Introducing Great Cow BASIC

Hello, and welcome to Great Cow BASIC help. This help file is intended to act as a backup to several other documents - to clarify topics which may not be made clear adequately in other places.
For information on installing GCBASIC and several other programs that may be helpful, please see the "Getting Started with Great Cow BASIC" page at http://gcbasic.sourceforge.net/starting.html.

If you're new to programming, you should try the Great Cow BASIC tutorial. This explains everything in a step-by-step manner, and assumes no prior knowledge.

If you've programmed in another language, then the demonstration files and command reference may be the best place to turn.

If there's anything else that you need help on, please visit the Great Cow BASIC forum.

Using GCBASIC

About GCBASIC

Need to compile a program with Great Cow BASIC, but don't know where to begin? Try these instructions:

1. First, download and install GCBASIC. It's best if you let everything install to its default directory (folder).
2. Open up the folder that contains GCBASIC. This is generally C:\Program Files\GCBASIC.
3. Open the folder that contains the program you want to compile. Make sure that you open it in a new window.
4. Drag the icon of the file you want to compile onto "gcbasic.exe". The compiler will open, compile the file, and then close again.
5. A file called "compiled.hex" will have been created. Download this to your microcontroller*, and you're set to go!

Additional, more detailed instructions are available on the Getting Started with GCBASIC page.

*You need a suitable programmer to do this, and instructions should be included with the programmer on how to download to the microcontroller.

Command Line Parameters

About the Command Line Parameters

GCBASIC [/O:output.asm] [/A:assembler] [/P:programmer] [/K:{C|A}] [/V] [/L] [/NP] filename

Switch	Description	Default
/O:*filename*	Sets the name of the assembly file generated to *filename*.	Same name as the input file, but with a .asm extension.
/A:assembler	Batch file used to call assembler*. If /A:GCASM is given, GCBASIC will use its internal assembler.	The program will not be assembled
/P:programmer	Batch file used to call programmer*. This parameter is ignored if the program is not assembled.	The program will not be downloaded.
/K:{C|A}	Keep original code in assembly output. /K:C will save comments, /K:A will preserve all input code.	No original code left in output.
/V	Verbose mode - compiler gives more detailed information about its activities.	-
/L	Show license and exit.	-
/NP	Do not pause on errors. Use with IDEs.	Pause when an error occurs, and wait for the user to press a key.
filename	The file to compile.	-

* For the /A: and /P: switches, there are special options available. If %FILENAME% is present, it will be replaced by the name of the .asm file. %FN_NOEXT% will be replaced by the name of the .asm file but without an extension, and %CHIPMODEL% will be replaced with the name of the chip. The name of the chip will be the same as that on the chip data file.

A batch to to load the ASM from GCBASIC into MPASM. I think that your m.bat should be like this:

```
@ECHO OFF

echo C:\progra~1\microc~1\mpasms~1\MPASMWIN /c- /o- /q+ /l- /x- /w1 %code%.asm

C:\progra~1\microc~1\mpasms~1\MPASMWIN /c- /o- /q+ /l- /x- /w1 %code%.asm
```

Frequently Asked Questions

Why doesn't anything come up when I run gcbasic.exe?

Great Cow BASIC is a command line compiler. To compile a file, you can drag and drop it onto the gcbasic.exe icon. There are also several Integrated Development Environments, or IDEs, available for GCBASIC. These will give you an area where you can edit your program and a button to send the program to the chip. Several are listed on the GCBASIC website.

What chips does Great Cow BASIC support?

Hopefully, all 8 bit PIC chips (those in the PIC10, PIC12, PIC16 and PIC18 families). If you find one that GCBASIC does not work with properly, please post about it in the Compiler Problems section of the GCBASIC forum. Support for Atmel AVR chips has been added.

Is Great Cow BASIC case sensitive?

No! For example, Set, SET, set, SeT, etc are all treated exactly the same way by GCBASIC.

Can I specify the bit of a variable to alter using another variable?

No. Set variable.othervariable On may not generate an error, but it will not act as expected.

Why is x feature not implemented?

Because it hasn't been thought of, or no-one has been able to implement it! If there are any features that you would like to see in Great Cow BASIC, please post them in the "Open Discussion" section of the GCBASIC forum. Or, if you can, have a go at adding the feature yourself!

Acknowledgements

Developers and Contributors

Hugh Considine - Main developer of Great Cow Basic
Stefano Bonomi - Two-wire LCD subroutines
Geordie Millar - Swap and Swap4 subroutines
Finn Stokes - 8-bit multiply routine, program memory access code
Evan Venn - Utilities, revised I2C routines and the update to this help file

Translation Contributors

Stefano Delfiore - Italian
Pablo Curvelo - Spanish
Murat Inceer - Turkish

Other Contributors

Russ Hensel - Great Cow BASIC Notes
Chuck Hellebuyck - His documentation for the GLCD and other pieces, see here.
Frank Steinberg - GCB@SYN IDE for Great Cow Basic, see here.
Alexy T. - SynWrite IDE used for GCB IDE, see here.
Thomas Henry for the Select Case and the Sine Table examples, see here and here respectively.

Conversion of Help File to PDF

Zamzar.com

Inputs/Outputs

About Inputs and Outputs

Most general purpose pins on a microcontroller can function in one of two modes: input mode, or output mode.

When acting as an input, the pin will be placed in high impedance state. The microcontroller will then sense the pin, and the program can read the state of the pin and make decisions based on it.

When in output mode, the microcontroller will connect the pin to either Vcc (the positive supply), or Vss (ground, or the negative supply). The program can then set the state of the pin.

GCBASIC will attempt to determine the direction of each pin, and set it appropriately. However, if the pin is read from and written to, then then pin must be configured to input / output mode by the program, using the appropriate Dir commands.

Configuration

About Configuration

*(Note: This section does not apply to AVR microcontrollers. AVR chips do have a similar configuration settings, but they are controlled through "Configuration Fuses". GCBASIC cannot set these - you **must** use the programmer software.)*

Every PIC chip has a CONFIG word. This is an area of memory on the chip that stores settings which govern the operation of the chip.

The following aspects of the chip are governed by the CONFIG word:

- Oscillator selection - will the chip run from an internal oscillator, or is an external one attached?
- Automatic resets - should the chip reset if the power drops too low? If it detects it is running the same piece of code over and over?
- Code protection - what areas of memory must be kept hidden once written to?
- Pin usage - which pins are available for programming, resetting the chip, or emitting PWM signals?

The exact configuration settings vary amongst chips. To find out a list of valid settings, please consult the datasheet for the PIC chip that you wish to use.

This can all be rather confusing - hence, GCBASIC will automatically set some config settings, unless told otherwise:

- **Low Voltage Programming (LVP) is turned off.** This enables the PGM pin (usually B3 or B4) to be used as a normal I/O pin.
- **Watchdog Timer (WDT) is turned off.** The WDT resets the chip if it runs the same piece of code over and over - this can cause trouble with some of the longer delay routines in GCBASIC.
- **Master Clear (MCLR) is disabled where possible.** On many newer chips this allows the MCLR pin (often PORTA.5) to be used as a standard input port. It also removes the need for a pull-up resistor on the MCLR pin.
- **An oscillator mode will be selected, based on the following rules:**
    1. If the PIC has an internal oscillator, and the internal oscillator is capable of generating the speed specified in the #chip line, then the internal oscillator will be used.
    2. If the clock speed is over 4 Mhz, the external HS oscillator is selected
    3. If the clock speed is 4 MHz or less, then the external XT oscillator mode is selected.

Note that these settings can easily be individually overridden whenever needed. For example, if the Watchdog Timer is needed, adding the line

```
#config WDT = ON
```

This will enable the watchdog timer, without affecting any other configuration settings.

Using Configuration

Once the necessary CONFIG options have been determined, adding them to the program is easy. On a new line type "#config" and then list the desired options separated by commas, such as in this line:

```
#config OSC = RC, BODEN = OFF
```

GCBASIC also supports this format on 10/12/16 series chips:

```
#config INTOSC_OSC_NOCLKOUT, BODEN_OFF
```

However, for upwards compatibility with 18F chips, you should use the = style config settings.

It is possible to have several #config lines in a program - for instance, one in the main program, and one in each of several #include files. However, care must then be taken to ensure that the settings in one file do not conflict with those in another.

For more help, see #config Directive

Arrays

About Arrays

An array is a special type of variable - one which can store several values at once. It is essentially a list of numbers in which each one can be addressed individually through the use of an "index". The index is a value in brackets immediately after the name of the array.

Examples of array names are:

Array/Index	Meaning
Fish(10)	Element 10 of an array called Fish
DataLog(2)	The second number in an array named DataLog
ButtonList(Temp)	An element in the array "ButtonList" that is selected according to the value in the variable "Temp"

Using Arrays

To use an array, its name is specified, then the index. Arrays can be used everywhere that a normal variable can be used.

Setting an entire array at once

It is possible to set several elements of an array with a single line of code. This short example shows how:

```
Dim TestVar(10)
TestVar = 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Element 0 of TestVar will be set to the number of items in the list, which in this case is 9. Each element of the array will then be loaded with the corresponding value in the list - so in the example, TestVar(1) will be set to 1, TestVar(2) to 2, and so on.

If there are many items in the array, it may be better to use a Lookup Table to store the items, and then copy some of the data items into a smaller array as needed.

For more help, see Declaring arrays with DIM

Comments

Usage:

Adding comments to your GCB program is done with an apostrophe before the comment line. You can also comment out sections of code if you want just by placing an apostrophe, a semi-colon or use the statement REM at the beginning of each line. The SynGCB IDE has a feature to do this automatically.

Example:

```
      ' The number of pins to flash
      #define FlashPins 2


REM You can create a header using an apostrophe before each line
      REM This is a great way to describe your program
      REM You can also use it to describe the hardware connections.


' You can place comments above the command or on the same line
      Dir PORTB Out   ' Initialise PORTB to all Outputs



; The Main loop
      do
          PORTB = 0         ' All Pins off
          Wait 1 S         ' Delay 1 second
          PORTB = 0xFF     ' All pins on
          Wait 1 s         ' Delay 1 second
      Loop
```

Conditions

About Conditions

In GCBASIC (and most other programming languages) a condition is a statement that can be either true or false. Conditions are used when the program must make a decision.

A condition is generally given as a value or variable, a relative operator (such as = or >), and another value or variable. Several conditions can be combined to form one condition through the use of logical operators such as AND and OR.

GCBASIC supports these relative operators:

Symbol	Meaning
=	Equal
<>	Not Equal
<	Less Than
>	Greater Than
<=	Less than or equal to
>=	Equal to or greater than

In addition, these logical operators can be used to combine several conditions into one:

Name	Abbreviation	Condition true if
AND	&	both conditions are true
OR	|	at least one condition is true
XOR	#	one condition is true
NOT	!	the condition is not true

NOT is slightly different to the other logical operators, in that it only needs one other condition. Other arithmetic operators may be combined in conditions, to change values before they are compared, for example.

GCBASIC has two built in conditions - TRUE, which is always true, and FALSE, which is always false. These can be used to create infinite loops.

It is also possible to test individual bits in conditions. To do this, specify the bit to test, then 1 or 0 (or ON and OFF respectively). Presently there is no way to combine bit tests with other conditions - NOT, AND, OR and XOR will not work.

Example conditions:

Condition	Comments
Temp = 0	Condition is true if Temp = 0
Sensor <> 0	Condition is true if Sensor is not 0
Reading1 >	

Reading2	True if Reading1 is more than Reading2
Mode = 1 AND Time > 10	True if Mode is 1 and Time is more than 10
Heat > 5 OR Smoke > 2	True if Heat is more than 5 or Smoke is more than 2
Light >= 10 AND (NOT Time > 7)	True if Light is 10 or more, and Time is 7 or less
Temp.0 ON	True if Temp bit 0 is on

Constants

About Defines

A define is a type of directive that tells the compiler to find a given word, and replace it with another word or number. Defines are useful for situations where a routine needs to be easily altered. For example, a define could be used to specify the amount of time to run an alarm for once triggered.

It is also possible to use defines to specify ports - thus defines can be used to aid in the creation of code that can easily be adapted to run on a different PIC with different ports.

GCBASIC makes considerable use of defines internally. For instance, the LCD code uses defines to set the ports that it must use to communicate with the LCD.

Using Defines

To create a define is a matter of using the #define directive. Here are some examples of defines:

```
#define Line 34
      #define Light PORTB.0
      #define LightOn Set PORTB.0 on
```

Line is a simple constant - GCBASIC will find "Line" in the program, and replace it with the number 34. This could be used in a line following program, to make it easier to calibrate the program for different lighting conditions.

Light is a port - it represents a particular pin on the PIC chip. This would be of use if the program had many lines of code that controlled the light, and there was a possibility that the port the light was attached to would need to change in the future.

LightOn is a define used to make the program more readable. Rather than typing "Set PORTB.0 on" over and over, it would then be made possible to type "LightOn", and have the compiler do the hard work.

Functions

About Functions

Functions are a special type of subroutine that can return a value. This means that when the name of the function is used in the place of a variable, GCBASIC will call the function, get a value from it, and then put the value into the line of code in the place of the variable.

Functions may also have parameters - these are treated in exactly the same way as parameters for subroutines. The only exception is that brackets are required around any parameters when calling a function.

Using Functions

This program uses a function called AverageAD to take two analog readings, and then make a decision based on the average:

```
        'Select chip
        #chip 16F88, 20

'Define ports
        #define LED PORTB.0
        #define Sensor AN0

'Set port directions
        dir LED out
        dir PORTA.0 in

'Main code
        Do
            Set PORTB.0 Off
            If AverageAD > 128 Then Set PORTB.0 On
            wait 10 ms
        Loop

Function AverageAD
            'Get 2 readings, divide by 2, store in AverageAD
            'Note the cast, the result of ReadAD needs to be converted to
            'a word before adding, or the result may overflow.
            AverageAD = ([word]ReadAD(Sensor) + ReadAD(Sensor)) / 2

        end function
```

See Also Subroutines, Exit

Labels

About Labels

Labels are used as markers throughout the program. Labels are used to mark a position in the program to 'jump to' from another position using a goto, gosub or other command.

Labels can be any word (that is not already a reserved keyword) and may contain digits and the underscore character. Labels must start with a letter or underscore (not digit), and are followed directly by a colon (:) at the marker position. The colon is not required within the actual commands.

The compiler is not case sensitive. Lower and/or upper case may be used at any time.

Example:

```
      'This program will flash the light until the button is pressed
      'off. Notice the label named SWITCH_OFF.

#chip 16F628A, 4

#define BUTTON PORTB.0
      #define LIGHT PORTB.1
      Dir BUTTON In
      Dir BUTTON Out

Do
          PulseOut LIGHT, 500 ms
          If BUTTON = 1 Then Goto SWITCH_OFF
          Wait 500 ms
          If BUTTON = 1 Then Goto SWITCH_OFF
      Loop

SWITCH_OFF:
      Set LIGHT Off
      'Chip will enter low power mode when program ends
```

For more help, see Goto, Gosub

Lookup Tables

About Lookup Tables

A lookup table is a list of values that are stored in the program memory of the chip, which can be accessed using the ReadTable command.

The advantage of lookup tables is that they are memory efficient, compared to an equivalent set of IF statements.

Using Lookup Tables

First, the table must be created. The code to create a lookup table is simple - a line that has "Table" and then the name of the table, a list of numbers (up to 254), and then "End Table".

Once the table is created, the ReadTable command is used to read data from it. The ReadTable command requires the name of the table it is to read, the location of the item to retrieve, and a variable to store the retrieved number in.

Lookup tables can store byte or word values. GCBASIC will automatically detect the type of the table depending on the values in it.

Item 0 of a lookup table stores the size of the table. If the ReadTable command attempts to read beyond the end of the table, the value 0 will be returned.

Advanced use of Lookup Tables

You can use the Table statement to store the data table in EEPROM. If the compiler is told to store a data table in "Data" memory, it will store it in the EEPROM.

NOTE: The limitation of of using EPPROM tables is that you can only store BYTEs. You cannot store WORD values in the EEPROM tables.

Here is some example code:

```
          #chip 16F628

'Read table item
          'Must use ReadTable and a variable for the index, or the table won't be
          downloaded to EEPROM

TableLoc = 2
          ReadTable TestDataSource, TableLoc, SomeVar

'Write to table , this is not required
          EPWrite 1, 45

'Table of values to write to EEPROM
          'EEPROM location 0 will store length of table
          'Subsequent locations will each store a value
```

```
Table TestDataSource Store Data
              12
              24
              36
              48
              60
              72
          End Table
```

For more help, see ReadTable

Miscellaneous

Combining multiple instructions

It is possible to combine multiple instructions on a single line, by separating them with a colon. For example, this code:

```
Set PORTB.0 On
      Set PORTB.1 On
      Wait 1 sec
      Set PORTB.0 Off
      Set PORTB.0 Off
```

could also be written as:

```
Set PORTB.0 On: Set PORTB.1 On
      Wait 1 sec
      Set PORTB.0 Off: Set PORTB.0 Off
```

In most cases, it will make no difference if commands share a line or not. However, special care should be taken with If commands, as this code:

```
Set PORTB.0 Off
      Set PORTB.1 Off
      If Temp > 10 Then Set PORTB.0 On: Set PORTB.1 On
      Wait 1 s
```

Will be equivalent to this:

```
Set PORTB.0 Off
      Set PORTB.1 Off
      If Temp > 10 Then
          Set PORTB.0 On
          Set PORTB.1 On
      End If
      Wait 1 s
```

Also, the commands used to start and end subroutines, data tables and functions must be alone on a line. For example, this is WRONG:

```
Sub Something: Set PORTB.0 Off: End Sub
```

ReadTable

Syntax:

ReadTable *TableName, Item, Output*

Command Availability:

Available on all microcontrollers.

Explanation:

The ReadTable command is used to read information from lookup tables. *TableName* is the name of the table that is to be read, *Item* is the line of the table to read, and *Output* is the variable to write the retrieved value in to.

Item is 1 for the first line of the table, 2 for the second, and so on. Item 0 gives the size of the table. Care must be taken to ensure that the program is not told to read beyond the end of the table, or strange effects will be observed.

The type of *Output* should match the type of data stored in the table. For example, if the table contains Word values then *Output* should be a Word variable. If the type does not match, GCBASIC will attempt to convert the value.

Example:

```
      'Chip Settings
      #chip 16F88, 20

'Hardware Settings
      #define LED PORTB.0
      Dir LED Out

'Main Routine
      ReadTable TimesTwelve, 4, Temp
      Set LED Off
      If Temp = 48 Then Set LED On

'Lookup table named "TimesTwelve"
      Table TimesTwelve
      12
      24
      36
      48
      60
      72
      84
      96
      108
      120
      132
      144
      End Table
```

For more help, see Lookup Tables

Scripts

About Scripts

A script is a small section of code that Great Cow BASIC runs when it starts to compile a program. Their main use is to perform calculations that are required to adjust the program for different speed chips.

Scripts are not compiled and downloaded to the microcontroller - GCBASIC reads them, runs them, removes them from the program and then allows any results they have calculated to be used as constants in the program.

Inside a script, constants are treated like variables. Scripts can read the values of constants, and set them to contain new values.

Using Lookup Tables

Scripts start with "#script" and end with "#endscript". Inside, they can consist of 3 commands:

- If
- Assignment (=)
- Error

If is similar to the If command in normal GCBASIC code, except that it doesn't have an Else clause. It is used to compare the values of constants.

= is identical to that in GCBASIC programs. The constant that is to be set goes on the left side of the =, and the new value goes on the right.

Error is used to display an error message. Anything after the Error command is displayed at the end of compilation, and is saved in the error log for the program.

Example Script

This script is used in the pwm.h file. It takes the values of the constants PWM_Freq, PWM_Duty and ChipMHz, and uses the equations shown in the PIC datasheets to calculate the correct values for the relevant system variables.

```
#script
    PR2Temp = int((1/PWM_Freq)/(4*(1/(ChipMHz*1000))))
    T2PR = 1
    If PR2Temp > 255 Then
        PR2Temp = Int((1 / PWM_Freq) / (16 * (1 / (ChipMHz * 1000))))
        T2PR = 4
        If PR2Temp > 255 Then
            PR2Temp = Int(( 1 / PWM_Freq) / (64 * (1 / (ChipMHz * 1000))))
            T2PR = 16
            If PR2Temp > 255 Then
                Error Invalid PWM Frequency value
            End If
        End If
     End If
```

```
DutyCycle = (PWM_Duty * 10.24) * PR2Temp / 1024
     DutyCycleH = (DutyCycle AND 1020) / 4
     DutyCycleL = DutyCycle AND 3
 #endscript
```

After this script has run, the values PR2Temp, DutyCycleH and DutyCycleL are used as constants to set up the required variables.

Subroutines

About Subroutines

A subroutine is a small program inside of the main program. Subroutines are typically used when a task needs to be repeated several times in different parts of the main program.

There are two main uses for subroutines:

- Keeping programs neat and easy to read
- Reducing the size of programs by allowing common sections of code to be reused.

When the PIC comes to a subroutine it saves its location in the current program before jumping to the start of, or calling, the subroutine. Once it reaches the end of the subroutine it returns to the main program, and continues to run the code where it left off previously.

Normally, it is possible for subroutines to call other subroutines. There are limits to the number of times that a subroutine can call another sub, which vary from chip to chip:

PIC Family	Instruction Width	Number of subs called
10F*, 12C5*, 12F5*, 16C5*, 16F5*	12	1
12C*, 12F*, 16C*, 16F*, except those above	14	7
18F*, 18C*	16	31

These limits are due to the amount of memory on the PIC which saves its location before it jumps to a new subroutine. Some GCBASIC commands are subroutines, so you should always allow for 2 or 3 subroutine calls more than your program has.

AVR microcontrollers have no fixed limit on how many subroutines can be called at a time, but if too many are called then some variables on the chip may be corrupted. To check if there are too many subroutines, work out the most that will be called at once, then multiply that number by 2 and create an array of that size. If an out of memory error message comes up, there are too many calls.

Another feature of subroutines is that they are able to accept parameters. These are values that are passed from the main program to the subroutine when it is called, and then passed back when the subroutine ends.

Using Subroutines

To call a subroutine is very simple - all that is needed is the name of the sub, and then a list of parameters. This code will call a subroutine named "Buzz" that has no parameters:

```
Buzz
```

If the sub has parameters, then they should be listed after the name of the subroutine. This would be the command to call a subroutine named "MoveArm" that has three parameters:

```
MoveArm NewX, NewY, 10
```

If a subroutine has parameters, you may choose to put brackets around them, like so:

```
MoveArm (NewX, NewY, 10)
```

All that this does is change the appearance of the code - it doesn't make any difference to what the code does. Decide which one meets your own personal preference, and then stick with it.

Creating subroutines

To create a subroutine is almost as simple as using one. There must be a line at the start which has "sub ", and then the name of the subroutine. Also, there needs to be a line at the end of the subroutine which reads "end sub". To create a subroutine called "Buzz", this is the required code:

```
sub Buzz

'code for the subroutine goes here

end sub
```

If the subroutine has parameters, then they need to be listed after the name. For example, to define the "MoveArm" sub used above, use this code:

```
sub MoveArm(ArmX, ArmY, ArmZ)

'code for the subroutine goes here

end sub
```

In the above sub, ArmX, ArmY and ArmZ are all variables. If the call from above is used, the variables will have these values at the start of the subroutine:

```
ArmX = NewX
      ArmY = NewY
      ArmZ = 10
```

When the subroutine has finished running, GCBASIC will copy the values back where possible. NewX will be set to ArmX, and NewY to ArmY. GCBASIC will not attempt to set the number 10 to ArmZ.

Controlling the direction data moves in

It is possible to instruct GCBASIC not to copy the value back after the subroutine is called. If a subroutine is defined like this:

```
sub MoveArm(In ArmX, In ArmY, In ArmZ)
        'code for the subroutine goes here

end sub
```

Then GCBASIC will copy the values to the subroutine, but will not copy them back.

GCBASIC can also be prevented from copying the values back, by adding "Out" before the parameter name. This is used in the EEPROM reading routines - there is no point copying a data value into the read subroutine, so Out has been used to avoid wasting time and memory. The EPRead routine is defined as "Sub EPRead(In Address, Out Data).

Many older sections of code use "#NR" at the end of the line where the parameters are specified. The "#NR" means "No Return", and when used has the same effect as adding "In" before every parameter. Use of "#NR" is not recommended, as it does not give the same level of control.

Using different data types for parameters

It is possible to use any type of variable a as parameter for a subroutine. Just add "As " and then the data type to the end of the parameter name. For example, to make all of the parameters for the MoveArm subroutine word variables, use this code:

```
sub MoveArm(ArmX As Word, ArmY As Word, ArmZ As Word)
        ...
    end sub
```

Optional parameters

Sometimes, the same value may be used over and over again for a parameter, except in a particular case. If this occurs, a default value may be specified for the parameter, and then a value for that parameter only needs to be given in a call if it is different to the default.

For example, suppose a subroutine to create an error beep is required. Normally it emits a 440 Hz tone, but sometimes a different tone is required. To create the sub, this code would be used:

```
Sub ErrorBeep(Optional OutTone As Word = 440)
      Tone OutTone, 100
    End Sub
```

Note the "Optional" before the parameter, and the " = 440" after it. This tells GCBASIC that

if no parameter is supplied, then set the OutTone parameter to 440.

If called using this line:

```
ErrorBeep
```

then a 440 Hz beep will be emitted. If called using this line:

```
ErrorBeep 1000
```

then the sub will produce a 1000 Hz tone.

When using several parameters, it is possible to make any number of them optional. If the optional parameter/s are at the end of the call, then no value needs to be specified. If they are at the start or in the middle, then you must insert commas to allow GCBASIC to tell where the optional parameters are.

Overloading

It is possible to have 2 subroutines with the same name, but different parameters. This is known as overloading, and GCBASIC will automatically select the most appropriate subroutine for each call.

An example of this is the Print routine in the LCD routines. There are actually several Print subroutines; for example, one has a byte parameter, one a word parameter, and one a string parameter. If this command is used:

```
Print 100
```

Then the Print (byte) subroutine will be called. However, if this command is used:

```
Print 30112
```

Then the Print (word) subroutine will be called. If there is no exact match for a particular call, GCBASIC will use the option that requires the least conversion of variable types. For example, if this command is used:

```
Print PORTB.0
```

The byte print will be used. This is because byte is the closest type to the single bit parameter.

See Also Functions, Exit

Variables

About Variables

A variable is an area of memory on the microcontroller that can be used to store a number or a series of letters. This is useful for many purposes, such as taking a sensor reading and acting on it, or counting the number of times the robot has performed a particular task.

Each variable must be given a name, such as "MyVariable" or "PieCounter". Choosing a name for a variable is easy - just don't include spaces or any symbols (other than _), and make sure that the name is at least 2 characters (letters and/or numbers) long.

Variable Types

There are several different types of variable, and each type can store a different sort of information. These are the variable types that Great Cow BASIC can currently use:

Variable type	Information that this variable can store	Example uses for this type of variable
Bit	A bit (0 or 1)	Flags to track whether or not a piece of code has run
Byte	A whole number between 0 and 255	General purpose storage of data, such as counters
Word	A whole number between 0 and 65535	Storage of extra large numbers
Integer	A whole number between - 32768 and 32767	Anything where a negative number will occur
Long	A whole number between 0 and 2^32 (4.29 billion)	Storing very, very big numbers
Array	A list of whole numbers ranging from 0 to 255	Logs of sensor readings
String	A series of letters, numbers and symbols.	Messages that are to be shown on a screen

Using Variables

Byte variables do not need any special commands to set them up - just put the name of the variable in to the command where the variable is needed.

Other types of variable can be used in a very similar way, except that they must be "dimensioned" first. This involves using the DIM command, to tell Great Cow BASIC that it is dealing with something other than a byte variable.

A key feature of variables is that it is possible the have the microcontroller check a variable, and only run a section of code if it is a given value. This can be done with the IF command.

String Variables

Strings are defined as follows:

```
'Create buffer variables to store received messages

Dim Buffer As String
```

String variables default to 10 bytes for chips with less than 16 bytes of RAM, 20 bytes for chips with 16 to 367 bytes , and 40 bytes for devices with more RAM than 367.

Defining a length for the string is the best way to limit memory usage. It is good practice if you need a string of a certain size to set the length of a strings, since the default length for a string variable changes depending on the amount of memory in the microcontroller (see above).

To set the length see the example below:

```
'Create buffer variables to store received messages as 16 bytes long

      Dim OutBuffer As String * 16
```

Variable Aliases

Some variables are aliases, which are used to refer to memory locations used by other variables. These are useful for joining predefined byte variable together to form word variables.

Alias are not like pointers in many languages - they must always refer to the same variable or variables and cannot be changed.

Casting

Casting changes the type of a variable or value. Placing the type that the value should be converted to in square brackets will tell the compiler to convert it. For example, this will cause two byte variables to be treated as word variables by the addition code:

```
Dim MyWord As Word
      MyWord = [word]ByteVar + AnotherByteVar
```

Why do this? If there are no casts, then GCBASIC will add the two values using the byte addition code, and then convert the result to a word to store in MyWord. Suppose that ByteVar is 150, and AnotherByteVar is 231. When added, this will come to 381 - which will overflow, leaving 125 in the result. However, when the cast is added, GCBASIC will treat

ByteVar as if it were a word, and so will use the word addition code. This will cause the correct result to be calculated.

Often, a cast will be used when calculating an average:

```
MyAverage = ([word]Value1 + Value2) / 2
```

It's also possible to cast the second value:

```
MyAverage = (Value1 + [word]Value2) / 2
```

The result will be exactly the same.

For more help, see: Declaring variables with DIM, Setting Variables

Doing things to individual bits of variables see, SET, ROTATE

Checking variables and doing different things based on their value, see IF, DO, FOR, Conditions

ADOff

This command is obsolete. There should be no need to call it. GCBASIC will automatically disable the A/D converter and set all pins to digital mode when starting the program, and after every use of the ReadAD function.
It is recommended that this command be removed from all programs.

ReadAD

Syntax:

var = ReadAD (*port*)

Command Availability:

Available on all PIC and AVR chips with an analog to digital converter module built in.

Explanation:

Return an 8 bit number [0-255]

ReadAD is a function that can be used to read the built-in analog to digital converter that many microcontroller chips include. *port* should be AN0, AN1, AN2, etc., up to the number of analog inputs available on the chip that is in use. Those familiar with AVR microcontrollers can also refer to the ports as ADC0, ADC1, etc. Refer to the datasheet for the microcontroller chip to find the number of ports available. (Note: it's perfectly acceptable to use ANx on AVR, or ADCx on PIC.)

Another function, ReadAD10, is almost identical to ReadAD. The only difference is that it returns a full 10 bit value in a word variable.

Example:

```
     #chip 16F819, 8
     #config osc = int

'Set the input pin direction
     Dir PORTA.0 In

'Loop to take readings until the EEPROM is full
     For CurrentAddress = 0 to 255

         'Take a reading and log it
         EPWrite CurrentAddress, ReadAD(AN0)

         'Wait 10 minutes before getting another reading
         Wait 10 min
     Next
```

See Also ReadAD10

ReadAD10

Syntax:

var = ReadAD10 (*port*)

Command Availability:

Available on all PIC and AVR chips with an analog to digital converter module built in.

Explanation:

Return an 10 bit number [0-1023] in a word variable

ReadAD is a function that can be used to read the built-in analog to digital converter that many microcontroller chips include. *port* should be AN0, AN1, AN2, etc., up to the number of analog inputs available on the chip that is in use. Those familiar with AVR microcontrollers can also refer to the ports as ADC0, ADC1, etc. Refer to the datasheet for the microcontroller chip to find the number of ports available. (Note: it's perfectly acceptable to use ANx on AVR, or ADCx on PIC.)

Another function, ReadAD is almost identical to ReadAD10. The only difference is that it returns a byte variable.

Example:

```
     #chip 16F819, 8
     #config osc = int

'Set the input pin direction
     Dir PORTA.0 In

'Loop to take readings until the EEPROM is full
     For CurrentAddress = 0 to 255

         'Take a reading and show it
         Print str(ReadAD10(AN0))

         'Wait 10 minutes before getting another reading
         Wait 10 min
     Next
```

See Also ReadAD

EPRead

Syntax:

EPRead *location, store*

Command Availability:

Available on all PIC and AVR chips with EEPROM data memory.

Explanation:

EPRead is used to read information from the EEPROM data storage that many microcontroller chips are equipped with. *location* represents the location to read data from, and varies from one chip to another. *store* is the variable in which to store the data after it has been read from EEPROM.

Example:

```
      'Program to turn a light on and off
      'Will remember the last status

#chip tiny2313, 1
      #define Button PORTB.0
      #define Light PORTB.1

Dir Button In
      Dir Light Out

'Load saved status
      EPRead 0, LightStatus

If LightStatus = 0 Then
          Set Light Off
      Else
          Set Light On
      End If

Do
          'Wait for the button to be pressed
          Wait While Button = On
          Wait While Button = Off

          'Toggle value, record
          LightStatus = !LightStatus
          EPWrite 0, LightStatus

    'Update light
          If LightStatus = 0 Then
              Set Light Off
          Else
              Set Light On
          End If
      Loop
```

For more help, see EPWrite

EPWrite

Syntax:

EPWrite *location, data*

Command Availability:

Available on all PIC and AVR chips with EEPROM data memory.

Explanation:

EPWrite is used to write information to the EEPROM data storage, so that it can be accessed later by a programmer on the PC, or by the EPRead command. *location* represents the location to read data from, and varies from one chip to another. *data* is the data that is to be written to the EEPROM, and can be a value or a variable.

Example:

```
     #chip 16F819, 8
     #config osc = int


'Set the input pin direction
     Dir PORTA.0 In


'Loop to take readings until the EEPROM is full
     For CurrentAddress = 0 to 255

         'Take a reading and log it
         EPWrite CurrentAddress, ReadAD(AN0)

         'Wait 10 minutes before getting another reading
         Wait 10 min

     Next
```

For more help, see EPRead

ProgramErase

Syntax:

ProgramErase(*location*)

Command Availability:

Available on all PIC chips with self write capability. Not available on AVR at present.

Explanation:

ProgramErase erases information from the program memory on chips that support this feature. The largest value possible for *location* depends on the amount of program memory on the PIC, which is given on the datasheet.

This command must be called before writing to a block of memory. It is slow in comparison to other GCBASIC commands. Note that is erases memory in 32-byte blocks - see the relevant PIC datasheet for more information.

This is an advanced command which should only be used by advanced developers. Care must be taken with this command, as it can easily erase the program that is running on the PIC.

For more help, see ProgramRead, ProgramWrite

ProgramRead

Syntax:

ProgramRead (*location, store*)

Command Availability:

Available on all PIC chips with self write capability. Not available on AVR at present.

Explanation:

ProgramRead reads information from the program memory on chips that support this feature. *location* and *store* are both word variables, meaning that they can store values over 255.

The largest value possible for *location* depends on the amount of program memory on the PIC, which is given on the datasheet. *store* is 14 bits wide, and thus can store values up to 16383.

This is an advanced command which should only be used by advanced developers.

Example:

For more help, see ProgramErase, ProgramWrite

ProgramWrite

Syntax:

ProgramWrite (*location, value*)

Command Availability:

Available on all PIC chips with self write capability. Not available on AVR at present.

Explanation:

ProgramWrite writes information to the program memory on chips that support this feature. *location* and value are both word variables.

The largest value possible for *location* depends on the amount of program memory on the PIC, which is given on the datasheet. *value* is 14 bits wide, and thus can store values up to 16383.

This is an advanced command which should only be used by advanced developers. ProgramErase must be used to clear a block of memory BEFORE ProgramWrite is called.

Example:

For more help, see ProgramErase, ProgramRead

Do

Syntax:

Do [{While | Until} *condition*]
...
program code
....
<condition> Exit Do
...
Loop [{While | Until} *condition*]

Command Availability:

Available on all microcontrollers.

Explanation:

The Do command will cause the code between the Do and the Loop to run repeatedly while *condition* is true or until *condition* is true, depending on whether While or Until has been specified.

Note that the While or Until and the condition can only be specified once, or not at all. If they are not specified, then the code will repeat endlessly.

Optionally, you can specify a condition to exit the Do-Loop immediately.

Example #1:

```
      'This code will flash a light until the button is pressed
      #chip 12F629, 4
      #config osc = int

#define BUTTON GPIO.3
      #define LIGHT GPIO.5

Dir BUTTON In
      Dir LIGHT Out

Do Until BUTTON = 1
          PulseOut LIGHT, 1 s
          Wait 1 s
      Loop
```

Example #2:

```
This code will also flash a light until the button is pressed. This exa,ple uses EXIT DO
within a continuous loop.

      #chip 12F629, 4
      #config osc = int
```

```
#define BUTTON GPIO.3
      #define LIGHT GPIO.5

Dir BUTTON In
      Dir LIGHT Out

Do
          PulseOut LIGHT, 1 s
          Wait 1 s
          if BUTTON = 1 then EXIT DO
      Loop
```

For more help, see Conditions

End

Syntax:

End

Command Availability:

Available on all microcontrollers.

Explanation:

When the End command is used, the program will immediately stop running. There are very few cases where this command is needed - generally, the program should be an endless loop.

Example:

```
'This program will turn on the red light, but not the green light
      Set RED On
      End
      Set GREEN On
```

Exit [option]

Syntax options:

Exit Sub | Exit Function | Exit Do | Exit For

Command Availability:

Available on all microcontrollers.

Explanation:

This command will make the program exit the routine it is currently in, as it would if it came to the end of the routine.

Applies to Subroutines, Functions, For-Next loops and Do-Loop loops.

Example:

```
      #chip tiny13, 1

#define SENSOR PORTB.0
      #define BUZZER PORTB.1
      #define LIGHT PORTB.2
      Dir SENSOR In
      Dir BUZZER Out
      Dir LIGHT Out

Do
          Burglar
      Loop

'Burglar Alarm subroutine
      Sub Burglar
          If SENSOR = 0 Then
              Set BUZZER Off
              Set LIGHT Off
              Exit Sub
          End If
          Set BUZZER On
          Set LIGHT On
      End Sub
```

For more help, see Do, For, Sub, Function

For

Syntax:

For *counter* = *start* To *end* [Step *increment*]
...
program code
<condition> Exit For
...
Next

Command Availability:

Available on all microcontrollers.

Explanation:

The For command is ideal for situations where a piece of code needs to be run a set number of times, and where it is necessary to keep track of how many times the code has run. When the For command is first executed, *counter* is set to *start*. Then, each successive time the program loops, *increment* is added to *counter*, until *counter* is equal to *end*. Then, the program continues beyond the Next.

Step and *increment* are optional. If Step is not specified, GCBASIC will increment *counter* by 1 each time the code is run.

The Exit For is optional and can used to exit the loop upon a specific condition.

Example #1:

```
     'This code will flash a green light 6 times.

#chip 16F88, 8
     #config Osc = Int

#define LED PORTB.0
     Dir LED Out

For LoopCounter = 1 to 6
         PulseOut Led, 1 s
         Wait 1 s
     Next
```

Example #2:

```
     'This code will flash alternate LEDS until the switch is pressed.

#chip 16F88, 8
     #config Osc = Int

#define LED1 PORTB.0
     Dir LED1 Out
     #define LED2 PORTB.2
     Dir LED2 Out
```

```
#define SWITCH1 PORTA.0
     Dir SWITCH1 In

     main:
         PulseOut LED1, 1 s
         For LoopCounterOut = 1 to 250
                 PulseOut LED2, 4 Ms
                 if switch = On then Exit For
         Next
         Set LED2 OFF
     goto main
```

See Also Repeat

Gosub

Syntax:

Gosub *label*

Command Availability:

Available on all microcontrollers.

Explanation:

The Gosub command is used to jump to a label as a subroutine, in a similar way to Goto. The difference is that Return can then be used to return to the line of code after the Goto.

Note: Gosub should not be used if it can be avoided. It is not required to call a subroutine that has been defined using Sub, just write the name of the subroutine.

Example:

```
      'This program will flash an LED on portb bit 0 and play a beep on
      'porta bit 4. until the robot is turned off.

#chip 16F628A, 4 'Change this to suit your circuit

#define SOUNDOUT PORTA.4
      #define LIGHT PORTB.0
      Dir LIGHT Out


Do
          'Flash Light
          PulseOut LIGHT, 1 s
          Wait 1 s
          'Beep
          Gosub PlayBeep
      Loop

PlayBeep:
      Tone 200, 10
      Tone 100, 10
      Return
```

For more help, see Goto, Labels

Goto

Syntax:

Goto *label*

Command Availability:

Available on all microcontrollers.

Explanation:

The Goto command will make the robot jump to the line specified, and continue running the program from there. The Goto command is mainly useful for exiting out of loops - if you need to create an infinite loop, use the Do command instead.

Be careful how you use Goto. If used too much, it can make programs very hard to read.

To define a label, put the name of the label alone on a line, with just a colon (:) after it.

Example:

```
'This program will flash the light until the button is pressed
'off. Notice the label named SWITCH_OFF.

#chip 16F628A, 4 'Change this line to suit your circuit

#define BUTTON PORTB.0
#define LIGHT PORTB.1
Dir BUTTON In
Dir BUTTON Out

Do
    PulseOut LIGHT, 500 ms
    If BUTTON = 1 Then Goto SWITCH_OFF
    Wait 500 ms
    If BUTTON = 1 Then Goto SWITCH_OFF
Loop

SWITCH_OFF:
Set LIGHT Off
'Chip will enter low power mode when program ends
```

For more help, see Gosub, Labels

If

Syntax:

Short form:

If *condition* Then *command*

Long form:

If *condition* Then
...
program code
...
End If

Using Else:

If *condition* Then
code to run if true
Else
code to run if false
End If

Command Availability:

Available on all microcontrollers.

Explanation:

The If command is the most common command used to make decisions. If *condition* is true, then *command* (short) or *program code* (long) will be run. If it is false, then the robot will skip to the code located on the next line (short) or after the End If (long form).

If Else is used, then the condition between If and Else will run if the condition is true, and the code between Else and End If will run if the condition is false.

Note:

ELSE must be on a separate line in the source code.

Supported

```
<instruction> ' is supported
ELSE
<instruction>
```

Not Supported

```
<instruction> ELSE ' is not supported but will compile
```

```
            <instruction>
```

Example:

```
      'Turn a light on or off depending on a light sensor

#chip 12F683, 8
      #config osc = int

#define LIGHT GPIO.1
      #define SENSOR AN3
      #define SENSOR_PORT GPIO.4

Dir LIGHT Out
      Dir SENSOR_PORT In

Do
          If ReadAD(SENSOR) > 128 Then
              Set LIGHT Off
          Else
              Set LIGHT On
          End If
      Loop
```

For more help, see Conditions

IndCall

Syntax:

IndCall *Address*

Command Availability:

Available on all microcontrollers.

Explanation:

IndCall provides a basic implementation of function pointers. *Address* is the program memory location of the subroutine that is to be called. There are two ways to specify this - either by providing a direct reference to the subroutine using the @ operator, or by specifying a word variable that contains the address.

This command is useful for callbacks. For example, a particular subroutine might read bytes from a serial connection, but different actions may need to be taken at different times. A different subroutine could be created for each action, and then the subroutine for the appropriate action could be passed to the serial connection reading routine each time it is called.

Note: Calling subroutines that have parameters using IndCall is not supported. Errors may occur. If data needs to be passed, use a variable instead.

Example:

```
'Flash an LED using an indirect call
#chip 12F683

'Create a word variable, and set it to the memory location of the
'Blink subroutine.
Dim FlashingSub As Word
FlashingSub = @Blink

'Main loop
Do
    'Indirect call to subroutine at location FlashingSub
    IndCall FlashingSub
Loop

'LED flashing subroutine
Sub Blink
    PulseOut GPIO.0, 500 ms
    Wait 500 ms
End Sub
```

Pause

Syntax:

Fixed Length Delay: Pause *time* ms

Command Availability:

Available on all microcontrollers.

Explanation:

The Pause command will cause the program to wait for either a specified amount of time in milliseconds.

Repeat

Syntax:

Repeat *times*
...
program code
...
End Repeat

Command Availability:

Available on all microcontrollers.

Explanation:

The Repeat command is ideal for situations where a piece of code needs to be run a set number of times. It uses less memory and runs faster than the For command, and should be used wherever it is not necessary to count how many times the code has run.

Repeat has a maximum repeat value of 65535.

Example:

```
     'This code will flash a green light 6 times.

#chip 16F88, 20

#define LED PORTB.0
     dir LED out

Repeat 6
         PulseOut LED, 1 s
         Wait 1 s
     End Repeat
```

See Also FOR

Select

Syntax:

Select Case *var*
Case *value1*
code1

Case *value2*
code2

Case Else
code3

End Select

Command Availability:

Available on all microcontrollers.

Explanation:

The Select Case control structure is used to select and run a particular section of code, based on the value of *var*. If *var* equals *value1* then *code1* will be run. Once *code1* has run, the chip will jump to the End Select command and continue running the program. If none of the other conditions are true, then the code under the Case Else section will be run.

Case Else is optional, and the program will function correctly without it.

If there is only one line of code after the Case, the code may look neater if the line is placed after the Case. This is shown below in the example, for cases 3, 4 and 5.

It is important to note that **only one section of code will be run** when using Select Case.

There are two example shown below.

Example #1:

```
     'Program to read a value from a potentiometer, and display a
     'different word based on the result


#chip 18F4550, 20


'LCD connection settings
     #define LCD_IO 4
     #define LCD_DB4 PORTD.4
     #define LCD_DB5 PORTD.5
     #define LCD_DB6 PORTD.6
```

```
      #define LCD_DB7 PORTD.7
      #define LCD_RS PORTD.0
      #define LCD_RW PORTD.1
      #define LCD_Enable PORTD.2


DIR PORTA.0 IN
      Do
          Temp = ReadAD(AN0) / 20
          CLS
          Select Case Temp
              Case 0
                  Print "None!"

        Case 1
                  Print "One"

        Case 2
                  Print "Two"

        Case 3: Print "Three"
              Case 4: Print "Four"
              Case 5: Print "Five"
              Case Else
                  Print "A lot!"
         End Select
         Wait 250 ms
      Loop
```

Example #2:

This code demonstrates how to receive codes from a handheld remote control unit. This has been tested and supports a Sony TV remote and also a universal remote set to Sony TV mode.

The program gets both the device number and the key number, and also translates the key number to English. The received results are displayed on an LCD.

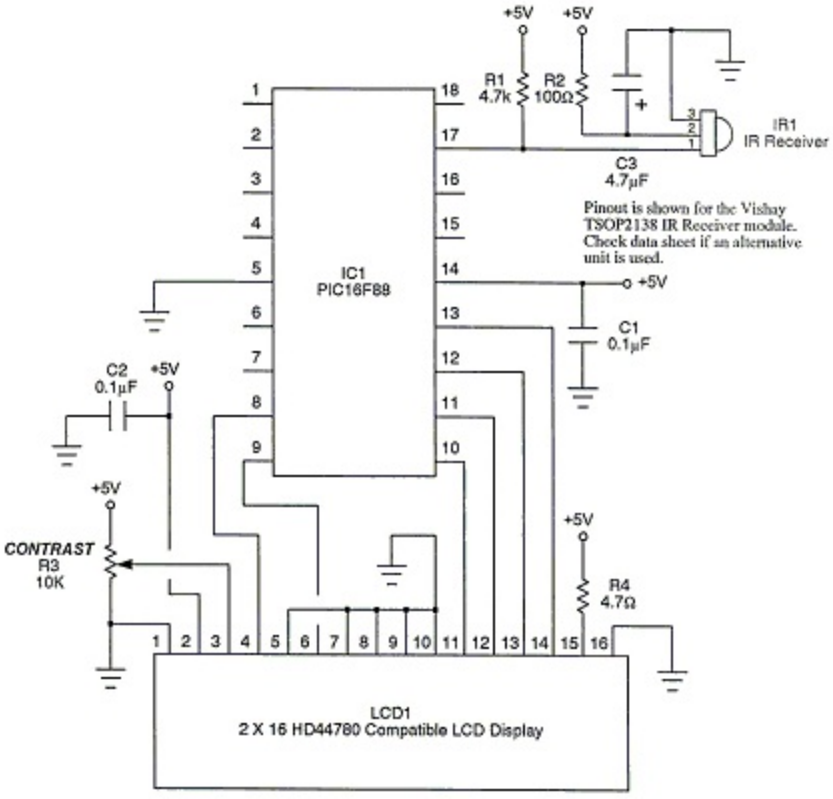
The circuit for the IR receiver and the chip is shown below.

```
       ;A program to receive IR codes sent by a Sony
       ;compatible handheld remote control.

#chip 16F88, 8                  ;PIC16F88 running at 8 MHz
       #config mclr=off                ;reset handled internally
       #config osc=int                 ;use internal clock

;----- Constants
```

```
#define LCD_IO      4             ;4-bit mode
       #define LCD_RS      PortB.2       ;pin 8 is Register Select
       #define LCD_Enable  PortB.3       ;pin 9 is Enable
       #define LCD_DB4     PortB.4       ;DB4 on pin 10
       #define LCD_DB5     PortB.5       ;DB5 on pin 11
       #define LCD_DB6     PortB.6       ;DB6 on pin 12
       #define LCD_DB7     PortB.7       ;DB7 on pin 13
       #define LCD_NO_RW   1             ;ground RW line on LCD
       #define IR          PortA.0       ;sensor on pin 17

;----- Variables

dim device, cmd, count, i as byte
       dim pulse(12)                     ;pulse count array
       dim button as string              ;ASCII for button label

;----- Program

dir PortA in                         ;A.0 is IR input
       dir PortB out                        ;B.2 - B.6 for LCD

cls                                  ;clear the LCD
       print "Dev:    Cmd:"                 ;logo for top line
       locate 1,0
       print "Button:"                      ;logo for second line

do
         getIR, cmd                         ;wait for IR signal
         printCmd                           ;show device and command
         printKey                           ;show key label
         wait 10 mS                         ;ignore any repeats
       loop                                 ;repeat forever

;----- Subroutines

sub getIR
         tarry1:
           count = 0                        ;wait for start bit
           do while IR = 0                  ;measure width (active low)
             wait 100 uS                    ;24 X 100 uS = 2.4 mS
             count += 1
           loop
         if count < 20 then goto tarry1  ;less than this so wait

  for i=1 to 12                      ;read/store the 12 pulses
           tarry2:
             count = 0
             do while IR = 0                ;zero = 6 units = 0.6 mS
               wait 100 uS                  ;one = 12 units = 1.2 mS
               count += 1
             loop
           if count < 4 then goto tarry2 ;too small to be legit
           pulse(i) = count               ;else store pulse width
         next

  cmd = 0                            ;command built up here
```

```
        for i = 1 to 7                  ;1st seven bits are the cmd
          cmd = cmd / 2                 ;shift into place
          if pulse(i) > 10 then         ;longer than 10 mS
             cmd = cmd + 64             ;so call it a one
          end if
        next

  device = 0                        ;device number built up here
        for i=8 to 12                   ;next 5 bits are device number
          device = device / 2
          if pulse(i) > 10 then
             device = device + 16
          end if
        next
      end sub

sub printCmd            ;print device number
        locate 0,5
        print "   "
        locate 0,5
        print device

  locate 0,13           ;print raw command number
        print "   "
        locate 0,13
        print cmd
      end sub

sub PrintKey            ;print translated button
        locate 1,9
        print "       "
        locate 1,9

  select case cmd       ;translate command code
          case 0
            button = "One"
          case 1
            button = "Two"
          case 2
            button = "Three"
          case 3
            button = "Four"
          case 4
            button = "Five"
          case 5
            button = "Six"
          case 6
            button = "Seven"
          case 7
            button = "Eight"
          case 8
            button = "Nine"
          case 9
            button = "Zero"
          case 10
            button = "#####"
          case 11
            button = "Enter"
          case 12
```

```
      button = "#####"
    case 13
      button = "#####"
    case 14
      button = "#####"
    case 15
      button = "#####"
    case 16
      button = "Chan+"
    case 17
      button = "Chan-"
    case 18
      button = "Vol+"
    case 19
      button = "Vol-"
    case 20
      button = "Mute"
    case 21
      button = "Power"
    case else
      button = "     "
  end select
  print button
end sub
```



Wait

Syntax:

Fixed Length Delay: Wait *time units*

Conditional Delay: Wait {While | Until} *condition*

Command Availability:

Available on all microcontrollers.

Explanation:

The Wait command will cause the program to wait for either a specified amount of time (such as 1 second), or while/until a condition is true.

When using the fixed-length delay, there is a variety of units that are available:

Unit	Length of unit	Delay range
us	1 microsecond	1 us - 65535 us
10us	10 microseconds	10 us - 2.55 ms
ms	1 millisecond	1 ms - 65535 ms
10ms	10 milliseconds	10 ms - 2.55 s
s	1 second	1 s - 255 s
m	1 minute	1 min - 255 min
h	1 hour	1 hour - 255 hours

At one stage, GCBASIC variables could not hold more than 255. The 10us and 10ms units were added as a way to work around this limit. There is now no such limit (Wait 1000 ms will work for example), so these are not really needed. However, you may see them in some older examples or programs, and the 10us units are sometimes the shortest delay that will work accurately.

Example:

```
'This code will wait until a button is pressed, then it will flash
'a light every half a second and produce a 440 Hz tone.

#chip 16F819, 8
#config osc = int

#define BUTTON PORTB.0
#define SPEAKER PORTB.1
#define LIGHT PORTB.2
Dir BUTTON In
Dir SPEAKER Out
Dir LIGHT Out
```

```
'Assumes Button switches on when pressed
     Wait Until BUTTON = 1
     Wait Until BUTTON = 0


Do
         'Flash the light
         Set LIGHT On
         Wait 500 ms
         Set LIGHT Off

         'Produce the tone
         '440 Hz = 880 changes = tone on for 1.14 ms
         Repeat 440
             PulseOut SPEAKER, 1140 us
             Wait 114 10us 'Wait for 114 x 10 us (1.14 ms)
         End Repeat
     Loop
```

For more help, see Conditions

Interrupts

Introduction

Interrupts are a feature of many microcontrollers. They allow the microcontroller to temporarily pause (interrupt) the code it is running and then start running another piece of code when some event occurs. Once it has dealt with the event, it will return to where it was and continue running the program.

Many events can trigger an interrupt, such as a timer reaching its limit, a serial message being received, or a special pin on the microcontroller receiving a signal.

Using Interrupts

There are two ways to use interrupts in GCBASIC. The first way is to use the On Interrupt command. This will automatically enable a given interrupt, and run a particular subroutine when the interrupt occurs.

The other way to deal with interrupts is to create a subroutine called Interrupt. GCBASIC will call this subroutine whenever an interrupt occurs, and then your code can check the "flag" bits to determine which interrupt has occured, and what should be done about it. If you use this approach, then you'll need to enable the desired interrupts manually. It is also essential that your code clears the flag bits, or else the interrupt routine will be called repeatedly.

Some combination of these two methods is also possible - the code generated by On Interrupt with check to see if the interrupt is one it recognises. If the interrupt is recognised, On Interrupt will deal with it - if not, the Interrupt subroutine will be called to deal with the interrupt.

The recommended way is to use On Interrupt, as it is both more efficient and easier to set up.

During some sections of code, it is desirable not to have any interrupts occur. If this is the case, then use the IntOff command to disable interrupts at the start of the section, and IntOn to re-enable them at the end. If any interrupt events occur while interrupts are disabled, then they will be processed as soon as interrupts are re-enabled. If the program does not use interrupts, IntOn and IntOff will be removed automatically by GCBASIC.

See Also IntOff, IntOn, On Interrupt

IntOff

Syntax:

IntOff

Command Availability:

Available on PIC and AVR microcontrollers with interrupt support. Will be automatically removed on chips without interrupts.

Explanation:

IntOff is used to disable interrupts on the microcontroller. It should be used at the start of code which is timing-sensitive, and which would not function correctly if paused and restarted.

It is essential that IntOn is used to turn interrupts on again after the timing-sensitive code has finished running. If not, no interrupts will be handled.

IntOff will be removed from the program if no interrupts are used. It is recommended that IntOff be placed before all code that is timing sensitive, in case interrupts are implemented later.

See also IntOn, Interrupts

IntOn

Syntax:

IntOn

Command Availability:

Available on PIC and AVR microcontrollers with interrupt support. Will be automatically removed on chips without interrupts.

Explanation:

IntOn is used to enable interrupts on the microcontroller after IntOff has disabled them. It should be used at the end of code which is timing-sensitive.

IntOn will be removed from the program if no interrupts are used.

See also IntOff, Interrupts

On Interrupt

Syntax:

On Interrupt *event* Call *handler*

On Interrupt *event* Ignore

Command Availability:

Available on PIC and AVR microcontrollers with interrupt support.

Explanation:

On Interrupt will add code to call the subroutine *handler* whenever the interrupt *event* occurs. When Call is specified, GCBASIC will enable the interrupt, and call the interrupt handler when it occurs. When Ignore is specified, GCBASIC will disable the interrupt handler and prevent it from being called when the event occurs. If the event occurs while the handler is disabled, then the handler will be called as soon as it is re-enabled. The only way to prevent this from happening is to manually clear the flag bit for the interrupt.

There are many possible interrupt events that can occur, and the events vary greatly from chip to chip. GCBASIC will display an error if a given chip cannot support the specified event.

In some cases, On Interrupt will not be able to set or clear the interrupt flag and/or enable bits. If this is the case, GCBASIC will display a warning. You will need to consult the chip datasheet and use the Set command to manually set/clear the flag and enable bits, both at the start of the program and inside the interrupt handler subroutine.

If On Interrupt is used to handle an event, then the Interrupt subroutine will not be called for that event. However, it will still be called for any events not dealt with by On Interrupt.

Events:

GCBASIC supports the events shown on the table below. Some events are only implemented on a few specialised chips. Events in grey are supported by PICs and AVRs, events in blue are only supported by some PICs, and events in red are only supported by AVRs.

Note that GCBASIC doesn't fully support all of the hardware which can generate interrupts - some work may be required with various system variables to control the unsupported peripherals.

Event Name	Description
ADCReady	The analog/digital converter has finished a conversion
BatteryFail	The battery has failed in some way. This is only implemented on the ATmega406
CANActivity	CAN bus activity is taking place
CANBadMessage	A bad CAN message has been received

CANError	Some CAN error has occured
CANHighWatermark	CAN high watermark reached
CANRx0Ready	New message present in buffer 0
CANRx1Ready	New message present in buffer 1
CANRx2Ready	New message present in buffer 2
CANRxReady	New message present
CANTransferComplete	Transfer of data has been completed
CANTx0Ready	Buffer 0 has been sent
CANTx1Ready	Buffer 1 has been sent
CANTx2Ready	Buffer 2 has been sent
CANTxReady	Sending has completed
CCADCAccReady	CC ADC accumulate conversion finished (ATmega406 only)
CCADCReady	CC ADC instantaneous conversion finished (ATmega406 only)
CCADCRegular	CC ADC regular conversion finished (ATmega406 only)
CCP1	The CCP1 module has captured an event
CCP2	The CCP2 module has captured an event
CCP3	The CCP3 module has captured an event
CCP4	The CCP4 module has captured an event
CCP5	The CCP5 module has captured an event
Comp0Change	The output of comparator 0 has changed
Comp1Change	The output of comparator 1 has changed
Comp2Change	The output of comparator 2 has changed
Crypto	The KEELOQ module has generated an interrupt
EEPROMReady	An EEPROM write has finished
Ethernet	The Ethernet module has generated an interrupt. This must be dealt with in the handler.
ExtInt0	External Interrupt pin 0 has been ed
ExtInt1	External Interrupt pin 1 has been ed
ExtInt2	External Interrupt pin 2 has been ed
ExtInt3	External Interrupt pin 3 has been ed
ExtInt4	External Interrupt pin 4 has been ed
ExtInt5	External Interrupt pin 5 has been ed
ExtInt6	External Interrupt pin 6 has been ed
ExtInt7	External Interrupt pin 7 has been ed
GPIOChange	The pins on port GPIO have changed
LCDReady	The LCD is about to draw a segment
LPWU	The Low Power Wake Up has been ed
OscillatorFail	The external oscillator has failed, and the PIC is running from an internal oscillator.
PinChange	Logic level of PCINT pin has changed
PinChange0	Logic level of PCINT0 pin has changed
PinChange1	Logic level of PCINT1 pin has changed
PinChange2	Logic level of PCINT2 pin has changed
PinChange3	Logic level of PCINT3 pin has changed
PinChange4	Logic level of PCINT4 pin has changed
PinChange5	Logic level of PCINT5 pin has changed
PinChange6	Logic level of PCINT6 pin has changed
PinChange7	Logic level of PCINT7 pin has changed
PMPReady	A Parallel Master Port read or write has finished
PORTAChange	The pins on port A have changed
PORTABChange	The pins on port A and/or B have changed

PORTBChange	The pins on port B have changed
PSC0Capture	The counter for Power Stage Controller 0 matches the value in a compare register, the value of the counter has been captured, or a synchronisation error has occurred
PSC0EndCycle	Power Stage Controller 0 has reached the end of its cycle
PSC1Capture	The counter for Power Stage Controller 1 matches the value in a compare register, the value of the counter has been captured, or a synchronisation error has occurred
PSC1EndCycle	Power Stage Controller 1 has reached the end of its cycle
PSC2Capture	The counter for Power Stage Controller 2 matches the value in a compare register, the value of the counter has been captured, or a synchronisation error has occurred
PSC2EndCycle	Power Stage Controller 2 has reached the end of its cycle
PSPReady	A Parallel Slave Port read or write has finished
PWMTimeBase	The PWM time base matches the PWM Time Base Period register (PTPER)
SPIReady	The SPI module has finished the previous transfer
SPMReady	A write to program memory by the spm instruction has finished
SPPReady	A SPP read or write has finished
SSP1Collision	SSP1 has detected a bus collision
SSP1Ready	The SSP/SSP1/MSSP1 module has finished sending or receiving
SSP2Collision	SSP2 has detected a bus collision
SSP2Ready	The SSP2/MSSP2 module has finished sending or receiving
Timer0Capture	An input event on the pin ICP0 has caused the value of Timer 0 to be captured in the ICR0 register
Timer0Match1	Timer 0 matches the Timer 0 output compare register A (OCR0A)
Timer0Match2	Timer 0 matches the Timer 0 output compare register B (OCR0B)
Timer0Overflow	Timer 0 has overflowed
Timer1Capture	An input event on the pin ICP1 has caused the value of Timer 1 to be captured in the ICR1 register
Timer1Error	The Timer 1 Fault Protection unit has been ed by an input on the INT0 pin
Timer1Match1	Timer 1 matches the Timer 1 output compare register A (OCR1A)
Timer1Match2	Timer 1 matches the Timer 1 output compare register B (OCR1B)
Timer1Match3	Timer 1 matches the Timer 1 output compare register C (OCR1C)
Timer1Match4	Timer 1 matches the Timer 1 output compare register D (OCR1D)
Timer1Overflow	Timer 1 has overflowed
Timer2Match1	Timer 2 matches the Timer 2 output compare register A (OCR2A)
Timer2Match2	Timer 2 matches the Timer 2 output compare register B (OCR2B)
Timer2Overflow	Timer 2 has overflowed
Timer3Capture	An input event on the pin ICP3 has caused the value of Timer 3 to be captured in the ICR3 register
Timer3Match1	Timer 3 matches the Timer 3 output compare register A (OCR3A)

Timer3Match2	Timer 3 matches the Timer 3 output compare register B (OCR3B)
Timer3Match3	Timer 3 matches the Timer 3 output compare register C (OCR3C)
Timer3Overflow	Timer 3 has overflowed
Timer4Capture	An input event on the pin ICP4 has caused the value of Timer 4 to be captured in the ICR4 register
Timer4Match1	Timer 4 matches the Timer 4 output compare register A (OCR4A)
Timer4Match2	Timer 4 matches the Timer 4 output compare register B (OCR4B)
Timer4Match3	Timer 4 matches the Timer 4 output compare register C (OCR4C)
Timer4Overflow	Timer 4 has overflowed
Timer5CAP1	An input on the CAP1 pin has caused the value of Timer 5 to be captured in CAP1BUF
Timer5CAP2	An input on the CAP2 pin has caused the value of Timer 5 to be captured in CAP2BUF
Timer5CAP3	An input on the CAP3 pin has caused the value of Timer 5 to be captured in CAP3BUF
Timer5Capture	An input event on the pin ICP5 has caused the value of Timer 5 to be captured in the ICR5 register
Timer5Match	Timer5 matches the PR5 register
Timer5Match1	Timer 5 matches the Timer 5 output compare register A (OCR5A)
Timer5Match2	Timer 5 matches the Timer 5 output compare register B (OCR5B)
Timer5Match3	Timer 5 matches the Timer 5 output compare register C (OCR5C)
Timer5Overflow	Timer 5 has overflowed
TWIConnect	The AVR has been connected to or disconnected from the TWI (I2C) bus
TWIReady	The TWI has finished the previous transmission and is ready to send or receive more data
UsartRX1Ready	UART/USART 1 has received data
UsartRX2Ready	UART/USART 2 has received data
UsartRX3Ready	UART/USART 3 has received data
UsartRX4Ready	UART/USART 4 has received data
UsartTX1Ready	UART/USART 1 is ready to send data
UsartTX1Sent	UART/USART 1 has finished sending data
UsartTX2Ready	UART/USART 2 is ready to send data
UsartTX2Sent	UART/USART 2 has finished sending data
UsartTX3Ready	UART/USART 3 is ready to send data
UsartTX3Sent	UART/USART 3 has finished sending data
UsartTX4Ready	UART/USART 4 is ready to send data
UsartTX4Sent	UART/USART 4 has finished sending data
USBEndpoint	A USB endpoint has generated an interrupt
USB	The USB module has generated an interrupt. This must be dealt with in the handler.
USIOverflow	The USI counter has overflowed from 15 to 0
USIStart	The USI module has detected a start condition
VoltageFail	The input voltage has dropped too low
VoltageRegulator	An interrupt has been generated by the voltage regulator (ATmega16HVA only)
WakeUp	The Wake-Up timer has overflowed

WDT	An interrupt has been generated by the Watchdog Timer

Example:

```
      'This program increments a counter every time Timer1 overflows
      #chip 16F877A, 20

'LCD connection settings
      #define LCD_IO 4
      #define LCD_DB4 PORTD.4
      #define LCD_DB5 PORTD.5
      #define LCD_DB6 PORTD.6
      #define LCD_DB7 PORTD.7
      #define LCD_RS PORTD.0
      #define LCD_RW PORTD.1
      #define LCD_Enable PORTD.2

InitTimer1 Osc, PS1_1/8
      StartTimer 1
      CounterValue = 0

Wait 100 ms
      Print "Int Test"

On Interrupt Timer1Overflow Call IncCounter

Do
          CLS
          Print CounterValue
          Wait 100 ms
      Loop

Sub IncCounter
          CounterValue ++
      End Sub
```

For more help, see The InitTimer0 article contains an example of using Timer 0 and On Interrupt to generate a Pulse Width Modulation signal to control a motor.

See also IntOff, IntOn

Overview

Introduction

The keypad routines allow for a program to read from a 4 x 4 matrix keypad. There are two ways that the keypad routines can be set up. One option is to connect the wires from the keypad in a particular order, and then to set the KeypadPort constant. The other option is to connect the keypad in whatever way is easiest, and then set the KEYPAD_ROW_x and KEYPAD_COL_x constants. The first option (setting KeypadPort) will generate slightly more efficient code.

Configuration using KEYPAD_ROW_x and KEYPAD_COL_x

These constants must be set:

Constant Name	Controls	Default Value
KEYPAD_ROW_1	The pin on the microcontroller that connects to the Row 1 pin on the keypad	N/A
KEYPAD_ROW_2	The pin on the microcontroller that connects to the Row 2 pin on the keypad	N/A
KEYPAD_ROW_3	The pin on the microcontroller that connects to the Row 3 pin on the keypad	N/A
KEYPAD_ROW_4	The pin on the microcontroller that connects to the Row 4 pin on the keypad	N/A
KEYPAD_COL_1	The pin on the microcontroller that connects to the Col 1 pin on the keypad	N/A
KEYPAD_COL_2	The pin on the microcontroller that connects to the Col 2 pin on the keypad	N/A
KEYPAD_COL_3	The pin on the microcontroller that connects to the Col 3 pin on the keypad	N/A
	The pin on the	

KEYPAD_COL_4	microcontroller that connects to the Col 4 pin on the keypad	N/A

If using a 3 x 3 keypad, do not set the KEYPAD_ROW_4 or KEYPAD_COL_4 constants.

Configuration using KeypadPort

When setting up the keypad code using the KeypadPort constant, only KeypadPort needs to be set:

Constant Name	Controls	Default Value
KeypadPort	The port on the microcontroller chip that the keypad is connected to.	N/A

For this to work, the keypad must be connected as follows:

Microcontroller port pin	Keypad connector
0	Row 1
1	Row 2
2	Row 3
3	Row 4
4	Column 1
5	Column 2
6	Column 3
7	Column 4

Note: To use a 3 x 3 keypad in this mode, the pins on the microcontroller for any unused columns must be pulled up.

KeypadData

Syntax:

var = KeypadData

Command Availability:

Available on all microcontrollers.

Explanation:

This function will return a value corresponding to the key that is pressed on the keypad. Note that if two or more keys are pressed, then only one value will be returned.

var can have one of the following values:

Value	Constant Name	Key Pressed
0		0
1		1
2		2
3		3
4		4
5		5
6		6
7		7
8		8
9		9
10	KEY_A	A
11	KEY_B	B
12	KEY_C	C
13	KEY_D	D
14	KEY_STAR	Asterisk/Star (*)
15	KEY_HASH	Hash (#)
255	KEY_NONE	None

Example:

```
     'Program to show the value of the last pressed key on the LCD
     #chip 18F4550, 20


'LCD connection settings
     #define LCD_IO 4
     #define LCD_DB4 PORTD.4
     #define LCD_DB5 PORTD.5
     #define LCD_DB6 PORTD.6
     #define LCD_DB7 PORTD.7
     #define LCD_RS PORTD.0
```

```
      #define LCD_RW PORTD.1
      #define LCD_Enable PORTD.2

'Keypad connection settings
      #define KeypadPort PORTB

'Main loop
      Do
          'Get key
          Temp = KeypadData

          'If a key is pressed, then display it
          If Temp <> KEY_NONE Then
              CLS
              Print Temp
              Wait 100 ms
          End If
      Loop
```

For more help, see Relevant Constants

KeypadRaw

Syntax:

largevar = KeypadRaw

Command Availability:

Available on all microcontrollers.

Explanation:

This function will return a 16 bit value, in which each bit corresponds to a key on the keypad. If the key is pressed its bit will hold 1, and if it is released its bit will contain a 0.

This table shows the key that each bit corresponds to:

Bit	Key Position (row, col)	Common Key Symbol
15	1,1	1
14	1,2	2
13	1,3	3
12	1,4	A
11	2,1	4
10	2,2	5
9	2,3	6
8	2,4	B
7	3,1	7
6	3,2	8
5	3,3	9
4	3,4	C
3	4,1	*
2	4,2	0
1	4,3	#
0	4,4	D

Example:

```
     'Program to show the keypad status using LEDs
     #chip 16F877A, 20

'Keypad connection settings
     #define KeypadPort PORTB

'LEDs
     #define LED1 PORTC
```

```
      #define LED2 PORTD
      Dir LED1 Out
      Dir LED2 Out

'Declare a 16 bit variable for the key value
      Dim KeyStatus As Word

'Main loop
      Do
          'Get key
          KeyStatus = KeypadRaw

          'Display
          LED1 = KeyStatus_H 'High Byte
          LED2 = KeyStatus 'Low Byte
      Loop
```

For more help, see Relevant Constants

GLCD Overview

The GLCD commands are used to control a graphical Liquid Crystal Display based on the KS0108 driver chip. These are often 128x64 pixel displays. They can draw graphical elements by enabling or disabling pixels.
GCB makes this type of device easier to control with the commands for the GLCD.

Setup:

You **must** include the glcd.h file at the top of your program. The file needs to be in brackets.
#include <GLCD.h>

Defines:

There are several connections that must be defined to use the GLCD commands with a KS0108 display. The *I/O pin* is the pin on the PIC Microcontroller that is connected to that specific pin on the graphic LCD.

```
#define GLCD_RW  I/O pin    'Read/Write pin connection
    #define GLCD_RESET  I/O pin    'Reset pin connection
    #define GLCD_CS1  I/O pin    'CS1 pin connection
    #define GLCD_CS2  I/O pin    'CS2 pin connection
    #define GLCD_RS  I/O pin       'RS pin connection
    #define GLCD_ENABLE  I/O pin    'Enable pin Connection
    #define GLCD_DB0  I/O pin    'Data pin 0 Connection
    #define GLCD_DB1  I/O pin    'Data pin 1 Connection
    #define GLCD_DB2  I/O pin    'Data pin 2 Connection
    #define GLCD_DB3  I/O pin    'Data pin 3 Connection
    #define GLCD_DB4  I/O pin    'Data pin 4 Connection
    #define GLCD_DB5  I/O pin    'Data pin 5 Connection
    #define GLCD_DB6  I/O pin    'Data pin 6 Connection
    #define GLCD_DB7  I/O pin     'Data pin 7 Connection
```

Commands:

```
    InitGLCD    'Initialize port pins

GLCDCLS    'Clear screen

GLCDPrint(In PrintLocX, In PrintLocY, PrintData As String)

GLCDDrawChar(In CharLocX, In CharLocY, In CharCode)
```

Box(In LineX1, In LineY1, In LineX2, In LineY2, Optional In LineColour = 1)

FilledBox(In LineX1, In LineY1, In LineX2, In LineY2, Optional In LineColour = 1)

Line(In LineX1, In LineY1, In LineX2, In LineY2, Optional In LineColour = 1)

PSet(In GLCDX, In GLCDY, In GLCDState)

GLCDWriteByte (In LCDByte)

GLCDReadByte

Example:

This example shows how to drive a KS0108 based Graphic LCD module with the built in commands of Great Cow Basic. See Graphic LCD for details, this is an external web site.

```
;Chip Settings
      #chip 16F886,16
      '#config MCLRE = on 'enable reset switch on CHIPINO
      #include <chipino.h>
      #include <GLCD.h>

;Defines (Constants)
      #define GLCD_RW PORTB.1  'D9 to pin 5 of LCD
      #define GLCD_RESET PORTB.5 'D13 to pin 17 of LCD
      #define GLCD_CS1 PORTB.3 'D12 to actually since CS1, CS2 are backward
      #define GLCD_CS2 PORTB.4 'D11 to actually since CS1, CS2 are backward
      #define GLCD_RS PORTB.0  'D8 to pin 4 D/I pin on LCD
      #define GLCD_ENABLE PORTB.2 'D10 to Pin 6 on LCD
      #define GLCD_DB0 PORTC.7 'D0 to pin 7 on LCD
      #define GLCD_DB1 PORTC.6 'D1 to pin 8 on LCD
      #define GLCD_DB2 PORTC.5 'D2 to pin 9 on LCD
      #define GLCD_DB3 PORTC.4 'D3 to pin 10 on LCD
      #define GLCD_DB4 PORTC.3 'D4 to pin 11 on LCD
      #define GLCD_DB5 PORTC.2 'D5 to pin 12 on LCD
      #define GLCD_DB6 PORTC.1 'D6 to pin 13 on LCD
      #define GLCD_DB7 PORTC.0 'D7 to pin 14 on LCD

InitGLCD
      Start:
      GLCDCLS
      GLCDPrint 0,10,"Hello" 'Print Hello
      wait 5 s
      GLCDPrint 0,10, "ASCII #:" 'Print ASCII #:
      Box 18,30,28,40                    'Draw Box Around ASCII Character
      for char = 15 to 129             'Print 0 through 9
      GLCDPrint 17, 20 , Str(char)
      GLCDdrawCHAR 20,30, char
      wait 1 s
```

```
next
line 0,50,127,50                'Draw Line using line command
for xvar = 0 to 80             'draw line using Pset command
pset xvar,63,on                     '
next                                          '
Wait 10 s
Goto Start
```

See also InitGLCD, GLCDCLS, GLCDDrawChar, GLCDPrint, GLCDReadByte, GLCDWriteByte, Pset

GLCDCLS

Syntax:

GLCDCLS

Explanation:

Clears the screen of a Graphic LCD.

GLCDDrawChar

Syntax:

GLCDDrawChar(CharLocX, CharLocY, CharCode)
CharLocX is the X corrdinate location for the character
CharLocY is the Y coordinate location for the character
CharCode is the ASCII number of the character to display. Can be decimal hex or binary.

Explanation:

Displays an ASCII character at a specified X and Y location.
On a 128x64 Graphic LCD
X = 1 to 128
Y = 1 to 64

GLCDPrint

Syntax:

GLCDPrint(PrintLocX, PrintLocY, PrintData)
PrintLocX is the X corrdinate location for the data
PrintLocY is the Y coordinate location for the data
PrintData is a String or String variable of the data to display

Explanation:

Prints string character(s) at a specified location on the GLCD screen.

On a 128x64 GLCD display
X is typically 0 to 128
Y is typically 0 to 64

GLCDReadByte

Syntax:

byte_variable = GLCDReadByte

Explanation:

Reads a byte of data from the Graphic LCD memory

GLCDWriteByte

Syntax:

GLCDWriteByte (LCDByte)

Explanation:

Writes a byte of data to the Graphic LCD memory

Line

Syntax:

Line(LineX1,LineY1, LineX2, LineY2, Optional LineColour = 1)

Explanation:

Draws a line on a graphic LCD from X1, Y1 location to X2,Y2 location. Must be horizontal or vertical. Diagonal will not work properly.

InitGLCD

Syntax:

InitGLCD

Explanation:

This initializes the Graphical LCD for operation. Here are the steps it takes:

Example:

```
'Set pin directions

Dir GLCD_RS Out
Dir GLCD_RW Out
Dir GLCD_ENABLE Out
Dir GLCD_CS1 Out
Dir GLCD_CS2 Out
Dir GLCD_RESET Out

'Reset

Set GLCD_RESET Off
Wait 1 ms
Set GLCD_RESET On
Wait 1 ms

'Select both chips

Set GLCD_CS1 On
Set GLCD_CS2 On

'Set on

Set GLCD_RS Off
GLCDWriteByte 63

'Set Z to 0

GLCDWriteByte 192

'Deselect chips

Set GLCD_CS1 Off
Set GLCD_CS2 Off

'Clear screen

GLCDCLS
```

Pset

Syntax:

PSet(GLCDX, GLCDY, GLCDState)

Explanation:

Sets or Clears a Pixel at the specified X, Y location. A one for GLCDState sets the pixel and a zero clears the pixel.

Box

Syntax:

Box(LineX1,LineY1, LineX2, LineY2, Optional LineColour = 1)

Explanation:

Draws a box on a graphic LCD from the upper corner of X1, Y1 location to X2,Y2 location.

See also FilledBox

FilledBox

Syntax:

FilledBox(LineX1,LineY1, LineX2, LineY2, Optional LineColour = 1)

Explanation:

Draws a filled box on a graphic LCD from the upper corner of X1, Y1 location to X2,Y2 location.

See also Box

LCD Overview

Introduction:

The routines in this section allow GCBASIC programs to interact with alphanumeric Liquid Crystal Displays based on the HD44780 IC. This covers most 16 x 2 and similar displays.

These routines allow the displays to be connected to the microcontroller in many different ways:

Connection Mode	Required Connections
0	None are required directly by the routines. However, the LCD routines must be provided with other subroutines which will handle the communication. This is useful for communicating with LCDs connected through RS232 or I2C.
2	Data and Clock lines. This mode is used when the LCD is connected through a 74LS174 shift register IC, as detailed at **the Internet way back engine** or [todo]
4	R/W, RS, Enable and the highest 4 data lines (DB4 through DB7).
8	R/W, RS, Enable and all 8 data lines. The data lines must all be connected to the same I/O port, in sequential order. For example, DB0 to PORTB.0, DB1 to PORTB.1 and so on, with DB7 going to PORTB.7.

Using 0-bit mode:

To use 0-bit connection mode, a subroutine to write a byte to the LCD must be provided. Optionally, another subroutine to read a byte from the LCD can also be given. If there is no way to read from the LCD, then the LCD_NO_RW constant must be set.

In 0-bit mode, the LCD_RS constant will be set automatically to a spare bit variable. The higher level LCD commands (such as Print and Locate) will set it, and the code responsible for writing to the LCD should read it and then set the RS pin on the LCD appropriately.

This code is an example of how to use 0-bit mode. It sends messages to another microcontroller, which has been programmed to read the messages and toggle the pins of an LCD appropriately:

```
#define LCD_IO 0
```

```
      #define LCDWriteByte MySendToLCD
      #define LCD_NO_RW

Sub MySendToLCD(In MyLCDByte)

          'Uses I2C
          'Sends an address byte (128)
          'Then a control byte, where bit 4 is the state of the RS pin
          'Then a data byte, which is sent to the LCD data pins.

          ControlByte = 0
          If LCD_RS = On Then ControlByte.4 = On

          I2CStart
          I2CSend 128
          I2CSend ControlByte
          I2CSend MyLCDByte
          I2CStop

    'Need to allow time for receiver of message to update LCD
          Wait 5 ms

      End Sub
```

If the LCD was to be read, then LCDReadByte would be set to the name of a function that reads the LCD and returns the data byte from the LCD.

Relevant Constants:

(Note: this section does not apply in 0 bit mode)

These constants are used to control settings for the Liquid Crystal Display routines included with GCBASIC. To set them, place a line in the main program file that uses #define to assign a value to the particular constant.

Some constants are used required for 4 and 8-bit modes, some are required for 4-bit mode, and some are used by 8-bit mode. When using 2-bit mode only three constants must be set - all others can be ignored. Check the "Modes" column to determine if you must set a constant.

Constant Name	Controls	Default Value	Modes
LCD_IO	The I/O mode. Can be 2, 4 or 8.	8	2, 4, 8
LCD_DB	The data pin used in 2-bit mode.	N/A - ***Must be set***	2
LCD_CB	The clock pin used in 2- bit mode.	N/A - ***Must be set***	2
LCD_RS	Specifies the output pin that is connected to Register Select on the LCD.	N/A - ***Must be set***	4, 8

LCD_RW	Specifies the output pin that is connected to Read/Write on the LCD. The R/W pin can be disabled*.	N/A - ***Must be set*** *(unless R/W is disabled)*	4, 8
LCD_Enable	Specifies the output pin that is connected to Read/Write on the LCD.	N/A - ***Must be set***	4, 8
LCD_DATA_PORT	Output port used to interface with LCD data bus	N/A - ***Must be set***	8 only
LCD_DB4	Output pin used to interface with bit 4 of the LCD data bus	N/A - ***Must be set***	4 only
LCD_DB5	Output pin used to interface with bit 5 of the LCD data bus	N/A - ***Must be set***	4 only
LCD_DB6	Output pin used to interface with bit 6 of the LCD data bus	N/A - ***Must be set***	4 only
LCD_DB7	Output pin used to interface with bit 7 of the LCD data bus	N/A - ***Must be set***	4 only

*The R/W pin can be disabled by setting the LCD_NO_RW constant. If this is done, there is no need for the R/W to be connected to the chip, and no need for the LCD_RW constant to be set. Ensure that the R/W line on the LCD is connected to ground if not used!

Print

Syntax:

Print *string*
Print *byte*
Print *word*
Print *integer*

Command Availability:

Available on all microcontrollers.

Explanation:

The Print command will show the contents of a variable on the LCD. It can display string, word or byte variables.

Example:

```
      'A Light Meter program.

'General hardware configuration
      #chip 16F877A, 20
      #define LightSensor AN0

'LCD connection settings
      #define LCD_IO 8
      #define LCD_DATA_PORT PORTC
      #define LCD_RS PORTD.0
      #define LCD_RW PORTD.1
      #define LCD_Enable PORTD.2

CLS
      Print "Light Meter:
      Locate 1,2
      Print "A GCBASIC Demo"
      Wait 2 s

Do
          CLS
          Print "Light Level: "
          Print ReadAD(LightSensor)
          Wait 250 ms
      Loop
```

For more help, see Relevant Constants

Locate

Syntax:

Locate *line, column*

Command Availability:

Available on all microcontrollers.

Explanation:

The Locate command is used to move the cursor on the LCD to the given location.

Example:

```
     'A Hello World program for GCBASIC.
     'Uses Locate to show "World" on the second line

'General hardware configuration
     #chip 16F877A, 20

'LCD connection settings
     #define LCD_IO 8
     #define LCD_DATA_PORT PORTC
     #define LCD_RS PORTD.0
     #define LCD_RW PORTD.1
     #define LCD_Enable PORTD.2

'Main routine
     Print "Hello"
     Locate 1, 5
     Print "World"
```

For more help, see Relevant Constants

Put

Syntax:

Put Line,Column, Character

Command Availability:

Available on all microcontrollers.

Explanation:

The Put command writes the given ASCII character code to the location on the LCD.

Example:

```
     'A scrolling star for GCBASIC

'Misc Settings
     #define SCROLL_DELAY 250 ms

'General hardware configuration
     #chip 16F877A, 20

'LCD connection settings
     #define LCD_IO 8
     #define LCD_DATA_PORT PORTC
     #define LCD_RS PORTD.0
     #define LCD_RW PORTD.1
     #define LCD_Enable PORTD.2

'Main routine
     For StarPos = 0 To 16
         If StarPos = 0 Then
             Put 0, 16, 32
             Put 0, 0, 42
         Else
             Put 0, StarPos - 1, 32
             Put 0, StarPos, 42
         End If
         Wait SCROLL_DELAY
     Next
```

For more help, see Relevant Constants

Get

Syntax:

var = Get(Line, Column)

Command Availability:

Available on all microcontrollers, except if the LCD is connected using 2 bit mode.

Explanation:

The Get function reads the ASCII character code at a given location on the LCD.

For more help, see Put, Relevant Constants

CLS

Syntax:

CLS

Command Availability:

Available on all microcontrollers.

Explanation:

The CLS command clears the contents of the LCD, and returns the cursor to the top left corner of the screen

Example:

```
'A Flashing Hello World program for GCBASIC

'General hardware configuration
     #chip 16F877A, 20

'LCD connection settings
     #define LCD_IO 8
     #define LCD_DATA_PORT PORTC
     #define LCD_RS PORTD.0
     #define LCD_RW PORTD.1
     #define LCD_Enable PORTD.2

'Main routine
     Do
         Print "Hello World"
         Wait 1 sec
         CLS
         Wait 1 sec
     Loop
```

For more help, see Relevant Constants

LCDCreateChar

Syntax:

LCDCreateChar *char, chardata()*

Command Availability:

Available on all microcontrollers.

Explanation:

The LCDCreateChar command is used to send a custom character to the LCD.

Each character on the LCD is made up from an 8 row by 5 column (5x8) matrix of pixels. The data to be sent to the LCD is composed of an 8 element array, where each element corresponds to a row. Inside each element, the 5 lowest bits make up the data for the corresponding row. When a bit is set a dot will be drawn at the matching location; when it is cleared, no dot will appear.

An array of more than 8 elements may be used, but only the first 8 will be read.

char is the ASCII value of the character to create. ASCII codes 0 through 7 are usually used to store custom characters.

chardata() is an array containing the data for the character.

Example:

```
     'This program draws a smiling face character

'General hardware configuration
     #chip 16F877A, 20

'LCD connection settings
     #define LCD_IO 8
     #define LCD_DATA_PORT PORTC
     #define LCD_RS PORTD.0
     #define LCD_RW PORTD.1
     #define LCD_Enable PORTD.2

'Create an array to store the character until it is copied
     Dim TempArray(8)

'Set the array to hold the character
     'Binary has been used to improve the readability of the code, but is not essential
     TempArray(1) = b'00011011'
     TempArray(2) = b'00011011'
     TempArray(3) = b'00000000'
     TempArray(4) = b'00000100'
     TempArray(5) = b'00000000'
     TempArray(6) = b'00010001'
     TempArray(7) = b'00010001'
     TempArray(8) = b'00001110'
```

```
'Copy the character from the array to the LCD
     LCDCreateChar 0, TempArray()

'Draw the custom character
     LCDWriteChar 0
```

For more help, see LCDWriteChar, Relevant Constants

LCDHex

Syntax:

LCDHex *value*

Command Availability:

Available on all microcontrollers.

Explanation:

The LCDHex command will show the specified value on the LCD, at the current cursor position. It will convert the byte it has been given into hexadecimal format.

Example:

```
      'A program to count from 0 to FF on an LCD screen.

'General hardware configuration
      #chip 16F877A, 20

'LCD connection settings
      #define LCD_IO 8
      #define LCD_DATA_PORT PORTC
      #define LCD_RS PORTD.0
      #define LCD_RW PORTD.1
      #define LCD_Enable PORTD.2

For Counter = 0 To 255
          CLS
          LCDHex Counter
          Wait 250 ms
      Next
```

LCDWriteChar

Syntax:

LCDWriteChar *char*

Command Availability:

Available on all microcontrollers.

Explanation:

The LCDWriteChar command will show the specified character on the LCD, at the current cursor position.

char is the ASCII value of the character to show. On most LCDs, characters 0 through 7 are user defined, and can be set using the LCDCreateChar command.

Example:

```
     'This program draws a smiling face character

'General hardware configuration
     #chip 16F877A, 20

'LCD connection settings
     #define LCD_IO 8
     #define LCD_DATA_PORT PORTC
     #define LCD_RS PORTD.0
     #define LCD_RW PORTD.1
     #define LCD_Enable PORTD.2

'Create an array to store the character until it is copied
     Dim TempArray(8)

'Set the array to hold the character
     TempArray(1) = b'00011011'
     TempArray(2) = b'00011011'
     TempArray(3) = b'00000000'
     TempArray(4) = b'00000100'
     TempArray(5) = b'00000000'
     TempArray(6) = b'00010001'
     TempArray(7) = b'00010001'
     TempArray(8) = b'00001110'

'Copy the character from the array to the LCD
     LCDCreateChar 0, TempArray()

'Draw the custom character
     LCDWriteChar 0
```

For more help, see LCDCreateChar, Relevant Constants

Overview

Introduction:

The routines described in this chapter allow the generation of Pulse Width Modulation signals. These allow for the microcontroller to control the speed of a motor, or the brightness of a light. The routines can also be used to generate the appropriate frequency signal to drive an infrared LED for remote control applications.

The PWMOn, PWMOff and HPWM routines use the PWM generation module on the microcontroller. They will only work on some microcontrollers - see the article on each command for details. PWMOn and HPWM will cause the PWM module on the microcontroller to start generating the PWM signal, which will then continue to be generated until the PWMOff instruction is run.

PWMOut does not make use of any special hardware. However, a signal is only generated while the PWMOut command is running - when the program moves on to the next command, the signal will stop.

Relevant Constants:

These constants are used to control settings for the Pulse Width Modulation module of the PIC chip. To set them, place a line in the main program file that uses #define to assign a value to the particular constant.

Note that there are two sets of constants: one for Hardware PWM, and one for Software PWM. Hardware PWM requires a CCP module on the PIC chip - Software PWM has no requirements regarding the PIC.

Hardware PWM

These constants are only required for PWMOn. HPWM and PWMOff do not require any constants to operate.

Constant Name	Controls	Default Value
PWM_Freq	Specifies the output frequency of the PWM module in KHz.	38
PWM_Duty	Sets the duty cycle of the PWM module output. Given as percentage.	50

Hardware PWM is only available through the "CCP1" or "CCP" pin. This is a hardware limitation of PIC microcontrollers.

Software PWM

Constant Name	Controls	Default Value
	The PWM Period. The length of any	

PWM_Delay	delay used will be multiplied by 255. If no value is specified, no delays will be inserted into the PWM routine.	Not defined - no delay
PWM_Out*n*	The port physical port on the PIC that corresponds to channel *n*. *n* can represent 1, 2, 3 or 4.	Not Defined

More than 4 channels are possible, but for this the PWMOut routine in include\lowlevel\stdbasic.h must be altered.

PWMOut

Syntax:

PWMOut *channel, duty cycle, cycles*

Command Availability:

Available on all microcontrollers.

Explanation:

This command uses a software PWM routine included in GCBASIC to produce a PWM signal on the selected port of the chip. This routine does NOT require a PWM module on the chip.

channel sets the channel that the PWM is to be generated on. This must have been defined previously by setting the constants SoftPWM1, SoftPWM2, etc. The maximum number of channels available is 4.

duty cycle specifies the PWM duty cycle, and ranges from 0 to 255. 255 corresponds to 100%, 127 to 50%, 63 to 25%, and so on.

cycles is used to set the amount of PWM pulses to supply. This is useful for situations in which a pulse of a specific length is required. The formula for calculating the time taken for one cycle is:

$T_{CYCLE} = (28 + 10C)T_{OSC} + 255PWM\_Delay,$

where C is the number of channels used and T_{OSC} is the length of time taken to execute 1 instruction on the chip (0.2 us on a 20 MHz chip, 1 us on a 4 Mhz chip). PWM_Delay is a length of time specified using the PWM_Delay constant.

Example:

```
'This program controls the brightness of an LED on PORTB.0     'using the software PWM routine and a potentiometer.

'Select chip model
#chip 16F877A, 20

'Set PWM Port
#define PWM_Out1 PORTB.0

'Set port directions
dir SoftPWM1 out    'PWM
dir PORTA.0 in    'Potentiometer

'Main routine
do
   PWMOut 1, ReadAD(AN0), 100     '100 cycles is a purely
                'arbitrary value here.
```

loop

PWMOff

Syntax:

PWMOff

Command Availability:

Only available on PIC microcontrollers with Capture/Compare/PWM (CCP) module.

Explanation:

This command will disable the output of the PWM module on the PIC chip.

Example:

```
'This program will enable a 76 Khz PWM signal, with a duty cycle
'of 80%. It will emit the signal for 10 seconds, then stop.
#define PWM_Freq 76    'Set frequency in KHz
#define PWM_Duty 80    'Set duty cycle to 80 %
PWMOn                  'Turn on the PWM
WAIT 10 s              'Wait 10 seconds
PWMOff                 'Turn off the PWM
```

For more help, see PWMOn

PWMOn

Syntax:

PWMOn

Command Availability:

Only available on PIC microcontrollers with Capture/Compare/PWM (CCP) module.

Explanation:

This command will enable the output of the PWM module on the PIC chip.

Example:

```
'This program will enable a 76 Khz PWM signal, with a duty cycle
'of 80%. It will emit the signal for 10 seconds, then stop.
#define PWM_Freq 76    'Set frequency in KHz
#define PWM_Duty 80    'Set duty cycle to 80 %
PWMOn                  'Turn on the PWM
WAIT 10 s              'Wait 10 seconds
PWMOff                 'Turn off the PWM
```

For more help, see PWMOff

HPWM

Syntax:

HPWM *channel, frequency, duty cycle*

Command Availability:

Only available on PIC microcontrollers with Capture/Compare/PWM (CCP) module.

Explanation:

This command sets up the hardware PWM module of the PIC chip to generate a PWM waveform of the given frequency and duty cycle. Once this command is called, the PWM will be emitted until PWMOff is called. If you only need one particular frequency and duty cycle, you should use PWMOn and the constants PWM_Freq and PWM_Duty instead.

channel is 1 or 2, and corresponds to the pins CCP1 and CCP2 respectively. On chips with only one CCP port, pin CCP or CCP1 is always used, and *channel* is ignored. (It should be set to 1 anyway to allow for future upgrades to more powerful PIC chips.)

frequency sets the frequency of the PWM output. It is measured in KHz. The maximum value allowed is 255 KHz. The minimum value varies depending on the clock speed. 1 KHz is the minimum on chips 16 MHz or under and 2 Khz is the lowest possible on 20 MHz chips. In situations that do not require a specific PWM frequency, the PWM frequency should equal approximately 1 five-hundredth the clock speed of the PIC (ie 40 Khz on a 20 MHz chip, 16 KHz on an 8 MHz chip). This gives the best duty cycle resolution possible.

duty cycle specifies the desired duty cycle of the PWM signal, and ranges from 0 to 255 where 255 is 100% duty cycle.

Example:

```
'This program will alter the brightness of an LED using
'hardware PWM.

'Select chip model and speed
#chip 16F877A, 20

'Set the CCP1 pin to output mode
DIR PORTC.2 out

'Main code
do
   'Turn up brightness over 2.5 seconds
   For Bright = 1 to 255
      HPWM 1, 40, Bright
      wait 10 ms
   next
   'Turn down brightness over 2.5 seconds
   For Bright = 255 to 1
```

```
        HPWM 1, 40, Bright
        wait 10 ms
    next
loop
```

For more help, see PWMOff

Overview

Introduction:

These routines allow GCBASIC to generate pseudo-random numbers. The generator uses a 16 bit linear feedback shift register to produce pseudo-random numbers. The most significant 8 bits of the LFSR are used to provide an 8 bit random number.

When compiling a program, GCBASIC will generate an initial seed for the generator. However, this seed will be the same every time the program runs, so the sequence of numbers produced by a given program will always be the same. To work around this, there is a Randomize subroutine. It can be provided with a new seed for the generator (which will cause the generator to move to a different point in the sequence). Alternatively, Randomize can be set to obtain a seed from some other source such as a timer every time it is run.

Relevant Constants:

These constants are used to control settings for the tone generation routines. To set them, place a line in the main program file that uses #define to assign a value to the particular constant.

Constant Name	Controls	Default Value
RANDOMIZE_SEED	Source of the random seed if Randomize is called without a parameter	Timer0

Random

Syntax:

var = Random

Command Availability:

Available on all microcontrollers

Explanation:

The Random function will generate a pseudo-random number between 0 and 255 inclusive.

The numbers generated by Random will follow the same sequence every time, until Randomize is used.

Example:

```
     'Set chip model
     #chip tiny2313, 1

'Use randomize, with the value on PORTD as the seed
     Randomize PORTD

'Generate random numbers, and output on PORTB
     Do
         PORTB = Random
         Wait 1 s
     Loop
```

Randomize

Syntax:

Randomize
Randomize *seed*

Command Availability:

Available on all microcontrollers

Explanation:

Randomize is used to seed the pseudo random number generator, so that it will produce a different sequence of numbers each time it is used.

If no *seed* is specified, then the RANDOMIZE_SEED constant will be used as the seed. If *seed* is specified, then it will be used to seed the generator.

It is important that the seed is different every time that Randomize is used. If the seed is always the same, then the sequence of numbers will always be the same. It is best to use a running timer, an input port, or the analog to digital converter as the source of the seed, since these will normally provide a different value each time the program runs.

Example:

```
     'Set chip model
     #chip tiny2313, 1

'Use randomize, with the value on PORTD as the seed
     Randomize PORTD

'Generate random numbers, and output on PORTB
     Do
         PORTB = Random
         Wait 1 s
     Loop
```

Overview

Introduction

The 7 segment display routines make it easier for GCBASIC programs to display numbers and letters on 7 segment LED displays.

There are two ways that the 7 segment display routines can be set up. One option is to connect the wires from the display/s in a particular order, and then to set the DisplayPort*n* and DispSelect*n* constants. The other option is to connect the display/s in whatever way is easiest, and then set the DISP_SEG_x and DISP_SEL_x constants. The first option (setting DisplayPort*x* and DispSelect*n)* will generate slightly more efficient code.

Configuration using DISP_SEG_x and DISP_SEL_x

When setting up the 7 segment code with DISP_SEG_x constants, these must be set:

Constant Name	Controls	Default Value
DISP_SEG_x	Controls the output ping used to control segment x of the displays. There are 7 of these constants, named DISP_SEG_A through DISP_SEG_G. One must be set for each segment.	N/A
DISP_SEG_DOT	Specifies the output pin used to control the decimal point on the displays.	N/A
DISP_SEL_x	The command used to select display *n*. Used to control addressing pins when several displays are multiplexed.	N/A

Note: Instead of setting DISP_SEL_x, it is possible to set DispSelectn and use these in conjunction with DISP_SEG_x.

Configuration using DisplayPort*n* and DispSelect*n*

To set the 7-Segment display routines supplied with GCBASIC using DisplayPortn, it is necessary to set these constants:

Constant Name	Controls	Default Value
	Controls the output	

DisplayPort*n*	port used to control display *n*. *n* is A, B, C or D, corresponding to displays 1, 2, 3 and 4, respectively.	N/A
DispSelect*n*	The command used to select display *n*. Used to control addressing pins when several displays are multiplexed.	nop

To set up the routines in this way, the displays must be connected as follows:

Microcontroller port pin	**Display Segment**
0	A
1	B
2	C
3	D
4	E
5	F
6	G

DisplayValue

Syntax:

DisplayValue *display, data*

Command Availability:

Available on all microcontrollers.

Explanation:

This command will display the given value on a seven segment LED display. *display* is the number of the display to use, and *data* is the value between 0 and 9 to show

Example:

```
'This program will count from 0 to 99 on two LED displays
#chip 16F819, 8
#config osc = int

#define DISP_SEG_A PORTB.0
#define DISP_SEG_B PORTB.1
#define DISP_SEG_C PORTB.2
#define DISP_SEG_D PORTB.3
#define DISP_SEG_E PORTB.4
#define DISP_SEG_F PORTB.5
#define DISP_SEG_G PORTB.6

#define DISP_SEL_1 PORTA.0
#define DISP_SEL_2 PORTA.1

Do
    For Counter = 0 To 99

        'Get the 2 digits
        Number = Counter
        Num1 = 0
        If Number >= 10 Then
            Num1 = Number / 10
            'SysCalcTempX stores remainder after division
            Number = SysCalcTempX
        End If
        Num2 = Number

        'Show the digits
        'Each DisplayValue will erase the other (multiplexing)
        'So they must be called often enough that the flickering
        'cannot be seen.
        Repeat 500
            DisplayValue 1, Num1
            Wait 1 ms
            DisplayValue 2, Num2
            Wait 1 ms
        End Repeat
    Next
Loop
```

For more help, see Relevant Constants

DisplayChar

Syntax:

DisplayChar (*display, character*)

Command Availability:

Available on all microcontrollers.

Explanation:

This command will display the given ASCII character on a seven segment LED display. *display* is the number of the display to use, and *character* is the ASCII character to show.

Example:

```
      'This program will show "Hello" on a LED display
      'The display should be connected to PORTB

#chip 16F877A, 20
      #define DisplayPortA PORTB

Dim Message As String
      Message = "Hello "
      For Counter = 1 to 6
          DisplayChar 1, Message(Counter)
          Wait 250 ms
      Next
```

For more help, see Relevant Constants

RS232 Software Overview

Introduction:

These routines allow the microcontroller to send and receive RS232 data. All functions are implemented using software, so no special hardware is required on the microcontroller. However, if the microcontroller has a hardware serial module (usually referred to as UART or USART), and the serial data lines are connected to the appropriate pins, the hardware routines should be used for smaller code, improved reliability and higher baud rates.

Relevant Constants:

These constants are used to control settings for the RS232 serial communication routines. To set them, place a line in the main program file that uses #define to assign a value to the particular constant.

Constant Name/s	Controls	Default Value
SendALow, SendBLow, SendCLow	These are used to define the commands used to send a low (0) bit on serial channels A, B and C respectively.	nop **(must be set)**
SendAHigh, SendBHigh, SendCHigh	These are used to define the commands used to send a high (1) bit on serial channels A, B and C respectively.	nop **(must be set)**
RecALow, RecBLow, RecCLow	The condition that is true when a low bit is being received	Sys232Temp.0 OFF **(must be set)**
RecAHigh, RecBHigh, RecCHigh	The condition that is true when a high bit is being received	Sys232Temp.0 ON **(must be set)**

InitSer

Syntax:

InitSer *channel, rate, start, data, stop, parity, invert*

Command Availability:

Available on all microcontrollers.

Explanation:

This command will set up the serial communications. The parameters are as follows:

- *channel* is 1, 2 or 3, and refers to the I/O ports that are used for communication.
- *rate* is the bit rate, which is given by the letter r and then the desired rate in bps. Acceptable units are r300, r600, r1200, r2400, r4800, r9600 and r19200.
- *start* gives the number of start bits, which is usually 1. To make the PIC wait for the start bit before proceeding with the receive, add 128 to *start*. (Note: it may be desirable to use the WaitForStart constant here.)
- *data* tells the program how many data bits are to be sent or received. In most situations this is 8, but it can range between 1 and 8, inclusive.
- *stop* is the number of stop bits. If *start* bit 7 is on, then this number will be ignored.
- *parity* refers to a system of error checking used by many devices. It can be odd (in which there must always be an odd number of high bits), even (where the number of high bits must always be even), or none (for systems that do not use parity).
- *invert* can be either "normal" or "invert". If it in "invert", then high bits will be changed to low, and low to high.

Example:

Please refer to SerSend for an example of InitSer

For more help, see Relevant Constants

SerSend

Syntax:

SerSend *channel, data*

Command Availability:

Available on all microcontrollers.

Explanation:

This command will send a byte given by *data* using the RS232 channel referred to as *channel* according to the rules set using InitSer.

Example:

```
      'This program will send a byte using PORTB.2, the value of which
      'depends on whether a button is pressed. This can be used with the 'example for
      SerReceive.

#chip 16F819, 8
      #config Osc = Int

#define SendAHigh Set PORTB.2 ON
      #define SendALow Set PORTB.2 OFF
      #define Button PORTA.0

Dir Button In
      Dir PORTB.2 Out

InitSer 1, r9600, 1+WaitForStart, 8, 1, none, normal
      Do
          If Button = On Then Temp = 100
          If Button = Off Then Temp = 0
          SerSend 1, Temp
          Wait 100 ms
      Loop
```

For more help, see Relevant Constants, InitSer

SerReceive

Syntax:

SerReceive *channel, output*

Command Availability:

Available on all microcontrollers.

Explanation:

This command will read a byte from the RS232 channel given by *channel* according to the rules set using InitSer, and store the received byte in the variable *output*.

Example:

```
      'This program will read a byte from PORTB.2, and set the LED on if
      'the byte is more than 50. This can be used with the SerSend
      'example program.

#chip 16F88, 8
      #config Osc = Int

#define RecAHigh PORTB.2 ON
      #define RecALow PORTB.2 OFF
      #define LED PORTB.0

Dir PORTB.0 Out
      Dir PORTB.2 In

InitSer 1, r9600, 1 + WaitForStart, 8, 1, none, normal
      Do
          SerReceive 1, Temp
          If Temp <= 50 Then Set LED Off
          If Temp > 50 Then Set LED On
      Loop
```

For more help, see Relevant Constants, InitSer

SerPrint

Syntax:

SerPrint *channel, value*

Command Availability:

Available on all microcontrollers.

Explanation:

SerPrint is used to send a value over the serial connection. *value* can be a string, word or byte - SerPrint is very similar to Print. *channel* is the serial connection to send data through.

SerPrint will not send any new line characters. If the chip is sending to a terminal, these commands should follow every SerPrint:

SerSend *channel*, 13

SerSend *channel*, 10

Example:

```
'This program will display any values received over the serial
'connection. If "pot" is received, the value of the analog sensor
'will be sent.

'Chip settings
#chip 18F2525, 8
#config Osc = Int

'LCD settings
#define LCD_IO 4
#define LCD_RS PORTC.7
#define LCD_RW PORTC.6
#define LCD_Enable PORTC.5
#define LCD_DB4 PORTC.4
#define LCD_DB5 PORTC.3
#define LCD_DB6 PORTC.2
#define LCD_DB7 PORTC.1

'Serial settings
#define SerInPort PORTB.6
#define SerOutPort PORTB.7

#define SendAHigh Set SerOutPort Off
#define SendALow Set SerOutPort On
#define RecAHigh SerInPort Off
#define RecALow SerInPort On

'Potentiometer
#define POT_PORT PORTA.0
#define POT_AN AN0

'Set pin directions
```

```
     Dir SerOutPort Out
     Dir SerInPort In
     Dir POT_PORT In

'Create buffer variables to store received messages
     Dim Buffer As String
     Dim OldBuffer As String
     BufferSize = 0

'Set up serial connection
     InitSer 1, r9600, 1 + WaitForStart, 8, 1, none, invert

'Show test messages
     Print "Serial Tester"
     Wait 1 s
     SerPrint 1, "GCBASIC RS232 Test"
     SerSend 1, 13
     SerSend 1, 10
     Wait 1 s

'Main loop
     Do
         'Get a byte from the terminal
         SerReceive 1, Temp

         'If Enter key was pressed, deal with buffer contents
         If Temp = 13 Then
             Buffer(0) = BufferSize

             'Try to execute commands in buffer
             If Not ExecCommand (Buffer) Then
                 'Show message on bottom line, last message on top.
                 CLS
                 Print OldBuffer
                 Locate 1, 0
                 Print Buffer
                 'Store the message for next time
                 OldBuffer = Buffer
             End If

             BufferSize = 0
         End If
         'Backspace code, delete last character in buffer
         If Temp = 8 Then
             If BufferSize > 0 Then BufferSize -= 1
         End If
         'Received ASCII code between 32 and 127, add to buffer
         If Temp >= 32 And Temp <= 127 Then
             BufferSize += 1
             Buffer(BufferSize) = Temp
         End If
     Loop

'Takes a sensor reading and sends it to terminal
     Sub SendSensorReading
         SerPrint 1, "Sensor Reading: "
         SerPrint 1, ReadAD10(POT_AN)
         SerSend 1, 13
         SerSend 1, 10
```

```
    End Sub

'Will check the buffer for a command
    'If command found, run it and return true
    'If not, return false
    Function ExecCommand (CmdIn As String)
        ExecCommand = False
        'If received command is "pot", show potentiometer value
        If CmdIn = "pot" Then
            SendSensorReading
            ExecCommand = True
        End If
    End Function
```

For more help, see Relevant Constants

RS232 Hardware Overview

Introduction

These subroutines allow GCBASIC programs to communicate more easily using RS232.

These hardware-based routines are intended for use on microcontrollers with built in RS232 modules - normally referred to in datasheets as USART or UART modules. To use these, the RS232 data lines must be connected to the pins on the microcontroller used by the serial module. If the RS232 lines are connected elsewhere, or the microcontroller has no RS232 module, then the software based routines must be used.

Relevant Constants

These constants affect the operation of the hardware RS232 routines:

Constant Name	Controls	Default Value
USART_BAUD_RATE	Baud rate (in bps) for the routines to operate at.	N/A
USART_BLOCKING	If set, this constant will cause the USART routines to delay until data can be sent or received. If not set, then the data will be buffered and send by the hardware when possible.	Not set

HSerPrint

Syntax:

HSerPrint *value*

Command Availability:

Available on all microcontrollers with a USART or UART module.

Explanation:

HSerPrint is used to send a value over the serial connection. *value* can be a string, integers, long, word or byte. HSerPrint is very similar to Print. IThe channel used will be the hardware serial connection.

HSerPrint will not send any new line characters. If the chip is sending to a terminal, these commands should follow every HSerPrint :

HSerPrint 13
HSerPrint 10

Example:

```
     'This program will display any values received over the serial
     'connection. If "pot" is received, the value of the analog sensor
     'will be sent.
     'Note: This has been adapted from the SerPrint example.

'Chip settings
     #chip 18F2525, 8
     #config Osc = Int

'LCD settings
     #define LCD_IO 4
     #define LCD_RS PORTC.7
     #define LCD_RW PORTC.6
     #define LCD_Enable PORTC.5
     #define LCD_DB4 PORTC.4
     #define LCD_DB5 PORTC.3
     #define LCD_DB6 PORTC.2
     #define LCD_DB7 PORTC.1

'USART settings
     #define USART_BAUD_RATE 9600

'Potentiometer
     #define POT_PORT PORTA.0
     #define POT_AN AN0

'Set pin directions
     Dir SerOutPort Out
     Dir SerInPort In
     Dir POT_PORT In
```

```
'Create buffer variables to store received messages
     Dim Buffer As String
     Dim OldBuffer As String
     BufferSize = 0

'Show test messages
     Print "Serial Tester"
     Wait 1 s
     HSerPrint "GCBASIC RS232 Test"
     HSerSend 13
     HSerSend 10
     Wait 1 s

'Main loop
     Do
         'Get a byte from the terminal
         HSerReceive Temp

         'If Enter key was pressed, deal with buffer contents
         If Temp = 13 Then
             Buffer(0) = BufferSize

             'Try to execute commands in buffer
             If Not ExecCommand (Buffer) Then
                 'Show message on bottom line, last message on top.
                 CLS
                 Print OldBuffer
                 Locate 1, 0
                 Print Buffer
                 'Store the message for next time
                 OldBuffer = Buffer
             End If

             BufferSize = 0
         End If
         'Backspace code, delete last character in buffer
         If Temp = 8 Then
             If BufferSize > 0 Then BufferSize -= 1
         End If
         'Received ASCII code between 32 and 127, add to buffer
         If Temp >= 32 And Temp <= 127 Then
             BufferSize += 1
             Buffer(BufferSize) = Temp
         End If
     Loop

'Takes a sensor reading and sends it to terminal
     Sub SendSensorReading
         HSerPrint "Sensor Reading: "
         HSerPrint ReadAD10(POT_AN)
         HSerSend 13
         HSerSend 10
     End Sub

'Will check the buffer for a command
     'If command found, run it and return true
     'If not, return false
     Function ExecCommand (CmdIn As String)
         ExecCommand = False
         'If received command is "pot", show potentiometer value
```

```
        If CmdIn = "pot" Then
            SendSensorReading
            ExecCommand = True
        End If
    End Function



HserPrintByteCRLF
    HserPrintCRLF
```

For more help, see also HserPrintByteCRLF and HserPrintCRLF

HSerReceive

Syntax:

Used as subroutine:

HSerReceive *output*

Used as function:

output = HSerReceive

Command Availability:

Available on all microcontrollers with a USART or UART module.

Explanation:

This command will read a byte from the hardware RS232 module. It can be used either as a subroutine or as a function. If used as a subroutine, a variable must be supplied to store the received value in. If used as a function, it will return the received value.

Example:

```
'This program will read a value from the USART, and display it on PORTB.

#chip 16F877A, 20

'USART settings
     #define USART_BAUD_RATE 9600

'Set PORTB to input
     Dir PORTB Out
     'Set USART receive pin to input
     Dir PORTC.7 In

'Main loop
     Do
         'Get and display value
         'If there is no new data, HSerReceive will return old value.
         HSerReceive PORTB
         'Could also write:
         'PORTB = HSerReceive
     Loop
```

HSerSend

Syntax:

HSerSend *data*

Command Availability:

Available on all microcontrollers with a USART or UART module.

Explanation:

This command will send a byte given by *data* using the hardware RS232 module.

Example:

```
     'This program will send the status of PORTB through the hardware
     'serial module.

#chip 16F877A, 20

'USART settings
     #define USART_BAUD_RATE 9600

'Set PORTB to input
     Dir PORTB In
     'Set USART transmit pin to output
     Dir PORTC.6 Out

'Main loop
     Do
         'Send PORTB value through USART
         HSerSend PORTB
         'Short delay for receiver to process message
         Wait 10 ms
     Loop
```

HserPrintByteCRLF

Syntax:

HserPrintByteCRLF *data*

Command Availability:

Available on all microcontrollers with a USART or UART module.

Explanation:

This command will send a byte given by *data* using the hardware RS232 module and then send the ASCII codes 13 and 10. ASCII codes 13 and 10 equate to a carriage return and line feed.

Example:

```
'This program will send the status of PORTB through the hardware serial module.

HserPrintByteCRLF 65    ' Will print a single A on the terminal
HserPrintByteCRLF "A"   ' Will print a single A on the terminal
```

See also HserPrintCRLF

HserPrintCRLF

Syntax:

HserPrintCRLF

Command Availability:

Available on all microcontrollers with a USART or UART module.

Explanation:

This command will send ASCII codes 13 and 10 only using the hardware RS232 module. ASCII codes 13 and 10 equate to a carriage return and line feed.

Example:

```
'This program will send the status of PORTB through the hardware serial module.

HserPrintCRLF    ' Will send a CR & LF to the terminal
```

See also HserPrintByteCRLF

PS/2 Overview

Introduction

These routines make it easier to communicate with a PS/2 device, particularly a keyboard.

Relevant Constants

These constants affect the operation of the PS/2 routines:

Constant Name	Controls	Default Value
PS2Data	Pin connected to PS/2 data line	N/A
PS2Clock	Pin connected to PS/2 clock line.	N/A
PS2_DELAY	This constant can be set to a delay, such as 10 ms. If set, a delay will be added at the end of every byte sent or received.	Not set

INKEY

Syntax:

output = INKEY

Command Availability:

Available on all microcontrollers.

Explanation:

The INKEY function will read the last pressed key from a PS/2 keyboard, and return an ASCII value corresponding to the key. If no key is pressed, then INKEY will return 0.

It will also monitor Caps Lock, Num Lock and Scroll Lock keys, and update the status LEDs as appropriate.

Example:

```
      'A program to accept messages from a standard PS/2 keyboard
      'Any keys pressed will be shown on an LCD screen.

'Hardware settings
      #chip 18F4620, 20

'LCD connection settings
      #define LCD_IO 4
      #define LCD_DB4 PORTD.4
      #define LCD_DB5 PORTD.5
      #define LCD_DB6 PORTD.6
      #define LCD_DB7 PORTD.7
      #define LCD_RS PORTD.0
      #define LCD_RW PORTD.1
      #define LCD_Enable PORTD.2

'PS/2 connection settings
      #define PS2Clock PORTC.1
      #define PS2Data PORTC.0
      #define PS2_DELAY 10 ms

'Set up key log
      Dim KeyLog(32)
      DataCount = 0
      KeyLog(1) = 32

Main:
          'Read the last pressed key
          KeyIn = INKEY
          'If no key pressed, try reading again
          If KeyIn = 0 Then Goto Main

          'Escape pressed - clear message
          If KeyIn = 27 Then
              DataCount = 0
              For DataPos = 1 to 32
```

```
            KeyLog(DataPos) = 32
        Next
        Goto DisplayData
    End If

    'Backspace pressed - delete last character
    If KeyIn = 8 Then
        If DataCount = 0 Then Goto Main
        KeyLog(DataCount) = 32
        DataCount = DataCount - 1
        Goto DisplayData
    End If

    'Otherwise, add the character to the buffer
    If KeyIn >= 31 And KeyIn <= 127 Then
        DataCount = DataCount + 1
        KeyLog(DataCount) = KeyIn
    End If

DisplayData:
    'Display key buffer
    'LCDWriteChar is used instead of Print for greater control
    CLS
    For DataPos = 1 to DataCount
        If DataPos = 17 then Locate 1, 0
        LCDWriteChar KeyLog(DataPos)
    Next

Goto Main
```

PS2SetKBLeds

Syntax:

PS2SetKBLeds (*LedStatus*)

Command Availability:

Available on all microcontrollers.

Explanation:

This routine will turn the status LEDs on a keyboard on or off. *LedStatus* is a variable, of which the lower 3 bits correspond to the 3 LEDs. Bit 0 is for Scroll Lock, bit 1 controls Num Lock and bit 2 controls Caps Lock.

Note that this routine does not alter the status variables within the INKEY routine - so even if the Caps Lock LED is turned on, Caps Lock will stay off.

Example:

```
     'A spinning LED program for a keyboard
     'Will flash Num Lock, then Caps Lock, then Scroll Lock.


'Hardware settings
     #chip 16F88, 8


#define PS2Clock PORTB.2
     #define PS2Data PORTB.3
     #define PS2_DELAY 10 ms


'Main Loop
     Do

         'Turn on only Num Lock (bit 1)
         PS2SetKBLeds b'00000010'
         Wait 250 ms

         'Turn on only Caps Lock (bit 2)
         PS2SetKBLeds b'00000100'
         Wait 250 ms

         'Turn on only Scroll Lock (bit 0)
         PS2SetKBLeds b'00000001'
         Wait 250 ms

     Loop
```

PS2ReadByte

Syntax:

output = PS2ReadByte

Command Availability:

Available on all microcontrollers.

Explanation:

PS2ReadByte will read a byte from the PS/2 bus. It will return the byte, or 0 if no data was returned by the PS/2 device.

The PS/2 bus will normally be held in the inhibit state. PS2ReadByte will uninhibit the bus for 25 ms. If a response is received, it will be read. Then, the bus will be placed back in the inhibit state.

Example:

For an example, please refer to the INKEY function in the ps2.h file.

PS2WriteByte

Syntax:

PS2WriteByte *data*

Command Availability:

Available on all microcontrollers.

Explanation:

PS2WriteByte will send a byte to a PS/2 device. Once the byte has been written, the PS/2 bus will be placed in the inhibit state.

Example:

For an example, please refer to the PS2SetKBLeds function in the ps2.h file.

SPIMode

Syntax:

SPIMode *Mode*

Command Availability:

Available on PIC microcontrollers with Synchronous Serial Port (SSP) module.

Explanation:

SPIMode sets the mode of the SPI module within the PIC chip. These are the possible SPI Modes:

Mode Name	Description
MasterSlow	Master mode, SPI clock is 1/64 of the frequency of the PIC.
Master	Master mode, SPI clock is 1/16 of the frequency of the PIC.
MasterFast	Master mode, SPI clock is 1/4 of the frequency of the PIC.
Slave	Slave mode
SlaveSS	Slave mode, with the Slave Select pin enabled.

See Also SPITransfer

SPITransfer

Syntax:

SPITransfer *tx, rx*

Command Availability:

Available on PIC microcontrollers with Synchronous Serial Port (SSP) module.

Explanation:

This command simultaneously sends and receives a byte of data using the SPI protocol. It behaves differently depending on whether the PIC has been set to act as a master or a slave.

When operating as a master, SPITransfer will initiate a transfer. The data in *tx* will be sent to the slave, whilst the byte that is buffered in the slave will be read into *rx.*

In slave mode, the SPITransfer command will pause the program until a transfer is initiated by the master. At this point, it will send the data in *tx* whilst reading the transmission from the master into the *rx* variable.

Example:

There are two example programs for this command - one to run on the slave PIC, and one on the master. A reading is taken from a sensor on the slave, and sent across to the master which shows the data on its LCD screen.

Slave Program:

```
'Select chip model and configuration
#chip 16F88, 20
#config MCLR_OFF

'Set directions of SPI pins
dir PORTB.2 out
dir PORTB.1 in
dir PORTB.4 in
'Set direction of analogue pin
dir PORTA.0 in

'Set SPI mode to slave
SPIMode Slave

'Allow other PIC to initialise LCD
Wait 1 sec

'Main loop - takes a reading, and then waits to send it across.
do
'Note that rx is 0 - this is because no data is to be received.
SPITransfer ReadAD(AN0), 0
loop
```

Master Program:

```
        'General hardware configuration
        #chip 16F877A, 20

'LCD connection settings
        #define LCD_IO 8
        #define LCD_DATA_PORT PORTC
        #define LCD_RS PORTD.0
        #define LCD_RW PORTD.1
        #define LCD_Enable PORTD.2

'Set SPI pin directions
        dir PORTC.5 out
        dir PORTC.4 in
        dir PORTC.3 out

'Set SPI Mode to master, with fast clock
        SPIMode MasterFast

'Main Loop
        do
        'Read a byte from the slave
        'No data to send, so tx is 0
        SPITransfer 0, Temp

'Display data
        if Temp > 0 then
          CLS
          Print "Light: "
          LCDInt Temp
          Temp = 0
        end if

'Wait to allow time for the LCD to show the given value
        wait 100 ms
        loop
```

I2C Overview

Introduction:

These routines allow GCBASIC programs to send and receive I2C messages. They can be configured to act as master or slave, and the speed can also be altered.

No hardware I2C module is required for these routines - all communication is handled in software. However, these routines will not work on 12-bit instruction PICs (10F, 12F5xx and 16F5xx chips).

Relevant Constants:

These constants control the setup of the software I2C routines:

Constant	Controls	Default Value
I2C_MODE	Mode of I2C routines (Master or Slave)	Master
I2C_DATA	Pin on microcontroller connected to I2C data	N/A
I2C_CLOCK	Pin on microcontroller connected to I2C clock	N/A
I2C_BIT_DELAY	Time for a bit (used for acknowledge detection)	2 us
I2C_CLOCK_DELAY	Time for clock pulse to remain high	1 us
I2C_END_DELAY	Time between clock pulses	1 us
I2C_USE_TIMEOUT	Set to true if slave mode I2C routines should stop waiting for the master and exit after a timeout occurs.	Not Set
I2C_DISABLE_INTERRUPTS	Disable interrupts during I2C routines. Important when an i2C clock is part of your solution	Only available in GCB compiler post 1/2014

Example:

This example examines the IC2 devices and displays on a terminal. This code will require adaption but the code shows an approach to discover the IC2 devices.

```
  ' change the processor
  #chip 16F1825,32
  #config Osc = intOSC, MCLRE_OFF
  #include "..\outputserial.h"



' Define I2C settings - CHANGE PORTS
    #define I2C_MODE Master
    #define I2C_DATA PORTC.5
    #define I2C_CLOCK PORTC.4
    #define I2C_DISABLE_INTERRUPTS ON


' THIS CONFIG OF THE SERIAL PORT WORKS WITH max232 THEN TO PC
    #define SendAHigh Set PORTC.3 OfF
    #define SendALow Set PORTC.3 ON
    Dir PORTC.3 Out
    #define SerSendDelayms 10
    'Set up serial connection
    InitSer 1, r9600, 1+128, 8, 1, none, INVERT
    wait SerSendDelayms ms

ANSIERASECREEN_SW
    ANSI_SW ( 0,0)
    SerPrint 1,  "Started: "
    wait SerSendDelayms ms
    SerPrint 1,  "Searching I2C address space: v0.82"
    sersend 1, ( 10)
    sersend 1, ( 13)






dim deviceID as byte

    for deviceID = 0 to 255
          I2CStart
          I2CSend ( deviceID )
          i2cstop

     if I2CSendState = ACK  then
           SerPrint 1,  "__"
          SerPrint 1,  "ID : 0x"
          SerPrint 1,  hex(deviceID)
          SerPrint 1,  " (d"
          SerPrint 1,  Str(deviceID)
          SerPrint 1,  ") - "
          SerSend  1, 9
          SerPrint 1,  "I2C Port ": Sersend 1, I2CSendState:        sersend 1, (
  10):sersend 1, ( 13)
          end if
```

```
    next
    SerPrint 1,  "End of Device Search": sersend 1, ( 10):sersend 1, ( 13)
END
```

I2CACKPOLL

Syntax:

I2CAckpoll (I2C_device_address)

Command Availability:

Only available in GCB I2C.h with release after 1/2014

Available on all microcontrollers except 12 bit instruction PICs (10F, 12F5xx, 16F5xx chips)

Explanation:

Should only be used when I2C routines are operating in Master mode, this command will look for a specific I2C device on the I2C bus.

This set a global variable I2CAckPollState that can be inspected in your calling routine.

Example:

```
.....
' ACK polling removes the need to for the 24xxxxx device to have a 5ms write time
I2CACKPOLL( eeprom_device )
' You check the exit state, see I2CAckPollState
.....
```

I2CReceive

Syntax:

I2CReceive *data*

I2CReceive *data*, *ack*

Command Availability:

Available on all microcontrollers except 12 bit instruction PICs (10F, 12F5xx, 16F5xx chips)

Explanation:

The I2CReceive command will send *data* through the I2C connection. If *ack* is TRUE, or no value is given for *ack*, then I2CReceive will send an ack.

If in master mode, I2CReceive will read the data immediately. If in slave mode, I2CReceive will wait for the master to send the data before reading.

Example:

```
     'This program reads an I2C register and sets an LED if it is over 100.
     'It will read from address 83, register 1.

'Chip settings
     #chip 18F2620, 8

'I2C settings
     #define I2C_MODE Master
     #define I2C_DATA PORTC.0
     #define I2C_CLOCK PORTC.1

'Misc settings
     #define LED PORTB.0

'Main loop
     Do
          'Send start
          I2CStart

          'Request value
          I2CSend 83
          I2CSend 1

    'Read value
          I2CReceive ValueIn

          'Send stop
          I2CStop

          'Turn on LED if received value > 100
          Set LED Off
          If ValueIn > 100 Then Set LED On

          'Delay
```

```
    Wait 20 ms

Loop
```

I2CRESET

Syntax:

I2CReset

Command Availability:

Only available in GCB I2C.h with release after 1/2014

Available on all microcontrollers except 12 bit instruction PICs (10F, 12F5xx, 16F5xx chips)

Explanation:

This will attempt a reset of the I2C by changing the state of the I2C bus.

Example:

```
.....
I2CReset
.....
```

I2CRestart

Syntax:

I2CRestart

Command Availability:

Only available in GCB I2C.h with release after 1/2014

Available on all microcontrollers except 12 bit instruction PICs (10F, 12F5xx, 16F5xx chips)

Explanation:

If the I2C routines are operating in Master mode, this command will send a start and restart condition in a single command.

Example:

```
.....
    I2CRESTART
.....
```

I2CSend

Syntax:

I2CSend *data*

I2CSend *data*, *ack*

Command Availability:

Available on all microcontrollers except 12 bit instruction PICs (10F, 12F5xx, 16F5xx chips)

Explanation:

The I2CSend command will send *data* through the I2C connection. If *ack* is TRUE, or no value is given for *ack*, then I2CSend will wait for an Ack from the receiver before continuing.

If in master mode, I2CSend will send the data immediately. If in slave mode, I2CSend will wait for the master to request the data before sending.

Example:

```
     'This program will act as an I2C analog to digital converter
     'When data is requested from address 83, registers 0 through
     '3, it will return the value of AN0 through AN3.


'Chip model
     #chip 16F88, 8


'I2C settings
     #define I2C_MODE Slave
     #define I2C_CLOCK PORTB.0
     #define I2C_DATA PORTB.1


#define I2C_DISABLE_INTERRUPTS ON


'Main loop
     Do
         'Wait for start condition
         I2CStart


    'Get address
         I2CReceive Address
         If Address = 83 Then
             'If address was this device's address, respond
             I2CReceive Register

             OutValue = ReadAD(Register)
             I2CSend OutValue
         End If

         I2CStop

         Wait 5 ms
     Loop
```

I2CSlaveDeviceReceive

Syntax:

I2CSlaveDeviceReceive (this_devices_address, current_device_address_on_bus, [optional ACK|NACK])

Command Availability

Only available in GCB I2C.h with release after 1/2014

Available on all microcontrollers except 12 bit instruction PICs (10F, 12F5xx, 16F5xx chips)

Slave Mode only.

Explanation:

This instruction inspects the i2C and scans for I2C request for 'this' device. The instruction essentially ensure that the request on the i2c bus is for 'this' slave device.

This returns the current I2C device that is being addressed on the I2C bus.

Example:

```
' This example receives data from a Picaxe PIC only when it is 'this' device. 'This' device responds with the value of the AD port.
      '
      ' Reads a byte from the I2C bus, and this device will send an ACK if this device is
      being addressed.



' Assumes CHIP and SERIAL PORT etc. etc. are operational prior this code

' This device ID as a SLAVE @ 0x7e
      thisI2CSlavedevice = 0b01111110


  Counter = 0
       Do
                  OutValue = ReadAD(0 )

             'Wait for start condition
                  I2CStart
                  ' Ensures the request is for this device
                   I2CSlaveDeviceReceive thisI2CSlavedevice, readDeviceAddress, TRUE
                  if readDeviceAddress = thisI2CSlavedevice then
                       counter++
                       I2CReceive PicaxeTimeinSeconds, TRUE
                       I2CStart
                       I2CReceive Read_DeviceAddress, TRUE

                  I2CSend (readDeviceAddress, TRUE)
                       ' Send back the data received
                       I2CSend (PicaxeTimeinSeconds, TRUE)
```

```
                    ' Send the read addres
                    I2CSend (TimeinSeconds, TRUE)
                    ' Send a counter
                    I2CSend (Counter, TRUE)
                    ' Send AD value
                    I2CSend (OutValue, TRUE)
                End if
                I2CStop
                SerPrint 1, "Time in Seconds Received "
                SerPrint 1, str(TimeinSeconds)
                SerSend  1, 9
                SerPrint 1, "Pot Value "
                SerPrint 1, str(OutValue)
            SerPrint 1, 13
            SerPrint 1, 10
    Loop
```

.....

For more help, see I2CReceive

I2CStart

Syntax:

I2CStart

Command Availability:

Available on all microcontrollers except 12 bit instruction PICs (10F, 12F5xx, 16F5xx chips)

Explanation:

If the I2C routines are operating in Master mode, this command will send a start condition. If routines are in Slave mode, it will pause the program until a start condition is sent by the master. It should be placed at the start of every I2C transmission.

If interrupt handling is enabled, this command will disable it.

Example:

Please see I2CSend and I2CReceive for an example.

I2CStartOccurred

Syntax:

I2CStartOccurred

Command Availability:

Available on all microcontrollers except 12 bit instruction PICs (10F, 12F5xx, 16F5xx chips)

Explanation:

Check if a start condition has occurred since the last run of this function

Only used in slave mode.

If interrupt handling is enabled, this command will disable it.

Example:

Please see I2CSend and I2CReceive for an example.

I2CStop

Syntax:

I2CStop

Command Availability:

Available on all microcontrollers except 12 bit instruction PICs (10F, 12F5xx, 16F5xx chips)

Explanation:

When in Master mode, this command will send an I2C stop condition, and re-enable interrupts if I2CStart disabled them. In Slave mode, it will re- enable interrupts. I2CStop should be called at the end of every I2C transmission.

Example:

Please see I2CSend and I2CReceive for an example.

Overview

Introduction:

These routines generate tones of a given frequency and duration.

Note: If an exact frequency is required, or a smaller program is needed, these routines should not be used. Instead, you should use code like this:

```
Repeat count
         PulseOut SoundOut, period us
         Wait period us
     End Repeat
```

Set *count* and *period* to the appropriate values:

- *period* to 1000000 / desired frequency / 2
- *count* to desired duration / *period*.

Relevant Constants:

These constants are used to control settings for the tone generation routines. To set them, place a line in the main program file that uses #define to assign a value to the particular constant.

Constant Name	Controls	Default Value
SoundOut	The output pin to produce sound on	N/A - ***Must be set***

Tone

Syntax:

Tone *Frequency, Duration*

Command Availability:

Available on all microcontrollers.

Explanation:

This command will produce the specified tone for the specified duration. *Frequency* is measured in Hz, and *Duration* is in 10 ms units.

Please note that this command may not produce the exact frequency specified. While it is accurate enough for error beeps and small pieces of monophonic music, it should not be used for anything that requires a highly precise frequency.

Example:

```
'Sample program to produce a constant A note (440 Hz)
'on PORTB bit 1.
#chip 16F877A, 20
#define SoundOut PORTB.1


Do
    Tone 440, 1000
Loop
```

For more help, see Relevant Constants

ShortTone

Syntax:

ShortTone Frequency, Duration

Command Availability:

Available on all microcontrollers.

Explanation:

This command will produce the specified tone for the specified duration. Frequency is measured in units of 10 Hz, and Duration is in 1 ms units.

Please note that this command may not produce the exact frequency specified. While it is accurate enough for error beeps and small pieces of monophonic music, it should not be used for anything that requires a highly precise frequency.

Example:

```
'Sample program to produce a tone on PORTB bit 1, based on the
'reading of an LDR on AN0 (usually PORTA bit 0).

#chip 16F88, 20
    #define SoundOut PORTB.1

Dir PORTA.0 In

Do
        ShortTone ReadAD(AN0), 100
    Loop
```

For more help, see Relevant Constants

Timer Overview

Several functions are provided to read the current timer value. They are:

- Timer0
- Timer1
- Timer2
- Timer3
- Timer4
- Timer5

Not all of these functions are available on all chips. For example, if a chip only has 3 timers, then only Timer0, Timer1 and Timer2 will be available. Timer0 and Timer2 return byte values, while Timer1, Timer3, Timer4 and Timer5 will return words. Please refer to the datasheet for your microcontroller to determine the number and size of the timers available.

InitTimer0

Syntax:

InitTimer0 *source, prescaler*

Command Availability:

Available on all microcontrollers with a Timer 0 module.

Explanation:

InitTimer0 will set up timer 0, according to the settings given. *source* can be Osc or Ext.

On a PIC microcontroller, *prescaler* can be one of the following settings:

- PS0_2
- PS0_4
- PS0_8
- PS0_16
- PS0_32
- PS0_64
- PS0_128
- PS0_256

These correspond to a prescaler of between 1/2 and 1/256 the oscillator speed. The prescaler will apply to either the oscillator or the external clock input.

On an AVR microcontroller, *prescaler* can be one of these settings:

- PS_1
- PS_8
- PS_64
- PS_256
- PS_1024

These correspond to a prescaler of between 1/1 and 1/1024 times the original input frequency. On the AVR, the prescaler will only apply when the timer is driven from the internal oscillator - the prescaler has no effect on the external clock pulse.

When the timer overflows from 255 to 0, a Timer0Overflow interrupt will be generated. This can be used in conjunction with On Interrupt to run a section of code periodically.

Example:

```
      'This code will use Timer 0 and On Interrupt to generate a Pulse
      'Width Modulation signal, that will allow the speed of a motor to
      'be easily controlled.

#chip 16F88, 8
      #config osc = int

#define MOTOR PORTB.0

'Call the initialisation routine
```

```
     InitMotorControl

'Main routine
     Do
         'Increase speed to full over 2.5 seconds
         For Speed = 0 to 100
             MotorSpeed = Speed
             Wait 25 ms
         Next
         'Hold speed
         Wait 1 s
         'Decrease speed to zero over 2.5 seconds
         For Speed = 100 to 0
             MotorSpeed = Speed
             Wait 25 ms
         Next
         'Hold speed
         Wait 1 s
     Loop

'Setup routine
     Sub InitMotorControl
         'Clear variables
         MotorSpeed = 0
         PWMCounter = 0

         'Add a handler for the interrupt
         On Interrupt Timer0Overflow Call PWMHandler

         'Set up the timer
         InitTimer0 Osc, PS0_1/2
         'Timer 0 starts automatically on a PIC
     End Sub

'PWM sub
     'This will be called when Timer 0 overflows
     Sub PWMHandler
         If MotorSpeed > PWMCounter Then
             Set MOTOR On
         Else
             Set MOTOR Off
         End If
         PWMCounter += 1
         If PWMCounter = 100 Then PWMCounter = 0
     End Sub
```

InitTimer1

Syntax:

InitTimer1 *source, prescaler*

Command Availability:

Available on all microcontrollers with a Timer 1 module.

Explanation:

InitTimer1 will set up timer 1, according to the settings given. *source* can be Osc , Ext or ExtOsc (ExtOsc is only available on PIC microcontrollers). On a PIC, *prescaler* can be one of the following settings:

- PS1_1
- PS1_2
- PS1_4
- PS1_8

On an AVR chip, the following options are available for *prescaler*:

- PS_1
- PS_8
- PS_64
- PS_256
- PS_1024

Example:

```
      'This example will measure that time that a switch stays on for
      #chip 16F819, 20
      #define Switch PORTA.0

Dir Switch In
      DataCount = 0
      InitTimer Osc, PS1_8

Dim TimerValue As Word

Do
          ClearTimer 1
          Wait Until Switch = On
          StartTimer 1
          Wait Until Switch = Off
          StopTimer 1

          'Read the timer
          TimerValue = Timer1

          'Log the timer value
          EPWrite(DataCount, TimerValue_H)
          EPWrite(DataCount + 1, TimerValue)
          DataCount += 2
      Loop
```

StartTimer

Syntax:

StartTimer *TimerNo*

Command Availability:

Available on all PIC and AVR microcontrollers with built in timer modules.

Explanation:

StartTimer is used to start the specified timer. Note that it cannot be used to start Timer 0 on a PIC - this timer always runs.

Example:

Please refer to the InitTimer1 article for an example.

ClearTimer

Syntax:

ClearTimer *TimerNo*

Command Availability:

Available on all PIC and AVR microcontrollers with built in timer modules.

Explanation:

ClearTimer is used to clear the specified timer.

Example:

Please refer to the InitTimer1 article for an example.

StopTimer

Syntax:

StopTimer *TimerNo*

Command Availability:

Available on all PIC and AVR microcontrollers with built in timer modules.

Explanation:

StopTimer is used to stop the specified timer. Note that it cannot be used to stop Timer 0 on a PIC - this timer always runs.

Example:

Please refer to the InitTimer1 article for an example.

Reading Timers

Several functions are provided to read the current timer value. They are:

- Timer0
- Timer1
- Timer2
- Timer3
- Timer4
- Timer5

Not all of these functions are available on all chips. For example, if a chip only has 3 timers, then only Timer0, Timer1 and Timer2 will be available. Timer0 and Timer2 return byte values, while Timer1, Timer3, Timer4 and Timer5 will return words.

Please refer to the datasheet for your microcontroller to determine the number and size of the timers available.

Using Variables

Variables

Using and accessing bytes within word and long numbers etc may be required when you are creating your solution. This can be done with some ease.

You can access the bytes within word and longs variables using the following as a guide using the Suffixes _H, _U and _E

```
              Dim workvariable as word
              workvariable = 21845
              Dim lowb as byte
              Dim highb as byte
              Dim upperb as byte
              Dim lastb as byte

lowb      = workvariable
              highb     = workvariable_H
              upperb    = workvariable_U
              lastb     = workvariable_E
```

To further explain, where

```
              Dim rB as Byte
              Dim sW as Word
              Dim sW as Word
```

To extract the bytes from a WORD of 16 bits use the Suffix _H

```
   'To use the bits 7-0 [lower byte] in the Word variable sW
              rB = sW

   'For bits 15-8 [upper byte] in the Word variable sW, use sw_H
              rB = sW_H
```

To extract the bytes from a LONG of 32 bits use the Suffixes _H, _U and _E, where

```
              Dim rB as Byte
              Dim sW as Word
              Dim tL as Long

    ' For bits 7-0 [lowest byte #0] in Long variable tL
              rB = tL

    ' For bits 15-8 [lower middle byte #1] in Long variable tL
              rB = tL_H

    ' For bits 23-16 [upper middle byte #2] in Long variable tL
              rB = tL_U

    ' For bits 31-24 [highest byte #3] in Long variable tL
```

```
rB = tL_E
```

To extract nibbles from the variable rB

```
lower_nibble = rB & 0x0F
upper_nibble = (rB & 0xF0) / 16
```

More on setting Variables and Constants

Variables

Within Great Cow Basic you can use regular variable assignments. But, you can also use C like maths assignments.

The following methods are also supported.

```
GLCDPrintLoc += 6
CharCode -= 15
CharCode++
CharCode---
```

Within Great Cow Basic you can define constants, see Constants. Please note what is and what is not support with respect to assigning numbers to constants. An example program examines what is supported.

```
  #chip 16F88, 4
  #config Osc = INT, MCLRE_OFF

' All these work
      #define Test0 b'11111111'
      #define Test1 0b11111111
      #define Test2 0B11111111
      #define Test3 255
      #define Test4 0xFF
      #define Test5 0xff
      #define Test6 0Xff

# Proof
    dir porta Out

  porta = test0
      porta = test1
      porta = test2
      porta = test3
      porta = test4
      porta = test5
      porta = test6
```

You can assigned values/numbers with all the methods shown above (for constants and variables) but please be aware that you must Use '0' not '00'. One zero equates to zero and two zeros will give you an unassigned variable.

Constants

A few critical constants are defined within Great Cow Basic , you can use this constants. They include:

```
#define ON 1
#define OFF 0
#define TRUE 255
#define FALSE 0
```

Setting Variables

Syntax:

Variable = *data*

Explanation:

Variable will be set to *data*. *Data* can be either a fixed value (such as 157), another variable, or a sum.

If *data* is a fixed value, it must be an integer between 0 and 255 inclusive.

If *data* is a calculation, then it may have any of the following operands:

- + (add)
- - (subtract, or negate if there is no value before it)
- * (multiply)
- / (divide)
- % (modulo)
- & (and)
- | (or)
- # (xor)
- ! (not)
- = (equal)
- <> (not equal)
- < (less than)
- > (greater than)
- <= (less than or equal)
- >= (more than or equal)

The final 6 operands are for checking conditions. They will return FALSE (0) if the condition is false, or TRUE (255) if the condition is true.

The And, Or, Xor and Not operators function both as bitwise and logical operators.

GCBASIC understands order of operations. If multiple operands are present, they will be processed in this order:

1. Brackets
2. Unary operations (not and negate)
3. Multiply/Divide/Modulo
4. Add/Subtract
5. Conditional operators
6. And/Or/Xor

There are several modes in which variables can be set. GCBASIC will automatically use a different mode for each calculation, depending on the type of variable being set. If a byte

variable is being set, byte mode will be used; if a word variable is being set, word mode will be used. If a byte is being set but the calculation involves numbers larger than 255, word mode can be used by adding [WORD] to the start of one of the values in the calculation. This is known as casting - refer to the Variables article for more information.

If you prefer, you can add "LET" to the start of the line. It will not alter the execution of the program, but is included for those who are used to including it in other BASIC dialects.

Example:

```
      'This program is to illustrate the setting of variables.
      Chipmunk = 46     'Sets the variable Chipmunk to 46
      Animal = Chipmunk    'Sets the variable Animal to the value of the variable
      Chipmunk
      Bear = 2 + 3 * 5        'Sets the variable Bear to the result of 2 + 3 * 5, 17.
      Sheep = (2 + 3) * 5    'Sets the variable Sheep to the result of (2 + 3) * 5, 25.
      Animal = 2 * Bear        'Sets the variable Animal to twice the value of Bear.


LargeVar = 321        'LargeVar must be set as a word - see DIM.
      Temp = LargeVar / [WORD]5 'Note the use of [WORD] to ensure that the calculation is
      performed correctly
```

Dim

Dim has two uses, both of which involve large variables. It can be used to define arrays, and to declare variables larger than 1 byte.

Syntax:

For Variables > 1 byte:

Dim *variable* [*, variable2*[*, variable3*]] [As *type*] [Alias *othervar* [,*othervar2*]] [At *location*]

For Arrays:

Dim *array*(*size*) [At *location*]

Command Availability:

Available on all microcontrollers.

Explanation:

The Dim variable command is used to inform GCBASIC of variables that are larger than 1 byte, or to create alternate names for other variables.

The Dim array command also sets up array variables. When using it for this, *size* must be a constant value up to 80.

type specifies the type of variable that is to be created. Different variable types can hold values over different ranges, and use different amounts of RAM. See the Variables article for more information.

When multiple variables are included on the one line, GCBASIC will set them all to the type that is specified at the end of the line. If there is no type specified, then GCBASIC will make the variable a byte.

Alias creates a variable using the same memory location as one or more other variables. It is mainly used internally in GCBASIC to treat system variables as a word. For example, this command is used to create a word variable, made up from the two memory locations used to store the result of an A/D conversion:

A variable can be placed at a specific location in the data memory of the chip using the At option. *location* will be used whether it is a valid location or not, but a warning will be generated if GCBASIC has already allocated the memory, or if the memory does not appear to be valid. This can be used for peripherals that have multi byte buffers in RAM.

Dim ADResult As Word Alias ADRESH, ADRESL

Example:

```
'This program will set up an array, and a word variable
```

```
dim DataList(10)
      dim Reading as word

Reading = 21978
      DataList(1) = 15

dim stringvariable as string
```

For more help, see:

The SerPrint article uses Dim to create string variables and Variables for more details in creating and managing strings lengths.

BcdToDec_GCB

Syntax:

BcdToDec_GCB (ByteVariable)

Command Availability:

Only available in GCB compiler post 1/2014

Available on all microcontrollers.

Support Bytes only.

Explanation:

Converts numbers from Binary Coded Decimal format to decimal.

GCB from Feb 2014 has this function built.

But, you can easily add it with this function below. Just add this to your GCB program and then call it when you need it.

Example:

```
Function BcdToDec(va) as byte
    BcdToDec=(va/16)*10+va%16
End Function
```

Also see DecToBcd_GCB

DecToBcd_GCB

Syntax:

DectoBcd_GCB (ByteVariable)

Command Availability:

Only available in GCB compiler post 1/2014

Available on all microcontrollers.

Support Bytes only.

Explanation:

Converts numbers from Decimal to Binary Coded Decimal format.

GCB from Feb 2014 has this function built.

But, you can easily add it with this function below. Just add this to your GCB program and then call it when you need it.

Example:

```
Function DecToBcd(va) as Byte
     DecToBcd=(va/10)*16+va%10
End Function
```

Also se BcdToDec_GCB

Rotate

Syntax:

Rotate *variable* {Left | Right} [Simple]

Command Availability:

Available on all microcontrollers.

Explanation:

The Rotate command will rotate *variable* one bit in a specified direction. The bit shifted will be placed in the Carry bit of the Status register (STATUS.C). STATUS.C acts as a ninth bit of the variable that is being rotated.

variable supports Bytes, Word and Long variables.

When a variable is **rotated right**, the bit in the STATUS.C location is placed into the MSB of the variable being rotated, and the LSB of the variable is placed into STATUS.C location.

When **rotated left** the opposite occurs. The MSB of the variable is shifted to the STATUS.C bit and the LSB of the variable will contain what was previously in the STATUS.C bit location.

This table shows the operation of the Rotate Left command

Command	***variable***	**STATUS.C**
Values at start:	b'01110011'	0
Rotate Left	b'11100110'	0
Rotate Left again	b'11001100'	1
Rotate Left third time	b'10011001'	1

As you may notice the STATUS.C bit added a 0 to the rotation. So this will take 9 shifts left to get back to the original value.

SIMPLE option

Many times you want to rotate the variable around like the STATUS.C bit wasn't there so the MSB of the variable fills the LSB of the variable on Rotate Left or the LSB fills the MSB on Rotate Right. That is where the SIMPLE option comes in. It adds a hidden step that shifts the STATUS.C bit twice so the bit moves from one end of the variable to the other.

Command	***variable***	**STATUS.C**
Values at start:	b'01110011'	0
Rotate Left	b'11100110'	0
Rotate Left again	b'11001101'	1
Rotate Left	b'10011011'	1

	third time	

(*) Known as SREG bit C, or simply C flag on AVR.

Example:

```
      'This program will use Rotate to show a chasing LED.
      '8 LEDs should be connected to PORTB, one on each pin.

#chip 16F819, 8
      #config osc = int

'Set port direction
      Dir PORTB Out

'Set initial state of port (bits 0 and 4 on)
      PORTB = b'00010001'

'Chase
      Do
          Rotate PORTB Right Simple
          Wait 250 ms
      Loop
```

Set

Syntax:

Set *variable.bit* {On | Off}

Command Availability:

Available on all microcontrollers.

Explanation:

The purpose of the Set command is to turn individuals bits on and off. The Set command is most useful for controlling output ports, but can also be used to set variables.

Often when controlling output ports, Set is used in conjunction with constants. This makes it easier to adapt the program for a new circuit later.

Example:

```
     'Blink LED sample program for GCBASIC
     'Controls an LED on PORTB bit 0.

'Set chip model and config options
     #chip 16F84A, 20

'Set a constant to represent the output port
     #define LED PORTB.0

'Set pin direction
     Dir LED Out

'Main routine
     Do
         Set LED On
         Wait 1 sec
         Set LED OFF
         Wait 1 sec
     Loop
```

SWAP4

Syntax:

SWAP4(VariableA)

Command Availability:

Available on all microcontrollers.

Support Bytes only.

Explanation:

A function that swaps (or exchanges) nibbles (or the 8 bits of a byte in nibbles).

Example:

```
      dim ByteVariable as Byte

' Set variable to  0x12
      ByteVariable = 0x12

ByteVariable = Swap4( ByteVariable )

HSerPrint hex(ByteVariable)

' Would return 0x21
```

SWAP

Syntax:

SWAP(VariableA, VariableB)

Command Availability:

Available on all microcontrollers.

Support Bytes and Words only.

Explanation:

A function that swaps (or exchanges) one byte or word for another. SWAP support the use of byte and word variables.

Asc

Syntax:

byte*var* = ASC(string, [position])

Command Availability:

Available on all microcontrollers

Only available in GCB I2C.h with release after 1/2014

Explanation:

Only available in GCB compiler post 1/2014

Returns the character code of the character at the specified position in a string.

ASC returns the character code of a particular character in the string. If the string is an ANSI string, the returned value will be in the range of 0 to 255. This function DOES NOT support UNICODE.

The optional position parameter determines which character is to be checked. The first character is one, the second two, etc. If the position parameter is missing, the first character is presumed.

CHR is the natural complement of ASC. CHR produces a one-character string corresponding to its ASCII

Restrictions:

If the string passed is null (zero-length) or the position is zero or greater than the length of the string the returned value will be 0.

Example:

```
charpos = ASC( "ABCD" ): ' Returns 65

charpos = ASC( "ABCD", 2 ): ' Returns 66
```

For more help, see Chr

ByteToBin

Syntax:

stringvar = BYTETOBIN(*bytevar*)

Command Availability:

Only available in GCB compiler post 1/2014

Available on all microcontrollers

Explanation:

The ByteToBin function creates a string of a ANSI (8-byte) characters. The function converts a number to a string consisting of ones and zeros that represents the binary value.

Supports BYTE variables only. For WORD variables use WordToBin

Example:

```
string = ByteToBin( 1 ): ' Returns "00000001"

string = ByteToBin( 254 ): ' Returns "11111110"
```

For more help, see WordToBin

CHR

Syntax:

stringvar = CHR(*bytevar*)

Command Availability:

Only available in GCB compiler post 1/2014

Available on all microcontrollers

Explanation:

The CHR function creates a string of a ANSI (1-byte) character.

ASC is the natural complement of CHR.

Example:

```
string = CHR( 65 ): ' Returns "A"

string = CHR( 66 ): ' Returns "B"
```

For more help, see Asc

Hex

Syntax:

stringvar = Hex(*number*)

Command Availability:

Available on all microcontrollers

Explanation:

The Hex function will convert a number into hexadecimal format. The input *number* should be a byte variable, or a fixed number between 0 and 255 inclusive. After running the function, the string variable *stringvar* will contain a 2 digit hexadecimal number.

Example:

```
     'Set chip model
     #chip 16F1936

'Set up hardware serial connection
     #define USART_BAUD_RATE 9600
     #define USART_BLOCKING

'Send EEPROM data over serial connection
     'Uses Hex to display as hexadecimal
     For CurrentLocation = 0 to 255
         'Send location
         HSerPrint Hex(CurrentLocation)
         HSerPrint ":"
         'Read byte and send
         EPRead CurrentLocation, CurrByte
         HSerPrint Hex(CurrByte)
         'Send carriage return/line feed
         HSerPrintCRLF
     Next
```

When using the functions HEX() do not leave space between the function call and the left brace. You will get a compiler error that is meaningless.

```
    ' use this, note this is no space between the HEX and the left brace!
        HEX(number_variable)
        ' do not use, note the space!
        HEX (number_variable)
```

See Also Str. Val

Instr

Syntax:

location = Instr(*source*, *find*)

Command Availability:

Available on all microcontrollers

Explanation:

The Instr function will search one string to find the location of another string within it. *source* is the string to search inside, and *find* is the string to find. The function will return the location of *find* within *source,* or 0 if *source* does not contain *find*.

Example:

```
     'Set chip model
     #chip 16F1936

'Set up hardware serial connection
     #define USART_BAUD_RATE 9600
     #define USART_BLOCKING

'Fill a string with a message
     Dim TestData As String
     TestData = "Hello, world!"

'Display the location of "world" within the string
     'Will return 8, because "w" in world is the 8th character
     'of "Hello, world!"
     HSerPrint Instr(TestData, "world")
     HSerPrintCRLF

'Display the location of "planet" within the string
     'Will display 0, because "planet" does not occur inside
     'the string "Hello, world!"
     HSerPrint Instr(TestData, "planet")
     HSerPrintCRLF
```

LCase

Syntax:

output = LCase(*source*)

Command Availability:

Available on all microcontrollers

Explanation:

The LCase function will convert all of the letters in the string *source* to lower case, and return the result.

Example:

```
     'Set chip model
     #chip 16F1936


'Set up hardware serial connection
     #define USART_BAUD_RATE 9600
     #define USART_BLOCKING


'Fill a string with a message
     Dim TestData As String
     TestData = "Hello, world!"


'Display the string in lower case
     'Will display "hello, world!"
     HSerPrint LCase(TestData)
     HSerPrintCRLF
```

See Also UCase

Left

Syntax:

output = Left(*source, count*)

Command Availability:

Available on all microcontrollers

Explanation:

The Left function will extract the leftmost *count* characters from the input string *source*, and return them in a new string.

Example:

```
      'Set chip model
      #chip 16F1936


'Set up hardware serial connection
      #define USART_BAUD_RATE 9600
      #define USART_BLOCKING


'Fill a string with a message
      Dim TestData As String
      TestData = "Hello, world!"


'Display the leftmost 5 characters
      'Will display "Hello"
      HSerPrint Left(TestData, 5)
      HSerPrintCRLF
```

See Also Mid, Right

Len

Syntax:

output = Len(string)

Command Availability:

Only available in GCB compiler post 1/2014

Available on all microcontrollers

Explanation:

The LEN function returns an byte value which is the length of a phrase or a sentence, including the empty spaces. The format is:

```
target_byte_variable = Len("Phrase")
```

or another example. This code will loop through the for-next loop 12 times as determined by the length of the string:

```
' create a test string of 12 characters

dim teststring as string * 12

teststring = "0123456789AB"
        for loopthrustring = 1 to len(teststring)
        hserprint mid(teststring, loopthrustring , 1)
        next
```

Ltrim

Syntax:

stringvar = LTRIM(*stringvar*)

Command Availability:

Only available in GCB compiler post 1/2014

Available on all microcontrollers

Explanation:

The LTRIM function will trim the 7-bit ASCII space character (value 32) from the LEFT hand side of a string.

Use LTRIM on text that you have received from another source that may have irregular spacing at the left hand end of the string.

See Also Trim, Rtrim

Mid

Syntax:

output = Mid(*source, start, count*)

Command Availability:

Available on all microcontrollers

Explanation:

The Mid function is used to extract characters from the middle of a string variable. *source* is the variable to extract from, *start* is the position of the first character to extract, and *count* is the number of characters to extract. If *count* is not specified, all characters from *start* to the end of the source string will be returned.

Example:

```
'Set chip model
#chip 16F1936

'Set up hardware serial connection
     #define USART_BAUD_RATE 9600
     #define USART_BLOCKING

'Fill a string with a message
     Dim TestData As String
     TestData = "The cat sat on the mat"

'Extract "cat". The c is at position 5, and 3 letters are needed
     HSerPrint "The animal is a "
     HSerPrint Mid(TestData, 5, 3)

'Extract the action. "sat" starts at position 9.
     HSerPrint "The animal "
     HSerPrint Mid(TestData, 9)
     HSerPrintCRLF
```

See Also Left, Right

Right

Syntax:

output = Right(*source, count*)

Command Availability:

Available on all microcontrollers

Explanation:

The Right function will extract the rightmost *count* characters from the input string *source*, and return them in a new string.

Example:

```
      'Set chip model
      #chip 16F1936


'Set up hardware serial connection
      #define USART_BAUD_RATE 9600
      #define USART_BLOCKING


'Fill a string with a message
      Dim TestData As String
      TestData = "Hello, world!"


'Display the rightmost 6 characters
      'Will display "world!"
      HSerPrint Right(TestData, 6)
      HSerPrintCRLF
```

See Also Left, Mid

Rtrim

Syntax:

stringvar = RTRIM(*stringvar*)

Command Availability:

Only available in GCB compiler post 1/2014
Available on all microcontrollers

Explanation:

The RTRIM function will trim the 7-bit ASCII space character (value 32) from the RIGHT hand side of a string.

Use RTRIM on text that you have received from another source that may have irregular spacing at the right hand end of the string.

See Also Trim, Ltrim

Str

Syntax:

stringvar = Str(*number*)

Command Availability:

Available on all microcontrollers

Explanation:

The Str function will convert a number into string. *number* can be any byte or word variable, or a fixed number between 0 and 65535 inclusive. The string variable *stringvar* will contain the same number, represented as a string.

This function is especially useful if a number needs to added to the end of a string, or if a custom data sending routine has been created but only supports the output of string variables.

Example:

```
      'Set chip model
      #chip 16F1936

'Set up hardware serial connection
      #define USART_BAUD_RATE 9600
      #define USART_BLOCKING

'Take an A/D reading
      SensorReading = ReadAD(AN0)

'Create a string variable
      Dim OutVar As String

'Fill string with sensor reading
      OutVar = Str(SensorReading)

'Send
      HSerPrint OutVar
      HSerPrintCRLF
```

When using the functions STR() do not leave space between the function call and the left brace. You will get a compiler error that is meaningless.

```
    ' use this, note this is no space between the STR and the left brace!
        STR(number_variable)
        ' do not use, note the space!
        STR (number_variable)
```

See Also Hex, Val

Trim

Syntax:

stringvar = TRIM(*stringvar*)

Command Availability:

Only available in GCB compiler post 1/2014

Available on all microcontrollers

Explanation:

The TRIM function will trim the 7-bit ASCII space character (value 32) from text.

TRIM removes all spaces from text except for single spaces between words. Use TRIM on text that you have received from another source that may have irregular spacing at the left or right hand ends of the string.

See Also Ltrim, Rtrim

UCase

Syntax:

output = UCase(*source*)

Command Availability:

Available on all microcontrollers

Explanation:

The UCase function will convert all of the letters in the string *source* to upper case, and return the result.

Example:

```
     'Set chip model
     #chip 16F1936


'Set up hardware serial connection
     #define USART_BAUD_RATE 9600
     #define USART_BLOCKING


'Fill a string with a message
     Dim TestData As String
     TestData = "Hello, world!"


'Display the string in upper case
     'Will display "HELLO, WORLD!"
     HSerPrint UCase(TestData)
     HSerPrintCRLF
```

See Also LCase

Val

Syntax:

var = Val(*string*)

Command Availability:

Available on all microcontrollers

Explanation:

The Val function will extract a number from a string variable, and store it in a word variable. One potential use is reading numbers that are sent in ASCII format over a serial connection.

Example:

```
      'Program for an RS232 controlled dimmer
      'Set chip model
      #chip 16F1936


'Set up hardware serial connection
      #define USART_BAUD_RATE 9600
      #define USART_BLOCKING


'Set pin directions for USART and PWM


'Variable for output level
      Dim OutputLevel As Word


'Variables for received bytes
      Dim DataIn As String
      DataInCount = 0


'Main Loop
      Do
          'Get serial byte
          Wait Until USARTHasData
          HSerReceive InByte

          'Process latest byte
          'Enter key?
          If InByte = 13 Then
              'Convert output level to numeric variable
              OutputLevel = Val(DataIn)

              'Output
              HPWM 1, 32, OutputLevel


         'Clear output buffer for next command
              DataIn = ""
              DataInCount = 0
          End If


     'Number?
          If InByte >= 48 and InByte <= 57 Then
```

```
        'Add to end of DataIn string
        DataInCount += 1
        DataIn(DataInCount) = InByte
        DataIn(0) = DataInCount
    End If
Loop
```

See Also Hex, Str

WORDTOBIN

Syntax:

stringvar = WORDTOBIN(*bytevar*)

Command Availability:

Only available in GCB compiler post 1/2014
Available on all microcontrollers

Explanation:

The ByteToBin function creates a string of a ANSI (8-byte) characters. The function converts a number to a string consisting of ones and zeros that represents the binary value.

Example:

```
string = WordToBin( 1 ): ' Returns "0000000000000001"

string = WordToBin( 37654 ): ' Returns "1001001100010110"
```

For more help, see ByteToBin

Dir

Syntax:

Dir *port.bit* {In | Out} ***(Individual Form)***

Dir *port* {In | Out | *DirectionByte*} ***(Entire Port Form)***

Command Availability:

Available on all microcontrollers. However, some low end PIC microcontrollers (10F, 12F5x and 16F5x chips) will only accept the entire port form.

Explanation:

The Dir command is used to set the direction of the ports of the microcontroller chip. The individual form sets the direction of one pin at a time, whereas the entire port form will set all bits in a port.

In the individual form, specify the port and bit (ie. PORTB.4), then the direction, which is either In or Out.

The entire port form is similiar to the TRIS instruction offered by some PIC chips. To use it, give the name of the port (ie. PORTA), and then a byte is to be written into the TRIS variable. *This form of the command is for those who are familiar with the PIC chip's internal architecture.*

WARNING: Entire port form will work differently on AVR when a value other than IN or OUT is used! AVR chips use 0 to indicate in and 1 to indicate out, whereas PICs use 0 for out and 1 for in. When IN and OUT are used there are no compatibility issues.

Example:

```
'This program sets PORTA bits 0 and 1 to in, and the rest to out.
'It also sets all of PORTB to output, except for B1.
'Individual form is used for PORTA:
DIR PORTA.0 IN
DIR PORTA.1 IN
DIR PORTA.2 OUT
DIR PORTA.3 OUT
DIR PORTA.4 OUT
DIR PORTA.5 OUT
DIR PORTA.6 OUT
DIR PORTA.7 OUT
'Entire port form used for B:
DIR PORTB b'00000010'

'Entire port form used for C:
DIR PORTC IN
```

Pot

Syntax:

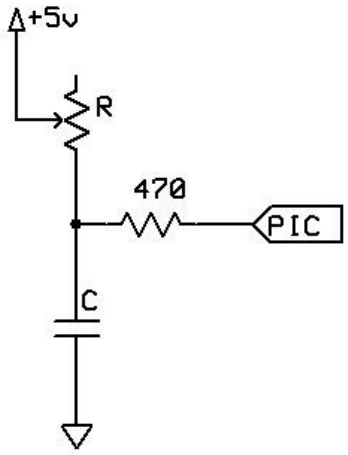
Pot *pin, output*

Command Availability:

Available on all microcontrollers.

Explanation:

Pot makes it possible to measure an analog resistance with a digital port, with the addition of a small capacitor. This is the required circuit:



The command works by using the microcontroller pin to discharge the capacitor, then measuring the time taken for the capacitor to charge again through the resistor.

The value for the capacitor must be adjusted depending on the size of the variable resistor. The charging time needs to be approximately 2.5 ms when the resistor is at its maximum value. For a typical 50 k potentiometer or LDR, a 50 nf capacitor is required.

This command should be used carefully. Each time it is inserted, 20 words of program memory are used on the chip, which as a rough guide is more than 15 times the size of the Set command.

pin is the port connected to the circuit. The direction of the pin will be dealt with by the Pot command.

output is the name of the variable that will receive the value.

Example:

```
     'This program will beep whenever a shadow is detected
     'A potentiometer is used to adjust the threshold

#chip 16F628A, 4
     #config INTOSC_OSC_NOCLKOUT

#define ADJUST PORTB.0
```

```
      #define LDR PORTB.1
      #define SoundOut PORTB.2

Dir SoundOut Out

Do
          Pot ADJUST, Threshold
          Pot LDR, LightLevel
          If LightLevel > Threshold Then
              Tone 1000, 100
          End If
      Loop
```

See Also http://ladyada.net/library/rccalc.html or www.cvs1.uklinux.net/cgi-bin/calculators/time_const.cgi for calculating capacitor value. These site are not associated with GCBASIC.

PulseOutInv

Syntax:

PulseOutInv *pin, time units*

Command Availability:

Available on all microcontrollers.

Explanation:

The PulseOutInv command will set the specified pin low, wait for the specified amount of time, and then set the pin high. The pin is specified in the same way as it is for the Set command, and the time is the same as for the Wait command.

Example:

```
'This program flashes an LED on GPIO.0 using PulseOutInv
#chip 12F629, 4
#config INTRC_OSC_NOCLKOUT

Dir GPIO.0 Out
    Do
        PulseOutInv GPIO.0, 1 sec     'Turn LED off for 1 sec
        Wait 1 sec                 'Wait 1 sec with LED on
    Loop
```

PulseOut

Syntax:

PulseOut *pin, time units*

Command Availability:

Available on all microcontrollers.

Explanation:

The PulseOut command will set the specified pin high, wait for the specified amount of time, and then set the pin low again. The pin is specified in the same way as it is for the Set command, and the time is the same as for the Wait command.

Example:

```
      'This program flashes an LED on GPIO.0 using PulseOut
      #chip 12F629, 4
      #config INTRC_OSC_NOCLKOUT


Dir GPIO.0 Out
      Do
          PulseOut GPIO.0, 1 sec 'Turn LED on for 1 sec
          Wait 1 sec 'Wait 1 sec with LED off
      Loop
```

Peek

Syntax:

OutputVariable = Peek (*location*)

Command Availability:

Available on all microcontrollers.

Explanation:

The Peek function is used to read information from the on-chip RAM of the microcontroller.

location is a word variable that gives the address to read. The exact range of valid values varies from chip to chip.

This command should not normally be used, as it will make the porting of code to another chip very difficult.

Example #1 :

```
'This program will read and check a value from PORTA
'Will only work on some PICs
Temp = PEEK(5)
IF Temp.2 ON THEN SET green ON
IF Temp.2 OFF THEN SET red ON
```

Example #2

```
 ' This subroutine will toggle the pin state.
 ' You must change the parameters for your specific chip.
 ' Usage as show in examples below.
 '
 '      Toggle @PORTE, 2 ' equates to RE1.
 '      Wait 100  ms
 '      Toggle @PORTE, 2
 '      Wait 100 ms



' Port , Pin address in Binary
 ' Pin0 = 1
 ' Pin1 = 2
 ' Pin2 = 4
 ' Pin3 = 8
 '
 ' You can toggle any number of pins.
 ' Toggle @PORTE, 0x55
 Sub Toggle ( In DestPort As word, In DestBit )
       Poke DestPort, Peek(DestPort) xor DestBit
 End sub
```

See Also Poke

Poke

Syntax:

Poke (*location, value*)

Command Availability:

Available on all microcontrollers.

Explanation:

The Poke command is used to write information to the on-chip RAM of the microcontroller.

location is a word variable that gives the address to write. The exact range of valid values varies from chip to chip.

value is the data to write to the location.

This command should not normally be used, as it will make the porting of code to another chip very difficult.

Example #1:

```
'This program will set all of the PORTB pins high
POKE (6, 255)
```

Example #2:

```
;Chip Settings
#chip 16F88

Do Forever
     FlashPin @PORTB, 8
     Wait 1 s
Loop

Sub FlashPin (In DestVar As word, In DestBit)
     Poke DestVar, Peek(DestVar) Or DestBit
     Wait 1 s
     Poke DestVar, Peek(DestVar) And Not DestBit
End Sub
```

Using @ before the name of a variable (including a special function register) will give you the address of that variable, which can then be stored in a word variable and used by Peek and Poke to indirectly access the location.

See Also Peek

ABS

Syntax:

integer_variable = ABS(integer_variable)

Command Availability:

Available on all microcontrollers

Explanation:

The ABS function will computes the absolute value of a integer number therefore in the range of -32678 to 32768.

Example:

```
absolute_value = Abs( -127 ): ' Will return 127

absolute_value = Abs( 127 ): ' Will return 127 also.  :-)
```

Average

Syntax:

integer_variable = Average(byte_variable1 , byte_variable2)

Command Availability:

Available on all microcontrollers

Explanation:

A function that returns the average of two numbers. This only supports byte variables. Provides a very fast way to calculate the average of two 8 bit numbers.

Example:

```
average_value = Average( 8 ,4 ): ' Will return 6
```

#chip

Syntax:

#chip *model, speed*

Explanation:

The #chip directive is used to specify the chip model and speed that GCBASIC must compile for. *model* is the model of the microcontroller chip - something along the lines of "16F819". *speed* is the speed of the chip in MHz, and is required for the delay and PWM routines.

If *speed* is omitted, GCBASIC will use the highest clock speed that the internal oscillator can support. If the chip does not have an internal oscillator, then GCBASIC will assume that the chip is being run at its maximum possible clock speed using an external crystal.

When using an AVR, there is not need to specify "AT" before the name.

Examples:

```
#chip 12F509, 4
#chip 18F4550, 48
#chip 16F88, 0.125
#chip tiny2313, 1
#chip mega8, 16
```

#config

Syntax:

#config *option1, option2, ... , optionn*

Explanation:

The #config directive is used to specify configuration options for the chip. There is a detailed explanation of #config in the Configuration section of help.

See Also Configuration

#define

Syntax:

#define *Find Replace*

Explanation:

#define will search through the program for *Find*, and replace it with the value given for *Replace.*

See Also Defines

#if

Syntax:

#if *Condition*

...

#endif

Explanation:

The #if directive is used to prevent a section of code from compiling unless *Condition* is true.

Condition has the same syntax as the condition in a normal GCBASIC if command. The only difference is that it uses constants instead of variables.

Example:

```
     'This program will pulse an adjustable number of pins on PORTB
     'The number of pins is controlled by the FlashPins constant
     #chip 16F88, 8

'The number of pins to flash
     #define FlashPins 2

'Initialise
     Dir PORTB Out

'Main loop
     Do
         #if FlashPins >= 1
             PulseOut PORTB.0, 250 ms
         #endif
         #if FlashPins >= 2
             PulseOut PORTB.1, 250 ms
         #endif
         #if FlashPins >= 3
             PulseOut PORTB.2, 250 ms
         #endif
         #if FlashPins >= 4
             PulseOut PORTB.3, 250 ms
         #endif
     Loop
```

#ifdef

Syntax:

#ifdef *Constant* | *Constant Value* | *Var(VariableName)*
...
#endif

Explanation:

The #ifdef directive is used to selectively enable sections of code. There are several ways in which it can be used:

- Checking if a constant is defined
- Checking if a constant is defined and has a particular value
- Checking if a system variable exists
- Checking if a system bit has been defined

The advantage of using #ifdef rather than an equivalent series of IF statements is the amount of code that is downloaded to the chip. #ifdef controls what code is compiled and downloaded, IF controls what is run once on the chip. #ifdef should be used whenever the value of a constant is to be checked.

GCBASIC also supports the #ifndef directive - this is the opposite of the #ifdef directive - it will remove code that #ifdef leaves, and vice versa.

(Note: The code in the following sections will not compile, as it is missing #chip directives and DIR commands. It is intended to act as an example only.)

Enabling code if a constant is defined

Syntax Example:

```
#define Blink1

#ifdef Blink1
    PulseOut PORTB.0, 1 sec
    Wait 1 sec
#endif
#ifdef Blink2
    PulseOut PORTB.1, 1 sec
    Wait 1 sec
#endif
```

This code will pulse PORTB.0, but not PORTB.1. This is because Blink1 has been defined, but Blink2 has not. If the line

#define Blink2

was added at the start of the program, then both pins would be pulsed. The value of the constant defined is not important and can be left off of the #define.

Enabling code if a constant is defined and has a given value

Syntax Example:

```
        #define PinsToFlash 2

    #ifdef PinsToFlash 1,2,3
            PulseOut PORTB.0, 1 sec
        #endif
        #ifdef PinsToFlash 2,3
            PulseOut PORTB.1, 1 sec
        #endif
        #ifdef PinsToFlash 3
            PulseOut PORTB.2, 1 sec
        #endif
```

This program uses a constant called PinsToFlash that controls how many lights are pulsed. PORTB.0 is pulsed when PinsToFlash is equal to 1, 2 or 3, PORTB.1 is pulsed when PinsToFlash equals 2 or 3, and PORTB.2 is flashed when PinsToFlash is 3.

Enabling code if a system variable is defined

Syntax Example:

```
        #ifdef NoVar(ANSEL)
             SET ADCON1.PCFG3 OFF
             SET ADCON1.PCFG2 ON
             SET ADCON1.PCFG1 ON
             SET ADCON1.PCFG0 OFF
        #endif
        #ifdef Var(ANSEL)
             ANSEL = 0
        #endif
```

The above section of code has been copied directly from a-d.h. It is used to disable the A/D function of pins, so that they can be used as standard digital I/O ports. If ANSEL is not declared as a system variable for a particular chip, then the program uses ADCON1 to control the port modes. If ANSEL is defined, then the chip is newer and its ports can be set to digital by clearing ANSEL.

Enabling code if a system bit is defined

Similar to above, except with Bit and NoBit in the place of Var and NoVar respectively.

See Also Defines, #define

#ifndef

Syntax:

#ifndef *Constant* | *Constant Value* | *Var(VariableName)*
...
#endif

Explanation:

The #ifndef directive is used to selectively enable sections of code. It is the opposite of the #ifdef directive - it will delete code in cases where #ifdef would leave it, and will leave code where #ifdef would delete it.

See the #ifdef article for more information.

#include

Syntax:

#include *filename*

Explanation:

#include tells GCBASIC to open up another file, read all of the subroutines and constants from it, and then copy them into the current program.

There are two forms of include - absolute, and relative.

Absolute is used to refer to files in the C:\Program Files\GCBASIC\include directory. The name of the file is specified in between < and > symbols. For instance, to include the file "srf04.h", the directive is:

```
#include <srf04.h>
```

Relative is used to read files in the same folder as the currently selected program. Filenames are given enclosed in quotation marks, such as:

```
#include "mycode.h"
```

where mycode.h is the name of the file that is to be read.

It is not essential that the include file name ends in .h - the important thing is that the name given to GCBASIC is the exact name of the file to be included.

Those who are familiar with #include in assembly or C should bear in mind that #include in GCBASIC works differently to #include in most other languages - code is not inserted at the location of the #include, but rather at the end of the current program.

#script

Syntax:

```
#script
    [scriptcommand1]
    [scriptcommand2]
    ...
    [scriptcommandn]
#endscript
```

Explanation:

The #script block is used to create small sections of code which GCBASIC runs during compilation. A detail explanation and example are included in the Scripts article.

See Also Scripts

#startup

Syntax:

#startup *SubName*

Explanation:

#startup is used in include files to automatically insert initialization routines. If a define or subroutine from the file is used in the program, then the specified subroutine will be called.

There are some limitations on this directive. It may only occur once within a file, and no parameters can be specified for the subroutine that is to be called.

#mem

This directive is obsolete. GCBASIC can now determine the amount of memory on a chip automatically, and will ignore the #mem directive.
It is recommended that this directive is removed from all programs.

#option bootloader

Syntax:

#option bootloader address

Explanation:

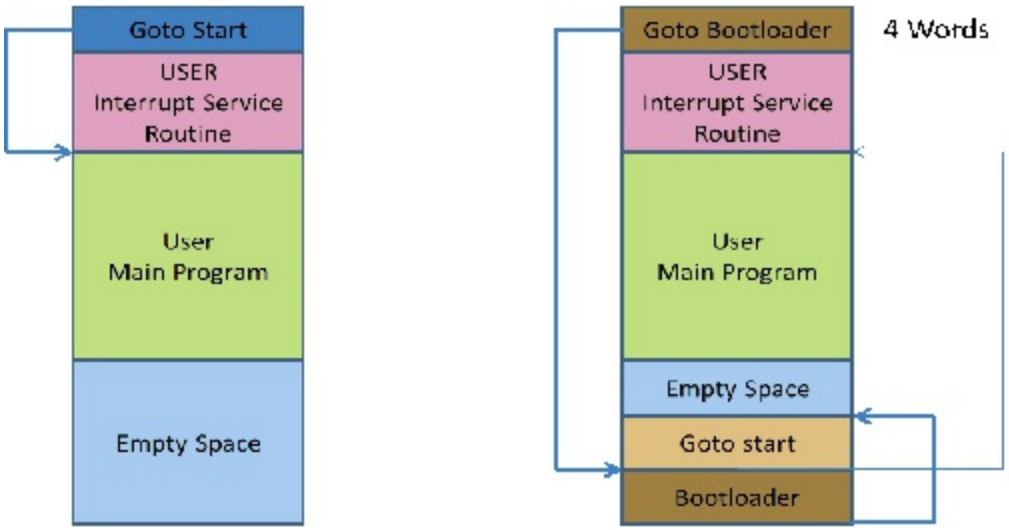
#option bootloader prevents the overwriting of any pre-loaded bootloader code, vectors, etc. below the <address> and have all GCBASIC code start at "address".

A bootloader is a program that stays in the microcontroller and communicates with the PC, typically through the serial interface. The bootloader receives a user program from the PC and writes it in the flash memory, then launches this program in execution. Bootloaders can only be used with those microcontrollers that can write their flash memory through software.

The bootloader itself must be written into the flash memory with an external programmer.

In order for the bootloader to be launched after each reset, a "goto bootloader" instruction must exist somewhere in the first 4 instructions; There are two types of bootloaders, some that require that the user reallocate the code and others that by themselves reallocate the first 4 instructions of the user program to another location and execute them when the bootloader exits.

The diagram below shows the architecture of a bootloader. The left hand is the operation of the instructions without a bootloader. The right hand shows the initial instruction of goto the bootoader, then, when the bootloader has initialised the execution of the start code.



See example bootload software.

Example:

#option bootloader 0x800

Assembler Overview

Explanation:

You can use assembler code within your Great Cow Basic code.

You can put the assembler code inline in with your source code. The assembler code will be passed through to the assembly file associated with your project.

GCBASIC should recognise all of the commands in the microcomputer datasheet.

The commands should be in lower case and have a space or tab in front of the command.

Even if the mnemonics are not being formatted properly, gputils/MPASM should still be capable of assemble the source code.

Example:

Format commands as follows:

```
btfsc STATUS,Z
bsf PORTB,1
```

Serial/RS232 Buffer Ring

Explanation:

This code implements a serial buffer ring. This means the incoming serial port data is placed into a buffer that can be read at anytime. This example uses an interrupt and a buffer.

```
;Chip Settings
      #chip 16F1937,32
      #config LVP=OFF, BODEN=OFF, WDT=OFF, OSC=XT

      ' [change to your config] This is the config for a serial terminal
      ' turn on the RS232 and terminal port.
      ' Define the USART port
      #define USART_BAUD_RATE 9600
      #define USART_BLOCKING true
      ' [[change to your config] Ensure these port addresses are correct
      #define SerInPort PORTc.7
      #define SerOutPort PORTc.6
      'Set pin directions
      Dir SerOutPort Out
      Dir SerInPort In



' [[change to your config] This assumes you are using an ANSI compatible terminal.  Use
PUTTY.EXE it is very easy.

HSerPrint "Started: Serial between two devices"
      HSerSend 10
      HSerSend 13

        ' Pot port
        DIR PORTA.0 IN

  'Interrupt Handlers
        On Interrupt UsartRX1Ready Call readUSART

  ' Constants etc required for Buffer Ring
        #define BUFFER_SIZE 8
        #define bkbhit (next_in <> next_out)
        ;Variables
        Dim buffer(BUFFER_SIZE)
        Dim next_in as byte: next_in = 1
        Dim next_out as byte: next_out = 1
        Dim syncbyte as Byte



  syncbyte = 0x55 ' you can use 255 - your choice


   Do
                ' Send some info to another device
                Repeat 3
                        HSerSend syncbyte
```

```
            end Repeat
            HSerprint readad(an0)

        do while bkbhit

           ' get three sync chars  then display the next char in buffer
               if bgetc = syncbyte and bgetc = syncbyte and bgetc = syncbyte then
                  HSerPrint "sync'ed '"
                  HSerPrint chr(bgetc)
                  HSerSend 10
                  HSerSend 13
               end if
            Loop

LOOP

  Sub readUSART

      buffer(next_in) = HSerReceive
           temppnt = next_in
           next_in = ( next_in + 1 ) % BUFFER_SIZE
           if ( next_in = next_out ) then  ' buffer is full!!
              next_in = temppnt
           end if

  End Sub

  function bgetc
           wait while !(bkbhit)
           bgetc = buffer(next_out)
           next_out=(next_out+1) % BUFFER_SIZE
     end Function
```

See Also Interrupts, HSerReceive

Flashing LEDs and an Interrupt

Explanation:

This code implements four flashing LEDs. This is based on the Microchip Low Pin Count Demo Board..

The example program will blink the four red lights in succession. Press the Push Button Switch, labeled SW1, and the sequence of the lights will reverse. Rotate the potentiometer, labeled RP1, and the light sequence will blink at a different rate.

This implements an interrupt for the switch press, reads the analog port and set the LEDs.

```
#chip 18F14K22, 32
     #config OSC = IRC, MCLRE_OFF

' Works with the low count demo board

'Set the input pin direction
         #define SwitchIn1 PORTa.3
         Dir SwitchIn1 In

#define LedPort PORTc
         DIR PORTC OUT

'Setup the ADC  pin direction
         Dir PORTA.0 In
         dim ADCreading as word



'Setup the input pin direction
         #define IntPortA PORTA.1
         Dir IntPortA In

' variable and constants
         #define intstate as byte
         intstate = 0
         #define minwait 1

    dim ccount as byte
         dim leddir as byte

         ccount = 8
         leddir = 0

    SET PORTC = 15
         WAIT 1 S
               SET PORTC = 0

          Set IOCA.3 on
               Dir porta.3 in
               On Interrupt PORTABCHANGE Call Setir
```

```
            setLedDirection

     main:
     DO
         INTON
         ADCreading = ReadAD10(AN0)
         if ADCreading < minwait then ADCreading = minwait

         ' Set LEDs
         Set PortC = ccount
         wait ADCreading ms

         if leddir = 0 then
           rotate ccount left simple
           ' Restart LED position
           if ccount = 16 then
               ccount = 128
           end if


    end if

         if leddir = 1 then
           rotate ccount Right simple
           ' restart LED position
           if ccount = 128 then
             ccount = 8
           end if

         end if
         ' reset interrupt - this may be been reset so set to zero so interrupt can
     operate.
         intstate = 0

     Loop


     sub Setir



         if IntPortA  = 0 and intstate = 0 then
           intstate = 1
           wait while SwitchIn1 = 0
           setLedDirection
         end if



end sub

sub setLedDirection

   ' set LED values
        select case leddir

    case 0
           leddir = 1
           ccount  = 8

         case 1
```

```
            leddir = 0
            ccount = 1

    end select

End Sub
```

See Also Interrupts, ReadAD10

Sine Tables

Explanation:

This code implements Sine tables lookup in Great Cow Basic.
The example program will output the Sine value of numbers between -720 and 720 degrees.

This is implemented to output the values to an LCD display.

```
      ;Implementing the Sine function in GC Basic
      ;Thomas Henry -- 3/19/2014

;This program demonstrates the Sine function by printing
      ;out its values for arguments from -720 degrees
      ;to 720 degrees, in one-degree increments. The actual
      ;function itself is very short. The main business of the
      ;demo program is tied up in formatting the results to
      ;appear neatly on an LCD.

;This Sine function is completely general purpose. It can
      ;handle positive, negative or zero arguments of any
      ;arbitrary size.

;The values returned are valid to four decimal places,
      ;equivalent to most printed tables, and more than accurate
      ;for most engineering purposes.

;The values are maintained as signed integers, scaled up by
      ;a factor of 10000. A lookup table stored in program
      ;memory is used.

;----- Configuration

#chip 16F88, 8              ;PIC16F88 running at 8 MHz
      #config mclr=off            ;reset handled internally
      #config osc=int             ;use internal clock

;----- Constants

#define LCD_IO      4       ;4-bit mode
      #define LCD_RS      PortB.2 ;pin 8 is LCD Register Select
      #define LCD_Enable  PortB.3 ;pin 9 is LCD Enable
      #define LCD_DB4     PortB.4 ;DB4 on pin 10
      #define LCD_DB5     PortB.5 ;DB5 on pin 11
      #define LCD_DB6     PortB.6 ;DB6 on pin 12
      #define LCD_DB7     PortB.7 ;DB7 on pin 13
      #define LCD_NO_RW   1       ;ground the RW line on LCD

#define degree      223     ;ASCII code for degree mark

;----- Variables
```

```
dim index as byte
      dim i, sign, value, arg as integer

;----- Program

dir PortB out                 ;all outputs to the LCD

for i = -720 to 720           ;values of sine from -720 to 720
        cls
        print "sin("                  ;print the label
        print i                       ;and the argument
        LCDWriteChar degree           ;print degree mark
        print ")="                    ;and closing parenthesis
        locate 1,0                    ;move to the next line

  value = sin(i)              ;get the value of the sine

  if value = 0 or value = -1 or value = 1 then
          print value                 ;handle whole number cases
        else
          if value < 0 then
            print "-"                 ;a negative result
            value = -1 * value
          end if

    print "0."                ;format fractional number

    if value < 1000 then      ;left pad smaller results
            print "0"
          end if
          if value < 100 then
            print "0"
          end if
            if value < 10 then
            print "0"
          end if

    print value               ;then print the fraction
         end if

  wait 1 S                    ;pause to view
       next i

;----- Subroutines

function sin(in arg as integer) as integer
        if arg < 0 then             ;sine is an odd function,
          sign = -1                 ;so negate negative argument
          arg = -1 * arg            ;and change sign of result
        else
          sign = 1                  ;else a positive argument
        end if

  arg = arg mod 360           ;reduce to 0 to 359 degrees

  if (arg > 270) then         ;Quadrant IV
```

```
          sign = -1 * sign
          arg = 360 - arg           ;make reference angle
        else
          if (arg > 180) then       ;Quadrant III
            sign = -1 * sign
            arg = arg - 180         ;make reference angle
          else                      ;Quadrant II
            if (arg > 90) then
              arg = 180 - arg       ;make reference angle
            end if
          end if                    ;Quadrant I by default
        end if

  index = [byte]arg+1         ;make index into table
        readTable sineTab, index, sin

  sin = sign * sin            ;create final result
      end function

;----- Data

table sineTab as word
        ;Sine values for 0 through 90 degrees, scaled up by 10000.
        0
        175
        349
        523
        698
        872
        1045
        1219
        1392
        1564
        1736
        1908
        2079
        2250
        2419
        2588
        2756
        2924
        3090
        3256
        3420
        3584
        3746
        3907
        4067
        4226
        4384
        4540
        4695
        4848
        5000
        5150
        5299
        5446
        5592
        5736
        5878
```

6018
6157
6293
6428
6561
6691
6820
6947
7071
7193
7314
7431
7547
7660
7771
7880
7986
8090
8192
8290
8387
8480
8572
8660
8746
8829
8910
8988
9063
9135
9205
9272
9336
9397
9455
9511
9563
9613
9659
9703
9744
9781
9816
9848
9877
9903
9925
9945
9962
9976
9986
9994
9998
1
end table

Troubleshooting

Problem	Common Causes	More Assistance
Cannot compile a program	There is an error in the program. Is Great Cow BASIC complaining about a particular line of code?	GCBASIC Forums
	Great Cow BASIC has not been installed correctly - reinstall it.	GCBASIC Forums
	There is a bug in Great Cow BASIC	Post on the forums, or send an email with your program attached to w_cholmondeley@users.sourceforge.net
A program compiles and downloads fine, but will not run	Oscillator not selected.	Configuration

Changes

Release v0.9hd
Updates to this Help are as follows:

Jan 14

New item(s)

- Len
- Asc
- Chr
- Trim
- Ltrim
- Rtrim
- Swap4
- Swap
- Abs
- Average
- Trim
- Ltrim
- Rtrim
- Wordtobin
- Bytetobin
- GLCD
- DectoBCD
- BCDtoDec
- Using variables
- More on constants and variables
- Acknowledgements

Changes to

- Str
- Hex
- Poke
- Else
- Readtable
- Exit (was exitsub)
- Command line parameters
- Frequently asked questions

Fixed typos.
Updated REPEAT maximum repeat value.
Updated most pages for layout.
Fixed links to external pages, again. This time downloaded as full html pages, for POT and LC.
Added LABEL, Bootloader and revise Select, add READAD10
Fix Double SWAP

@ v0.9hb Mar 2014

Added PulseOutInv
I2CRestart
Add new variants to use of Comments
Added Assembler Section

@ v0.9hc Mar 2014

Revised HSERPRINT to show Integers and Longs are supported and changed the text to be correct.
Added HserPrintByteCRLF and HserPrintCRLF
Added Sine Table Example
Revised TABLE to show the limitation with respect to using WORDS when placing TABLES in EEPROM

@ v0.9hd

Revised Rotate to clarify type supported.