ' Create a small stack capable of holding 20 bytes

   Dim WordVar1 as Word          ' Create a Word variable
   Dim WordVar2 as Word          ' Create another Word variable
   Dim Receipt as Word           ' Create a variable to hold result

   WordVar1 = 1234               ' Load the Word variable with a value
   WordVar2 = 567                ' Load the other Word variable with a value
' Call the subroutine and return a value
   GoSub AddThem[WordVar1, WordVar2], Receipt
   Print Dec Receipt             ' Display the result as decimal
   Stop

' Subroutine starts here. Add two parameters passed and return the result
AddThem:
   Dim AddWordVar1 as Word       ' Create two uniquely named variables
   Dim AddWordVar2 as Word

   Pop AddWordVar2               ' Pop the last variable pushed
   Pop AddWordVar1               ' Pop the first variable pushed
   AddWordVar1 = AddWordVar1 + AddWordVar2  ' Add the values together
   Return AddWordVar1            ' Return the result of the addition
```


In reality, what's happening with the **Return** in the above program is simple, if we break it into its constituent events:

```
Push AddWordVar1
Return
```

Notes
The same rules apply for the variable returned as they do for **Pop**, which is after all, what is happening when a variable is returned.

Return resumes execution at the statement following the **GoSub** which called the subroutine.

See also : **Call**, **GoSub**, **Push**, **Pop** .

Right$

Syntax
Destination String = **Right$** (Source String, Amount of characters)

Overview
Extract n amount of characters from the right of a source string and copy them into a destination string.

Operators
- ***Destination String*** can only be a String variable, and should be large enough to hold the correct amount of characters extracted from the Source String.
- ***Source String*** can be a String variable, or a Quoted String of Characters. See below for more variable types that can be used for Source String.
- ***Amount of characters*** can be any valid variable type, expression or constant value, that signifies the amount of characters to extract from the right of the Source String. Values start at 1 for the rightmost part of the string and should not exceed 255 which is the maximum allowable length of a String variable.

Example 1
```
' Copy 5 characters from the right of SourceString into DestString

  Dim SourceString as String * 20   ' Create a String of 20 characters
  Dim DestString as String * 20     ' Create another String

  SourceString = "HELLO WORLD"      ' Load the source string with characters
' Copy 5 characters from the source string into the destination string
  DestString = Right$ (SourceString, 5)
  Hrsout DestString,13              ' Display the result, which will be "WORLD"
  Stop
```

Example 2
```
' Copy 5 characters from right of a Quoted Character String to DestString
  Dim DestString as String * 20     ' Create a String of 20 characters

' Copy 5 characters from the quoted string into the destination string
  DestString = Right$ ("HELLO WORLD", 5)
  Hrsout DestString, 13             ' Display the result, which will be "WORLD"
  Stop
```

The Source String can also be a Byte, Word, Byte Array, Word Array or Float variable, in which case the value contained within the variable is used as a pointer to the start of the Source String's address in RAM.

Example 3

```
' Copy 5 characters from the right of SourceString into DestString using a
' pointer to SourceString

  Dim SourceString as String * 20   ' Create a String of 20 characters
  Dim DestString as String * 20     ' Create another String
' Create a Word variable to hold the address of SourceString
  Dim StringAddr as Word

  SourceString = "HELLO WORLD"      ' Load the source string with characters
' Locate the start address of SourceString in RAM
  StringAddr = VarPtr (SourceString)
' Copy 5 characters from the source string into the destination string
  DestString = Right$ (StringAddr, 5)
  Hrsout DestString, 13             ' Display the result, which will be "WORLD"
  Stop
```

A third possibility for Source String is a Label name, in which case a null terminated Quoted String of Characters is read from a **Cdata** table.

Example 4

```
' Copy 5 characters from the right of a Cdata table into DestString

  Dim DestString as String * 20     ' Create a String of 20 characters

' Copy 5 characters from label Source into the destination string
  DestString = Right$ (Source, 5)
  Hrsout DestString, 13             ' Display the result, which will be "WORLD"
  Stop

' Create a null terminated string of characters in code memory
Source:
  Cdata "HELLO WORLD", 0
```

See also : **Creating and using Strings, Creating and using Virtual Strings with Cdata, Cdata, Len, Left$, Mid$, Str$, ToLower, ToUpper, VarPtr.**

Rsin

Syntax
Variable = **Rsin**, { Timeout Label }

or

Rsin { Timeout Label }, Modifier..Variable {, Modifier.. Variable...}

Overview
Receive one or more bytes from a predetermined pin at a predetermined baud rate in standard asynchronous format using 8 data bits, no parity and 1 stop bit (8N1). The pin is automatically made an input.

Operators
- ***Modifiers*** may be one of the serial data modifiers explained below.
- ***Variable*** can be any user defined variable.
- An optional ***Timeout Label*** may be included to allow the program to continue if a character is not received within a certain amount of time. Timeout is specified in units of 1 millisecond and is specified by using a **Declare** directive.

Example
```
  Declare Rsin_Timeout = 2000       ' Timeout after 2 seconds
  Dim Var1 as Byte
  Dim WordVar as Word
  Var1 = Rsin, { TimeoutLabel }     ' Receive with a timeout
  Rsin Var1, WordVar
  Rsin { Label }, Var1, WordVar

TimeoutLabel:
  { do something when timed out }
```

Declares
There are four Declares for use with **Rsin**. These are :

Declare Rsin_Pin Port . Pin
Assigns the Port and Pin that will be used to input serial data by the **Rsin** command. This may be any valid port on the microcontroller.

If the **Declare** is not used in the program, then the default Port and Pin is PortB.1.

Declare Rsin_Mode Inverted, True or 1, 0
Sets the serial mode for the data received by **Rsin**. This may be inverted or true. Alternatively, a value of 1 may be substituted to represent inverted, and 0 for true.

If the **Declare** is not used in the program, then the default mode is Inverted.

Declare Serial_Baud 0 to 65535 bps (baud)
Informs the **Rsin** and **Rsout** routines as to what baud rate to receive and transmit data.

Virtually any baud rate may be transmitted and received, but there are standard bauds:

300, 600, 1200, 2400, 4800, 9600, and 19200.

When using a 4MHz crystal, the highest baud rate that is reliably achievable is 9600. However, an increase in the oscillator speed allows higher baud rates to be achieved, including 38400 baud.

If the **Declare** is not used in the program, then the default baud is 9600.

Declare Rsin_Timeout 0 to 65535 milliseconds (ms)
Sets the time, in milliseconds, that **Rsin** will wait for a start bit to occur.

Rsin waits in a tight loop for the presence of a start bit. If no timeout value is used, then it will wait forever. The **Rsin** command has the option of jumping out of the loop if no start bit is detected within the time allocated by timeout.

If the **Declare** is not used in the program, then the default timeout value is 10000ms or 10 seconds.

Rsin Modifiers.
As we already know, **Rsin** will wait for and receive a single byte of data, and store it in a variable . If the microcontroller were connected to a PC running a terminal program and the user pressed the "A" key on the keyboard, after the **Rsin** command executed, the variable would contain 65, which is the ASCII code for the letter "A"

What would happen if the user pressed the "1" key? The result would be that the variable would contain the value 49 (the ASCII code for the character "1"). This is an important point to remember: every time you press a character on the keyboard, the computer receives the ASCII value of that character. It is up to the receiving side to interpret the values as necessary. In this case, perhaps we actually wanted the variable to end up with the value 1, rather than the ASCII code 49.

The **Rsin** command provides a modifier, called the decimal modifier, which will interpret this for us. Look at the following code:

```
Dim SerData as Byte
Rsin Dec SerData
```

Notice the decimal modifier in the **Rsin** command that appears just to the left of the SerData variable. This tells **Rsin** to convert incoming text representing decimal numbers into true decimal form and store the result in SerData. If the user running the terminal software pressed the "1", "2" and then "3" keys followed by a space or other non-numeric text, the value 123 will be stored in the variable SerData, allowing the rest of the program to perform any numeric operation on the variable.

Without the decimal modifier, however, you would have been forced to receive each character ("1", "2" and "3") separately, and then would still have to do some manual conversion to arrive at the number 123 (one hundred twenty three) before you can do the desired calculations on it.

The decimal modifier is designed to seek out text that represents decimal numbers. The characters that represent decimal numbers are the characters "0" through "9". Once the **Rsin** command is asked to use the decimal modifier for a particular variable, it monitors the incoming serial data, looking for the first decimal character. Once it finds the first decimal character, it will continue looking for more (accumulating the entire multi-digit number) until is finds a non-decimal numeric character. Remember that it will not finish until it finds at least one decimal character followed by at least one non-decimal character.

To illustrate this further, examine the following examples (assuming we're using the same code example as above):

Serial input: "ABC"
Result: The program halts at the **Rsin** command, continuously waiting for decimal text.

Serial input: "123" (with no characters following it)
Result: The program halts at the **Rsin** command. It recognises the characters "1", "2" and "3" as the number one hundred twenty three, but since no characters follow the "3", it waits continuously, since there's no way to tell whether 123 is the entire number or not.

Serial input: "123" (followed by a space character)
Result: Similar to the above example, except once the space character is received, the program knows the entire number is 123, and stores this value in SerData. The **Rsin** command then ends, allowing the next line of code to run.

Serial input: "123A"
Result: Same as the example above. The "A" character, just like the space character, is the first non-decimal text after the number 123, indicating to the program that it has received the entire number.

Serial input: "ABCD123EFGH"
Result: Similar to examples 3 and 4 above. The characters "ABCD" are ignored (since they're not decimal text), the characters "123" are evaluated to be the number 123 and the following character, "E", indicates to the program that it has received the entire number.

The final result of the **Dec** modifier is limited to 16 bits (up to the value 65535). If a value larger than this is received by the decimal modifier, the end result will be incorrect because the result rolled-over the maximum 16-bit value. Therefore, **Rsin** modifiers may not (at this time) be used to load Dword (32-bit) variables.

The decimal modifier is only one of a family of conversion modifiers available with **Rsin** See below for a list of available conversion modifiers. All of the conversion modifiers work similar to the decimal modifier (as described above). The modifiers receive bytes of data, waiting for the first byte that falls within the range of characters they accept (e.g., "0" or "1" for binary, "0" to "9" for decimal, "0" to "9" and "A" to "F" for hex. Once they receive a numeric character, they keep accepting input until a non-numeric character arrives, or in the case of the fixed length modifiers, the maximum specified number of digits arrives.

While very effective at filtering and converting input text, the modifiers aren't completely foolproof. As mentioned before, many conversion modifiers will keep accepting text until the first non-numeric text arrives, even if the resulting value exceeds the size of the variable. After **Rsin**, a Byte variable will contain the lowest 8 bits of the value entered and a Word (16-bits) would contain the lowest 16 bits. You can control this to some degree by using a modifier that specifies the number of digits, such as Dec2, which would accept values only in the range of 0 to 99.

Conversion Modifier	Type of Number Numeric	Characters Accepted
Dec{1..10}	Decimal, optionally limited to 1 - 10 digits	0 through 9
Hex{1..8}	Hexadecimal, optionally limited to 1 - 8 digits	0 through 9, A through F
Bin{1..32}	Binary, optionally limited to 1 - 32 digits	0, 1

A variable preceded by **Bin** will receive the ASCII representation of its binary value.
For example, if **Bin** Var1 is specified and "1000" is received, Var1 will be set to 8.

A variable preceded by **Dec** will receive the ASCII representation of its decimal value.
For example, if **Dec** Var1 is specified and "123" is received, Var1 will be set to 123.

A variable preceded by **Hex** will receive the ASCII representation of its hexadecimal value.
For example, if **Hex** Var1 is specified and "FE" is received, Var1 will be set to 254.

SKIP followed by a count will skip that many characters in the input stream.
For example, SKIP 4 will skip 4 characters.

The **Rsin** command can be configured to wait for a specified sequence of characters before it retrieves any additional input. For example, suppose a device attached to the microcontroller is known to send many different sequences of data, but the only data you wish to observe happens to appear right after the unique characters, "XYZ". A modifier named Wait can be used for this purpose:

```
Rsin Wait("XYZ"), SerData
```

The above code waits for the characters "X", "Y" and "Z" to be received, in that order, then it receives the next data byte and places it into variable SerData.

Str modifier.
The **Rsin** command also has a modifier for handling a string of characters, named **Str**.

The **Str** modifier is used for receiving a string of characters into a byte array variable.

A string is a set of characters that are arranged or accessed in a certain order. The characters "ABC" would be stored in a string with the "A" first, followed by the "B" then followed by the "C". A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string "ABC" would be stored in a byte array containing three bytes (elements).

Below is an example that receives ten bytes and stores them in the 10-byte array, SerString:

```
Dim SerString[10] as Byte     ' Create a 10-byte array.
Rsin Str SerString            ' Fill the array with received data.
Print Str SerString           ' Display the string.
```

If the amount of received characters is not enough to fill the entire array, then a formatter may be placed after the array's name, which will only receive characters until the specified length is reached. For example:

```
Dim SerString[10] as Byte     ' Create a 10-byte array.
Rsin Str SerString\5          ' Fill the first 5-bytes of the array
Print Str SerString\5         ' Display the 5-character string.
```

The example above illustrates how to fill only the first n bytes of an array, and then how to display only the first n bytes of the array. n refers to the value placed after the backslash.

Because of its complexity, serial communication can be rather difficult to work with at times. Using the guidelines below when developing a project using the **Rsin** and **Rsout** commands may help to eliminate some obvious errors:

Always build your project in steps.
Start with small, manageable pieces of code, (that deal with serial communication) and test them, one individually.
Add more and more small pieces, testing them each time, as you go.
Never write a large portion of code that works with serial communication without testing its smallest workable pieces first.

Pay attention to timing.
Be careful to calculate and overestimate the amount of time, operations should take within the micro-controller for a given oscillator frequency. Misunderstanding the timing constraints is the source of most problems with code that communicate serially. If the serial communication in your project is bi-directional, the above statement is even more critical.

Pay attention to wiring.
Take extra time to study and verify serial communication wiring diagrams. A mistake in wiring can cause strange problems in communication, or no communication at all. Make sure to connect the ground pins (Vss) between the devices that are communicating serially.

Verify port setting on the PC and in the Rsin / Rsout commands.
Unmatched settings on the sender and receiver side will cause garbled data transfers or no data transfers. This is never more critical than when a line transceiver is used(i.e. MAX232). Always remember that a line transceiver inverts the serial polarity.
If the serial data received is unreadable, it is most likely caused by a baud rate setting error, or a polarity error.

If receiving data from another device that is not a microcontroller, try to use baud rates of 9600 and below.

Because of additional overheads in the microcontroller, and the fact that the **Rsin** command offers no hardware receive buffer for serial communication, received data may sometimes be missed or garbled. If this occurs, try lowering the baud rate, or increasing the crystal frequency. Using simple variables (not arrays) will also increase the chance that the microcontroller will receive the data properly.

See also : **Declare**, **Rsout**, **Serin**, **Serout**, **HRsin**, **HRsout**, **Hserin**, **Hserout**.

Rsout

Syntax
Rsout Item {, Item... }

Overview
Send one or more Items to a predetermined pin at a predetermined baud rate in standard asynchronous format using 8 data bits, no parity and 1 stop bit (8N1). The pin is automatically made an output.

Operators
- Item may be a constant, variable, expression, or string.

There are no operators as such, instead there are modifiers. For example, if the text **Dec** precedes an item, the ASCII representation for each digit is transmitted.

The modifiers are listed below:

Modifier	Operation
At ypos,xpos	Position the cursor on a suitable serial LCD
Cls	Clear a suitable serial LCD (also creates a 30ms delay)
Bin{1..32}	Send binary digits
Dec{1..10}	Send decimal digits
Hex{1..8}	Send hexadecimal digits
Sbin{1..32}	Send signed binary digits
Sdec{1..10}	Send signed decimal digits
Shex{1..8}	Send signed hexadecimal digits
Ibin{1..32}	Send binary digits with a preceding '%' identifier
Idec{1..10}	Send decimal digits with a preceding '#' identifier
Ihex{1..8}	Send hexadecimal digits with a preceding '$' identifier
ISbin{1..32}	Send signed binary digits with a preceding '%' identifier
ISdec{1..10}	Send signed decimal digits with a preceding '#' identifier
IShex{1..8}	Send signed hexadecimal digits with a preceding '$' identifier
Rep c\n	Send character c repeated n times
Str array\n	Send all or part of an array
Cstr cdata	Send string data defined in a **Cdata** statement.

The numbers after the **Bin**, **Dec**, and **Hex** modifiers are optional. If they are omitted, then the default is all the digits that make up the value will be displayed.

If a floating point variable is to be displayed, then the digits after the **Dec** modifier determine how many remainder digits are send. i.e. numbers after the decimal point.

```
Dim FloatVar as Float
FloatVar = 3.145
Rsout Dec2 FloatVar,13     ' Send 2 values after the decimal point
```

The above program will transmit 3.14

If the digit after the **Dec** modifier is omitted, then 3 values will be displayed after the decimal point.


```
Dim FloatVar as Float
FloatVar = 3.1456
Rsout Dec FloatVar,13   ' Send 3 values after the decimal point
```

The above program will send 3.145

There is no need to use the **Sdec** modifier for signed floating point values, as the compiler's **Dec** modifier will automatically display a minus result:

```
Dim FloatVar as Float
FloatVar = -3.1456
Rsout Dec FloatVar,13   ' Send 3 values after the decimal point
```

The above program will send -3.145

Hex or **Bin** modifiers cannot be used with floating point values or variables.

The Xpos and Ypos values in the **At** modifier both start at 1. For example, to place the text "HELLO WORLD" on line 1, position 1, the code would be:

```
Rsout At 1, 1, "Hello World\r"
```

Example 1

```
Dim Var1 as Byte
Dim WordVar as Word
Dim DwordVar as Dword

Rsout "Hello World", 13               ' Display the text "Hello World"
Rsout "Var1= ", Dec Var1, 13          ' Display the decimal value of Var1
Rsout "Var1= ", Hex Var1, 13          ' Display the hexadecimal value of Var1
Rsout "Var1= ", Bin Var1, 13          ' Display the binary value of Var1
Rsout "DwordVar= ", Hex6 DwordVar, 13 ' Display 6 hex chars of DwordVar
```

Example 2

```
' Display a negative value on a serial terminal.
Symbol Negative = -200
Rsout At 1,1, Sdec Negative
```

Example 3

```
' Display a negative value on a serial terminal with a preceding identifier.
Rsout At 1,1, IShex -$1234
```

Example 3 will produce the text "$-1234" on the serial terminal.

Combining the unique features of the self modifying microcontroller's with a string format, the compiler is capable of reducing the overhead of printing, or transmitting large amounts of text data.

The **Cstr** modifier may be used in commands that deal with text processing i.e. **Serout**, **HRsout**, and **Print** etc.

The **Cstr** modifier is used in conjunction with the **Cdata** command. The **Cdata** command is used for initially creating the string of characters:

```
String1: Cdata "Hello World", 0
```

The above line of case will create, in flash memory, the values that make up the ASCII text "HELLO WORLD", at address String1. Note the null terminator after the ASCII text.

null terminated means that a zero (null) is placed at the end of the string of ASCII characters to signal that the string has finished.

To display, or transmit this string of characters, the following command structure could be used:

```
  Rsout Cstr String1
```

The label that declared the address where the list of **Cdata** values resided, now becomes the string's name. In a large program with lots of text formatting, this type of structure can save quite literally hundreds of bytes of valuable code space.

Try both these small programs, and you'll see that using **Cstr** saves a few bytes of code:

First the standard way of displaying text:

```
  Rsout "Hello World\r"
  Rsout "How are you?\r"
  Rsout "I am fine!\r"
  Stop
```

Now using the **Cstr** modifier:

```
  Rsout Cstr Text1
  Rsout Cstr Text2
  Rsout Cstr Text3
  Stop

Text1: Cdata "Hello World", 13, 0
Text2: Cdata "How are you?", 13, 0
Text3: Cdata "I am fine!", 13, 0
```

Again, note the null terminators after the ASCII text in the **Cdata** commands. Without these, the microcontroller will continue to transmit data in an endless loop.

The term 'virtual string' relates to the fact that a string formed from the **Cdata** command cannot be written too, but only read from.

The **Str** modifier is used for sending a string of bytes from a byte array variable. A string is a set of bytes sized values that are arranged or accessed in a certain order.

The values 1, 2, 3 would be stored in a string with the value 1 first, followed by 2 then followed by the value 3. A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string 1,2,3 would be stored in a byte array containing three bytes (elements).

Below is an example that displays four bytes (from a byte array):

```
Dim MyArray[10] as Byte     ' Create a 10-byte array.
MyArray[0] = "H"            ' Load the first 5 bytes of the array
MyArray[1] = "E"            ' With the data to send
MyArray[2] = "L"
MyArray[3] = "L"
MyArray[4] = "O"
Rsout Str MyArray\5         ' Display a 5-byte string.
```

Note that we use the optional \n argument of **Str**. If we didn't specify this, the microcontroller would try to keep sending characters until all 10 bytes of the array were transmitted. Since we do not wish all 10 bytes to be transmitted, we chose to tell it explicitly to only send the first 5 bytes.

The above example may also be written as:

```
Dim MyArray[10] as Byte     ' Create a 10-byte array.
Str MyArray = "HELLO"       ' Load the first 5 bytes of the array
Rsout Str MyArray\5         ' Send 5-byte string.
```

The above example, has exactly the same function as the previous one. The only difference is that the string is now constructed using **Str** as a command instead of a modifier.

Declares
There are four Declares for use with **Rsout**. These are :

Declare Rsout_Pin Port . Pin
Assigns the Port and Pin that will be used to output serial data from the **Rsout** command. This may be any valid port on the microcontroller.

If the **Declare** is not used in the program, then the default Port and Pin is PortB.0.

Declare Rsout_Mode Inverted, True or 1, 0
Sets the serial mode for the data transmitted by **Rsout**. This may be inverted or true. Alternatively, a value of 1 may be substituted to represent inverted, and 0 for true.

If the **Declare** is not used in the program, then the default mode is INVERTED.

Declare Serial_Baud 0 to 65535 bps (baud)
Informs the **Rsin** and **Rsout** routines as to what baud rate to receive and transmit data.

Virtually any baud rate may be transmitted and received, but there are standard bauds:

300, 600, 1200, 2400, 4800, 9600, and 19200.

When using a 4MHz crystal, the highest baud rate that is reliably achievable is 9600. However, an increase in the oscillator speed allows higher baud rates to be achieved, including 38400 baud.

If the **Declare** is not used in the program, then the default baud is 9600.

Declare Rsout_Pace 0 to 65535 microseconds (us)
Implements a delay between characters transmitted by the **Rsout** command.

On occasion, the characters transmitted serially are in a stream that is too fast for the receiver to catch, this results in missed characters. To alleviate this, a delay may be implemented between each individual character transmitted by **Rsout**.

If the **Declare** is not used in the program, then the default is no delay between characters.

Notes
Rsout is oscillator independent as long as the crystal frequency is declared at the top of the program. If no declare is used, then **Rsout** defaults to a 64MHz crystal frequency for its bit timing.

The **At** and **Cls** modifiers are primarily intended for use with suitable serial LCD modules. Using the following command sequence will first clear the LCD, then display text at position 5 of line 2:

```
Rsout Cls, At 2, 5, "Hello World"
```

The values after the **At** modifier may also be variables.

See also : **Declare**, **Rsin** , **Serin**, **Serout**, **HRsin**, **HRsout**, **Hserin**, **Hserout**.

Seed

Syntax
Seed Value

Overview
Seed the random number generator, in order to obtain a more random result.

Operator
- ***Value*** can be a variable, constant or expression, with a value from 1 to 65535. A value of $0345 is a good starting point.

Example
```
' Create and display a Random number

  Dim Rnd as Word

  Seed $0345                ' Create a starting point for the random number generator
  While 1 = 1               ' Create an infinite loop
    Rnd = Random            ' Get a pseudo random value
    Hrsout Dec Rnd, 13      ' Display the result on the serial terminal
    DelayMs 500             ' Delay to the results can be viewed
  Wend                      ' Do it forever
```

See also: **Random**.

Select..Case..EndSelect

Syntax
Select Expression
  Case Condition(s)
    Instructions
  {
  Case Condition(s)
    Instructions

  Case Else
    Statement(s)
  }
EndSelect

The curly braces signify optional conditions.

Overview
Evaluate an Expression then continually execute a block of BASIC code based upon comparisons to Condition(s). After executing a block of code, the program continues at the line following the **EndSelect**. If no conditions are found to be True and a **Case Else** block is included, the code after the **Case Else** leading to the **EndSelect** will be executed.

Operators
- ***Expression*** can be any valid variable, constant, expression or inline command that will be compared to the Conditions.
- ***Condition(s)*** is a statement that can evaluate as True or False. The Condition can be a simple or complex relationship, as described below. Multiple conditions within the same **Case** can be separated by commas.
- ***Instructions*** can be any valid BASIC command that will be operated on if the **Case** condition produces a True result.

Example
```
' Load variable Result according to the contents of variable Var1
' Result will return a value of 255 if no valid condition was met

  Dim Var1 as Byte
  Dim Result as Byte
  Result = 0            ' Clear the result variable before we start
  Var1 = 1              ' Variable to base the conditions upon
  Select Var1
    Case 1              ' Is Var1 equal to 1 ?
      Result = 1        ' Load Result with 1 if yes
    Case 2              ' Is Var1 equal to 2 ?
      Result = 2        ' Load Result with 2 if yes
    Case 3              ' Is Var1 equal to 3 ?
      Result = 3        ' Load Result with 3 if yes
    Case Else           ' Otherwise...
      Result = 255      ' Load Result with 255
  EndSelect
  Hrsout Dec Result, 13 ' Display the result on the serial terminal
```


Notes
Select..**Case** is simply an advanced form of the **If**..**Then**..**ElseIf**..**Else** construct, in which multiple **ElseIf** statements are executed by the use of the **Case** command.

Taking a closer look at the **Case** command:

```
Case Conditional_Op Expression
```

Where Conditional_Op can be an = operator (which is implied if absent), or one of the standard comparison operators <>, <, >, >= or <=. Multiple conditions within the same **Case** can be separated by commas. If, for example, you wanted to run a **Case** block based on a value being less than one or greater than nine, the syntax would look like:

```
Case <1, >9
```

Another way to implement **Case** is:

```
Case value1 to value2
```

In this form, the valid range is from Value1 to Value2, inclusive. So if you wished to run a **Case** block on a value being between the values 1 and 9 inclusive, the syntax would look like:

```
Case 1 to 9
```

For those of you that are familiar with C or Java, you will know that in those languages the statements in a **Case** block fall through to the next **Case** block unless the keyword break is encountered. In BASIC however, the code under an executed **Case** block jumps to the code immediately after **EndSelect**.

Shown below is a typical **Select Case** structure with its corresponding **If**..**Then** equivalent code alongside.

```
Select Var1
  Case 6, 9, 99, 66
  ' If Var1 = 6 or Var1 = 9 or Var1 = 99 or Var1 = 66 Then
    Hrsout "OR Values\r"
  Case 110 to 200
  ' ElseIf Var1 >= 110 and Var1 <= 200 Then
    Hrsout "AND Values\r"
  Case 100
  ' ElseIf Var1 = 100 Then
    Print "Equal Value\r"
  Case >300
  ' ElseIf Var1 > 300 Then
    Hrsout "Greater Value\r"
  Case Else
  ' Else
    Hrsout "Default Value\r"
EndSelect
' EndIf
```

See also : **If**..**Then**..**ElseIf**..**Else**..**EndIf**.

Serin

Syntax
Serin Rpin { \ Fpin }, Baudmode, { Plabel, } { Timeout, Tlabel, } [InputData]

Overview
Receive asynchronous serial data (i.e. RS232 data).

Operators
- ***Rpin*** is a Port.Bit constant that specifies the I/O pin through which the serial data will be received. This pin will be set to input mode.
- ***Fpin*** is an optional Port.Bit constant that specifies the I/O pin to indicate flow control status on. This pin will be set to output mode.
- ***Baudmode*** may be a variable, constant, or expression (0 - 65535) that specifies serial timing and configuration.
- ***Plabel*** is an optional label indicating where the program should jump to in the event of a parity error. This argument should only be provided if Baudmode indicates that parity is required.
- ***Timeout*** is an optional constant (0 - 65535) that informs **Serin** how long to wait for incoming data. If data does not arrive in time, the program will jump to the address specified by Tlable.
- ***Tlabel*** is an optional label that must be provided along with Timeout, indicating where the program should go in the event that data does not arrive within the period specified by Timeout.
- ***InputData*** is list of variables and modifiers that informs **Serin** what to do with incoming data. **Serin** may store data in a variable, array, or an array string using the **Str** modifier.

Notes
One of the most popular forms of communication between electronic devices is serial communication. There are two major types of serial communication; asynchronous and synchronous. The **Rsin**, **Rsout**, **Serin** and **Serout** commands are all used to send and receive asynchronous serial data. While the **Shin** and **Shout** commands are for use with synchronous communications.

The term asynchronous means 'no clock.' More specifically, 'asynchronous serial communication' means data is transmitted and received without the use of a separate 'clock' line. Data can be sent using as few as two wires; one for data and one for ground. The PC's serial ports (also called COM ports or RS232 ports) use asynchronous serial communication. Note: the other kind of serial communication, synchronous, uses at least three wires; one for clock, one for data and one for ground.

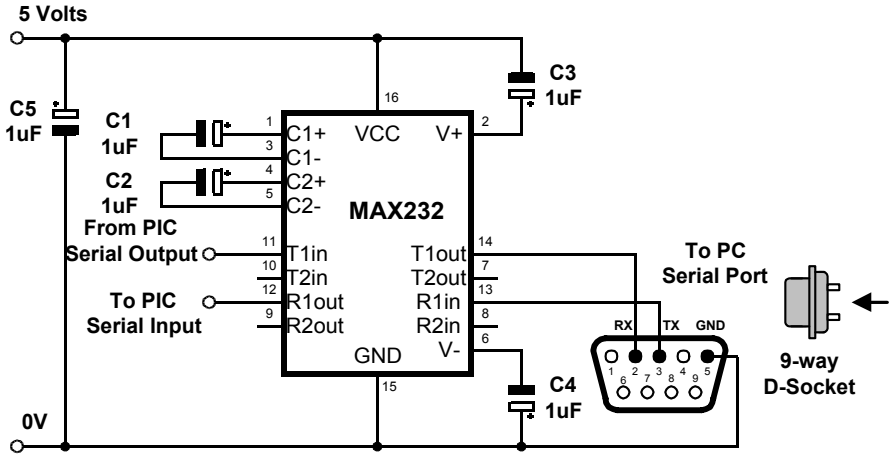
RS232 is the electrical specification for the signals that PC serial ports use. Unlike standard TTL logic, where 5 volts is a logic 1 and 0 volts is logic 0, RS232 uses -12 volts for logic 1 and +12 volts for logic 0. This specification allows communication over longer wire lengths without amplification.

Most circuits that work with RS232 use a line driver / receiver (transceiver). This component does two things:

Convert the ±12 volts of RS-232 to TTL compatible 0 to 5 volt levels.
Invert the voltage levels, so that 5 volts = logic 1 and 0 volts = logic 0.

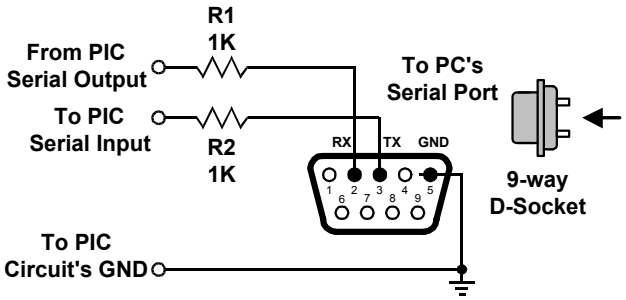
By far, the most common line driver device is the MAX232 from Maxim semiconductor. With the addition of a few capacitors, a complete 2-way level converter is realised. Figure 1 shows a typical circuit for one of these devices. The MAX232 is not the only device available, there are

other types that do not require any external capacitors at all. Visit Maxim's excellent web site at www.maxim.com, and download one of their many detailed datasheets.



Typical MAX232 RS232 line-transceiver circuit.

Because of the excellent IO capabilities of the microcontroller, and the adoption of TTL levels on most modern PC serial ports, a line driver is often unnecessary unless long distances are involved between the transmitter and the receiver. Instead a simple current limiting resistor is all that's required. As shown below:



Directly connected RS232 circuit.

You should remember that when using a line transceiver such as the MAX232, the serial mode (polarity) is inverted in the process of converting the signal levels, however, if using the direct connection, the mode is untouched. This is the single most common cause of errors when connecting serial devices, therefore you must make allowances for this within your software.

Asynchronous serial communication relies on precise timing. Both the sender and receiver must be set for identical timing, this is commonly expressed in bits per second (bps) called baud. **Serin** requires a value called Baudmode that informs it of the relevant characteristics of the incoming serial data; the bit period, number of data and parity bits, and polarity.

The Baudmode argument for **Serin** accepts a 16-bit value that determines its characteristics: 1-stop bit, 8-data bits/no-parity or 7-data bits/even-parity and most speeds from as low as 300 baud to 38400 baud (depending on the crystal frequency used). The following table shows how Baudmode is calculated, while table 1 shows some common baudmodes for standard serial baud rates.

Step 1.	Determine the bit period. (bits 0 – 11)	(1,000,000 / baud rate) – 20
Step 2.	data bits and parity. (bit 13)	8-bit/no-parity = step 1 + 0 7-bit/even-parity = step 1 + 8192
Step 3.	Select polarity. (bit 14)	True (noninverted) = step 2 + 0 Inverted = step 2 + 16384

Baudmode calculation.

Add the results of steps 1, 2 and 3 to determine the correct value for the Baudmode operator.

BaudRate	8-bit no-parity inverted	8-bit no-parity true	7-bit even-parity inverted	7-bit even-parity true
300	19697	3313	27889	11505
600	18030	1646	26222	9838
1200	17197	813	25389	9005
2400	16780	396	24972	8588
4800	16572	188	24764	8380
9600	16468	84	24660	8276

Table 1. Common baud rates and corresponding Baudmodes.

If communications are with existing software or hardware, its speed and mode will determine the choice of baud rate and mode. In general, 7-bit/even-parity (7E) mode is used for text, and 8-bit/no-parity (8N) for byte-oriented data. Note: the most common mode is 8-bit/no-parity, even when the data transmitted is just text. Most devices that use a 7-bit data mode do so in order to take advantage of the parity feature. Parity can detect some communication errors, but to use it you lose one data bit. This means that incoming data bytes transferred in 7E (even-parity) mode can only represent values from 0 to 127, rather than the 0 to 255 of 8N (no-parity) mode.

The compiler's serial commands **Serin** and **Serout**, have the option of still using a parity bit with 4 to 8 data bits. This is through the use of a **Declare**:

With parity disabled (the default setting):

```
Declare Serial_Data 4   ' Set Serin and Serout data bits to 4
Declare Serial_Data 5   ' Set Serin and Serout data bits to 5
Declare Serial_Data 6   ' Set Serin and Serout data bits to 6
Declare Serial_Data 7   ' Set Serin and Serout data bits to 7
Declare Serial_Data 8   ' Set Serin and Serout data bits to 8 (default)
```

With parity enabled:

```
Declare Serial_Data 5   ' Set Serin and Serout data bits to 4
Declare Serial_Data 6   ' Set Serin and Serout data bits to 5
Declare Serial_Data 7   ' Set Serin and Serout data bits to 6
Declare Serial_Data 8   ' Set Serin and Serout data bits to 7 (default)
Declare Serial_Data 9   ' Set Serin and Serout data bits to 8
```

Serial_Data data bits may range from 4 bits to 8 (the default if no **Declare** is issued). Enabling parity uses one of the number of bits specified.

Declaring Serial_Data as 9 allows 8 bits to be read and written along with a 9th parity bit.

Parity is a simple error-checking feature. When a serial sender is set for even parity (the mode the compiler supports) it counts the number of 1s in an outgoing byte and uses the parity bit to make that number even. For example, if it is sending the 7-bit value: %0011010, it sets the parity bit to 1 in order to make an even number of 1s (four).

The receiver also counts the data bits to calculate what the parity bit should be. If it matches the parity bit received, the serial receiver assumes that the data was received correctly. Of course, this is not necessarily true, since two incorrectly received bits could make parity seem correct when the data was wrong, or the parity bit itself could be bad when the rest of the data was correct.

Many systems that work exclusively with text use 7-bit/ even-parity mode. For example, to receive one data byte through bit-0 of PortA at 9600 baud, 7E, inverted:

Serin PortA.0, 24660, [SerData]

The above example will work correctly, however it doesn't inform the program what to do in the event of a parity error.

Below, is an improved version that uses the optional Plabel argument:

```
   Serin PortA.0, 24660, ParityError, [SerData]
   Print Dec SerData
   Stop
ParityError:
   Print "Parity Error"
   Stop
```

If the parity matches, the program continues at the **Print** instruction after **Serin**. If the parity doesn't match, the program jumps to the label ParityError. Note that a parity error takes precedence over other InputData specifications (as soon as an error is detected, **Serin** aborts and jumps to the Plabel routine).

In the examples above, the only way to end the **Serin** instruction (other than Reset or power-off) is to give **Serin** the serial data it needs. If no serial data arrives, the program is stuck in an endless loop. However, you can force **Serin** to abort if it doesn't receive data within a specified number of milliseconds.

For example, to receive a value through bit-0 of PortA at 9600 baud, 8N, inverted and abort **Serin** after 2 seconds (2000 ms) if no data arrives:

```
   Serin PortA.0, 16468, 2000, TimeoutError, [SerData]
   Print Cls, Dec Result
   Stop
TimeoutError:
   Print Cls, "Timed Out"
   Stop
```

If no serial data arrives within 2 seconds, **Serin** aborts and continues at the label TimeoutError.

Both Parity and Serial Timeouts may be combined. Below is an example to receive a value through bit-0 of PortA at 2400 baud, 7E, inverted with a 10-second timeout:


```
   Dim SerData as Byte
Again:
   Serin PortA.0, 24660, ParityError, 10000, TimeoutError, [SerData]
   Print Cls, Dec SerData
   GoTo Again
TimeoutError:
   Print Cls, "Timed Out"
   GoTo Again
ParityError:
   Print Cls, "Parity Error"
   GoTo Again
```

When designing an application that requires serial communication between microcontrollers, you should remember to work within these limitations:

When the microcontroller is sending or receiving data, it cannot execute other instructions.

When the microcontroller is executing other instructions, it cannot send or receive data.
The compiler does not offer a serial buffer as there is in PCs. At lower crystal frequencies, and higher serial rates, the microcontroller cannot receive data via **Serin**, process it, and execute another **Serin** in time to catch the next chunk of data, unless there are significant pauses between data transmissions.

These limitations can sometimes be addressed by using flow control; the Fpin option for **Serin** and **Serout**. Through Fpin, **Serin** can inform another microcontroller sender when it is ready to receive data. (Fpin flow control follows the rules of other serial handshaking schemes, however most computers other than the microcontroller cannot start and stop serial transmission on a byte-by-byte basis.)

Below is an example using flow control with data through bit-0 of PortA, and flow control through bit-1 of PortA, 9600 baud, N8, noninverted:

```
   Serin PortA.0\PortA.1, 84, [SerData]
```

When **Serin** executes, bit-0 of PortA (Rpin) is made an input in preparation for incoming data, and bit-1 of PortA (Fpin) is made an output low, to signal "go" to the sender. After **Serin** finishes receiving data, bit-1 of PortA is brought high to notify the sender to stop. If an inverted BaudMode had been specified, the Fpin's responses would have been reversed. The table below illustrates the relationship of serial polarity to Fpin states.

Serial Polarity	Ready to Receive ("Go")	Not Ready to Receive ("Stop")
Inverted	Fpin is High (1)	Fpin is Low (0)
Non-inverted	Fpin is Low (0)	Fpin is High (1)

See the following circuit for a flow control example using two 16F84 devices. In the demonstration program example, the sender transmits the whole word "HELLO!" in approx 6 ms. The receiver catches the first byte at most; by the time it got back from the first 1-second delay (**DelayMs** 1000), the rest of the data would be long gone. With flow control, communication is flawless since the sender waits for the receiver to catch up.


```
' Sender Code. Program into the Sender microcontroller.
Loop:
   Serout RA0\RA1, 16468, ["HELLO!"]    ' Send the message
   DelayMs 2500                         ' Delay for 2.5 seconds
   GoTo Loop                            ' Repeat the message forever

' Receiver Code. Program into the Receiver microcontroller.
   Dim Message as Byte
Again:
   Serin RA0\RA1, 16468, [Message]      ' Get 1 byte
   Print Message                        ' Display the byte on LCD.
   DelayMs 1000                         ' Delay for 1 second.
   GoTo Again                           ' Repeat forever
```

Serin Modifiers.

The **Serin** command can be configured to wait for a specified sequence of characters before it retrieves any additional input. For example, suppose a device attached to the microcontroller is known to send many different sequences of data, but the only data you wish to observe happens to appear right after the unique characters, “XYZ”. A modifier named Wait can be used for this purpose:

```
   Serin PortA.0, 16468, [Wait("XYZ"), SerData]
```

The above code waits for the characters “X”, “Y” and “Z” to be received, in that order, then it receives the next data byte and p[laces it into variable SerData.

The compiler also has a modifier for handling a string of characters, named **Str**.

The **Str** modifier is used for receiving a string of characters into a byte array variable.

A string is a set of characters that are arranged or accessed in a certain order. The characters "ABC" would be stored in a string with the "A" first, followed by the "B" then followed by the "C". A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string "ABC" would be stored in a byte array containing three bytes (elements).

Below is an example that receives ten bytes through bit-0 of PortA at 9600 bps, N81/inverted, and stores them in the 10-byte array, SerString:

```
Dim SerString[10] as Byte                ' Create a 10-byte array.
Serin PortA.0, 16468, [Str SerString]    ' Fill the array with data.
Print Str SerString                      ' Display the string.
```

If the amount of received characters is not enough to fill the entire array, then a formatter may be placed after the array's name, which will only receive characters until the specified length is reached. For example:

```
Dim SerString[10] as Byte                ' Create a 10-byte array.
Serin PortA.0, 16468, [Str SerString\5]  ' Fill first 5-bytes of array
Print Str SerString\5                    ' Display the 5-character string.
```

The example above illustrates how to fill only the first n bytes of an array, and then how to display only the first n bytes of the array. n refers to the value placed after the backslash.

Because of its complexity, serial communication can be rather difficult to work with at times. Using the guidelines below when developing a project using the **Serin** and **Serout** commands may help to eliminate some obvious errors:

Always build your project in steps.
Start with small, manageable pieces of code, (that deal with serial communication) and test them, one individually.
Add more and more small pieces, testing them each time, as you go.
Never write a large portion of code that works with serial communication without testing its smallest workable pieces first.
Pay attention to timing.
Be careful to calculate and overestimate the amount of time, operations should take within the microcontroller for a given oscillator frequency. Misunderstanding the timing constraints is the source of most problems with code that communicate serially. If the serial communication in your project is bi-directional, the above statement is even more critical.
Pay attention to wiring.
Take extra time to study and verify serial communication wiring diagrams. A mistake in wiring can cause strange problems in communication, or no communication at all. Make sure to connect the ground pins (Vss) between the devices that are communicating serially.
Verify port setting on the PC and in the* Serin */* Serout *commands.
Unmatched settings on the sender and receiver side will cause garbled data transfers or no data transfers. This is never more critical than when a line transceiver is used(i.e. MAX232). Always remember that a line transceiver inverts the serial polarity.

If the serial data received is unreadable, it is most likely caused by a baud rate setting error, or a polarity error. If receiving data from another device that is not a microcontroller, try to use baud rates of 9600 and below, or alternatively, use a higher frequency crystal.

Because of additional overheads in the microcontroller, and the fact that the **Serin** command offers no hardware receive buffer for serial communication, received data may sometimes be missed or garbled. If this occurs, try lowering the baud rate, or increasing the crystal frequency. Using simple variables (not arrays) will also increase the chance that the microcontroller will receive the data properly.

See also : **HRsin**, **HRsout**, **Hserin**, **Hserout**, **Rsin**, **Rsout**.

Serout

Syntax
Serout Tpin { \ Fpin }, Baudmode, { Pace, } { Timeout, Tlabel, } [OutputData]

Overview
Transmit asynchronous serial data (i.e. RS232 data).

Operators
- ***Tpin*** is a Port.Bit constant that specifies the I/O pin through which the serial data will be transmitted. This pin will be set to output mode while operating. The state of this pin when finished is determined by the driver bit in Baudmode.
- ***Fpin*** is an optional Port.Bit constant that specifies the I/O pin to monitor for flow control status. This pin will be set to input mode. Note: Fpin must be specified in order to use the optional Timeout and Tlabel operators in the **Serout** command.
- ***Baudmode*** may be a variable, constant, or expression (0 - 65535) that specifies serial timing and configuration.
- ***Pace*** is an optional variable, constant, or expression (0 - 65535) that determines the length of the delay between transmitted bytes. Note: Pace cannot be used simultaneously with Timeout.
- ***Timeout*** is an optional variable or constant (0 - 65535) that informs **Serout** how long to wait for Fpin permission to send. If permission does not arrive in time, the program will jump to the address specified by Tlable. Note: Fpin must be specified in order to use the optional Timeout and Tlabel operators in the **Serout** command.
- ***Tlabel*** is an optional label that must be provided along with Timeout. Tlabel indicates where the program should jump to in the event that permission to send data is not granted within the period specified by Timeout.
- ***OutputData*** is list of variables, constants, expressions and modifiers that informs **Serout** how to format outgoing data. **Serout** can transmit individual or repeating bytes, convert values into decimal, hex or binary text representations, or transmit strings of bytes from variable arrays, and **Cdata** constructs. These actions can be combined in any order in the OutputData list.

Notes
One of the most popular forms of communication between electronic devices is serial communication. There are two major types of serial communication; asynchronous and synchronous. The **Rsin**, **Rsout**, **Serin** and **Serout** commands are all used to send and receive asynchronous serial data. While the **Shin** and **Shout** commands are for use with synchronous communications.

The term asynchronous means 'no clock.' More specifically, 'asynchronous serial communication' means data is transmitted and received without the use of a separate 'clock' line. Data can be sent using as few as two wires; one for data and one for ground. The PC's serial ports (also called COM ports or RS232 ports) use asynchronous serial communication. Note: the other kind of serial communication, synchronous, uses at least three wires; one for clock, one for data and one for ground.

RS232 is the electrical specification for the signals that PC serial ports use. Unlike standard TTL logic, where 5 volts is a logic 1 and 0 volts is logic 0, RS232 uses -12 volts for logic 1 and +12 volts for logic 0. This specification allows communication over longer wire lengths without amplification.

Most circuits that work with RS232 use a line driver / receiver (transceiver). This component does two things:

Convert the ±12 volts of RS-232 to TTL compatible 0 to 5 volt levels.
Invert the voltage levels, so that 5 volts = logic 1 and 0 volts = logic 0.

By far, the most common line driver device is the MAX232 from MAXIM semiconductor. With the addition of a few capacitors, a complete 2-way level converter is realised (see **Serin** for circuit).

The MAX232 is not the only device available, there are other types that do not require any external capacitors at all. Visit Maxim's excellent web site at www.maxim.com <http://www.maxim.com>, and download one of their many detailed datasheets.

Because of the excellent IO capabilities of the microcontroller range of devices, and the adoption of TTL levels on most modern PC serial ports, a line driver is often unnecessary unless long distances are involved between the transmitter and the receiver. Instead a simple current limiting resistor is all that's required (see **Serin** for circuit).

You should remember that when using a line transceiver such as the MAX232, the serial mode (polarity) is inverted in the process of converting the signal levels, however, if using the direct connection, the mode is untouched. This is the single most common cause of errors when connecting serial devices, therefore you must make allowances for this within your software.

Asynchronous serial communication relies on precise timing. Both the sender and receiver must be set for identical timing, this is commonly expressed in bits per second (bps) called baud. **Serout** requires a value called Baudmode that informs it of the relevant characteristics of the incoming serial data; the bit period, number of data and parity bits, and polarity.

The Baudmode argument for **Serout** accepts a 16-bit value that determines its characteristics: 1-stop bit, 8-data bits/no-parity or 7-data bits/even-parity and virtually any speed from as low as 300 baud to 38400 baud (depending on the crystal frequency used). Table 2 below shows how Baudmode is calculated, while table 3 shows some common baudmodes for standard serial baud rates.

Step 1.	Determine the bit period. (bits 0 – 11)	(1,000,000 / baud rate) – 20
Step 2.	data bits and parity. (bit 13)	8-bit/no-parity = step 1 + 0 7-bit/even-parity = step 1 + 8192
Step 3.	Select polarity. (bit 14)	True (noninverted) = step 2 + 0 Inverted = step 2 + 16384

Baudmode calculation.

Add the results of steps 1, 2 3, and 3 to determine the correct value for the Baudmode operator

BaudRate	8-bit no-parity inverted	8-bit no-parity true	7-bit even-parity inverted	7-bit even-parity true
300	19697	3313	27889	11505
600	18030	1646	26222	9838
1200	17197	813	25389	9005
2400	16780	396	24972	8588
4800	16572	188	24764	8380
9600	16468	84	24660	8276

Note
For 'open' baudmodes used in networking, add 32768 to the values from the previous table.

If communications are with existing software or hardware, its speed and mode will determine the choice of baud rate and mode. In general, 7-bit/even-parity (7E) mode is used for text, and 8-bit/no-parity (8N) for byte-oriented data. Note: the most common mode is 8-bit/no-parity, even when the data transmitted is just text. Most devices that use a 7-bit data mode do so in order to take advantage of the parity feature. Parity can detect some communication errors, but to use it you lose one data bit. This means that incoming data bytes transferred in 7E (even-parity) mode can only represent values from 0 to 127, rather than the 0 to 255 of 8N (no-parity) mode.

The compiler's serial commands **Serout** and **Serin**, have the option of still using a parity bit with 4 to 8 data bits. This is through the use of a **Declare**:

With parity disabled (the default setting):

```
Declare Serial_Data 4   ' Set Serout and Serin data bits to 4
Declare Serial_Data 5   ' Set Serout and Serin data bits to 5
Declare Serial_Data 6   ' Set Serout and Serin data bits to 6
Declare Serial_Data 7   ' Set Serout and Serin data bits to 7
Declare Serial_Data 8   ' Set Serout and Serin data bits to 8 (default)
```

With parity enabled:

```
Declare Serial_Data 5   ' Set Serout and Serin data bits to 4
Declare Serial_Data 6   ' Set Serout and Serin data bits to 5
Declare Serial_Data 7   ' Set Serout and Serin data bits to 6
Declare Serial_Data 8   ' Set Serout and Serin data bits to 7 (default)
Declare Serial_Data 9   ' Set Serout and Serin data bits to 8
```

Serial_Data data bits may range from 4 bits to 8 (the default if no **Declare** is issued). Enabling parity uses one of the number of bits specified.

Declaring Serial_Data as 9 allows 8 bits to be read and written along with a 9th parity bit.

Parity is a simple error-checking feature. When the **Serout** command's Baudmode is set for even parity (compiler default) it counts the number of 1s in the outgoing byte and uses the parity bit to make that number even. For example, if it is sending the 7-bit value: %0011010, it sets the parity bit to 1 in order to make an even number of 1s (four).

The receiver also counts the data bits to calculate what the parity bit should be. If it matches the parity bit received, the serial receiver assumes that the data was received correctly. Of course, this is not necessarily true, since two incorrectly received bits could make parity seem correct when the data was wrong, or the parity bit itself could be bad when the rest of the data was correct. Parity errors are only detected on the receiver side.

Normally, the receiver determines how to handle an error. In a more robust application, the receiver and transmitter might be set up in such that the receiver can request a re-send of data that was received with a parity error.

Serout Modifiers.
The example below will transmit a single byte through bit-0 of PortA at 2400 baud, 8N1, inverted:

```
Serout PortA.0, 16780, [65]
```

In the above example, **Serout** will transmit a byte equal to 65 (the ASCII value of the character "A") through PortA.0. If the microcontroller was connected to a PC running a terminal program such as HyperTerminal set to the same baud rate, the character "A" would appear on the screen. Always remembering that the polarity will differ if a line transceiver such as the MAX232 is used.

What if you wanted the value 65 to appear on the PC's screen? As was stated earlier, it is up to the receiving side (in serial communication) to interpret the values. In this case, the PC is interpreting the byte-sized value to be the ASCII code for the character "A". Unless you're also writing the software for the PC, you cannot change how the PC interprets the incoming serial data, therefore to solve this problem, the data needs to be translated before it is sent.

The **Serout** command provides a modifier which will translate the value 65 into two ASCII codes for the characters "6" and "5" and then transmit them:

```
Serout PortA.0, 16780, [Dec 65]
```

Notice that the decimal modifier in the **Serout** command is the word **Dec**, this instructs the **Serout** to convert the number into separate ASCII characters which represent the value in decimal form. If the value 65 in the code were changed to 123, the **Serout** command would send three bytes (49, 50 and 51) corresponding to the characters "1", "2" and "3".

This is exactly the same modifier that is used in the **Rsout** and **Print** commands.

As well as the **Dec** modifier, **Serout** may use **Hex**, or **Bin** modifiers, again, these are the same as used in the **Rsout** and **Print** commands. Therefore, please refer to the **Rsout** or **Print** command descriptions for an explanation of these. The **Serout** command sends quoted text exactly as it appears in the OutputData list:

```
Serout PortA.0, 16780, ["HELLO WORLD", 13]
Serout PortA.0, 16780, ["Num = ", Dec 100]
```

The above code will display "HELLO WORLD" on one line and "Num = 100" on the next line. Notice that you can combine data to output in one **Serout** command, separated by commas. In the example above, we could have written it as one line of code:

```
Serout PortA.0, 16780, ["HELLO WORLD", 13, "Num = ", Dec 100]
```


Serout also has some other modifiers. These are listed below:

Modifier	Operation
Bin{1..32}	Send binary digits
Dec{1..10}	Send decimal digits
Hex{1..8}	Send hexadecimal digits
Sbin{1..32}	Send signed binary digits
Sdec{1..10}	Send signed decimal digits
Shex{1..8}	Send signed hexadecimal digits
Ibin{1..32}	Send binary digits with a preceding '%' identifier
Idec{1..10}	Send decimal digits with a preceding '#' identifier
Ihex{1..8}	Send hexadecimal digits with a preceding '$' identifier
ISbin{1..32}	Send signed binary digits with a preceding '%' identifier
ISdec{1..10}	Send signed decimal digits with a preceding '#' identifier
IShex{1..8}	Send signed hexadecimal digits with a preceding '$' identifier
Rep c\n	Send character c repeated n times
Str array\n	Send all or part of an array
Cstr cdata	Send string data defined in a **Cdata** statement.

If a floating point variable is to be displayed, then the digits after the **Dec** modifier determine how many remainder digits are printed. i.e. numbers after the decimal point.

```
Dim FloatVar as Float
FloatVar = 3.145
Serout PortA.0, 16780, [Dec2 FloatVar]   ' Send 2 values after decimal point
```

The above program will send 3.14

If the digit after the **Dec** modifier is omitted, then 3 values will be displayed after the decimal point.

```
Dim FloatVar as Float
FloatVar = 3.1456
Serout PortA.0, 16780, [Dec FloatVar]    ' Send 3 values after decimal point
```

The above program will send 3.145

There is no need to use the **Sdec** modifier for signed floating point values, as the compiler's **Dec** modifier will automatically display a minus result:

```
Dim FloatVar as Float
FloatVar = -3.1456
Serout PortA.0, 16780, [Dec FloatVar]    ' Send 3 values after decimal point
```

The above program will send -3.145

Hex or **Bin** modifiers cannot be used with floating point values or variables.

Using Strings with Serout.

The **Str** modifier is used for transmitting a string of characters from a byte array variable. A string is a set of characters that are arranged or accessed in a certain order. The characters "ABC" would be stored in a string with the "A" first, followed by the "B" then followed by the "C". A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string "ABC" would be stored in a byte array containing three bytes (elements).

Below is an example that transmits five bytes (from a byte array) through bit-0 of PortA at 9600 bps, N81/inverted:

```
Dim SerString[10] as Byte       ' Create a 10-byte array.
SerString[0] = "H"              ' Load the first 5 bytes of the array
SerString[1] = "E"              ' With the word "HELLO"
SerString[2] = "L"
SerString[3] = "L"
SerString[4] = "O"
Serout PortA.0, 16468, [Str SerString\5]    ' Send 5-byte string.
```

Note that we use the optional \n argument of **Str**. If we didn't specify this, the microcontroller would try to keep sending characters until all 10 bytes of the array were transmitted, or it found a byte equal to 0 (a null terminator). Since we didn't specify a last byte of 0 in the array, and we do not wish the last five bytes to be transmitted, we chose to tell it explicitly to only send the first 5 characters.

The above example may also be written as:

```
Dim SerString[10] as Byte                ' Create a 10-byte array.
Str SerString = "HELLO" ,0               ' Load the first 6 bytes of the array
Serout PortA.0, 16468, [Str SerString]   ' Send first 5-bytes of string.
```

In the above example, we specifically added a null terminator to the end of the string (a zero). Therefore, the **Str** modifier within the **Serout** command will output data until this is reached. An alternative to this would be to create the array exactly the size of the text. In our example, the array would have been 5 elements in length.

Another form of string is used by the **Cstr** modifier. Note: Because this uses the **Cdata** command to create the individual elements it is only for use with PICs that support self-modifying features, such as the 16F87X, and 18XXXX range of devices.

Below is an example of using the **Cstr** modifier. It's function is the same as the above examples, however, no RAM is used for creating arrays.

```
Serout PortA.0, 16468, [Cstr SerString]

SerString: Cdata "HELLO", 0
```

The **Cstr** modifier will always be terminated by a null (i.e. zero at the end of the text or data). If the null is omitted, then the **Serout** command will continue transmitting characters forever.

The **Serout** command can also be configured to pause between transmitted bytes. This is the purpose of the optional Pace operator. For example (9600 baud N8, inverted):


```
Serout PortA.0, 16468, 1000, ["Send this message Slowly"]
```

Here, the microcontroller transmits the message "Send this message Slowly" with a 1 second delay between each character.

A good reason to use the Pace feature is to support devices that require more than one stop bit. Normally, the microcontroller sends data as fast as it can (with a minimum of 1 stop bit between bytes). Since a stop bit is really just a resting state in the line (no data transmitted), using the Pace option will effectively add multiple stop bits. Since the requirement for 2 or more stop bits (on some devices) is really just a minimum requirement, the receiving side should receive this data correctly.

Serout Flow Control.
When designing an application that requires serial communication between PICs, you need to work within these limitations:

When the microcontroller is sending or receiving data, it cannot execute other instructions.
When the microcontroller is executing other instructions, it cannot send or receive data.
The compiler does not offer a serial buffer as there is in PCs. At lower crystal frequencies, and higher serial rates, the microcontroller cannot receive data via **Serin**, process it, and execute another **Serin** in time to catch the next chunk of data, unless there are significant pauses between data transmissions.

These limitations can sometimes be addressed by using flow control; the Fpin option for **Serout** and **Serin**. Through Fpin, **Serin** can inform another microcontroller sender when it is ready to receive data and **Serout** (on the sender) will wait for permission to send. Fpin flow control follows the rules of other serial handshaking schemes, however most computers other than the microcontroller cannot start and stop serial transmission on a byte-by-byte basis. That is why this discussion is limited to communication between PICmicros.

Below is an example using flow control with data through bit-0 of PortA, and flow control through bit-1 of PortA, 9600 baud, N8, noninverted:

```
Serout PortA.0\PortA.1, 84, [SerData]
```

When **Serin** executes, bit-0 of PortA (Tpin) is made an output in preparation for sending data, and bit-1 of PortA (Fpin) is made an input, to wait for the "go" signal from the receiver. The table below illustrates the relationship of serial polarity to Fpin states.

Serial Polarity	Ready to Receive ("Go")	Not Ready to Receive ("Stop")
Inverted	Fpin is High (1)	Fpin is Low (0)
Non-inverted	Fpin is Low (0)	Fpin is High (1)

See **Serin** for a flow control circuit.

The **Serout** command supports open-drain and open-source output, which makes it possible to network multiple PICs on a single pair of wires. These 'open baudmodes' only actively drive the Tpin in one state (for the other state, they simply disconnect the pin; setting it to an input mode). If two PICs in a network had their **Serout** lines connected together (while a third device listened on that line) and the PICs were using always-driven baudmodes, they could simultaneously output two opposite states (i.e. +5 volts and ground). This would create a short circuit. The heavy current flow would likely damage the I/O pins or the PICs themselves.

Since the open baudmodes only drive in one state and float in the other, there's no chance of this kind of short happening.

The polarity selected for **Serout** determines which state is driven and which is open as shown in the table below.

Serial Polarity	State(0)	State(1)	Resistor Pulled to:
Inverted	Open	Driven	Gnd (Vss)
Non-inverted	Driven	Open	+5V (Vdd)

Since open baudmodes only drive to one state, they need a resistor to pull the networked line into the opposite state, as shown in the above table and in the circuits below. Open baudmodes allow the micro-controller to share a line, however it is up to your program to resolve other networking issues such as who talks when, and how to detect, prevent and fix data errors.

See also : **Rsin**, **Rsout**, **HRsin**, **HRsout**, **Hserin**, **Hserout**, **Serin**.

Servo

Syntax
Servo Pin, Rotation Value

Overview
Control a remote control type servo motor.

Operators
- ***Pin*** is a Port.Pin constant that specifies the I/O pin for the attachment of the motor's control terminal.
- ***Rotation Value*** is a 16-bit (0-65535) constant or Word variable that dictates the position of the motor. A value of approx 500 being a rotation to the farthest position in a direction and approx 2500 being the farthest rotation in the opposite direction. A value of 1500 would normally centre the servo but this depends on the motor type.

Example
```
' Control a servo motor attached to pin 3 of PortA

   Dim Pos as Word               ' Servo Position
   Symbol Pin = PortA.3          ' Alias the servo pin
   Pos = 1500                    ' Centre the servo
   PortA = 0                     ' PortA lines low to read buttons
   TrisA = %00000111             ' Enable the button pins as inputs

' Check any button pressed to move servo
Main:
   If PortA.0 = 0 And Pos < 3000 Then Pos = Pos + 1  ' Move servo left
   If PortA.1 = 0 Then Pos = 1500                    ' Centre servo
   If PortA.2 = 0 And Pos > 0 Then Pos = Pos - 1     ' Move servo right
   Servo Pin, Pos
   DelayMs 5                                         ' Servo update rate
   Hrsout "Position=", Dec Pos, 13
   GoTo Main
```

Notes
Servos of the sort used in radio-controlled models are finding increasing applications in this robotics age we live in. They simplify the job of moving objects in the real world by eliminating much of the mechanical design. For a given signal input, you get a predictable amount of motion as an output.

To enable a servo to move it must be connected to a 5 Volt power supply capable of delivering an ampere or more of peak current. It then needs to be supplied with a positioning signal. The signal is normally a 5 Volt, positive-going pulse between 1 and 2 milliseconds (ms) long, repeated approximately 50 times per second.

The width of the pulse determines the position of the servo. Since a servo's travel can vary from model to model, there is not a definite correspondence between a given pulse width and a particular servo angle, however most servos will move to the centre of their travel when receiving 1.5ms pulses.

Servos are closed-loop devices. This means that they are constantly comparing their commanded position (proportional to the pulse width) to their actual position (proportional to the resistance of an internal potentiometer mechanically linked to the shaft). If there is more than a small difference between the two, the servo's electronics will turn on the motor to eliminate the error. In addition to moving in response to changing input signals, this active error correction means that servos will resist mechanical forces that try to move them away from a commanded position. When the servo is unpowered or not receiving positioning pulses, the output shaft may be easily turned by hand. However, when the servo is powered and receiving signals, it won't move from its position.

Driving servos with the compiler is extremely easy. The **Servo** command generates a pulse in 1 microsecond (µs) units, so the following code would command a servo to its centred position and hold it there:

```
Again:
   Servo PortA.0, 1500
   DelayMs 20
   GoTo Again
```

The 20ms delay ensures that the program sends the pulse at the standard 50 pulse-per-second rate. However, this may be lengthened or shortened depending on individual motor characteristics.

The **Servo** command is oscillator independent and will always produce 1us pulses regardless of the crystal frequency used.

See also : **PulseOut**.

SetBit

Syntax
SetBit Variable, Index

Overview
Set a bit of a variable or register using a variable index to the bit of interest.

Operators
- ***Variable*** is a user defined variable, of type Byte, Word, or Dword.
- ***Index*** is a constant, variable, or expression that points to the bit within Variable that requires setting.

Example
```
' Clear then Set each bit of variable ExVar
  Dim ExVar as Byte
  Dim Index as Byte
  ExVar = %11111111
  For Index = 0 to 7              ' Create a loop for 8 bits
    ClearBit ExVar,Index          ' Clear each bit of ExVar
    Hrsout Bin8 ExVar  , 13       ' Display the binary result
    DelayMs 100                   ' Slow things down to see what's happening
  Next                            ' Close the loop
  Hrsout 13
  For Index = 7 to 0 Step -1      ' Create a loop for 8 bits
    SetBit ExVar,Index            ' Set each bit of ExVar
    Hrsout Bin8 ExVar  , 13       ' Display the binary result
    DelayMs 100                   ' Slow things down to see what's happening
  Next                            ' Close the loop
  Hrsout 13
```

Notes
There are many ways to set a bit within a variable, however, each method requires a certain amount of manipulation, either with rotates, or alternatively, the use of indirect addressing using the FSR, and INDF registers. Each method has its merits, but requires a certain amount of knowledge to accomplish the task correctly. The **SetBit** command makes this task extremely simple using a register rotate method, however, this is not necessarily the quickest method, or the smallest, but it is the easiest. For speed and size optimisation, there is no shortcut to experience.

To Set a known constant bit of a variable or register, then access the bit directly using Port.n.

```
  PortA.1 = 1
```
or
```
  Var1.4 = 1
```

If a Port is targeted by **SetBit**, the Tris register is not affected.

See also : **ClearBit**, **GetBit**, **LoadBit**.

Set

Syntax
Set Variable or Variable.Bit

Overview
Place a variable or bit in a high state. For a variable, this means setting all the bits to 1. For a bit this means setting it to 1.

Operators
- ***Variable*** can be any variable or register.
- ***Variable.Bit*** can be any variable and bit combination.

Example
```
Set Var1.3      ' Set bit 3 of Var1
Set Var1        ' Load Var1 with the value of 255
Set STATUS.0    ' Set the carry flag high
Set Array       ' Set all of an Array variable. i.e. set to 255 or 65535
Set String1     ' Set all of a String variable. i.e. set to spaces (ASCII 32)
Set             ' Load all RAM with 255
```

Notes
Set does not alter the Tris register if a Port is targeted.
If no variable follows the **Set** command then all user RAM will be loaded with the value 255.

See also : **Clear, High, Low.**

Shin

Syntax
Shin dpin, cpin, mode, [result { \bits } { ,result { \bits }...}]

or

Var = **Shin** dpin, cpin, mode, shifts

Overview
Shift data in from a synchronous-serial device.

Operators
- ***Dpin*** is a Port.Pin constant that specifies the I/O pin that will be connected to the synchronous-serial device's data output. This pin's I/O direction will be changed to input and will remain in that state after the instruction is completed.
- ***Cpin*** is a Port.Pin constant that specifies the I/O pin that will be connected to the synchronous-serial device's clock input. This pin's I/O direction will be changed to output.
- ***Mode*** is a constant that tells **Shin** the order in which data bits are to be arranged and the relationship of clock pulses to valid data. Below are the symbols, values, and their meanings:

Symbol	Value	Description
MSBPRE MSBPRE_L	0	Shift data in highest bit first. Read data before sending clock. Clock idles low
LSBPRE LSBPRE_L	1	Shift data in lowest bit first. Read data before sending clock. Clock idles low
MSBPOST MSBPOST_L	2	Shift data in highest bit first. Read data after sending clock. Clock idles low
LSBPOST LSBPOST_L	3	Shift data in highest bit first. Read data after sending clock. Clock idles low
MSBPRE_H	4	Shift data in highest bit first. Read data before sending clock. Clock idles high
LSBPRE_H	5	Shift data in lowest bit first. Read data before sending clock. Clock idles high
MSBPOST_H	6	Shift data in highest bit first. Read data after sending clock. Clock idles high
LSBPOST_H	7	Shift data in lowest bit first. Read data after sending clock. Clock idles high

Result is a bit, byte, or word variable in which incoming data bits will be stored.

Bits is an optional constant specifying how many bits (1-16) are to be input by **Shin**. If no bits entry is given, **Shin** defaults to 8 bits.
Shifts informs the **Shin** command as to how many bit to shift in to the assignment variable, when used in the inline format.

Notes
Shin provides a method of acquiring data from synchronous-serial devices, without resorting to the hardware SPI modules resident on some microcontroller types. Data bits may be valid after the rising or falling edge of the clock line. This kind of serial protocol is commonly used by controller peripherals such as ADCs, DACs, clocks, memory devices, etc.

The **Shin** instruction causes the following sequence of events to occur:

- Makes the clock pin (cpin) output low.
- Makes the data pin (dpin) an input.
- Copies the state of the data bit into the msb (lsb-modes) or lsb (msb modes) either before (-pre modes) or after (-post modes) the clock pulse.
- Pulses the clock pin high.
- Shifts the bits of the result left (msb- modes) or right (lsb-modes).
- Repeats the appropriate sequence of getting data bits, pulsing the clock pin, and shifting the result until the specified number of bits is shifted into the variable.

Making **Shin** work with a particular device is a matter of matching the mode and number of bits to that device's protocol. Most manufacturers use a timing diagram to illustrate the relationship of clock and data.

```
Symbol CLK = PortB.0
Symbol DTA = PortB.1
Shin DTA, CLK, MSBPRE, [Var1]     ' Shiftin msb-first, pre-clock.
```

In the above example, both **Shin** instructions are set up for msb-first operation, so the first bit they acquire ends up in the msb (leftmost bit) of the variable.

The post-clock Shift in, acquires its bits after each clock pulse. The initial pulse changes the data line from 1 to 0, so the post-clock Shiftin returns %01010101.

By default, **Shin** acquires eight bits, but you can set it to shift any number of bits from 1 to 16 with an optional entry following the variable name. In the example above, substitute this for the first **Shin** instruction:

```
Shin DTA, CLK, MSBPRE, [Var1\4]   ' Shift in 4 bits.
```

Some devices return more than 16 bits. For example, most 8-bit shift registers can be daisy-chained together to form any multiple of 8 bits; 16, 24, 32, 40... You can use a single **Shin** instruction with multiple variables.
Each variable can be assigned a particular number of bits with the
backslash (\) option. Modify the previous example:

```
' 5 bits into Var1; 8 bits into Var2.
Shin DTA, CLK, MSBPRE, [Var1\5, Var2]
Hrsout "1st variable: ", Bin8 Var1, 13
Hrsout "2nd variable: ", Bin8 Var2, 13
```

Inline Shin Command.
The structure of the inline **Shin** command is:

```
Var = Shin dpin, cpin, mode, shifts
```

DPin, CPin, and Mode have not changed in any way, however, the inline structure has a new operand, namely Shifts. This informs the **Shin** command as to how many bit to shift in to the assignment variable. For example, to shift in an 8-bit value from a serial device, we would use:

```
Var1 = Shin DT, CK, MSBPRE, 8
```

To shift 16-bits into a Word variable:

```
WordVar = Shin DT, CK, MSBPRE, 16
```


Shout

Syntax
Shout Dpin, Cpin, Mode, [OutputData {\Bits} {,OutputData {\Bits}..}]

Overview
Shift data out to a synchronous serial device.

Operators
- ***Dpin*** is a Port.Pin constant that specifies the I/O pin that will be connected to the synchronous serial device's data input. This pin will be set to output mode.
- ***Cpin*** is a Port.Pin constant that specifies the I/O pin that will be connected to the synchronous serial device's clock input. This pin will be set to output mode.
- ***Mode*** is a constant that tells **Shout** the order in which data bits are to be arranged. Below are the symbols, values, and their meanings:

Symbol	Value	Description
LSBFIRST LSBFIRST _L	0	Shift data out lowest bit first. Clock idles low
MSBFIRST MSBFIRST_L	1	Shift data out highest bit first. Clock idles low
LSBFIRST _H	4	Shift data out lowest bit first. Clock idles high
MSBFIRST_H	5	Shift data out highest bit first. Clock idles high

- ***OutputData*** is a variable, constant, or expression containing the data to be sent.
- ***Bits*** is an optional constant specifying how many bits are to be output by **Shout**. If no Bits entry is given, **Shout** defaults to 8 bits.

Notes
Shin and **Shout** provide a method of acquiring data from synchronous serial devices. Data bits may be valid after the rising or falling edge of the clock line. This kind of serial protocol is commonly used by controller peripherals like ADCs, DACs, clocks, memory devices, etc.

At their heart, synchronous-serial devices are essentially shift-registers; trains of flip flops that receive data bits in a bucket brigade fashion from a single data input pin. Another bit is input each time the appropriate edge (rising or falling, depending on the device) appears on the clock line.

The **Shout** instruction first causes the clock pin to output low and the data pin to switch to output mode. **Then**, **Shout** sets the data pin to the next bit state to be output and generates a clock pulse. **Shout** continues to generate clock pulses and places the next data bit on the data pin for as many data bits as are required for transmission.

Making **Shout** work with a particular device is a matter of matching the mode and number of bits to that device's protocol. Most manufacturers use a timing diagram to illustrate the relationship of clock and data. One of the most important items to look for is which bit of the data should be transmitted first; most significant bit (MSB) or least significant bit (LSB).

Example

```
Shout DTA, CLK, MSBFIRST, [ 250 ]
```

In the above example, the **Shout** command will write to I/O pin DTA (the Dpin) and will generate a clock signal on I/O CLK (the Cpin). The **Shout** command will generate eight clock pulses while writing each bit (of the 8-bit value 250) onto the data pin (Dpin). In this case, it will start with the most significant bit first as indicated by the Mode value of MSBFIRST.

By default, **Shout** transmits eight bits, but you can set it to shift any number of bits from 1 to 16 with the Bits argument. For example:

```
Shout DTA, CLK, MSBFIRST, [250\4]
```

Will only output the lowest 4 bits (%0000 in this case). Some devices require more than 16 bits. To solve this, you can use a single **Shout** command with multiple values. Each value can be assigned a particular number of bits with the Bits argument. As in:

```
Shout DTA, CLK, MSBFIRST, [250\4, 1045\16]
```

The above code will first shift out four bits of the number 250 (%1111) and then 16 bits of the number 1045 (%0000010000010101). The two values together make up a 20 bit value.

See also : **Shin**.

Snooze

Syntax
Snooze Period

Overview
Enter sleep mode for a short period. Power consumption is reduced to approx 50 µA assuming no loads are being driven.

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

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

Example
```
Snooze 6     ' Low power mode for approx 1.152 seconds
```

Notes
Snooze intervals are directly controlled by the watchdog timer without compensation. Variations in temperature, supply voltage, and manufacturing tolerance of the microcontroller chip you are using can cause the actual timing to vary by as much as -50, +100 percent

See also : **Sleep**.

Sleep

Syntax
Sleep { Length }

Overview
Places the microcontroller into low power mode for approx n seconds. i.e. power down but leaves the port pins in their previous states.

Operator
- ***Length*** is an optional variable or constant (1-65535) that specifies the duration of sleep in seconds. If ***length*** is omitted, then the **Sleep** command is assumed to be the assembler mnemonic, which means the microcontroller will sleep continuously, or until an internal or external event awakes it.

Example
```
   Symbol LED = RA0
Again:
   High LED        ' Turn LED on.
   DelayMs 1000    ' Wait 1 second.
   Low LED         ' Turn LED off.
   Sleep 60        ' Sleep for 1 minute.
   GoTo Again
```

Notes
Sleep will place the microcontroller into a low power mode for the specified period of seconds. Period is 16 bits, so delays of up to 65,535 seconds are the limit (a little over 18 hours) **Sleep** uses the Watchdog Timer so it is independent of the oscillator frequency. The smallest units is about 2.3 seconds and may vary depending on specific environmental conditions and the device used.

The **Sleep** command is used to put the microcontroller in a low power mode without resetting the registers. Allowing continual program execution upon waking up from the **Sleep** period.

The microcontroller has the ability to be placed into a continual low power mode, consuming micro Amps of current.

The command for doing this is **Sleep**. The compiler's **Sleep** command or the assembler's **Sleep** instruction may be used. The compiler's **Sleep** command differs somewhat to the assembler's in that the compiler's version will place the microcontroller into low power mode for n seconds (where n is a value from 0 to 65535). The assembler's version still places the microcontroller into low power mode, however, it does this forever, or until an internal or external source wakes it. This same source also wakes the microcontroller when using the compiler's command.

Many things can wake the microcontroller from its sleep, the WatchDog Timer is the main cause and is what the compiler's **Sleep** command uses.

Another method of waking the microcontroller is an external one, a change on one of the port pins. We will examine more closely the use of an external source. There are two main ways of waking the microcontroller using an external source. One is a change on bits 4..7 of PortB.

Four of the PortB pins (RB<7:4>) are individually configurable as interrupt-on-change pins. Control bits in the IOCB register enable (when set) or disable (when clear) the interrupt function for each pin. When set, the RBIE bit of the INTCON register enables interrupts on all pins which also have their corresponding IOCB bit set. When clear, the RBIE bit disables all interrupt-on-changes. Only pins configured as inputs can cause this interrupt to occur (i.e., any RB<7:4> pin configured as an output is excluded from the interrupt-on-change comparison).

For enabled interrupt-on-change pins, the values are compared with the old value latched on the last read of PortB. The 'mismatch' outputs of the last read are OR'd together to set the PortB Change Interrupt flag bit (RBIF) in the INTCON register.

This interrupt can wake the device from the Sleep mode, or any of the Idle modes. The user, in the Interrupt Service Routine, can clear the interrupt in the following manner:

- Any read or write of PortB to clear the mismatch condition (except when PortB is the source or destination of a **movff** instruction).
- Clear the flag bit, RBIF.

A mismatch condition will continue to set the RBIF flag bit. Reading or writing PortB will end the mismatch condition and allow the RBIF bit to be cleared. The latch holding the last read value is not affected by a MCLR nor Brown-out Reset. After either one of these Resets, the RBIF flag will continue to be set if a mismatch is present.

The interrupt-on-change feature is recommended for wake-up on key depression operation and operations where PortB is only used for the interrupt-on-change feature. Polling of PortB is not recommended while using the interrupt-on-change feature.

```
Symbol LED = RB0              ' Assign the LED's pin to bit-0 of PortB
INTCONbits_GIE = 0            ' Turn Off global interrupts
Input RB4                     ' Set bit-4 of PortB (RB4) as an Input
PortB_Pullups = On            ' Enable PortB Pull-up Resistors
INTCONbits_RBIE = 1           ' Enable PortB[4..7] interrupt
IOCBbits_IOCB4 = 1            ' Enable pin-4 of PortB for interrupt source

While 1 = 1                   ' Create an infinite loop
  DelayMs 100                 ' Delay for 100ms
  Low LED                     ' Turn off the LED
  INTCONbits_RBIF = 0         ' Clear the PortB[4..7] interrupt flag
  Sleep                       ' Put the PICmicro to sleep
  DelayMs 100                 ' When it wakes up, delay for 100ms
  High LED                    ' Then light the LED
Wend                          ' Do it forever
```


SonyIn

Syntax
Variable = **SonyIn**

Overview
Receive Sony SIRC (Sony Infrared Remote Control) data from a predetermined pin. The pin is automatically made an input.

Operator
- ***Variable*** is a bit, byte, word, dword, or float variable, that will be loaded by **SonyIn**. The return data from the **SonyIn** command consists of two bytes, the *System* byte containing the type of remote used. i.e. TV, Video etc, and the *Command* byte containing the actual button value. The order of the bytes is *Command* (low byte) then System (high byte). If a byte variable is used to receive data from the infrared sensor then only the *Command* byte will be received.

Example
```
' Receive Sony SIRC data from an infrared sensor attached to PortC.0
  Declare SonyIn_Pin = PortC.0    ' Choose port and pin for infrared sensor
  Dim SonyIn_Word as Word         ' Create a Word variable to receive the SIRC data
' Alias the Command byte to SonyIn_Word low byte
  Dim SonyCommand as SonyIn_Word.Lowbyte
' Alias the Command byte to SonyIn_Word high byte
  Dim SonySystem as SonyIn_Word.Highbyte

  While 1 = 1                     ' Create an infinite loop
    Repeat
      SonyIn_Word = SonyIn        ' Receive a signal from the infrared sensor
    Until SonyCommand <> 255      ' Keep looking until a valid header found
     Hrsout "System  ", Dec SonySystem, 13   ' Display the System value
    Hrsout "Command ", Dec SonyCommand, 13 ' Display the Command value
  Wend
```

There is a single **Declare** for use with **SonyIn**:

Declare SonyIn_Pin Port . Pin
Assigns the Port and Pin that will be used to input infrared data by the **SonyIn** command. This may be any valid port on the microcontroller.

If the **Declare** is not used in the program, then the default Port and Pin is PortB.0.

Notes
The **SonyIn** command will return with both Command and System bytes containing 255 if a valid header was not received. The CARRY (STATUS.0) flag will also be set if an invalid header was received. This is an ideal method of determining if the signal received is of the correct type.

Sound

Syntax
Sound Pin, [Note,Duration {,Note,Duration...}]

Overview
Generates tone and/or white noise on the specified Pin. Pin is automatically made an output.

Operators
- ***Pin*** is a Port.Pin constant that specifies the output pin on the microcontroller.
- ***Note*** can be an 8-bit variable or constant. 0 is silence. Notes 1-127 are tones. Notes 128-255 are white noise. Tones and white noises are in ascending order (i.e. 1 and 128 are the lowest frequencies, 127 and 255 are the highest). Note 1 is approx 78.74Hz and Note 127 is approx 10,000Hz.
- ***Duration*** can be an 8-bit variable or constant that determines how long the Note is played in approx 10ms increments.

Example
```
' Star Trek The Next Generation...Theme and ship take-off
  Dim Loop as Byte
  Symbol Pin = PortB.0

Theme:
  Sound Pin,_
          [50,60,70,20,85,120,83,40,70,20,50,20,70,20,90,120,90,20,98,160]
  DelayMs 500
  For Loop = 128 to 255         ' Ascending white noises
    Sound Pin, [Loop,2]         ' For warp drive sound
  Next
  Sound Pin, [43,80,63,20,77,20,71,80,51,20,_
              90,20,85,140,77,20,80,20,85,20,_
             90,20,80,20,85,60,90,60,92,60,87,_
             60,96,70,0,10,96,10,0,10,96,10,0,_
             10,96,30,0,10,92,30,0,10,87,30,0,_
             10,96,40,0,20,63,10,0,10,63,10,0,_
             10,63,10,0,10,63,20]
  DelayMs 10000
  GoTo Theme
```

Notes
With the excellent I/O characteristics of the microcontroller, a speaker can be driven through a capacitor directly from the pin of the microcontroller. The value of the capacitor should be determined based on the frequencies of interest and the speaker load. Piezo speakers can be driven directly.

See also : **FreqOut**, **DTMFout**, **Sound2**.

Sound2

Syntax
Sound2 Pin2, Pin2, [Note1\Note2\Duration {,Note1,Note2\Duration...}]

Overview
Generate specific notes on each of the two defined pins. With the **Sound2** command more complex notes can be played by the microcontroller.

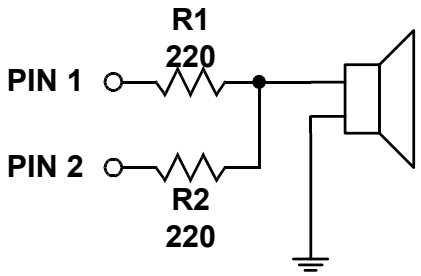
Operators
- ***Pin1*** and ***Pin2*** are Port.Pin constants that specify the output pins on the microcontroller.
- ***Note1*** is a variable or constant specifying frequency in Hertz (Hz, 0 to 16000) of a tone.
- ***Note2*** is a variable or constant specifying frequency in Hertz (Hz, 0 to 16000) of a tone.
- ***Duration*** can be a variable or constant that determines how long the Notes are played. In approx 1ms increments (0 to 65535).

Example 1
```
' Generate a 2500Hz tone and a 3500Hz tone for 1 second.
' The 2500Hz note is played from the first pin specified (PortB.0),
' and the 3500Hz note is played from the second pin specified (PortB.1).
  Symbol Pin1 = PortB.0
  Symbol Pin2 = PortB.1
  Sound2 Pin1, Pin2, [2500\3500\1000]
  Stop
```

Example 2
```
' Play two sets of notes 2500Hz and 3500Hz for 1 second
' and the second two notes, 500Hz and 1500Hz for 2 seconds.
  Symbol Pin1 = PortB.0
  Symbol Pin2 = PortB.1
  Sound2 Pin1, Pin2, [2500\3500\1000, 500\1500\2000]
  Stop
```

Notes
Sound2 generates two pulses at the required frequency one on each pin specified. The **Sound2** command can be used to play tones through a speaker or audio amplifier. **Sound2** can also be used to play more complicated notes. By generating two frequencies on separate pins, a more defined sound can be produced. **Sound2** is somewhat dependent on the crystal frequency used for its note frequency, and duration.

Sound2 does not require any filtering on the output, and produces a cleaner note than **FreqOut**. However, unlike **FreqOut**, the note is not a Sine wave. See diagram:



See also : **FreqOut**, **DTMFout**, **Sound**.

Stop

Syntax
Stop

Overview
Stop halts program execution by sending the microcontroller into an infinite loop.

Example
```
If A > 12 Then Stop
{ code data }
```

If variable A contains a value greater than 12 then stop program execution. code data will not be executed.

See also : **End**, **Sleep**, **Snooze**.

Strn

Syntax
Strn Byte Array = Item

Overview
Load a Byte Array with null terminated data, which can be likened to creating a pseudo String variable.

Operators
- ***Byte Array*** is the variable that will be loaded with values.
- ***Item*** can be another **Strn** command, a **Str** command, **Str$** command, a String variable, or a quoted character string

Example
```
' Load the Byte Array String1 with null terminated characters

  Dim String1[21] as Byte    ' Create a Byte array with 21 elements

  Strn String1 = "HELLO WORLD"
  ' Load String1 with characters and null terminate it
  Hrsout Str String1, 13     ' Display the string
  Stop
```

See also: **Arrays as Strings**, **Str$**.

Str$

Syntax
Str Byte Array = **Str$** (Modifier Variable)
or
String = **Str$** (Modifier Variable)

Overview
Convert a Decimal, Hexadecimal, Binary, or Floating Point value or variable into a null terminated string held in a byte array, or a String variable. For use only with the **Str** and **Strn** commands, and real String variables.

Operators
- ***Modifier*** is one of the standard modifiers used with **Print**, **Rsout**, **Hserout** etc. See list below.
- ***Variable*** is a variable that holds the value to convert. This may be a Bit, Byte, Word, Dword, or Float.
- ***Byte Array*** must be of sufficient size to hold the resulting conversion and a terminating null character (0).
- ***String*** must be of sufficient size to hold the resulting conversion.

Notice that there is no comma separating the Modifier from the Variable. This is because the compiler borrows the format and subroutines used in **Print**. Which is why the modifiers are the same:

Bin{1..32}	Convert to binary digits
Dec{1..10}	Convert to decimal digits
Hex{1..8}	Convert to hexadecimal digits
Sbin{1..32}	Convert to signed binary digits
Sdec{1..10}	Convert to signed decimal digits
Shex{1..8}	Convert to signed hexadecimal digits
Ibin{1..32}	Convert to binary digits with a preceding '%' identifier
Idec{1..10}	Convert to decimal digits with a preceding '#' identifier
Ihex{1..8}	Convert to hexadecimal digits with a preceding '$' identifier
ISbin{1..32}	Convert to signed binary digits with a preceding '%' identifier
ISdec{1..10}	Convert to signed decimal digits with a preceding '#' identifier
IShex{1..8}	Convert to signed hexadecimal digits with a preceding '$' identifier

Example 1
```
' Convert a Word variable to a String of characters in a Byte array.
  Dim String1[11] as Byte              ' Create byte array for converted value
  Dim WordVar1 as Word
  WordVar1 = 1234                      ' Load the variable with a value
  Strn String1 = Str$(Dec WordVar1)    ' Convert the Integer to a String
  Hrsout Str String1,13                ' Display the string
  Stop
```

Example 2
```
' Convert a Dword variable to a String of characters in a Byte array.
  Dim String1[11] as Byte              ' Create byte array for converted value
  Dim DwordVar1 as Dword
  DwordVar1 = 1234                     ' Load the variable with a value
  Strn String1 = Str$(Dec DwordVar1)   ' Convert the Integer to a String
  Hrsout Str String1, 13               ' Display the string
  Stop
```


Example 3

```
' Convert a Float variable to a String of characters in a Byte array.
  Dim String1[11] as Byte              ' Create byte array for converted value
  Dim Flt1 as Float
  Flt1 = 3.14                          ' Load the variable with a value
  Strn String1 = Str$(Dec Flt1 )       ' Convert the Float to a String
  Hrsout Str String1, 13               ' Display the string
  Stop
```

Example 4

```
' Convert a Word variable to a Binary String of characters in an array.
  Dim String1[34] as Byte              ' Create byte array for converted value
  Dim WordVar1 as Word
  WordVar1 = 1234                      ' Load the variable with a value
  Strn String1 = Str$(Bin WordVar1 )   ' Convert the Integer to a String
  Hrsout Str String1, 13               ' Display the string
  Stop
```

If we examine the resulting string (Byte Array) converted with example 2, it will contain:

character 1, character 2, character 3, character 4, 0

The zero is not character zero, but value zero. This is a null terminated string.

Notes

The Byte Array created to hold the resulting conversion, must be large enough to accommodate all the resulting digits, including a possible minus sign and preceding identifying character. %, $, or # if the I version modifiers are used. The compiler will try and warn you if it thinks the array may not be large enough, but this is a rough guide, and you as the programmer must decide whether it is correct or not. If the size is not correct, any adjacent variables will be overwritten, with potentially catastrophic results.

See also : **Creating and using Strings, Strn, Arrays as Strings**.

Swap

Syntax
Swap Variable, Variable

Overview
Swap any two variable's values with each other.

Operators
- ***Variable*** is the value to be swapped

Example
```
' If Dog = 2 and Cat = 10 then by using the swap command
' Dog will now equal 10 and Cat will equal 2.

   Var1 = 10           ' Var1 equals 10
   Var2 = 20           ' Var2 equals 20
   Swap Var1, Var2     ' Var2 now equals 20 and Var1 now equals 10
```


Symbol

Syntax
Symbol Name { = } Value

Overview
Assign an alias to a register, variable, or constant value

Operators
- ***Name*** can be any valid identifier.
- ***Value*** can be any previously declared variable, system register, or a Register.Bit combination.

The equals '=' character is optional, and may be omitted if desired.

When creating a program it can be beneficial to use identifiers for certain values that don't change:

```
Symbol Meter = 1
Symbol Centimetre = 100
Symbol Millimetre = 1000
```

This way you can keep your program very readable and if for some reason a constant changes later, you only have to make one change to the program to change all the values. Another good use of the constant is when you have values that are based on other values.

```
Symbol Meter = 1
Symbol Centimetre = Meter / 100
Symbol Millimetre = Centimetre / 10
```

In the example above you can see how the centimetre and millimetre were derived from the Meter.

Another use of the **Symbol** command is for assigning Port.Bit constants:

```
Symbol LED = PortA.0
High LED
```

In the above example, whenever the text LED is encountered, Bit-0 of PortA is actually referenced.

Floating point constants may also be created using **Symbol** by simply adding a decimal point to a value.

```
Symbol PI = 3.14      ' Create a floating point constant named PI
Symbol FlNum = 5.0    ' Create a floating point constant with the value 5
```

Floating point constant can also be created using expressions.

```
' Create a floating point constant holding the result of the expression
  Symbol Quanta = 5.0 / 1024
```

Notes
Symbol cannot create new variables, it simply aliases an identifier to a previously assigned variable, or assigns a constant to an identifier.

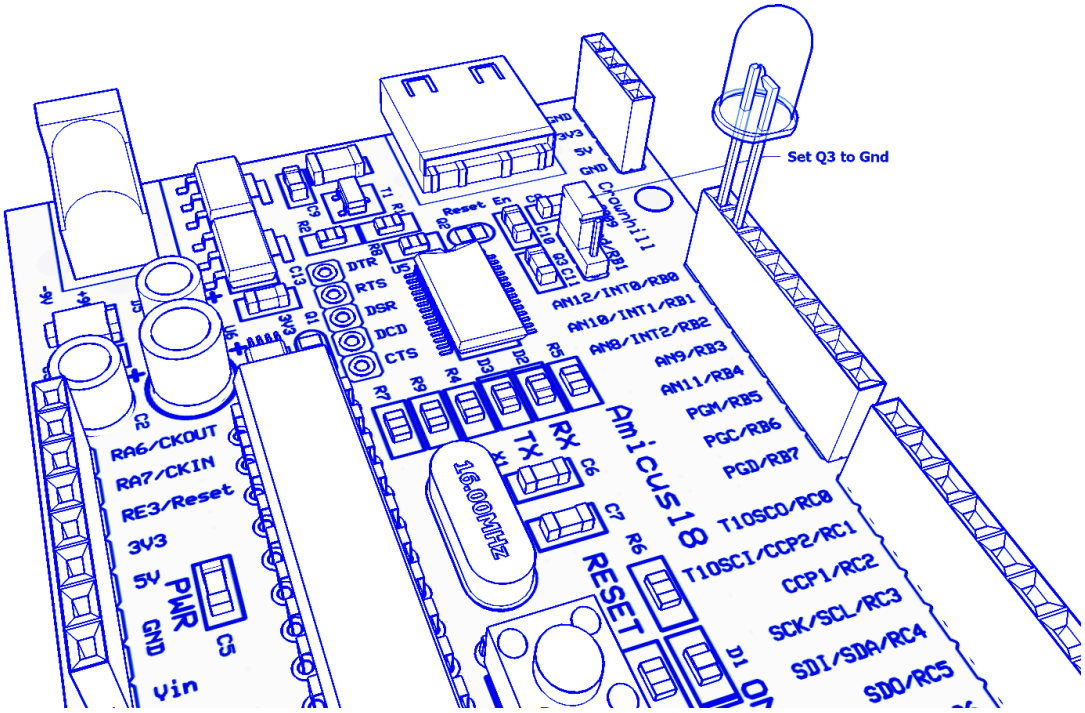
Toggle

Syntax
Toggle Port.Bit

Overview
Sets a pin to output mode and reverses the output state of the pin, changing 0 to 1 and 1 to 0.

Operators
- ***Port.Bit*** can be any valid Port and Bit combination.

Example
```
Low RB0              ' Make bit-0 of PortB an output an pull low
While 1 = 1          ' Create an infinite loop
  Toggle RB0         ' And now reverse the pin
  DelayMs 500        ' Wait for 500 ms
Wend
```



See also : **Clear**, **High**, **Low, Set**.

ToLower

Syntax
Destination String = **ToLower** Source String

Overview
Convert the characters from a source string to lower case.

Overview
- ***Destination String*** can only be a String variable, and should be large enough to hold the correct amount of characters extracted from the Source String.
- ***Source String*** can be a String variable, or a Quoted String of Characters. The Source String can also be a Byte, Word, Byte Array, Word Array or Float variable, in which case the value contained within the variable is used as a pointer to the start of the Source String's address in RAM. A third possibility for Source String is a LABEL name, in which case a null terminated Quoted String of Characters is read from a **Cdata** table.

Example 1
```
' Convert the characters from SourceString to lowercase into DestString

  Dim SourceString as String * 20   ' Create a String of 20 characters
  Dim DestString as String * 20     ' Create another String

  SourceString = "HELLO WORLD"      ' Load the source string with characters
  DestString = ToLower SourceString ' Convert to lowercase
  Hrsout DestString  , 13           ' Display the result, which will be "hello world"
```

Example 2
```
' Convert the characters from a Quoted Character String to lowercase
' into DestString

  Dim DestString as String * 20     ' Create a String of 20 characters

  DestString = ToLower "HELLO WORLD"  ' Convert to lowercase
  Hrsout DestString, 13             ' Display the result, which will be "hello world"
```

Example 3
```
' Convert to lowercase from SourceString into DestString using a pointer to
' SourceString

  Dim SourceString as String * 20   ' Create a String of 20 characters
  Dim DestString as String * 20     ' Create another String
  ' Create a Word variable to hold the address of SourceString
  Dim StringAddr as Word
  SourceString = "HELLO WORLD"      ' Load the source string with characters
  ' Locate the start address of SourceString in RAM
  StringAddr = VarPtr SourceString
  DestString = ToLower StringAddr   ' Convert to lowercase
  Hrsout DestString  ,13            ' Display the result, which will be "hello world"
```


Example 4

```
' Convert chars from a Cdata table to lowercase and place into DestString

   Dim DestString as String * 20     ' Create a String of 20 characters

   DestString = ToLower Source       ' Convert to lowercase
   Hrsout DestString, 13         ' Display the result, which will be "hello world"
   Stop

' Create a null terminated string of characters in code memory
Source:
   Cdata "HELLO WORLD", 0
```

See also : **Creating and using Strings, Creating and using Virtual Strings with Cdata, Cdata, Len, Left$, Mid$, Right$, Str$, ToUpper, VarPtr**.

ToUpper

Syntax
Destination String = **ToUpper** (Source String)

Overview
Convert the characters from a source string to UPPER case.

Overview
- ***Destination String*** can only be a String variable, and should be large enough to hold the correct amount of characters extracted from the Source String.
- ***Source String*** can be a String variable, or a Quoted String of Characters . The Source String can also be a Byte, Word, Byte Array, Word Array or Float variable, in which case the value contained within the variable is used as a pointer to the start of the Source String's address in RAM. A third possibility for Source String is a LABEL name, in which case a null terminated Quoted String of Characters is read from a **Cdata** table.

Example 1
```
' Convert the characters from SourceString to UpperCase and place into
' DestString

  Dim SourceString as String * 20     ' Create a String of 20 characters
  Dim DestString as String * 20       ' Create another String

  SourceString = "hello world"        ' Load the source string with characters
  DestString = ToUpper SourceString   ' Convert to uppercase
  Hrsout DestString  , 13    ' Display the result, which will be "HELLO WORLD"
  Stop
```

Example 2
```
' Convert the chars from a Quoted Character String to UpperCase
' and place into DestString

  Dim DestString as String * 20       ' Create a String of 20 characters

  DestString = ToUpper "hello world"  ' Convert to uppercase
  Hrsout DestString, 13      ' Display the result, which will be "HELLO WORLD
  Stop
```

Example 3
```
' Convert to UpperCase from SourceString into DestString using a pointer to
' SourceString
  Dim SourceString as String * 20     ' Create a String of 20 characters
  Dim DestString as String * 20       ' Create another String
  ' Create a Word variable to hold the address of SourceString
  Dim StringAddr as Word
  ' Load the source string with characters
  SourceString = "hello world"
  ' Locate the start address of SourceString in RAM
  StringAddr = VarPtr SourceString
  DestString = ToUpper StringAddr     ' Convert to uppercase
  Hrsout DestString, 13      ' Display the result, which will be "HELLO WORLD"
  Stop
```


Example 4

```
' Convert chars from Cdata table to uppercase and place into DestString

   Dim DestString as String * 20     ' Create a String of 20 characters

   DestString = ToUpper Source       ' Convert to uppercase
   Hrsout DestString, 13         ' Display the result, which will be "HELLO WORLD"
   Stop

' Create a null terminated string of characters in code memory
Source:
   Cdata "hello world", 0
```

See also : **Creating and using Strings, Creating and using Virtual Strings with Cdata, Cdata, Len, Left$, Mid$, Right$, Str$, ToLower, VarPtr** .

Toshiba_Command

Syntax
Toshiba_Command Command, Value

Overview
Send a command with or without parameters to a Toshiba T6963 graphic LCD.

Operators
- ***Command*** can be a constant, variable, or expression, that contains the command to send to the LCD. This will always be an 8-bit value.
- ***Value*** can be a constant, variable, or expression, that contains an 8-bit or 16-bit parameter associated with the command. An 8-bit value will be sent as a single parameter, while a 16-bit value will be sent as two parameters. Parameters are optional as some commands do not require any. Therefore if no parameters are included, only a command is sent to the LCD.

Because the size of the parameter is vital to the correct operation of specific commands, you can force the size of the parameter sent by issuing either the text "Byte" or "Word" prior to the parameter's value.

```
Toshiba_Command $C0, Byte $FF01   ' Send the low byte of the 16-bit value.
Toshiba_Command $C0, Word $01     ' Send a 16-bit value regardless.
```

The explanation of each command is too lengthy for this document, however they can be found in the Toshiba T6963 datasheet. The example program shown below contains a condensed list of commands.

Example
```
' Pan two pages of text left and right on a 128x64 Toshiba T6963 graphic LCD
  Declare LCD_Type = Toshiba' Use a Toshiba T6963 graphic LCD
'
' LCD interface pin assignments
'
  Declare LCD_DTPort = PortB          ' LCD's Data port
  Declare LCD_WRPin = PortC.2         ' LCD's WR line
  Declare LCD_RDPin = PortC.1         ' LCD's RD line
  Declare LCD_CEPin = PortC.0         ' LCD's CE line
  Declare LCD_CDPin = PortA.1         ' LCD's CD line
  Declare LCD_RSTPin = PortA.0        ' LCD's Reset line (Optional)
'
' LCD characteristics
'
  Declare LCD_Text_Pages = 2          ' Choose two text pages
  Declare LCD_RAM_Size = 8192         ' Amount of RAM the LCD contains
  Declare LCD_X_Res = 128             ' LCD's X Resolution
  Declare LCD_Y_Res = 64              ' LCD's Y Resolution
  Declare LCD_Font_Width = 6          ' The width of the LCD's font
  Declare LCD_Text_Home_Address = 0   ' Ensure text RAM starts at address 0
```



```
' LCD Display Constants:
' Register set commands:
  Symbol T_Cursor_POINTER_Set = $21 ' Cursor Pointer Set
' Offset Register Set (CGRAM start address offset)
  Symbol T_OFFset_REG_Set  = $22
  Symbol T_Addr_POINTER_Set = $24    ' Address Pointer Set
' Control Word Set commands:
  Symbol T_Text_Home_Set = $40    ' Text Home Address Set
  Symbol T_Text_AREA_Set = $41    ' Text Area Set
  Symbol T_GRAPH_Home_Set = $42   ' Graphics Home address Set
  Symbol T_GRAPH_AREA_Set = $43   ' Graphics Area Set
' Mode Set commands:
  Symbol T_or_MODE = $80          ' or mode
  Symbol T_xor_MODE = $81         ' xor mode
  Symbol T_and_MODE = $83         ' and mode
  Symbol T_Text_ATTR_MODE = $84   ' Text Attribute mode
  Symbol T_INT_CG_MODE = $80      ' Internal CG ROM mode
  Symbol T_EXT_CG_MODE = $88      ' External CG RAM mode
' Display Mode commands (or together required bits):
  Symbol T_DISPLAY_OFF = $90      ' Display off
  Symbol T_BLINK_ON = $91         ' Cursor Blink on
  Symbol T_Cursor_ON = $92        ' Cursor on
  Symbol T_Text_ON = $94          ' Text mode on
  Symbol T_Graphic_ON = $98       ' Graphic mode on
  Symbol T_Text_and_GRAPH_ON = $9C   ' Text and graphic mode on
' Cursor Pattern Select:
  Symbol T_Cursor_1LINE = $A0     ' 1 line cursor
  Symbol T_Cursor_2LINE = $A1     ' 2 line cursor
  Symbol T_Cursor_3LINE = $A2     ' 3 line cursor
  Symbol T_Cursor_4LINE = $A3     ' 4 line cursor
  Symbol T_Cursor_5LINE = $A4     ' 5 line cursor
  Symbol T_Cursor_6LINE = $A5     ' 6 line cursor
  Symbol T_Cursor_7LINE = $A6     ' 7 line cursor
  Symbol T_Cursor_8LINE = $A7     ' 8 line cursor
' Data Auto Read/Write:
  Symbol T_Data_AUTO_WR = $B0     ' Data write with auto increment of address
  Symbol T_Data_AUTO_RD = $B1     ' Data read with auto increment of address
  Symbol T_AUTO_Data_Reset = $B2 ' Disable auto read/write
' Data Read/Write:
  Symbol T_Data_WR_Inc = $C0      ' Data write and increment address
  Symbol T_Data_RD_Inc = $C1      ' Data read and increment address
  Symbol T_Data_WR_Dec = $C2      ' Data write and decrement address
  Symbol T_Data_RD_Dec = $C3      ' Data read and decrement address
  Symbol T_Data_WR = $C4          ' Data write with no address change
  Symbol T_Data_RD = $C5          ' Data read with no address change
' Screen Peek:
  Symbol T_SCREEN_Peek = $E0      ' Read the display
' Screen Copy:
  Symbol T_SCREEN_COPY = $E8      ' Copy a line of the display
' Bit Set/Reset (or with bit number 0-7):
  Symbol T_Bit_Reset = $F0        ' Pixel clear
  Symbol T_Bit_Set = $F8          ' Pixel set
```



```
' Create two variables for the demonstration
  Dim PAN_Loop as Byte          ' Holds the amount of pans to perform
  Dim Ypos as Byte              ' Holds the Y position of the displayed text
'
' The Main program loop starts here
'
  DelayMs 200                   ' Wait for things to stabilise
  Cls                           ' Clear and initialise the LCD
' Place text on two screen pages
  For Ypos = 1 to 6
    Print At Ypos, 0, "  THIS IS PAGE ONE      THIS IS PAGE TWO"
  Next
' Draw a box around the display
  Line 1, 0, 0, 127, 0          ' Top line
  LineTo 1, 127, 63             ' Right line
  LineTo 1, 0, 63               ' Bottom line
  LineTo 1, 0, 0                ' Left line
' Pan from one screen to the next then back
  While 1 = 1                   ' Create an infinite loop
    For PAN_Loop = 0 to 22
      ' Increment the Text home address
      Toshiba_Command T_Text_Home_Set, Word PAN_Loop
      DelayMs 200
    Next
    DelayMs 200
    For PAN_Loop = 22 to 0 Step -1
      ' Decrement the Text home address
      Toshiba_Command T_Text_Home_Set, Word PAN_Loop
      DelayMs 200
    Next
    DelayMs 200
  Wend                          ' Do it forever
```

Notes

When the Toshiba LCD's Declares are issued within the BASIC program, several internal variables and constants are automatically created that contain the Port and Bits used by the actual interface and also some constant values holding valuable information concerning the LCD's RAM boundaries and setup. These variables and constants can be used within the BASIC or Assembler environment. The internal variables and constants are:

Variables.

_ _LCD_DTPort	The Port where the LCD's data lines are attached.
_ _LCD_WRPort	The Port where the LCD's WR pin is attached.
_ _LCD_RDPort	The Port where the LCD's RD pin is attached.
_ _LCD_CEPort	The Port where the LCD's CE pin is attached.
_ _LCD_CDPort	The Port where the LCD's CD pin is attached.
_ _LCD_RSTPort	The Port where the LCD's RST pin is attached.

Constants.

_ _LCD_Type	The type of LCD targeted. 0 = Alphanumeric, 1 = Samsung, 2 = Toshiba.
_ _LCD_WRPin	The Pin where the LCD's WR line is attached.
_ _LCD_RDPin	The Pin where the LCD's RD line is attached.
_ _LCD_CEPin	The Pin where the LCD's CE line is attached.
_ _LCD_CDPin	The Pin where the LCD's CD line is attached.
_ _LCD_RSTPin	The Pin where the LCD's RST line is attached.
_ _LCD_Text_Pages	The amount of TEXT pages chosen.
_ _LCD_Graphic_Pages	The amount of Graphic pages chosen.
_ _LCD_RAM_Size	The amount of RAM that the LCD contains.
_ _LCD_X_Res	The X resolution of the LCD. i.e. Horizontal pixels.
_ _LCD_Y_Res	The Y resolution of the LCD. i.e. Vertical pixels.
_ _LCD_Font_Width	The width of the font. i.e. 6 or 8.
_ _LCD_Text_AREA	The amount of characters on a single line of TEXT RAM.
_ _LCD_Graphic_AREA	The amount of characters on a single line of Graphic RAM.
_ _LCD_Text_Home_Address	The Starting address of the TEXT RAM.
_ _LCD_Graphic_Home_Address	The Starting address of the Graphic RAM.
_ _LCD_CGRAM_Home_Address	The Starting address of the CG RAM.
_ _LCD_End_OF_Graphic_RAM	The Ending address of Graphic RAM.
_ _LCD_CGRAM_OFFset	The Offset value for use with CG RAM.

Notice that each name has *two* underscores preceding it. This should ensure that duplicate names are not defined within the BASIC environment.

It may not be apparent straight away why the variables and constants are required, however, the Toshiba LCDs are capable of many tricks such as panning, page flipping, text manipulation etc, and all these require some knowledge of RAM boundaries and specific values relating to the resolution of the LCD used.

See also : **LCDRead**, **LCDWrite**, **Pixel**, **Plot**, **Toshiba_UDG**,**UnPlot. See Print for circuit.**

Toshiba_UDG

Syntax
Toshiba_UDG Character, [Value {, Values }]

Overview
Create User Defined Graphics for a Toshiba T6963 graphic LCD.

Operators
- ***Character*** can be a constant, variable, or expression, that contains the character to define. User defined characters start from 160 to 255.
- ***Value|s*** is a list of constants, variables, or expressions, that contain the information to build the User Defined character. There are also some modifiers that can be used in order to access UDG data from various tables.

Example
```
' Create four User Defined Characters using four different methods
  Declare LCD_Type = Toshiba ' Use a Toshiba T6963 graphic LCD
'
' LCD interface pin assignments
'
  Declare LCD_DTPort = PortB          ' LCD's Data port
  Declare LCD_WRPin = PortC.2         ' LCD's WR line
  Declare LCD_RDPin = PortC.1         ' LCD's RD line
  Declare LCD_CEPin = PortC.0         ' LCD's CE line
  Declare LCD_CDPin = PortA.1         ' LCD's CD line
  Declare LCD_RSTPin = PortA.0        ' LCD's Reset line (Optional)
'
' LCD characteristics
'
  Declare LCD_X_Res = 128             ' LCD's X Resolution
  Declare LCD_Y_Res = 64              ' LCD's Y Resolution
  Declare LCD_Font_Width = 8          ' The width of the LCD's font
  Dim UDG_3[8] as Byte                ' Create a byte array to hold UDG data
  Dim DemoChar as Byte                ' Create a variable for the demo loop
' Create some User Defined Graphic data in eeprom memory
UDG_1 Edata $18, $18, $3C, $7E, $DB, $99, $18, $18
'
' The main demo loop starts here
  DelayMs 200                 ' Wait for things to stabilise
  Cls                         ' Clear both text and graphics of the LCD
' Load the array with UDG data
  Str UDG_3 = $18, $18, $99, $DB, $7E, $3C, $18, $18
'
' Print user defined graphic chars 160, 161, 162, and 162 on the LCD
'
  Print At 1, 0, "Char 160 = ", 160
  Print At 2, 0, "Char 161 = ", 161
  Print At 3, 0, "Char 162 = ", 162
  Print At 4, 0, "Char 163 = ", 163
```



```
Toshiba_UDG 160, [Estr UDG_1]          ' Place UDG edata into character 160
Toshiba_UDG 161, [UDG_2]               ' Place UDG cdata into character 161
Toshiba_UDG 162, [Str UDG_3\8]         ' Place UDG array into character 162
' Place values into character 163
Toshiba_UDG 163, $0C, $18, $30, $FF, $FF, $30, $18, $0C]
While 1 = 1                            ' Create an infinite loop
  For DemoChar = 160 to 163            ' Cycle through characters 160 to 163
    Print At 0, 0, DemoChar            ' Display the character
    DelayMs 200                        ' A small delay
  Next                                 ' Close the loop
Wend                                   ' Do it forever
'
' Create some User Defined Graphic data in code memory
UDG_2: Cdata $30, $18, $0C, $FF, $FF, $0C, $18, $30
```

Notes

User Defined Graphic values can be stored in on-board eeprom memory by the use of **Edata** tables, and retrieved by the use of the Estr modifier. Eight, and only Eight, values will be read with a single Estr:

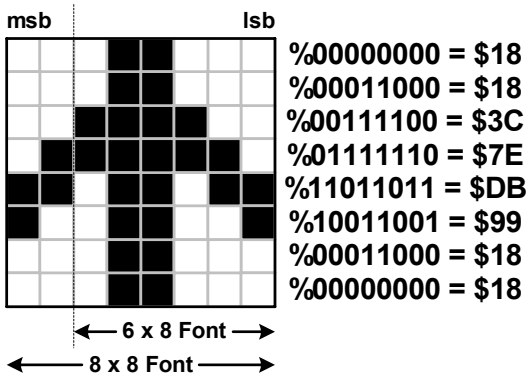
```
UDG_1 Edata $18, $18, $3C, $7E, $DB, $99, $18, $18
  Toshiba_UDG 160, [Estr UDG_1]
```

User Defined Graphic values can also be stored in code memory, on devices that can access their own code memory, and retrieved by the use of a label name associated with a **Cdata** table. Eight, and only Eight, values will be read with a single label name:

```
   Toshiba_UDG 161, [UDG_2]
UDG_2:
   Cdata $30, $18, $0C, $FF, $FF, $0C, $18, $30
```

The use of the **Str** modifier will retrieve values stored in an array, however, this is not recommended as it will waste precious RAM.

The Toshiba LCD's font is designed in an 8x8 grid or a 6x8 grid depending on the font size chosen. The diagram below shows a designed character and its associated values.



See also : **LCDRead**, **LCDWrite**, **Pixel**, **Plot**, **Toshiba_Command**, **UnPlot**.
See Print for circuit.

UnPlot

Syntax
UnPlot Ypos, Xpos

Overview
Clear an individual pixel on a graphic LCD.

Operators
- ***Xpos*** can be a constant, variable, or expression, pointing to the X-axis location of the pixel to clear. This must be a value of 0 to the X resolution of the LCD. Where 0 is the far left row of pixels.
- ***Ypos*** can be a constant, variable, or expression, pointing to the Y-axis location of the pixel to clear. This must be a value of 0 to the Y resolution of the LCD. Where 0 is the top column of pixels.

Example
```
  Declare LCD_Type = Graphic      ' Use a Graphic LCD

' Graphic LCD Pin Assignments
  Declare LCD_DTPort = PortB
  Declare LCD_RSPin = PortC.2
  Declare LCD_RWPin = PortA.0
  Declare LCD_ENPin = PortA.5
  Declare LCD_CS1Pin = PortA.1
  Declare LCD_CS2Pin = PortA.2

  Dim Xpos as Byte
  Cls                             ' Clear the LCD
' Draw a line across the LCD
  While 1 = 1                     ' Create an infinite loop
    For Xpos = 0 to 127
      Plot 20, Xpos
      DelayMs 10
    Next
' Now erase the line
    For Xpos = 0 to 127
      UnPlot 20, Xpos
      DelayMs 10
    Next
Wend
```

See also : **LCDRead**, **LCDWrite**, **Pixel**, **Plot. See Print for circuit.**

Val

Syntax
Variable = **Val** (Array Variable, Modifier)

Overview
Convert a Byte Array or String containing Decimal, Hexadecimal, or Binary numeric text into it's integer equivalent.

Operators
- ***Array Variable*** is a byte array or string containing the alphanumeric digits to convert and terminated by a null (i.e. value 0).
- ***Modifier*** can be **Hex**, **Dec**, or **Bin**. To convert a Hexadecimal string, use the **Hex** modifier, for Binary, use the **Bin** modifier, for Decimal use the **Dec** modifier.
- ***Variable*** is a variable that will contain the converted value. Floating point characters and variables cannot be converted, and will be rounded down to the nearest integer value.

Example 1
```
' Convert a string of hexadecimal characters to an integer
  Dim String1 as String * 10      ' Create a String
  Dim WordVar as Word             ' Create a variable to hold result
  String1 = "12AF"                ' Load the String with Hex digits
  WordVar1 = Val(String1,Hex)     ' Convert the String into an integer
  Hrsout Hex WordVar, 13          ' Display the integer as Hex
```

Example 2
```
' Convert a string of decimal characters to an integer
  Dim String1 as String * 10      ' Create a String
  Dim WordVar as Word             ' Create a variable to hold result
  String1 = "1234"                ' Load the String with Decimal digits
  WordVar1 = Val(String1,Dec)     ' Convert the String into an integer
  Hrsout Dec WordVar, 13          ' Display the integer as Decimal
```

Example 3
```
' Convert a string of binary characters to an integer
  Dim String1 as String * 17      ' Create a String
  Dim WordVar1 as Word            ' Create a variable to hold result
  String1 = "1010101010000000"    ' Load the String with Binary
  WordVar1 = Val(String1,Bin)     ' Convert the String into an integer
  Hrsout Bin WordVar1, 13         ' Display the integer as Binary
```


Notes

The **Val** command is *not* recommended inside an expression, as the results are not predictable. However, the **Val** command can be used within an **If**-**Then**, **While**-**Wend**, or **Repeat**-**Until** construct, but the code produced is not as efficient as using it outside a construct, because the compiler must assume a worst case scenario, and use Dword comparisons.

```
Dim String1 as String * 17          ' Create a String
String1 = "123"                     ' Load the String with Dec digits
If Val(String1,Hex) = 123 Then      ' Compare the result
  Hrsout Dec Val(String1,Hex), 13
Else
  Hrsout "Not Equal\r"
EndIf
```

See also: **Str**, **Strn**, **Str$**.

VarPtr

Syntax
Assignment Variable = **VarPtr** Variable

Overview
Returns the address of the variable in RAM. Commonly known as a pointer to a variable.

Operators
- ***Assignment Variable*** can be any of the compiler's variable types, and will receive the pointer to the variable's address.
- ***Variable*** can be any variable name used in the BASIC program.

While...Wend

Syntax
While Condition
Instructions
Instructions
Wend

or

While Condition { Instructions : } **Wend**

Overview
Execute a block of instructions while a condition is true.

Example
```
Var1 = 1
While Var1 <= 10
  Hrsout Dec Var1, 13
  Var1 = Var1 + 1
Wend
```

or

```
While PortA.0 = 1: Wend ' Wait for a change on the Port
```

Notes
While-**Wend**, repeatedly executes Instructions While Condition is true. When the Condition is no longer true, execution continues at the statement following the **Wend**. Condition may be any comparison expression.

See also : **If**-**Then**, **Repeat**-**Until**, **For**-**Next**.

Xin

Syntax
Xin DataPin, ZeroPin, {Timeout, Timeout Label}, [Variable{,...}]

Overview
Receive X-10 data and store the House Code and Key Code in a variable.

Operators
- ***DataPin*** is a constant (0 - 15), Port.Bit, or variable, that receives the data from an X-10 interface. This pin is automatically made an input to receive data, and should be pulled up to 5 Volts with a 4.7KΩ resistor.
- ***ZeroPin*** is a constant (0 - 15), Port.Bit, or variable, that is used to synchronise to a zero-cross event. This pin is automatically made an input to received the zero crossing timing, and should also be pulled up to 5 Volts with a 4.7KΩ resistor.
- ***Timeout*** is an optional value that allows program continuation if X-10 data is not received within a certain length of time. Timeout is specified in AC power line half-cycles (approximately 8.33 milliseconds).
- ***Timeout*** Label is where the program will jump to upon a timeout.

Example
```
  Dim HouseKey as Word
Loop:
' Receive X-10 data, go to NoData if none
  Xin PortA.2, PortA.0, 10, NoData, [HouseKey]
' Display X-10 data on the serial terminal
  Hrsout "House=", Dec HouseKey.Byte1, "Key=", Dec HouseKey.Byte0, 13
  GoTo Loop                  ' Do it forever
NoData:
  Hrsout "No Data", 13
  Stop
```

Xout and Xin Declares
In order to make the **Xin** command's results more in keeping with the BASIC Stamp interpreter, two declares have been included for both **Xin** and **Xout** These are.

Declare Xout_Translate = On/Off, True/False or 1/0

and

Declare Xin_Translate = On/Off, True/False or 1/0

Notes
Xin processes data at each zero crossing of the AC power line as received on ZeroPin. If there are no transitions on this line, **Xin** will effectively wait forever.

Xin is used to receive information from X-10 devices that can transmit the appropriate data. X-10 modules are available from a wide variety of sources under several trade names. An interface is required to connect the microcontroller to the AC power line. The TW-523 for two-way X-10 communications is required by **Xin**. This device contains the power line interface and isolates the microcontroller from the AC line.

If Variable is a Word sized variable, then each House Code received will be stored in the upper 8-bits of the Word And each received Key Code will be stored in the lower 8-bits of the Word variable. If Variable is a Byte sized variable, then only the Key Code will be stored.

The House Code is a number between 0 and 15 that corresponds to the House Code set on the X-10 module A through P.

The Key Code can be either the number of a specific X-10 module or the function that is to be performed by a module. In normal operation, a command is first sent, specifying the X-10 module number, followed by a command specifying the desired function. Some functions operate on all modules at once so the module number is unnecessary. Key Code numbers 0-15 correspond to module numbers 1-16.

Warning. Under no circumstances should the microcontroller be connected directly to the AC power line. Voltage potentials carried by the power line will not only instantly destroy the microcontroller, but could also pose a serious health hazard.

See also : **Xout**.

Xout

Syntax
Xout DataPin, ZeroPin, [HouseCode\KeyCode {\Repeat} {, ...}]

Overview
Transmit a HouseCode followed by a KeyCode in X-10 format.

Operators
- ***DataPin*** is a constant (0 - 15), Port.Bit, or variable, that transmits the data to an X-10 interface. This pin is automatically made an output.
- ***ZeroPin*** is a constant (0 - 15), Port.Bit, or variable, that is used to synchronise to a zero-cross event. This pin is automatically made an input to received the zero crossing timing, and should also be pulled up to 5 Volts with a 4.7KΩ resistor.
- ***HouseCode*** is a number between 0 and 15 that corresponds to the House Code set on the X-10 module A through P. The proper HouseCode must be sent as part of each command.
- ***KeyCode*** can be either the number of a specific X-10 module, or the function that is to be performed by a module. In normal use, a command is first sent specifying the X-10 module number, followed by a command specifying the function required. Some functions operate on all modules at once so the module number is unnecessary. KeyCode numbers 0-15 correspond to module numbers 1-16.
- ***Repeat*** is an optional operator, and if it is not included, then a repeat of 2 times (the minimum) is assumed. Repeat is normally reserved for use with the X-10 Bright and **Dim** commands.

Example
```
   Dim House as Byte
   Dim Unit as Byte
' Create some aliases of the keycodes
   Symbol UnitOn = %10010        ' Turn module on
   Symbol UnitOff = %11010       ' Turn module off
   Symbol UnitsOff = %11100      ' Turn all modules off
   Symbol LightsOn = %10100      ' Turn all light modules on
   Symbol LightsOff = %10000     ' Turn all light modules off
   Symbol Bright = %10110        ' Brighten light module
   Symbol DimIt = %11110         ' Dim light module
' Create aliases for the pins used
   Symbol DataPin = PortA.1
   Symbol ZeroC = PortA.0
   House = 0                     ' Set house to 0 (A)
   Unit = 8                      ' Set unit to 8 (9)
' Turn on unit 8 in house 0
   Xout DataPin ,ZeroC,[House \ Unit,House \ UnitOn ]
' Turn off all the lights in house 0
   Xout DataPin ,ZeroC,[House \ LightsOff ]
   Xout DataPin ,ZeroC,[House \ 0]
' Blink light 0 on and off every 10 seconds
Loop:
   Xout DataPin ,ZeroC,[House \ UnitOn ]
   DelayMs 10000                 ' Wait 10 seconds
   Xout DataPin ,ZeroC,[House \ UnitOff ]
   DelayMs 10000                 ' Wait 10 seconds
   GoTo Loop
```


Xout and Xin Declares
In order to make the **Xout** command's results more in keeping with the BASIC Stamp interpreter, two declares have been included for both **Xin** and **Xout**. These are.

Declare Xout_Translate = On/Off, True/False or 1/0
and
Declare Xin_Translate = On/Off, True/False or 1/0

Notes
Xout only transmits data at each zero crossing of the AC power line, as received on ZeroPin. If there are no transitions on this line, **Xout** will effectively wait forever.

Xout is used to transmit information from X-10 devices that can receive the appropriate data. X-10 modules are available from a wide variety of sources under several trade names. An interface is required to connect the microcontroller to the AC power line. Either the PL-513 for send only, or the TW-523 for two-way X-10 communications are required. These devices contain the power line interface and isolate the microcontroller from the AC line.

The KeyCode numbers and their corresponding operations are listed below:

KeyCode	KeyCode No.	Operation
UnitOn	%10010	Turn module on
UnitOff	%11010	Turn module off
UnitsOff	%11100	Turn all modules off
LightsOn	%10100	Turn all light modules on
LightsOff	%10000	Turn all light modules off
Bright	%10110	Brighten light module
Dim	%11110	Dim light module

Wiring to the X-10 interfaces requires 4 connections. Output from the X-10 interface (zero crossing and receive data) are open-collector, which is the reason for the pull-up resistors on the microcontroller.

Wiring for each type of interface is shown below:

PL-513 Wiring

Wire No.	Wire Colour	Connection
1	**Black**	Zero crossing output
2	Red	Zero crossing common
3	**Green**	X-10 transmit common
4	**Yellow**	X-10 transmit input

TW-523 Wiring

Wire No.	Wire Colour	Connection
1	**Black**	Zero crossing output
2	Red	Common
3	**Green**	X-10 receive output
4	**Yellow**	X-10 transmit input

Warning. Under no circumstances should the microcontroller be connected directly to the AC power line. Voltage potentials carried by the power line will not only instantly destroy the microcontroller, but could also pose a serious health hazard.

See also : **Xin**.

Using the Optimiser

The underlying assembler code produced by the compiler is the single most important element to a good language, because compact assembler not only means more can be squeezed into the tight confines of the microcontroller, but also the code runs faster which allows more complex operations to be performed. This is why the compiler now has a "dead code removal" pass as standard which will remove redundant mnemonics, and replace certain combinations of mnemonics with a single mnemonic. WREG tracking is also implemented as standard which helps eliminate unnecessary loading of a constant value into the WREG.

And even though the compiler already produces good underlying assembler mnemonics, there is always room for improvement, and that improvement is achieved by a separate optimising pass.

The optimiser is enabled by issuing the **Declare**:

```
Declare Optimiser_Level = n
```

Where n is the level of optimisation required.

The **Declare** should be placed at the top of the BASIC program, but anywhere in the code is actually acceptable because once the optimiser is enabled it cannot be disabled later in the same program.

The optimiser has 3 levels, 4 if you include Off as a level.

- Level 0 disables the optimiser.
- Level 1 (Compiler Default) Chooses the appropriate branching mnemonics and will replace **Call** with **RCall** and **GoTo** with **Bra** whenever appropriate, saving 1 byte of code space every time.
- Level 2 Further re-arranging of branching operations.
- Level 3 Re-arranges conditional branching operations. This is an important optimising pass because a single program can implement many decision making mnemonics.

You must be aware that optimising code further that value 1, can, in some circumstances, have a detrimental effect on a program, this is true of all optimisation on all compilers and is something that you should take into account. Therefore, always try to write and test your program without the optimiser pass. Then once it's working as expected, enable the optimiser a level at a time. However, this is not always possible with larger programs that will not fit within the microcontroller without optimisation. In this circumstance, choose level 1 optimisation whenever the code is reaching the limits of the microcontroller, testing the code as you go along.

Caveats
Of course there's no such thing as a free lunch, and there are some features that cannot be used when implementing the optimiser.

Do not use the **Movfw** macro as this will cause problems withing the Asm listing, use the correct mnemonic of **Movf** Var, W.

Do not use the assembler *LIST* and *NOLIST* directives, as the optimiser uses these to sculpt the final Asm used.

```
Declare Dead_Code_Remove = On/Off
```

The above declare removes some redundant op-codes from the underlying Asm code.

- Removal of redundant Bank Switching mnemonics.
- Removal of redundant **Movwf** mnemonics if preceded by a **Movf** Var,W mnemonic.
- Removal of redundant **Movf** Var,W mnemonics if preceded by a **Movwf** mnemonic.
- Removal of redundant **Andlw** mnemomics if preceded by another **Andlw** mnemonic.
- Replaced a **Call**-**Return** mnemonic pair with a single **GoTo** mnemonic.

Using the Preprocessor

A preprocessor directive is a non executable statement that informs the compiler how to compile. For example, some microcontroller devices have certain hardware features that others don't. A pre-processor directive can be used to inform the compiler to add or remove source code, based on that particular devices ability to support that hardware.

It's important to note that the preprocessor works with directives on a line by line basis. It is therefore important to ensure that each directive is on a line of its own. Don't place directives and source code on the same line.

It's also important not to mistake the compiler's preprocessor with the assembler's preprocessor. Any directive that starts with a dollar "$" is the compiler's preprocessor, and any directive that starts with a hash "#" is the assembler's preprocessor. They cannot be mixed, as each has no knowledge of the other.

Preprocessor directives can be nested in the same way as source code statements. For example:

```
$ifdef MyValue
   $if MyValue = 10
      Symbol CodeConst = 10
   $else
      Symbol CodeConst = 0
   $endif
$endif
```

Preprocessor directives are lines included in the code of the program that are not BASIC language statements but directives for the preprocessor itself. The preprocessor is actually a separate entity to the compiler, and, as the name suggests, preprocesses the BASIC code before the actual compiler sees it. Preprocessor directives are always preceded by a dollar sign "$".

Preprocessor Directives

To define preprocessor macros the directive **$define** is used. Its format is:

```
$define identifier replacement
```

When the preprocessor encounters this directive, it replaces any occurrence of identifier in the rest of the code by replacement. This replacement can be an expression, a statement, a block, or simply anything. The preprocessor does not understand BASIC, it simply replaces any occurrence of identifier by replacement.

```
$define TableSize 100
  Dim Table1[TableSize] as Byte
  Dim Table2[TableSize] as Byte
```

After the preprocessor has replaced TableSize, the code becomes equivalent to:

```
  Dim Table1[100] as Byte
  Dim Table2[100] as Byte
```


The use of **$define** as a constant definer is only one aspect of the preprocessor, and **$define** can also work with parameters to define pseudo function macros. The syntax then is:

```
$define identifier (parameter list) replacement
```

A simple example of a function-like macro is:

```
$define RadToDeg(x) ((x) * 57.29578)
```

This defines a radians to degrees conversion which can be used as:

```
Var1 = RadToDeg(34)
```

This is expanded in-place, so the caller does not need to clutter copies of the multiplication constant throughout the code.

Precedence
Note that the example macro RadToDeg(x) given above uses normally unnecessary parentheses both around the argument and around the entire expression. Omitting either of these can lead to unexpected results. For example:

Macro defined as:
```
$define RadToDeg(x) (x * 57.29578)
```
will expand
```
RadToDeg(a + b)
```
to
```
(a + b * 57.29578)
```

Macro defined as
```
$define RadToDeg(x) (x) * 57.29578
```
will expand
```
1 / RadToDeg(a)
```
to
```
1 / (a) * 57.29578
```

neither of which give the intended result.

Not all replacement tokens can be passed back to an assignment using the equals operator. If this is the case, the code needs to be similar to BASIC Stamp syntax, where the assignment variable is the last parameter:

```
$define GetMax(x,y,z) If x > y Then z = x : Else : z = y
```

This would replace any occurrence of GetMax followed by three parameter (argument) by the replacement expression, but also replacing each parameter by its identifier, exactly as would be expected of a function.

```
Dim Var1 as Byte
Dim Var2 as Byte
Dim Var3 as Byte

Var1 = 100
Var2 = 99
GetMax(Var1, Var2, Var3)
```


The previous would be placed within the BASIC program as:

```
Dim Var1 as Byte
Dim Var2 as Byte
Dim Var3 as Byte

Var1 = 100
Var2 = 99
If Var1 > Var2 Then Var3 = Var1 : Else : Var3 = Var2
```

Notice that the third parameter "Var3" is loaded with the result.

A macro lasts until it is undefined with the **$undef** preprocessor directive:

```
$define TableSize 100
  Dim Table1[TableSize] as Byte
$undef TableSize
$define TableSize 200
  Dim Table2[TableSize] as Byte
```

This would generate the same code as:

```
Dim Table1[100] as Byte
Dim Table2[200] as Byte
```

Because preprocessor replacements happen before any BASIC syntax check, macro definitions can be a tricky feature, so be careful. Code that relies heavily on complicated macros may be difficult to understand, since the syntax they expect is, on many occasions, different from the regular expressions programmers expect in Proton Amicus18 BASIC.

Preprocessor directives only extend across a single line of code. As soon as a newline character is found (end of line), the preprocessor directive is considered to end. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a comment character (') followed by a new line. No comment text can follow the comment character. For example:

```
$define GetMax(x,y,z) If x > y Then  '
                          z = x      '
                      Else           '
                          z = y      '
                      EndIf

GetMax(Var1, Var2, Var3)
```

The compiler will see:

```
If Var1 > Var2 Then
  Var3 = Var1
Else
  Var3 = Var2
EndIf
```

Note that parenthesis is always required around the **$define** declaration and its use within the program. Parenthesis for **$define** can be either round or square brackets.

If the replacement argument is not included within the **$define** directive, the identifier argument will output nothing. However, it can be used as an identifier for conditional code:

$define DoThis

```
$ifdef DoThis
{Rest of Code here}
$endif
```

$undef identifier
This removes any existing definition of the user macro identifier.

$eval Expression
In normal operation, the **$define** directive simply replaces text, however, using the $eval directive allows constant value expressions to be evaluated before replacement within the BASIC code. For example:

```
$define Expression(Prm1) $eval Prm1 << 1
```

The above will evaluate the constant parameter Prm1, shifting it left one position.

```
  Var1 = Expression(1)
```

Will be added to the BASIC code as:

```
  Var1 = 2
```

Because 1 shifted left one position is 2.

Several operators are available for use with an expression. These are +, -, *, -, ~, <<, >>, =, >, <, >=, <=, <>, And, Or, Xor.

Conditional Directives (**$ifdef**, **$ifndef**, **$if**, **$endif**, **$else** and **$elseif**)
Conditional directives allow parts of the code to be included or discarded if a certain condition is met.

$ifdef allows a section of a program to be compiled only if the macro that is specified as the parameter has been defined, no matter what its value is. For example:

```
$ifdef TableSize
  Dim Table[TableSize] as Byte
$endif
```

In the above condition, the line of code **Dim** Table[TableSize] **as Byte** is only compiled if TableSize was previously defined with **$define**, independent of its value. If it was not defined, the line will not be included in the program compilation.

$ifndef serves for the exact opposite: the code between **$ifndef** and **$endif** directives is only compiled if the specified identifier has not been previously defined. For example:

```
$ifndef TableSize
$define TableSize 100
$endif
  Dim Table[TableSize] as Byte
```


In the previous code, when arriving at this piece of code, the TableSize directive has not been defined yet. If it already existed it would keep its previous value since the **$define** directive would not be executed.

A valuable use for **$ifdef** is that of a code guard with include files. This allows multiple insertions of a file, but only the first will be used.

A typical code guard looks like:

```
$ifndef IncludeFileName
$define IncludeFileName
{ BASIC Code goes Here }
$endif
```

The logic of the above snippet is that if the include file has not previously been loaded into the program, the **$define** *IncludeFileName* will not have been created, thus allowing the inclusion of the code between **$if**ndef and **$endif**. However, if the include file has been previously loaded, the **$define** will have already been created, and the condition will be false, thus not allowing the code to be included.

IncludeFileName must be unique to each file. Therefore, it is recommended that a derivative of the Include File's name is used, or a preceding and following underscore surround it.

$if expression
This directive invokes the arithmetic evaluator and compares the result in order to begin a conditional block. In particular, note that the logical value of expression is always true when it cannot be evaluated to a number.

$else
This toggles the logical value of the current conditional block. What follows is evaluated if the preceding **$if** evaluated as false.

$endif
This ends a conditional block started by the **$if** directive.

$elseif expression
This directive can be used to avoid nested **$if** conditions. **$if..$elseif..$endif** is equivalent to **$if..$else $if ..$endif $endif**.

The **$if**, **$else** and **$elseif** directives serve to specify some condition to be met in order for the portion of code they surround to be compiled. The condition that follows **$if** or **$elseif** can only evaluate constant expressions, including macro expressions. For example:

```
$if TableSize > 200
$undef TableSize
$define TableSize 200

$elseif TableSize < 50
$undef TableSize
$define TableSize 50

$else
$undef TableSize
$define TableSize 100
$endif

   Dim Table[TableSize] as Byte
```

Notice how the whole structure of **$if**, **$elseif** and **$else** chained directives ends with **$endif**.

The behaviour of **$ifdef** and **$ifndef** can also be achieved by using the special built-in user directive **_defined** and **! _defined** respectively, in any **$if** or **$elseif** condition. These allow more flexibility than **$ifdef** and **$ifndef**. For example:

```
$if _defined(MyDefine) and _defined(AnotherDefine)
{ BASIC Code Here }
$endif
```

The argument for the **_defined** user directive must be surrounded by parenthesis. The preceding character "!" means "not".

$error message
This directive causes an error message with the current filename and line number. Subsequent processing of the code is then aborted.

```
   $error Error Message Here
```


Definition File

The Amicus18 microcontroller has an associated .Def file containing **$define** directives of all the SFRs, SFR bit names and device information. These are invaluable for conditional compilation, or keeping consistent register.bit names with differing devices.

The definition file is located within the compiler's "Includes\Sources" directory, and the appropriate one is added to the program automatically.

Each defined SFR name is preceded by an underscore, so that they don't get confused with the actual SFR's used by the compiler and assembler:

```
$define _SSPADD 4040
```

The SFR's address is the replacement text, and is also invaluable for conditional compilation.

```
   $ifdef _SSPADD
     SSPADD = 0
   $endif
```

or

```
   $if _defined(_SSPADD) and _SSPADD > 128
   { Do some Conditional Code Here }
   $endif
```

Notice that the defined name is not used, and cannot be used, as the actual SFR.

Each bit associated with an SFR is also defined, and is extremely useful for code clarity and consistency. Each bit name has the format:

SFR Namebits_Bit Name SFR.bit

So the bit GO_DONE bit of the ADCON0 SFR looks like:

```
   $define ADCON0bits_GO_DONE ADCON0.2
```


Built in Peripheral Macros

The compiler has several built-in macros for configuring the most popular peripheral modules contained with the microcontroller, these are the ADC (Analogue to Digital Converter), Timers, SPI (Serial Peripheral Interface),

ADC macro introduction

The ADC (Analogue to Digital Converter) peripheral on the Amicus18 is supported with the following macros. The macros are a mixture of compiler types and preprocessor types, and can be found in "Includes\Sources\ADC.inc"

A/D Converter Macros

- BusyADC Is A/D converter currently performing a conversion?
- CloseADC Disable the A/D converter.
- ConvertADC Start an A/D conversion.
- OpenADC Configure the A/D converter.
- ReadADC Read the results of an A/D conversion.
- SetChanADC Select A/D channel to be used.
- SelChanConvADC Select A/D channel to be used and start an A/D conversion.

BusyADC

Syntax
Variable = **BusyADC**()

Include file
ADC.inc

Overview
This macro indicates if the A/D peripheral is in the process of converting a value.

Return Value
- 1 if the A/D peripheral is performing a conversion.
- 0 if the A/D peripheral isn't performing a conversion.

CloseADC

Syntax
CloseADC()

Include file
ADC.inc

Overview
This macro disables the A/D converter and A/D interrupt mechanism.

ConvertADC

Syntax
ConvertADC()

Include file
ADC.inc

Overview
This macro starts an A/D conversion. The **BusyADC**() macro or A/D interrupt may be used to detect completion of the conversion. The result is held in registers ADRESL and ADRESH.

OpenADC

Syntax
OpenADC(pConfig, pConfig2, pPortConfig)

Include file
ADC.inc

Overview
This macro resets the A/D-related registers to the POR state and then Configures the clock, result format, voltage reference, port and channel.

Operators

- ***Pconfig*** A bitmask that is created by performing a bitwise AND operation ('&') with a value from each of the categories listed below. These values are defined in the file adcdefs.inc.

A/D clock source:

ADC_FOSC_2	Fosc / 2
ADC_FOSC_4	Fosc / 4
ADC_FOSC_8	Fosc / 8
ADC_FOSC_16	Fosc / 16
ADC_FOSC_32	Fosc / 32
ADC_FOSC_64	Fosc / 64
ADC_FOSC_RC	Internal RC Oscillator

A/D result justification:

ADC_RIGHT_JUST	Result in Least Significant bits (Used for 10-bit ADC result)
ADC_LEFT_JUST	Result in Most Significant bits (Used for 8-bit ADC result)

A/D acquisition time select:

ADC_0_TAD	0 Tad
ADC_2_TAD	2 Tad
ADC_4_TAD	4 Tad
ADC_6_TAD	6 Tad
ADC_8_TAD	8 Tad
ADC_12_TAD	12 Tad
ADC_16_TAD	16 Tad
ADC_20_TAD	20 Tad

- ***pConfig2*** A bitmask that is created by performing a bitwise AND operation ('&'), as shown in the example at the end of this document, with a value from each of the categories listed below. These values are defined in the file adcdefs.inc.

Channel:

ADC_CH0	Channel 0
ADC_CH1	Channel 1
ADC_CH2	Channel 2
ADC_CH3	Channel 3
ADC_CH4	Channel 4
ADC_CH5	Channel 5
ADC_CH6	Channel 6
ADC_CH7	Channel 7
ADC_CH8	Channel 8
ADC_CH9	Channel 9
ADC_CH10	Channel 10
ADC_CH11	Channel 11
ADC_CH12	Channel 12

A/D Vref+ and Vref- Configuration:

ADC_REF_VDD_VREFMINUS	VREF+ = VDD & VREF- = Ext.
ADC_REF_VREFPLUS_VREFMINUS	VREF+ = Ext. & VREF- = Ext.
ADC_REF_VREFPLUS_VSS	VREF+ = Ext. & VREF- = VSS
ADC_REF_VDD_VSS	VREF+ = VDD & VREF- = VSS

- ***pPortConfig*** The *pPortConfig* can have 8192 different combination, few are defined below:

ADC_0ANA	All digital
ADC_1ANA	analogue: AN0
ADC_2ANA	analogue: AN0-AN1
ADC_3ANA	analogue: AN0-AN2
ADC_4ANA	analogue: AN0-AN3
ADC_5ANA	analogue: AN0-AN4
ADC_6ANA	analogue: AN0-AN5
ADC_7ANA	analogue: AN0-AN6
ADC_8ANA	analogue: AN0-AN7
ADC_9ANA	analogue: AN0-AN8
ADC_10ANA	analogue: AN0-AN9
ADC_11ANA	analogue: AN0-AN10
ADC_12ANA	analogue: AN0-AN11

Example

```
'
' Open the ADC  :
'                   Fosc/32
'                   Right justified for 10-bit operation
'                   Tad value of 2
'                   Vref+ at Vcc : Vref- at Gnd
'                   Make AN0 an analogue input
'
  OpenADC(ADC_FOSC_32 & ADC_RIGHT_JUST & ADC_2_TAD, ADC_REF_VDD_VSS, ADC_1ANA)
```

ReadADC

Syntax

Variable = **ReadADC**(pChannel)

Include file

ADC.inc

Overview

This macro returns the Word (10 bit) result of the A/D conversion. Based on the configuration of the A/D converter (e.g., using the **OpenADC**() macro).

Operators

pChannel is an *optional* ADC channel to take the reading from

SetChanADC

Syntax
SetChanADC(pChannel)

Include file
ADC.inc

Overview
Selects the pin that will be used as input to the A/D Converter.

Operator
- ***pChannel*** One of the following values (defined in adcdefs.inc):

ADC_CH0	Channel 0
ADC_CH1	Channel 1
ADC_CH2	Channel 2
ADC_CH3	Channel 3
ADC_CH4	Channel 4
ADC_CH5	Channel 5
ADC_CH6	Channel 6
ADC_CH7	Channel 7
ADC_CH8	Channel 8
ADC_CH9	Channel 9
ADC_CH10	Channel 10
ADC_CH11	Channel 11
ADC_CH12	Channel 12
ADC_CH13	Channel 13
ADC_CH14	Channel 14
ADC_CH15	Channel 15
ADC_CH_CTMU	Channel 13
ADC_CH_VDDCORE	Channel 14
ADC_CH_VBG	Channel 15

SelChanConvADC

Syntax
SelChanConvADC(pChannel)

Include file
ADC.inc

Overview
Selects the pin that will be used as input to the A/D converter. And starts an A/D conversion. The **BusyADC**() macro or A/D interrupt may be used to detect completion of the conversion.

Operator
pChannel One of the values used for the SetChanADC macro.

Example
```
SelChanConvADC(ADC_CH0)
```


ADC_IntEnable() Enables the ADC interrupt i.e. sets PEIE and ADIE bits.
ADC_IntDisable() Disables the ADC interrupt i.e. clears ADIE bit.

Example use of the A/D Converter Macros:

```
  Include "ADC.inc"                ' Load the ADC macros into the program

  Dim Result as Word
'
' Open the ADC:
'                Fosc / 32
'                Right justified for 10-bit operation
'                Tad value of 2
'                Vref+ at Vcc : Vref- at Gnd
'                Make AN0 an analogue input
'
  OpenADC(ADC_FOSC_32 & ADC_RIGHT_JUST & ADC_2_TAD, ADC_REF_VDD_VSS, ADC_1ANA)
  DelayUs 2                       ' Delay for 2 microSeconds
  Result = ReadADC(0)             ' Read result of AN0
  CloseADC()                      ' Disable A/D converter
```


Timer macros Introduction

The timer peripherals are supported with the following macros. The macros are a mixture of compiler types and preprocessor types, and can be found in "Includes\Sources\Timers.inc"

- CloseTimerx Disable timer x.
- OpenTimerx Configure and enable timer x.
- ReadTimerx Read the value of timer x.
- WriteTimerx Write a value into timer x.
- SetTmrCCPSrc Configure the timer as a clock source to CCP module.

CloseTimer0

Syntax
CloseTimer0()

Include file
Timers.inc

Overview
This macro disables timer0 and it's interrupt.

CloseTimer1

Syntax
CloseTimer1()

Include file
Timers.inc

Overview
This macro disables timer1 and it's interrupt.

CloseTimer2

Syntax
CloseTimer2()

Include file
Timers.inc

Overview
This macro disables timer2 and it's interrupt.

CloseTimer3

Syntax
CloseTimer3()

Include file
Timers.inc

Overview
This macro disables timer3 and it's interrupt.

OpenTimer0

Syntax
OpenTimer0(pConfig)

Include file
Timers.inc

Overview
This macro configures timer0 according to the options specified and then enables it.

Operator
pConfig A bitmask that is created by performing either a bitwise AND operation ('&'), which is user configurable, with a value from each of the categories listed below. These values are defined in the file TimerDefs.inc.

Enable Timer0 Interrupt:

TIMER_INT_ON	Interrupt enabled
TIMER_INT_OFF	Interrupt disabled

Timer Width:

T0_8BIT	8-bit mode
T0_16BIT	16-bit mode

Clock Source:

T0_SOURCE_EXT	External clock source (I/O pin)
T0_SOURCE_INT	Internal clock source (Tosc)

External Clock Trigger (for T0_SOURCE_EXT):

T0_EDGE_FALL	External clock on falling edge
T0_EDGE_RISE	External clock on rising edge

Prescale Value:

T0_PS_1_1	1:1 prescale
T0_PS_1_2	1:2 prescale
T0_PS_1_4	1:4 prescale
T0_PS_1_8	1:8 prescale
T0_PS_1_16	1:16 prescale
T0_PS_1_32	1:32 prescale
T0_PS_1_64	1:64 prescale
T0_PS_1_128	1:128 prescale
T0_PS_1_256	1:256 prescale

Example

```
OpenTimer0(TIMER_INT_OFF & T0_8BIT & T0_SOURCE_INT & T0_PS_1_32)
```


OpenTimer1

Syntax
OpenTimer1(pConfig)

Include file
Timers.inc

Overview
This macro configures timer1 according to the options specified and then enables it.

Operator
pConfig A bitmask that is created by performing either a bitwise AND operation ('&'), which is user configurable, with a value from each of the categories listed below. These values are defined in the file TimerDefs.inc.

Enable Timer1 Interrupt:

TIMER_INT_ON	Interrupt enabled
TIMER_INT_OFF	Interrupt disabled

Timer Width:

T1_8BIT_RW	8-bit mode
T1_16BIT_RW	16-bit mode

Clock Source:

T1_SOURCE_EXT	External clock source (I/O pin)
T1_SOURCE_INT	Internal clock source (Tosc)

Prescaler:

T1_PS_1_1	1:1 prescale
T1_PS_1_2	1:2 prescale
T1_PS_1_4	1:4 prescale
T1_PS_1_8	1:8 prescale

Oscillator Use:

T1_OSC1EN_ON	Enable Timer1 oscillator
T1_OSC1EN_OFF	Disable Timer1 oscillator

Synchronise Clock Input:

T1_SYNC_EXT_ON	Sync external clock input
T1_SYNC_EXT_OFF	Don't sync external clock input

Example

```
OpenTimer1(TIMER_INT_ON & T1_8BIT_RW & T1_SOURCE_EXT & T1_PS_1_1)
```


OpenTimer2

Syntax
OpenTimer2(pConfig)

Include file
Timers.inc

Overview
This macro configures timer2 according to the options specified and then enables it.

Operator
pConfig A bitmask that is created by performing either a bitwise AND operation ('&'), which is user configurable, with a value from each of the categories listed below. These values are defined in the file TimerDefs.inc.

Enable Timer2 Interrupt:

TIMER_INT_ON	Interrupt enabled
TIMER_INT_OFF	Interrupt disabled

Prescale Value:

T2_PS_1_1	1:1 prescale
T2_PS_1_4	1:4 prescale
T2_PS_1_16	1:16 prescale

Postscale Value:

T2_POST_1_1	1:1 postscale
T2_POST_1_2	1:2 postscale
T2_POST_1_3	1:3 postscale
T2_POST_1_4	1:4 postscale
T2_POST_1_5	1:5 postscale
T2_POST_1_6	1:6 postscale
T2_POST_1_7	1:7 postscale
T2_POST_1_8	1:8 postscale
T2_POST_1_9	1:9 postscale
T2_POST_1_10	1:10 postscale
T2_POST_1_11	1:11 postscale
T2_POST_1_12	1:12 postscale
T2_POST_1_13	1:13 postscale
T2_POST_1_14	1:14 postscale
T2_POST_1_15	1:15 postscale
T2_POST_1_16	1:16 postscale

Example

```
OpenTimer2(TIMER_INT_OFF & T2_PS_1_1 & T2_POST_1_8)
```


OpenTimer3

Syntax
OpenTimer3(pConfig)

Include file
Timers.inc

Overview
This macro configures timer3 according to the options specified and then enables it.

Operator
pConfig A bitmask that is created by performing either a bitwise AND operation ('&'), which is user configurable, with a value from each of the categories listed below. These values are defined in the file TimerDefs.inc.

Enable Timer3 Interrupt:

TIMER_INT_ON	Interrupt enabled
TIMER_INT_OFF	Interrupt disabled

Timer Width:

T3_8BIT_RW	8-bit mode
T3_16BIT_RW	16-bit mode

Clock Source:

T3_SOURCE_EXT	External clock source (I/O pin)
T3_SOURCE_INT	Internal clock source (Tosc)

Prescale Value:

T3_PS_1_1	1:1 prescale
T3_PS_1_2	1:2 prescale
T3_PS_1_4	1:4 prescale
T3_PS_1_8	1:8 prescale

Synchronise Clock Input:

T3_SYNC_EXT_ON	Sync external clock input
T3_SYNC_EXT_OFF	Don't sync external clock input

Example

```
OpenTimer3(T3_8BIT_RW & T3_SOURCE_EXT & T3_PS_1_1 & T3_SYNC_EXT_OFF)
```


ReadTimer0

Syntax
Variable = **ReadTimer0**()

Include file
Timers.inc

Overview
This macro reads the value of the timer0 register pair.
Timer0: TMR0L,TMR0H

ReadTimer1

Syntax
Variable = **ReadTimer1**()

Include file
Timers.inc

Overview
This macro reads the value of the timer1 register pair.
Timer1: TMR1L,TMR1H

ReadTimer2

Syntax
Var = **ReadTimer2**()

Include file
Timers.inc

Overview
This macro reads the value of the timer2 register.
Timer2: TMR2

ReadTimer3

Syntax
Variable = **ReadTimer3**()

Include file
Timers.inc

Overview
This macro reads the value of the timer3 register pair.
Timer3: TMR3L,TMR3H

WriteTimer0

Syntax
WriteTimer0(pTimer)

Include file
Timers.inc

Overview
This macro writes a value to the timer0 register pair:
Timer0: TMR0L,TMR0H

Operator
pTimer The value that will be loaded into timer0.

Example
```
WriteTimer0(10000)
```

WriteTimer1

Syntax
WriteTimer1(pTimer)

Include file
Timers.inc

Overview
This macro writes a value to the timer1 register pair:
Timer1: TMR1L,TMR1H

Operator
pTimer The value that will be loaded into timer1.

Example
```
WriteTimer1(10000)
```

WriteTimer2

Syntax
WriteTimer2(pTimer)

Include file
Timers.inc

Overview
This macro writes a value to the timer1 register:
Timer2: TMR2

Operator
pTimer The value that will be loaded into timer2.

Example
```
WriteTimer2(100)
```


WriteTimer3

Syntax
WriteTimer3(pTimer)

Include file
Timers.inc

Overview
This macro writes a value to the timer1 register pair:
Timer3: TMR3L,TMR3H

Operator
pTimer The value that will be loaded into timer3.

Example
```
WriteTimer3(10000)
```

SetTmrCCPSrc

Syntax
SetTmrCCPSrc(pConfig)

Include file
Timers.inc

Overview
This macro configures a timer as a clock source for the CCP module.

Operator
pConfig A constant value from the list below. The values are defined in the file TimerDefs.inc.

T3_SOURCE_CCP	Timer3 source for both CCP's
T1_CCP1_T3_CCP2	Timer1 source for CCP1 and Timer3 source for CCP2
T1_SOURCE_CCP	Timer1 source for both CCP's

Example
```
SetTmrCCPSrc(T34_SOURCE_CCP12)
```


T3_OSC1EN_ON

Syntax
T3_OSC1EN_ON()

Include file
Timers.inc

Overview
This Macro enables the oscillator associated with Timer1 as source of external clock input for Timer3.

T3_OSC1EN_OFF

Syntax
T3_OSC1EN_OFF()

Include file
Timers.inc

Overview
This Macro disables the oscillator associated with Timer1 and selects the signal on pin T13CKI as source of external clock input for Timer3.

Example Use of the Timer0 Macro:

```
  Include "Timers.Inc"              ' Load the Timer Macros into the program

  Dim Result As Word

' Configure Timer0
  OpenTimer0(TIMER_INT_OFF & T0_SOURCE_INT & T0_PS_1_32 & T0_16BIT)

  HRSOut "Press a Key\r"
  While 1 = 1
    While InKey = 16 : Wend         ' Wait for a Keypress on the keypad
    Result = ReadTimer0()           ' Read Timer0
    WriteTimer0(0)                  ' Reset Timer0
    HRSOut "Timer0 Value = ", Dec Result,13 ' Display the value of Timer0
    While InKey <> 16 : Wend        ' Wait for the key to released
    DelayMS 50
  Wend
  CloseTimer0()                     ' Close timer0
```


SPI macros Introduction

The following macros are provided for the SPI™ peripheral:

- CloseSPI — Disable the SSP module used for SPI™ communications.
- DataReadySPI — Determine if a new value is available from the SPI buffer.
- OpenSPI — Initialise the SSP module used for SPI communications.
- ReadSPI — Read a byte from the SPI bus.
- WriteSPI — Write a byte to the SPI bus.

CloseSPI

Syntax
CloseSPI()

Include file
SPI.inc

Overview
This Macro disables the SSP module. Pin I/O returns under the control of the appropriate TRIS and LAT registers.

DataReadySPI

Syntax
Variable = **DataReadySPI**()

Include file
SPI.inc

Overview
This Macro determines if there is a byte to be read from the SSPBUF register.

Return Values
0 if there is no data in the SSPBUF register
1 if there is data in the SSPBUF register

Example
```
While DataReadySPI() = 0 : Wend
```


OpenSPI

Syntax
OpenSPI(pSyncMode, pBusMode, pSmpPhase)

Include file
SPI.inc

Overview
This Macro sets up the SSPx module for use with a SPIx bus device.

Operators
pSyncMode One of the following values, defined in SPIdefs.inc:

SPI_FOSC_4	SPI Master mode, clock = Fosc / 4
SPI_FOSC_16	SPI Master mode, clock = Fosc / 16
SPI_FOSC_64	SPI Master mode, clock = Fosc / 64
SPI_FOSC_TMR2	SPI Master mode, clock = TMR2 output / 2
SLV_SSON	SPI Slave mode, /SS pin control enabled
SLV_SSOFF	SPI Slave mode, /SS pin control disabled

pBusMode One of the following values, defined in SPIdefs.inc:

MODE_00	Setting for SPI bus Mode 0,0
MODE_01	Setting for SPI bus Mode 0,1
MODE_10	Setting for SPI bus Mode 1,0
MODE_11	Setting for SPI bus Mode 1,1

pSmpPhase One of the following values, defined in SPIdefs.inc:

SMPEND	Input data sample at end of data out
SMPMID	Input data sample at middle of data out

Example
```
OpenSPI(SPI_FOSC_16, MODE_00, SMPEND)
```


ReadSPI

Syntax
Variable = **ReadSPI**()

Include file
SPI.inc

Overview
This Macro initiates a SPIx bus cycle for the acquisition of a byte of data.

WriteSPI

Syntax
WriteSPI(pDataOut)

Include file
SPI.inc

Overview
This Macro writes a single data byte out.

Operator
pDataOut Value to be written to the SPI bus.

Example of SPI macros

```
Include "SPI.inc"

Dim SPI_data as Byte

OpenSPI(SPI_FOSC_64 , MODE_01 , SMPMID)
WriteSPI($55)
SPI_data = ReadSPI()
DataReadySPI()
CloseSPI()
```


Analogue Comparator macro Introduction

The Analogue comparator peripheral is supported with the following macros:

- CloseComp1 Disable the Analogue comparator1.
- CloseComp2 Disable the Analogue comparator2.
- OpenComp1 Configure and Enable the Analogue comparator1.
- OpenComp2 Configure and Enable the Analogue comparator2.

CloseComp1

Syntax
CloseComp1()

Include file
AnComp.inc

Overview
This macro disables the Analogue comparator1 and its interrupt mechanism.

CloseComp2

Syntax
CloseComp2()

Include File
AnComp.inc

Overview
This macro disables the Analogue comparator2 and its interrupt mechanism.

Comp1_IntEnable
Comp2_IntEnable

Syntax
Comp1_IntEnable()
Comp2_IntEnable()

Include File
AnComp.inc

Overview
These macros enable the comparator1 or comparator2 interrupt mechanism.

Comp1_IntDisable
Comp2_IntDisable

Syntax
Comp1_IntDisable()
Comp2_IntDisable()

Include File
AnComp.inc

Overview
These macros disable the comparator1 or comparator2 interrupt mechanism.

OpenComp1

Syntax
OpenComp1(pConfig)

Inlcude file
AnComp.inc

Overview
This macro resets the Analogue comparator peripheral to the POR (Power On Reset) state and configures the Analogue comparator related SFRs (Special Function Registers) according to the options specified, also configures the inputs and outputs for the specified options.

Operator
pConfig A constant bitmask that is created by performing a bitwise AND operation ('&'), user configurable, with a value from each of the categories listed below. These values are defined in the file AnComp.inc.

Comparator1 Output enable:

COMP_OP_EN	C1OUT is present on C1OUT pin (PortA.4)
COMP_OP_DIS	C1OUT is Internal only

Comparator1 output polarity select:

COMP_OP_INV	Comparator1 output logic is Inverted
COMP_OP_NINV	Comparator1 output logic is not Inverted

Comparator1 Ref (C1VREF)select:

COMP_REF_FVR	C1VREF input connects to FVR (fixed 1.2V)
COMP_REF_CVREF	C1VREF input connects to CVREF

Comparator1 Speed/Power select:

COMP_HSPEED	Comparator1 operates at normal power higher speed mode
COMP_LSPEED	Comparator1 operates at low power low speed mode

Comparator1 Ref (C1VIN+) select:

COMP_VINP_PIN	C1VIN+ connects to C1VIN+ pin
COMP_VINP_VREF	C1VIN+ connects to C1VREF output

Comparator1 channel select:

COMP_VINM_IN0	C12IN0- pin connects to C1VIN-
COMP_VINM_IN1	C12IN1- pin connects to C1VIN-
COMP_VINM_IN2	C12IN2- pin connects to C1VIN-
COMP_VINM_IN3	C12IN3- pin connects to C1VIN-

Example

```
OpenComp1(COMP_OP_NINV & COMP_OP_EN & COMP_REF_CVREF & COMP_VINM_IN0)
```


OpenComp2

Syntax
OpenComp2(pConfig)

Inlcude file
AnComp.inc

Overview
This macro resets the analogue comparator 2 peripheral to the POR (Power On Reset) state and configures the analogue comparator 2 related SFRs (Special Function Registers) according to the options specified, also configures the inputs and outputs for the specified options.

Operator
pConfig A constant bitmask that is created by performing a bitwise AND operation ('&'), user configurable, with a value from each of the categories listed below. These values are defined in the file AnComp.inc.

Comparator2 Output enable:

COMP_OP_EN	C2OUT is present on C2OUT pin (PortA.5)
COMP_OP_DIS	C2OUT is Internal only

Comparator2 output polarity select:

COMP_OP_INV	Comparator2 output logic is Inverted
COMP_OP_NINV	Comparator2 output logic is not Inverted

Comparator2 Ref (C2VREF)select:

COMP_REF_FVR	C2VREF input connects to FVR (fixed 1.2V)
COMP_REF_CVREF	C2VREF input connects to CVREF

Comparator2 Speed/Power select:

COMP_HSPEED	Comparator2 operates at normal power higher speed mode
COMP_LSPEED	Comparator2 operates at low power low speed mode

Comparator2 Ref (C2VIN+) select:

COMP_VINP_PIN	C2VIN+ connects to C2VIN+ pin
COMP_VINP_VREF	C2VIN+ connects to C2VREF output

Comparator2 channel select:

COMP_VINM_IN0	C12IN0- pin connects to C2VIN-
COMP_VINM_IN1	C12IN1- pin connects to C2VIN-
COMP_VINM_IN2	C12IN2- pin connects to C2VIN-
COMP_VINM_IN3	C12IN3- pin connects to C2VIN-

Example

```
OpenComp2(COMP_OP_NINV & COMP_OP_EN & COMP_REF_CVREF & COMP_VINM_CIN0)
```


Hardware PWM macro Introduction

The PWM peripheral is supported with the following macros:

- CloseAnalog1 Disable the CCP1 peripheral
- CloseAnalog2 Disable the CCP2 peripheral
- OpenAnalog1 Enable and configure the CCP1 peripheral
- OpenAnalog2 Enable and configure the CCP2 peripheral
- WriteAnalog1 Output an 8-bit or 10-bit Pulse Width Modulated waveform from CCP1
- WriteAnalog2 Output an 8-bit or 10-bit Pulse Width Modulated waveform from CCP2

CloseAnalog1

Syntax
CloseAnalog1()

Include file
hpwm8.inc for 8-bit PWM
or
hpwm10.inc for 10-bit PWM

Overview
Disable the CCP1 peripheral and set it's appropriate pin as an input.

CloseAnalog2

Syntax
CloseAnalog2()

Include file
hpwm8.inc for 8-bit PWM
or
hpwm10.inc for 10-bit PWM

Overview
Disable the CCP2 peripheral and set it's appropriate pin as an input.

OpenAnalog1

Syntax
OpenAnalog1()

Include file
hpwm8.inc for 8-bit PWM
or
hpwm10.inc for 10-bit PWM

Overview
Enable and configure the CCP1 peripheral and set it's appropriate pin as an output.

OpenAnalog2

Syntax
OpenAnalog2()

Include file
hpwm8.inc for 8-bit PWM
or
hpwm10.inc for 10-bit PWM

Overview
Enable and configure the CCP2 peripheral and set it's appropriate pin as an output.

WriteAnalog1

Syntax
WriteAnalog1(pValue)

Include file
hpwm8.inc for 8-bit PWM
or
hpwm10.inc for 10-bit PWM

Note. The CCP1 peripheral will be operating at the highest frequency possible for 8-bit (0 to 255) or 10-bit (0 to 1023). With the default 64MHz oscillator this will be 62.5KHz for 10-bit and 250KHz for 8-bit.

Only one of the above include files may be used within a program an any one time.

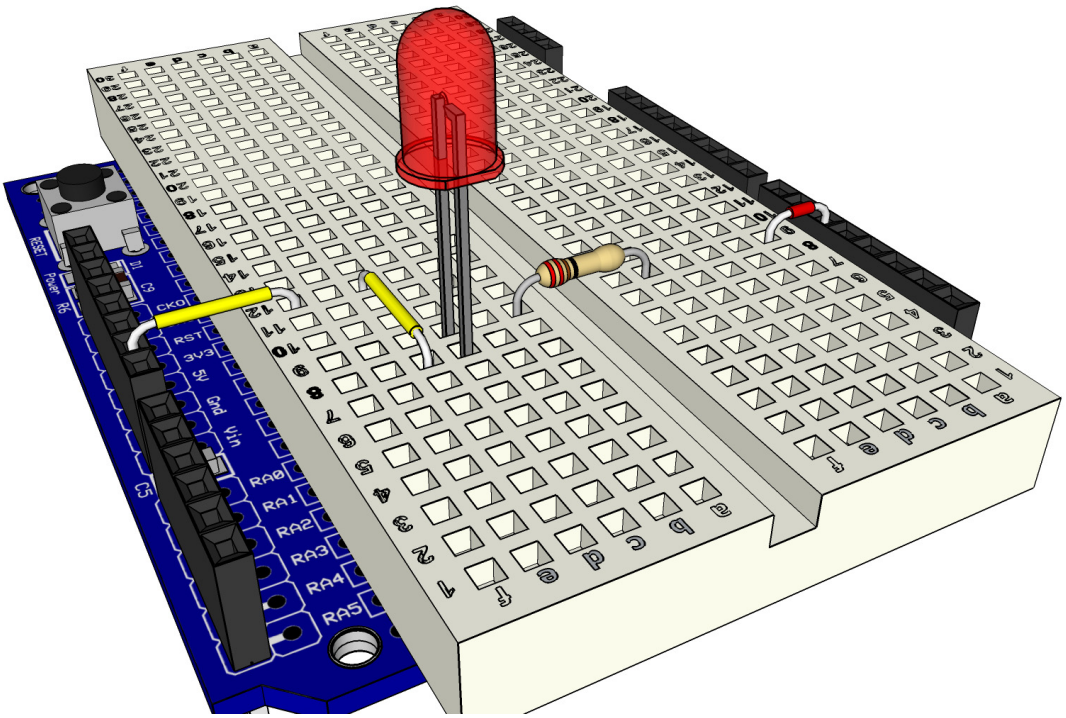
Overview
Output an 8-bit or 10-bit PWM waveform from the CCP1 peripheral's pin (RC2).

Operator
pValue a constant, variable, or expression that will alter the duty cycle of the PWM Waveform.

Example

```
' An LED attached to RC2 will increase illumination, then dim, repeatedly
' The voltage produced by the PWM signal is displayed on the serial terminal
'
   Include "Hpwm10.inc"                  ' Load the 10-bit PWM macros into the program
   Float_Display_Type = fast             ' Faster, more accurate float display
   Dim Volts As Float                    ' Holds the Voltage calculation
   Dim WordVar As Word                   ' Holds the duty cycle value for the PWM
'
' Quantasise the Voltage. i.e. Volts per-bit, based upon 3.3 Volts at 10-bits
'
   Symbol Quanta = 3.3 / 1023
   OpenAnalog1()                         ' Enable and configure the CCP1 peripheral
   While 1 = 1                           ' Create an infinite loop
      '
      ' Increase LED illumation
      ' Cycle the full range of 10-bits. i.e. 0 to 1023
      For WordVar = 0 To 1023
         WriteAnalog1(WordVar)           ' PWM on CCP1 (Bit-2 of PortC)
         Volts = WordVar * Quanta        ' Calculate the Voltage
         HRSOut Dec WordVar, " = ", Dec Volts, " Volts", 13 ' Display Voltage
      Next
      '
      ' Decrease LED illumination
      ' Cycle the full range of 10-bits (reversed). i.e. 1023 to 0
      For WordVar = 1023 To 0 Step -1
         WriteAnalog1 (WordVar)          ' PWM on CCP1 (Bit-2 of PortC)
         Volts = WordVar * Quanta        ' Calculate the Voltage
         HRSOut Dec WordVar, " = ", Dec Volts, " Volts", 13 ' Display Voltage
      Next
   Wend                                  ' Do it forever
```

A suitable layout for the above program built on the Companion Shield using a solderless breadboard is shown below:



WriteAnalog2

Syntax
WriteAnalog2(pValue)

Include file
hpwm8.inc for 8-bit PWM
or
hpwm10.inc for 10-bit PWM

Note. The CCPx peripherals will be operating at the highest frequency possible for 8-bit (0 to 255) or 10-bit (0 to 1023). With the default 64MHz oscillator this will be 62.5KHz for 10-bit and 250KHz for 8-bit.

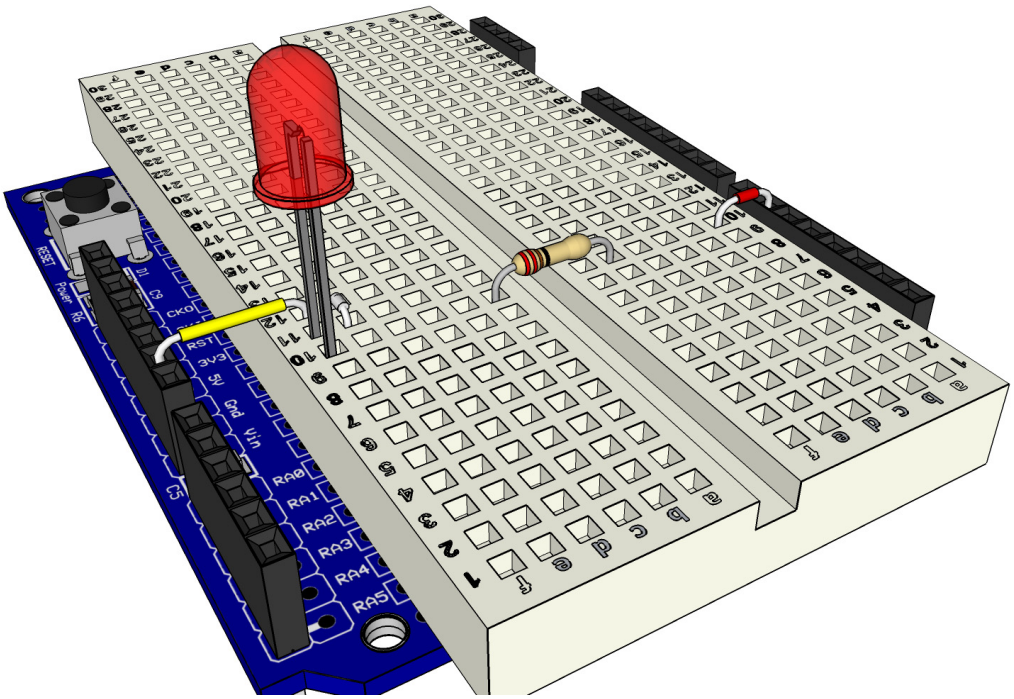
Only one of the above include files may be used within a program an any one time.

Overview
Output an 8-bit or 10-bit PWM waveform from the CCP2 peripheral's pin (RC1).

Operator
pValue a constant, variable, or expression that will alter the duty cycle of the PWM Waveform.

Example
```
' An LED attached to RC1 will increase illumination, then dim, repeatedly
' The voltage produced by the PWM signal is displayed on the serial terminal
'
   Include "Hpwm10.inc"             ' Load the 10-bit PWM macros into the program
   Float_Display_Type = fast        ' Faster, more accurate float display
   Dim Volts As Float               ' Holds the Voltage calculation
   Dim WordVar As Word              ' Holds the duty cycle value for the PWM
' Quantasise the Voltage. i.e. Volts per-bit, based upon 3.3 Volts at 10-bits
   Symbol Quanta = 3.3 / 1023
   OpenAnalog2()                        ' Enable and configure the CCP2 peripheral
   While 1 = 1                          ' Create an infinite loop
      '
      ' Increase LED illumation
      ' Cycle the full range of 10-bits. i.e. 0 to 1023
      For WordVar = 0 To 1023
         WriteAnalog2(WordVar)          ' PWM on CCP2 (Bit-1 of PortC)
         Volts = WordVar * Quanta       ' Calculate the Voltage
         HRSOut Dec WordVar, " = ", Dec Volts, " Volts", 13 ' Display Voltage
      Next
      '
      ' Decrease LED illumination
      ' Cycle the full range of 10-bits (reversed). i.e. 1023 to 0
      For WordVar = 1023 To 0 Step -1
         WriteAnalog2 (WordVar)         ' PWM on CCP1 (Bit-1 of PortC)
         Volts = WordVar * Quanta       ' Calculate the Voltage
         HRSOut Dec WordVar, " = ", Dec Volts, " Volts", 13 ' Display Voltage
      Next
   Wend                                 ' Do it forever
```


A suitable layout for the previous program built on the Companion Shield using a solderless breadboard is shown below:



Protected Compiler Words

Below is a list of protected words that the compiler or the Mpasm assembler uses internally. Be sure not to use any of these words as variable or label names, otherwise errors will be produced.

(A)
Abs, Acos, Actual_Banks, ADC_Resolution, AdcIn, Addlw, Addwf, Addwfc, Adin, Adin_Delay, Adin_Res, Adin_Stime, Adin_Tad, All_Digital, Amicus18_Start_Address, Andlw, Asin, Asm, Atan, Auto_Context_Save, Available_RAM
(B)
Bank0_End, Bank0_Start, Bank10_End, Bank10_Start, Bank11_End, Bank11_Start, Bank12_End, Bank12_Start, Bank13_End, Bank13_Start, Bank14_End, Bank14_Start, Bank15_End, Bank15_Start, Bank1_End, Bank1_Start, Bank2_End, Bank2_Start, Bank3_End, Bank3_Start, Bank4_End, Bank4_Start, Bank5_End, Bank5_Start, Bank6_End, Bank6_Start, Bank7_End, Bank7_Start, Bank8_End, Bank8_Start, Bank9_End, Bank9_Start, Bank_Select_Switch, BankiSel, BankSel, Bc, Bcf, Bin, Bin1, Bin10, Bin11, Bin12, Bin13, Bin14, Bin15, Bin16, Bin17, Bin18, Bin19, Bin2, Bin20, Bin21, Bin22, Bin23, Bin24, Bin25, Bin26, Bin27, Bin28, Bin29, Bin3, Bin30, Bin31, Bin32, Bin4, Bin5, Bin6, Bin7, Bin8, Bin9, Bit, Bn, Bnc, Bnn, Bnov, Bnz, Bootloader, Bov, Box, Bra, Branch, Branchl, Break, Brestart, Bsf, Bstart, Bstop, Btfsc, Btfss, Btg, Bus_DelayMs, Bus_SCL, BusAck, Busin, Busout, Button, Button_Delay, Byte, Byte_Math, BZ, Bit_Bit, Bit_Byte, Bit_Dword, Bit_Float, Bit_Word, Bit_Wreg, Byte_Bit, Byte_Byte, Byte_Dword, Byte_Float, Byte_Word, Byte_Wreg
(C)
Call, Case, Cblock, CCP1_Pin, CCP2_Pin, CCP3_Pin, CCP4_Pin, CCP5_Pin, Cdata, Cerase, CF_ADPort, CF_ADPort_Mask, CF_CD1Pin, CF_CE1Pin, CF_DTPort, CF_Init, CF_OEPin, CF_RDYPin, CF_Read, CF_Read_Write_Inline, CF_RSTPin, CF_Sector, CF_WEPin, CF_Write, Chr$, Circle, Clear, ClearBit, Clrf, Clrw, Cls, Code, Comf, Config, Constant, Context, Core, Cos, Count, Counter, Cpfseq, Cpfsgt, Cpfslt, Cread, Cursor, Cwrite
(D)
Da, Data, Daw, Db, Dc, Dcd, Dcfsnz, De, Dead_Code_Remove, Dword_Bit, Dword_Byte, Dword_Dword, Dword_Float, Dword_Word, Dword_Wreg, Debug_Req, Debugin, Dec, Dec, Dec1, Dec1, Dec10, Dec2, Dec2, Dec3, Dec3, Dec4, Dec4, Dec5, Dec5, Dec6, Dec6, Dec7, Dec7, Dec8, Dec8, Dec9, Decf, Decfsz, Declare, Decrement, Define, DelayMs, DelayUs, Device, Dig, Dim, Disable, Div32, Djc, Djnc, Djnz, Djz, Dt, DTMFout, Dw, Dword
(E)
Edata, Eeprom_Size, Else, ElseIf, Enable, End, EndAsm, EndIf, EndM, EndSelect, Equ, Eread, Error, ErrorLevel, Ewrite, ExitM, Exp, Expand
(F)
Fill, Fix16_8Add, Fix16_8Div, Fix16_8Greater, Fix16_8GreaterEqual, Fix16_8Less, Fix16_8LessEqual, Fix16_8Mul, Fix16_8Sub, Fix16_8ToFloat, Fix16_8ToInt, Fix8_8Add, Fix8_8Div, Fix8_8Greater Fix8_8GreaterEqual, Fix8_8Less, Fix8_8LessEqual, Fix8_8Mul, Fix8_8Sub, Fix8_8ToFloat, Fix8_8ToInt, Flash_Capable, Float, Float_Display_Type, Float_Rounding, Float_Rounding_Code, FloatToFix16_8, FloatToFix8_8, Font_Addr, For, FreqOut, Float_Bit, Float_Byte, Float_Dword, Float_Float, Float_Word, Float_Wreg
(G)
GetBit, GLCD_CS_Invert, GLCD_Fast_Strobe, GLCD_Read_Delay, GLCD_Strobe_Delay, GoSub, GoTo
(H)
HbRestart, HbStart, HbStop, Hbus_Bitrate, HbusAck, Hbusin, Hbusout, Hex, Hex1, Hex2, Hex3, Hex4, Hex4, Hex5, Hex6, Hex7, Hex8, High, High_Int_Sub_End, High_Int_Sub_Start, HighLow_Tris_Reverse, Hpwm, HRsin, Hrsin2, HRsout, HRsout2, Hserial2_Baud, Hserial2_Clear, Hserial2_Parity, Hserial2_RCSTA, Hserial2_SPBRG, Hserial2_TXSTA, Hserial_Baud, Hserial_Clear, Hserial_Parity, Hserial_RCSTA, Hserial_SPBRG, Hserial_TXSTA, Hserin, Hserin2, Hserout, Hserout2

(I)
I2C_Bus_SCL, I2C_Slow_Bus, I2Cin, I2Cout, I2CWrite, I2CRead, ICD_Req, Icos, Idata, If, Ijc, Ijnc, Ijnz, Ijz, Inc, Incf, Incfsz, Include, Increment, Infsnz, Inkey, Input, Int_Sub_End, Int_Sub_Start,
Internal_Bus, Internal_Font, IntToFix16_8, IntToFix8_8, Iorlw, Iorwf, IrIn, IrIn_Pin, Isin, ISqr
(K)
Keep_Hex_File, Keyboard_CLK_Pin, Keyboard_DTA_Pin, Keyboard_In, Keypad_Port
(L)
Label_Word, Label_Bank_Resets, LCD_CDPin, LCD_CEPin, LCD_CommandUS, LCD_CS1Pin, LCD_CS2Pin, LCD_DataUs, LCD_DTPin, LCD_DTPort, LCD_ENPin, LCD_Font_HEIGHT, LCD_Font_Width, LCD_Graphic_Pages, LCD_Interface, LCD_Lines, LCD_RAM_Size, LCD_RDPin, LCD_RSPin, LCD_RSTPin, LCD_RWPin, LCD_Text_Home_Address, LCD_Text_Pages, LCD_Type, LCD_WRPin, LCD_X_Res, LCD_Y_Res, LCDread, LCDwrite, Cdata, Left$, Len, Let, Lfsr,
Library_Core, Line, LineTo, LoadBit, Local, Log, Log10, LookDown,
LookDownL, LookUp, LookUpL, Low, Low_Int_Sub_End, Low_Int_Sub_Start, Lread, Lread16, Lread32, Lread8
(M)
Macro_Params, Max, Mid$, Min, Mouse_CLK_Pin, Mouse _Data_Pin, Mouse _In, Movf, Movff, Movlw, Movwf, MSSP_Type, Mullw, Mulwf
(N)
Ncd, Negf, Next, Nop, Num _Bit, Num _Byte, Num_Dword, Num _Float, Num _FSR,
Num _FSR0, Num _FSR2, Num _Word, Num _Wreg, Num_SFR
(O)
On_Hard_Interrupt, On_Hardware_Interrupt, On_Interrupt, On_Low_Interrupt, On_Soft_Interrupt, On_Software_Interrupt,
Onboard_Adc, Onboard_Uart, Onboard_Usb, Optimise_Bit_Test,
Optimiser_Level, Oread, Org, Output, Owin, Owout, Owrite
(P)
Page, PageSel, Pause, Pauseus, Peek, PIC_Pages, Pixel, PLL_Req, Plot, Poke, Pop, Portb_Pullups, Pot, Pow, Print, Prm_1, Prm_10, Prm_11, Prm_12, Prm_13, Prm_14, Prm_15, Prm_2, Prm_3, Prm_4, Prm_5, Prm_6, Prm_7, Prm_8, Prm_9, Prm_Count, PulsIn, PulsIn_Maximun, PulseOut, Push, Pwm
(R)
RAM_Bank, RAM_Banks, Random, RC5in, RC5in_Extended, RC5in_Pin, RCall, RCin, RcTime, Read, Rem, Remarks, Reminders, Rep, Repeat, RES, Reserve_RAM, Reset_Bank, Restore, Resume, Retfie, Retlw, Return, Return_Type, Return_Var, Rev, Right$, Rlcf, Rlf, Rlncf, Rol, Ror, Rrcf, Rrf, Rrncf, Rsin, Rsin_Mode, Rsin_Pin, Rsin_Timeout, Rsout, Rsout_Baud, Rsout_Mode, Rsout_Pace, Rsout_Pin,
Return_Bit, Return_Byte, Return_Dword, Return_Float,
Return_Word, Return_Wreg
(S)
SBreak, SCL_Pin, SDA_Pin, Seed, Select, Serial_Baud, Serial_Data, Serial_Parity, Serin, Serout, Servo, Set, Set_Bank, Set_Defaults, Set_OSCCAL, SetBit, SetF, Shift_DelayUs, ShiftIn, Shin, Shout, Show_Expression_Parts, Show_System_Variables, Signed_Dword_Terms, Sin, Single_Page_Model, SizeOf, Sleep, Slow_Bus, Small_Micro_Model, Snooze, SonyIn, SonyIn_Pin, Sound, Sound2, Sqr, Stack_Size, Stamp _Cos, Stamp_Sin, Stamp _Sqr, Step, Stop, Str, Str$, Str$, StrCmp, String, Strn, Subfwb, Sublw, Subwf, Subwfb, Swap, Swapf, Symbol
(T)
Tan, Tblrd, Tblwt, TCase, Then, to, Toggle, ToLower, Toshiba_Command, Toshiba_UDG, ToUpper, Tstfsz
(U)
Udata, UnPlot, Unsigned_Dwords, Until, Upper
(V)
Val, Var, Variable, VarPtr

(W)
Wait, Warnings, WatchDog, Wend, While, Word, Write, Word_Bit, Word_Byte, Word_Dword, Word_Float, Word_Word, Word_Wreg, Wreg_Bit, Wreg_Byte, Wreg_Dword, Wreg_Float, Wreg_Word
(X)
Xin, Xorlw, Xorwf, Xout, Xtal

_adc, _adcres, _code, _core, _defined, _device, _eeprom, _flash, _mssp, _ports, _ram, _uart
_usb, _xtal

Notes.

