BASIC Commander® & InnoBASIC™ Workshop

Reference Manual

Version 1.08

Contents

Preface

Welcome

We are delighted that you have chosen to purchase our Innovati® products. Whether you are a newcomer to microcontrollers or whether you have arrived at this point with some experience under your belt, we feel assured you will enjoy immensely the unique Innovati® approach to this fascinating area of electronics. This manual will give you the information you need to get going and how it is used will depend upon your skill level and previous experience. Users who are new to the world of microcontrollers would benefit from some selected other background reading on basic electronics and on the BASIC language before jumping in. Those with some electronic and programming experience could be more selective in what they read from this manual. However, wherever your interests lie and no matter what level you find yourself at, the Innovati® team sincerely wish you a fascinating journey into the world of microcontrollers.

The unique Innovati® approach in supplying high functioning modules allows you to develop hardware applications with superior functions running in a very short space of time and with a minimum of design effort. As to what you could achieve, well how about remote camera control in your model off-road explorer, or a sophisticated automatic lighting control system for your home, or perhaps a programmable robot, etc. Or perhaps you are just interested in educating yourself about microcontrollers. Or maybe you are a high school teacher and would like to

introduce the subject of microcontrollers into your science curriculum. Make no mistake, as the Innovati® system contains an industrial standard commercial high quality microcontroller at its core, the possibilities and applications are truly endless, and limited only by your ingenuity and experience. Whatever your interests may be, we are sure that the Innovati® system will not only result in the creation of ingenious and creative inventions and devices but will also be a real learning experience into the workings of microcontrollers.

Microcontrollers Everywhere

From that first cup of coffee and slice of toast you made yourself this morning to the alarm clock you set when you went to bed that night, your day has almost certainly been influenced and made more convenient by numerous microcontrollers in these electrical home appliances.

A car probably contains numerous microcontrollers, controlling everything from the mundane air conditioning and heating to the more sophisticated suspension and engine management systems. Several of the instruments on the dashboard will certainly contain microcontrollers to indicate to you that all is well, or perhaps that all is not well with your car! Did you watch TV, or perhaps a DVD? Undoubtedly these household appliances and their remote controllers contain microcontrollers. What was in the past a device intended for industrial applications or a tool for research applications in a laboratory has now found its way into all aspects of our lives. So if microcontrollers are so useful and can be used everywhere, why am I not able to use them for my own projects? Well now you can with this unique and high functioning Innovati® system. Perhaps you would like to put some automatic control into your model railway, or build a sophisticated alarm system for your home, or maybe even some unique automotive projects. Or how about some solar power projects, the list really can go on and on and is only limited by your creativity and imagination.

Innovati® Approach

The development of microcontroller projects can be approached from many angles. The traditional way would be to write your software using a low level assembly language, or use a so-called high-level language, which would not offer much convenience in terms of programming, with their corresponding long learning curves. Along with this, hardware for your specific application would need to be designed to interface the microcontroller to the real world. This could consist of things such as LCD displays, switches, LEDs, etc. All of this takes time and perhaps is an approach that is OK for industrial companies designing specific and specialized high volume products. However, Innovati® has taken the toil and time out of this approach by providing an intuitive BASIC style of programming language which we call innoBASIC™, and which is featured with sophisticated commands through which conventional functions can be accomplished in an easy manner. Additionally, through these commands, innoBASIC™ is featured with a most unique capability to interface Innovati's peripheral modules which further integrates both hardware and software into one. Complicated hardware control will be seamlessly migrated into the software world. In this way the amount of time and effort you need to get your idea up and running from a concept to a real working project is drastically reduced. Microcontrollers no longer are something just for the professionals, but can now easily be used and programmed by all.

System Overview

1

Introduction

The system is composed of several different parts, which all work together to form the overall system. At the top is the innoBASIC™ Workshop environment which is a software utility running in the PC. The user's program is edited with our powerful featured language innoBASIC™ editor which integrates all the resources perfectly. After editing, the program is compiled and downloaded from within this environment. During run-time, a Terminal Window is available as the Human Interface or Debug console. Then via the USB interface, the finished program code is downloaded to the BASIC Commander® Single Board Computer, which is the heart of the system. This small dual in-line Printed Circuit Board, PCB for short, is the unit where your program will be stored and run. After the program is stored in the program memory, the Single Board Computer is set to run.

There are three kinds of resources in which the BASIC Commander® can be used. First is the General-Purpose I/Os, for which built-in commands are provided for sophisticated functions; second is the cmdBUS™, where up to 32 Innovati® featured Peripheral Modules can be connected. These totally object-oriented modules make this functional expansion one of the most special features of the system. Thirdly is the debug interface, where information can be exchanged between the BASIC Commander® and the innoBASIC™ Workshop Terminal Window, not only for debugging but also for human-machine interfacing.

innoBASIC™ Workshop:
Editor, Compiler,
Downloader & Human
Interface Debug Window

USB:
Program Download and Run-
time Human Interface

On Line

Off Line

General I/O
Circuits

GIOs

cmdBUS:
Up to 32 Innovati's
Object-oriented
Peripheral Modules

cmdBUS™

Built-in Commands:
Featured for
General Purpose I/O
Functions

Peripheral
Module

Peripheral
Module

Peripheral
Module
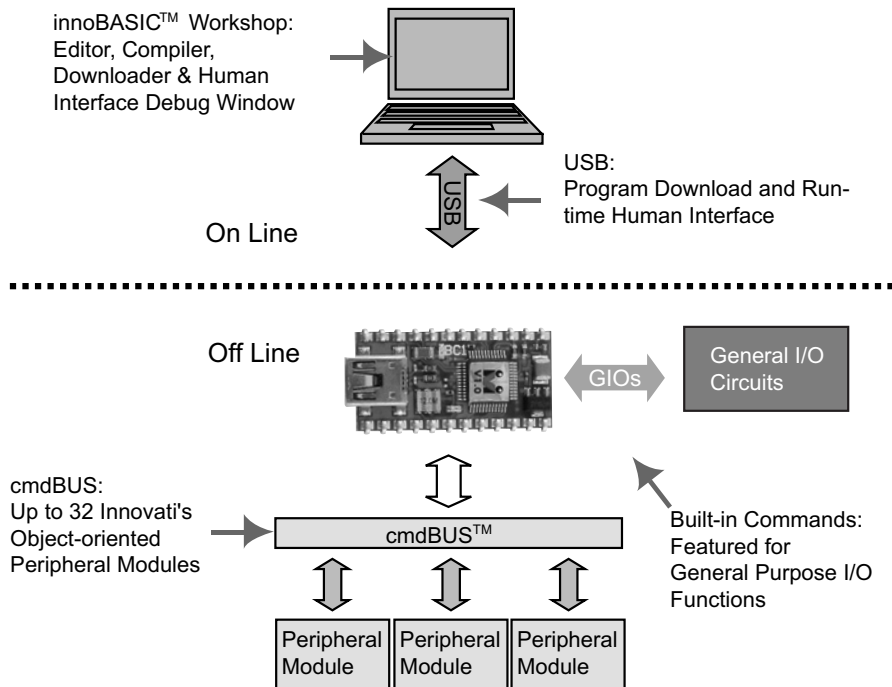
Figure 1-1   BASIC Commander® and innoBASIC™ Workshop System Overview

InnoBASIC™ Programming Language

With the usual focus on ensuring that user projects are up and running in a minimum
amount of time, the innoBASIC™ programming language was developed by Innovati's
software engineers to ensure that users can easily and quickly learn the rudiments of
programming techniques. Although innoBASIC™ is easy to use, it should not in any
way be viewed as a functionally restricted language. It is in fact a highly capable and
functionally rich language with a host of useful features. While based on the
universally popular BASIC language it has its own special characteristics, one of

which is the easy ability to control the external hardware modules. The InnoBASIC™ language does not employ the traditional interpreter methodology but rather uses a compiler methodology, which dramatically increases the execution speed.

BASIC Commander®

The BASIC Commander® is the central hardware unit of the system and is actually a miniature Single-Board Computer. On this small electronic board is a high performance microcontroller integrated circuit device, together with some other peripheral components such as those required for the power supply, PC interfacing, etc. When editing the program for your personal project, the BASIC Commander® will be connected to your PC with the supplied USB cable to enable easy program downloading and debug to be implemented. However, when the program development has been completed, the BASIC Commander® can of course be disconnected from the PC and run independently within your project without the PC connection.
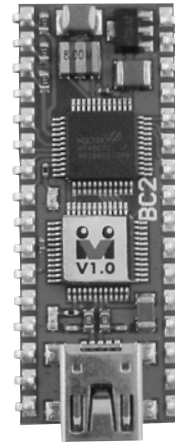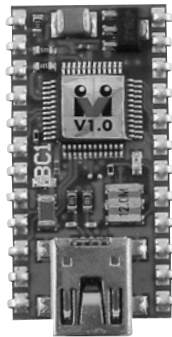
Figure 1-2a 24-pin BASIC Commander®   Figure 1-2b 32-pin BASIC Commander®

As the BASIC Commander® contains sensitive electronic components, handling with care is required to eliminate damage due to electrostatic discharge, often known by the abbreviation ESD. When inserting into a socket or a breadboard, ensure that the pins are in the right position, otherwise damage to the BASIC Commander® may occur. The I/O input voltage levels must not exceed their maximum rating of 6.0 Volts, exceeding this level may also cause damage to the I/O pins on the BASIC Commander® itself.

Peripheral Modules

The use of modules is one of the unique features of the Innovati® system. Comprising of modules such as those for I/O expansion, LCD display driving, motor control, GPS and so on, these external plug and play modules give the user a means of easily expanding their systems to incorporate a host of complex and useful features. The name Innovati® Peripheral Modules is a collective name for an ever-growing list of plug-in modules. Check Innovati's web site to find about the latest module developments. Each Peripheral Module has its own product name, which is not only used for product identification, but also used during programming. For example, the LCD module has an identification name "LCD2X16A".

Figure 1-3a LCD2X16A front view
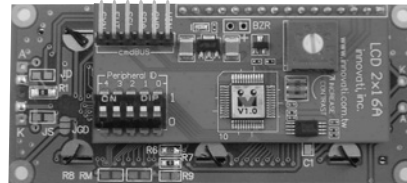
Figure 1-3b LCD2X16A rear view

For each of the Peripheral Modules, you will need a flat cable to connect the module to the cmdBUS™ on the Education Board.

Figure 1-4 6-wired cmdBUS™ cable

Caution must be taken when plugging-in the cable, as an incorrect connection could cause serious damage to the device.

Education Board

There is also an Education Board where custom-built projects can be constructed and in which a socket is included for plugging in the BASIC Commander® Single Board computer. This Education Board is equipped with a breadboard and with power management, providing users with a solderless means of constructing their projects. You may simply use jump wires to connect the power and I/O pins from the female header to the breadboard.
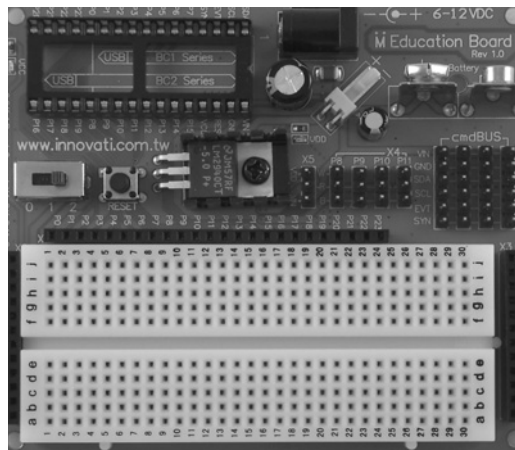
Figure 1-5 Education Board

As the Education Board comes without the BASIC Commander®, you may choose either a 24-pin or a 32-pin version of BASIC Commander® to plug into the Education Board. Be sure to check the insertion instruction drawn on the Education Board as incorrect alignment may cause serious damage to the BASIC Commander®.

How to Use this Reference Manual

To get ultimate enjoyment from your system, please take time to examine the manual and familiarize yourself with the key components of the system. Those with some microcontroller experience will no doubt want to push forward faster and could probably skip the introductory chapters. For those who are new to this fascinating area, we sincerely recommend that you sit down and work your way through the book as time spent at this initial learning stage will be a good investment for the future when you get into the real nuts and bolts of more complicated projects. However, we may view the reading of manuals, for some it is seen as an enjoyable learning process or for others as a necessary evil, to obtain essential information, don't let it stand in the way of having fun with the system.

Installation and Getting Started

2

Introduction

This section is dedicated to getting you up and running as quickly as possible and will demonstrate the basic functions of the system such as program writing and editing, compiling, downloading and debug. It is necessary to have either a BASIC Commander® or Education Board connected as without this the program cannot be downloaded.

Of course it is necessary to have an IBM or Compatible PC running under Windows 98 or above version/2000/ME/XP/Vista. A CD-ROM Drive is recommended, you can use the CD-ROM provided to install the innoBASIC™ Workshop. A USB 1.1 / 2.0 port is a must, which is essential for your program downloading and debugging.

Installing the InnoBASIC™ Workshop

Insert the supplied CD into your CD-ROM Drive and follow the on-screen instructions to install the innoBASIC™ workshop. Another and perhaps better method would be to go to the Innovati® website and download the latest versions of the innoBASIC™ workshop software. Although the supplied CD software will run perfectly well, using the website version will ensure you have the latest up-to-date version of the software. Installation should be trouble free and you should be presented with the innoBASIC™ workshop window after installation and running. This is the area in which you will work to write and edit your program.
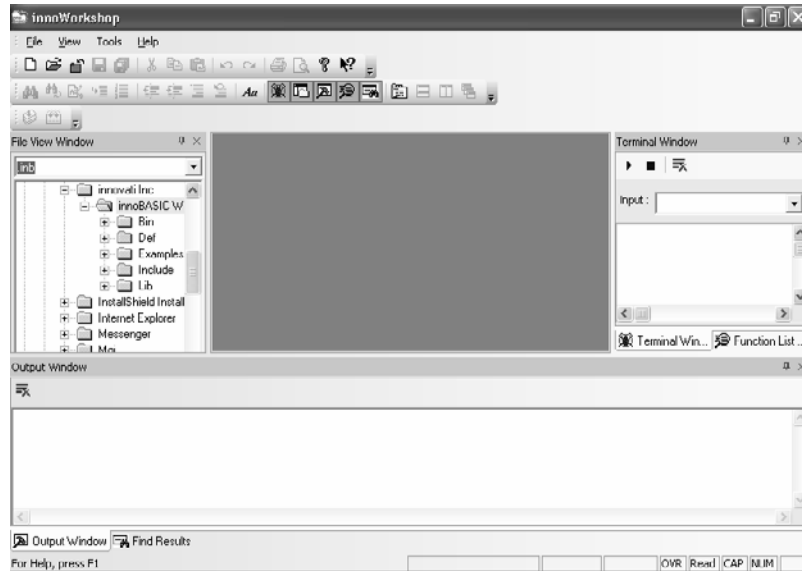
Figure 2-1 The innoBASIC™ Workshop Window

Hardware Installation

Using the supplied USB cable, connect either a BASIC Commander® or the Innovati®
Education Board to a USB socket on your PC. As the USB cable will supply the
necessary power to the hardware, no other power supply is necessary. Nevertheless,
the USB port should not be seen as a major power provider for your applications. If a
total of more than 500mA is demanded by your application, you should take proper
care of power management by using an external power supply.

TO PC USB Port   mini USB B
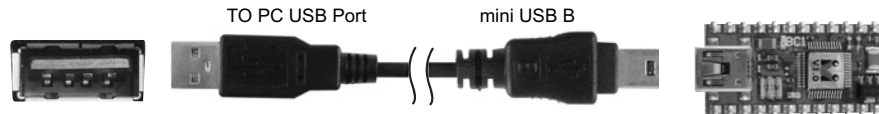
Figure 2-2 Connecting the Hardware

By following the on-screen instructions, where you will see the usual installation
screens for USB hardware installation, the hardware drivers should be successfully
installed.

Producing Your First Program

With the hardware and software successfully installed we are now ready to do
something with the system, in other words to write our first program. The following
steps should enable you to do this:

1. Open a new file using the File/New menu command or by selecting the "New"
   Icon. You will now see a blank area appear on the screen where the cursor will be
   flashing. This is the area where the program can now be written. You will also see
   the name Untitle1 appear which is the name the system will assign to your new
   file. Of course it can be changed to your own selected name but more about this
   later.

2. In this blank editing area enter the instructions shown below. Don't be too concerned at this stage about what the instructions mean, this information can be picked up at a later stage, however it is important to copy the instructions exactly:

```
Sub Main ()
    Dim s As String *10
    Debugin "Please enter your name: ", s, CR
    Debug s, ", Welcome to the Innovati World!"
End Sub
```
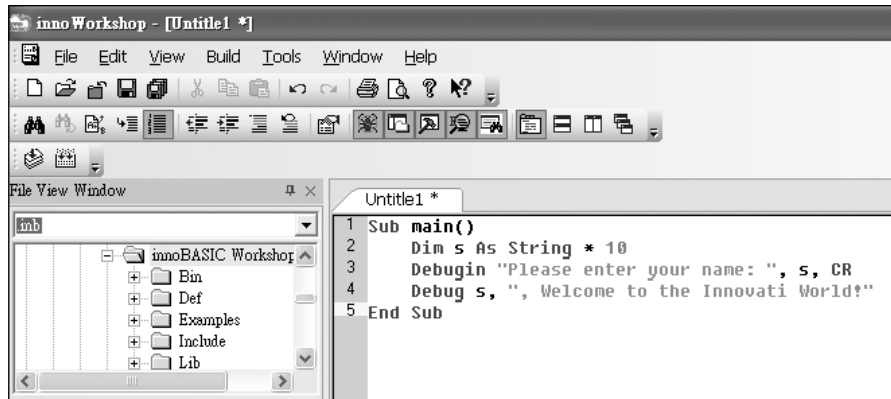

Figure 2-3 Editing the Program

3. This program should now be saved using the usual File/Save menu commands or using the standard Save File Icon. You can select the location where the file is to be placed using the directory structure in the File View Window. The file can be placed into the "Example" folder which has been created under the innoBASIC™ Workshop folder as shown. Of course it can be placed into any other folder you may have created using the Windows File Manager. Note that the file should be saved with the extension ".inb", which stands for innoBASIC™.
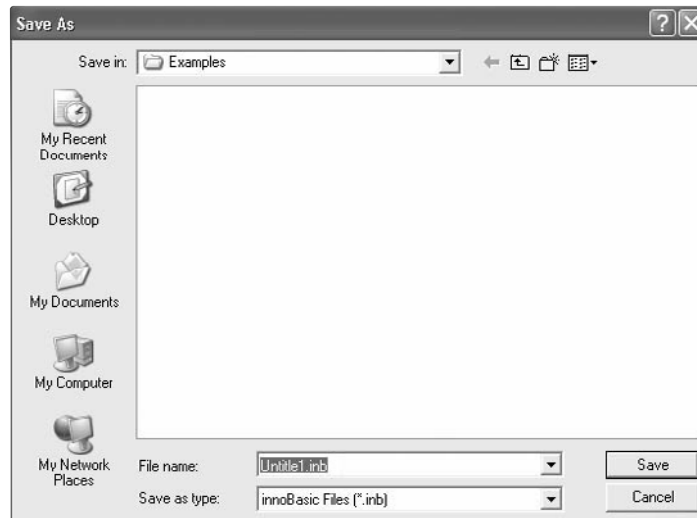
Figure 2-4 Saving the Program

4. Now the program can be compiled, which is implemented by selecting the Compile commander under the Build menu or by clicking on the Compile Icon. If the program has been written and edited correctly, you will receive a message in the Output Window telling you so, if not an error message will be received, and your program should be modified. A simple comparison of the above instructions

with what you have typed in should reveal where the error lies, however the error messages will actually give an indication as to where the error is located. Besides the Compile command, you may select the Build icon, which will compile the file first, and if no error is found, the compiled machine code will be downloaded via the USB cable into the BASIC Commander®. You may see a green LED on the BASIC Commander® flashing indicating that the download process is in progress.
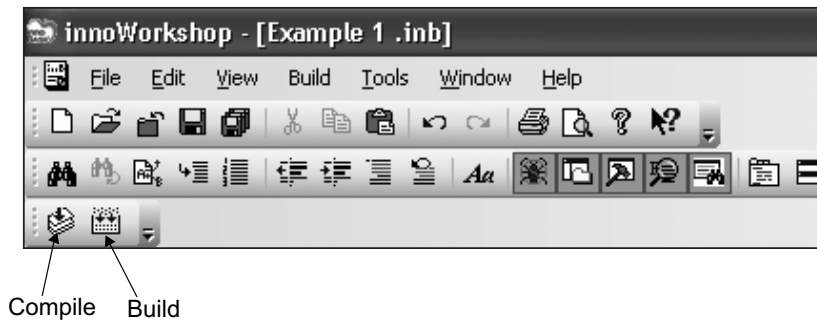
Figure 2-5 Compiling and Downloading the Program

5.   Your program has been downloaded and is presently resident in the non-volatile memory of the BASIC Commander® and will now run automatically. You should see a message in the Terminal Window asking you to enter your name. Please enter your first name here and then press Enter. The system should respond with a welcome message containing your name. You should note at this stage that all interaction between yourself and the BASIC Commander® hardware is taking place through the Terminal Window.

Data Input Area
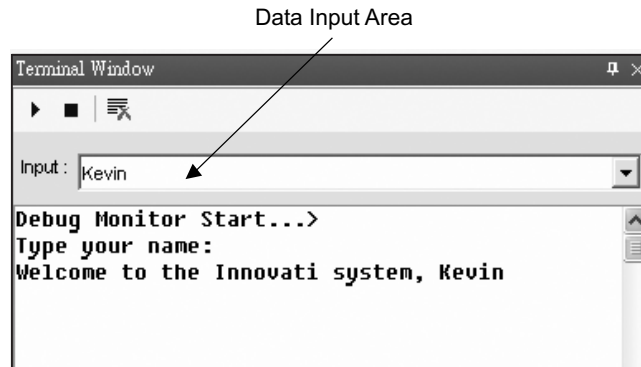
Figure 2-6 Welcome Message in the Terminal Window

If you have succeeded in implementing the above example, we extend our congratulations and hope that you can now move on with more confidence. By reading more of the manual you will understand more about the capabilities of the system, hopefully guiding you into the development of more complex projects and applications.

What Has Just Happened?

If you were successful in completing the example, what you have actually done is to have gone through the full process of program writing, editing, compiling, downloading, debugging and execution. It may seem a very simple example program, but actually it has still utilized all of the steps required in more complex applications. It has also demonstrated how the Terminal Window is used for direct communication between the user and the BASIC Commander® hardware. At this stage you may like to use this opportunity to modify your simple program to provide more complex operations, perhaps by modifying the instructions or adding a few new ones of your own. You may choose rather to continue to study the manual in more depth before proceeding.

3

InnoBASIC™ Workshop

Introduction

After working through the previously provided example program, it is now time to get down to learning more about the system and getting to work on some more serious projects. Each time the innoBASIC™ workshop system software is executed the result will be shown on the screen display as shown in the diagram. As the name "Workshop" suggests, this is the area in which you will work to create, debug, compile and download your program. As you are probably familiar with the Windows operating system and its many standard programs, you will no doubt recognize many of the standard function icons shown on the screen. In an area called the Program Editing Window you will write and edit your program and in the area called the Terminal Window you will communicate with the BASIC Commander® hardware.

Screen Layout

The screen is subdivided into several main individual working area windows each with their own functions. These are listed below with a short description.

File View Window   Program Editing Window   Function View Window

Program List   Output Window   Terminal Window
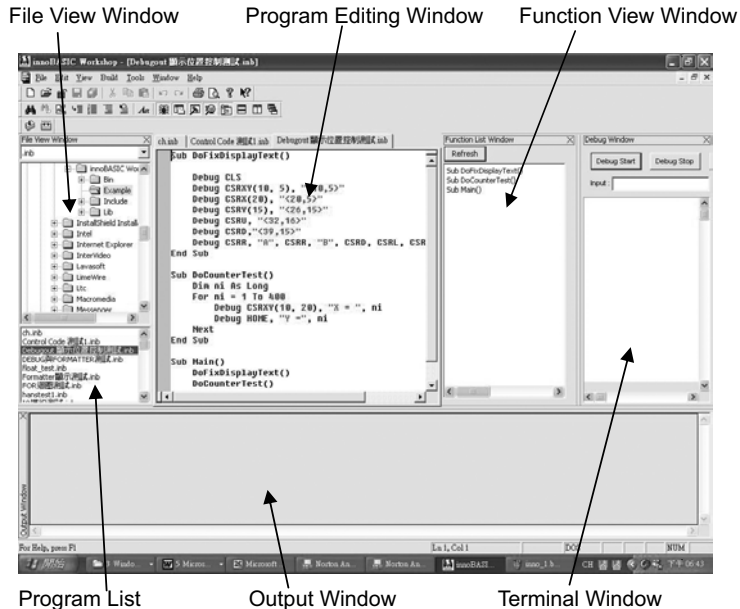
Figure 3-1 Workshop Windows and Menus

File View Window

This window shows the familiar windows file manager display from which your working program can be selected. Here you can select different directories to retrieve already stored files. Just below the file manager area are listed all the files in the selected folder. This area can be selected to display only the innoBASIC™ files, which are the files with an .inb extension next to the filename, or to display all files which is the *.* option. If an innoBASIC™ file is selected by double clicking its filename, the file contents will be displayed in the area to the right of the File View Window. Optional filename tabs displayed above the file contents window, which can be selected to show its contents.

Program Editing Window

This is the area in which you will write and edit your project's program. If no file is selected from the File View Window, this area will remain blank, however after a file is selected, its contents will be displayed within this window. As the user's project will often use several files, more than one file can be opened but only the contents of one file can be displayed in this area. Each opened file will have a representing tab and all tabs will be listed at the top of the Program Editing Window. The tab which is highlighted represents the file whose contents are presently displayed. Selecting other tabs with the mouse gives convenient editing access to other open files.

Output Window

When your program is compiled or downloaded, the various steps that the system goes through and the status of the system are shown within this window. Any errors occurring during compilation or downloading will be displayed here.

Function View Window

This area basically shows the list of functions or subroutines within the main program, giving a kind of index to the overall program. Rather than searching through what could be a long program, this window area displays the main subroutines within the program. By clicking on the name of the subroutine in the Function View Window, the cursor will automatically jump to the subroutine location in the program.

Terminal Window

The result of the execution of any DEBUG or DEBUGIN command will be displayed in this area. This is the place where the BASIC Commander® communicates with the user or where the user sends data to the BASIC Commander®. In the Terminal Window you will see three function icons, Start, Stop and Clear. The Clear function is used to clear the Terminal Window screen only, and has no effect on the BASIC Commander® or program operation. The Debug Start will execute a reset to the BASIC Commander® and cause the program to run again from the first instruction.

The Stop function will cease the communication process between the BASIC Commander® and Terminal, which are implemented with the DEBUG or DEBUGIN commands.

Some of these windows possess their own control icon which allows the window to be displayed or switched off. Each icon has a simple toggle function, click once to turn on the window and click again to turn off. The same on/off windows function are also available from within the View Menu.

Terminal Window

File View Window Icon

Function View Window Icon

Output Window Icon
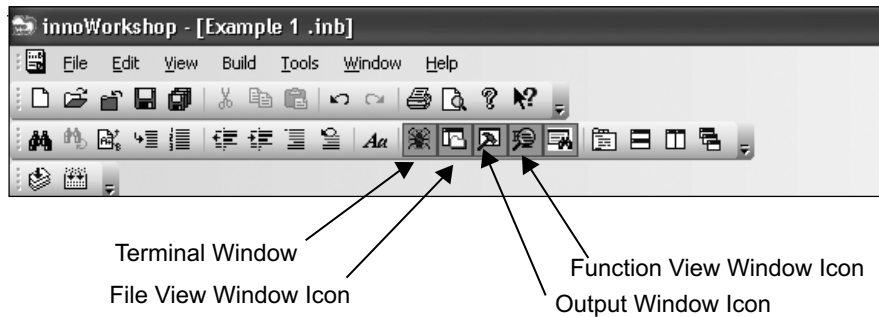
Figure 3-2 Workshop Window Control Icons

Try pressing each of the window toggle icons in turn and you will see how each window can be displayed or hidden on the overall innoBASIC™ Workshop window. You may prefer to work with all of the windows on as shown.

Steps to Creating a Program

For new beginners the creation of your first real program and application is perhaps a daunting prospect. This is simply due to a series of unknowns, which with an explanation on our part and a bit of effort on your part can quickly be dispelled. From the moment you install your system to the time you finally get your program and application running, there are certain steps which must be gone through to achieve the desired results. If you have worked through the simple example provided, you will have seen what these steps are, now for more complex examples you will soon start to see those previous computing mysteries and fears vanish before your eyes.

Program Editing	Input Instructions
Compile Program	Converts Instructions to Machine Code
Download Program	Place machine code into BASIC Commander®
Debug Program	Detect Errors using Terminal Window
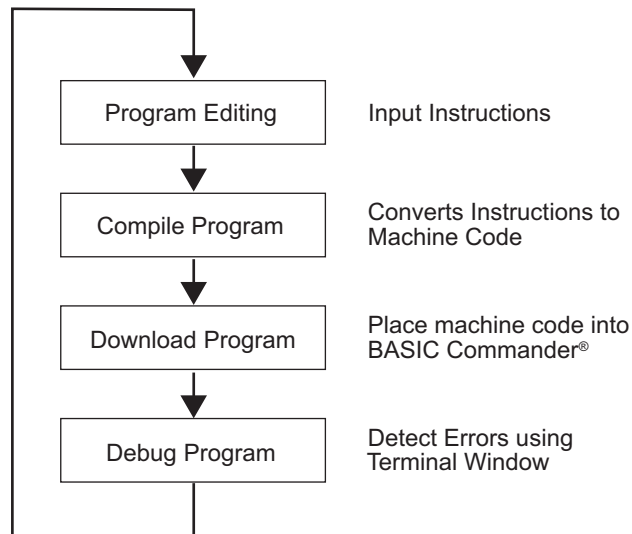
Figure 3-3 Program Creation Steps

Editing the Program

Similar to a simple text editor or word processor, the Program Editing Window is first used to input your program commands line by line. When you input, try and type in everything correctly, however don't worry too much if you make errors, these will be picked up later by the syntax checker. We also recommend you don't type in your whole program right away. Try a few statements at first, download it and run to see that everything is going according to plan. By breaking things up into separate application units, the overall project becomes much easier to manage, and with one section running correctly you can confidently move on to the next section.

Compiling and Building the Program

After your program has been entered into the innoBASIC™ Workshop, it must be converted into what is known as machine language before being downloaded into the BASIC Commander®. This process of conversion to machine language is known as Compiling; however before conversion it will first check the program contents for errors. If errors such as wrongly typed instructions are entered, the system will produce a corresponding error message, which will be displayed in the Output Window. Double Clicking on the error message in the Output Window will indicate in the Program Editing Window in which line the error is located. The error message itself should also contain some information regarding the nature of the error. The process known as "Build", will actually repeat all the process steps in the compiling action but will then take the compiled program and after some manipulation transmit it to the BASIC Commander® hardware via the USB cable link. During program editing, as users may not always want to send their program to the BASIC Commander®, and may only wish to check for correct program entry, the Compile and Build functions are provided with their own separate Icons.

Debugging your Program

Well in theory, after finishing your hardware and writing your program, it should just be downloaded into the BASIC Commander® and after running, everything would be fine. It doesn't quite work out that way, as unless the program is a very simple one, it is rarely likely to work the way you want it the first time. There will invariably be errors within the program that will have to be removed to achieve the desired application result, a perfectly normal process known as debugging. Thankfully the innoBASIC™ Workshop is equipped with debug tools, for which the Terminal Window is heavily utilized to assist with the highlighting of problem areas and to get your program up and running in as short a time as possible. The debugging of programs, seen by some as the real fun part of project development, is invariably an invaluable learning opportunity, and is in fact the area where you will learn the most about microcontroller systems. Patience and clear thinking during this stage will be well spent and lead to worthwhile experience for future and more complex projects.

Menus and Commands

This is the core of the innoBASIC™ Workshop and the place where a greater part of your development work will be done. With a format very similar to that of other Windows based programs, most users should be well acquainted with their style. The innoBASIC™ Workshop window is subdivided into several sections, which you should become familiar with to get the best use out of the system.

File Menu

This is virtually identical to most other Windows based programs with the usual Open, Close, Save, Print functions, etc. A useful feature is the list of recently opened files allowing easy and rapid selection.

Edit Menu

With the usual Windows Cut, Copy, Paste, Find and Replace functions are also the Redo and Undo function to allow easy recovery of typed errors. There are some additional functions in this menu however that requires further explanation:

Comment Selection

This useful command enables users to easily, with a few clicks of the mouse, comment out a single line or multiple lines of the program. By making a line of program a comment, it provides a simple method of forcing the program to ignore these instructions when the program is run, which is an extremely useful technique during program debug. Of course the same result can also be obtained by typing the comment instruction operator, which is a single quote character, in front of each statement.

Uncomment Selection

This command simply removes the comment operator from any program instruction lines which have been previously setup as comments. It forms a useful way of removing multiple lines of comments from a program without having to individually edit each line.

Goto Line

When programs are large it can be quite a task to jump to other locations, however if the line number of the program is known, this menu command can be used to directly jump to a specific line. The line numbers of the program will be generated automatically by the system and can be displayed by selecting the Line Counter option from within the View Menu or by selecting its own icon in the Editor Bar.

View Menu

Here is located the controls for whatever sections of the innoBASIC™ Workshop you wish to display or hide. By selecting the listed items, various menu bars or functional windows can be displayed or hidden. One worthy of a special mention is perhaps the Line Counter, which can display line numbers next to each instruction in your program. These numbers are automatically generated by the software and cannot be changed by the user. Note also that for the window display controls, an icon also exists in the Editor Bar, providing an easier control option for the displaying or non-displaying of the functional windows.

Build Menu

Here you can find the controls to compile your program and to download it from the PC into the BASIC Commander®.

Compile

During the early stages of program editing and debug, it is not usually necessary to always download the program, hence, under the Build main menu is the option to compile only. The compiling action takes the innoBASIC™ instructions from your program and transforms them into machine code language. Any errors in your program such as spelling or wrong specifications will be indicated in the Output Window. The source of the error in your program can be located by double clicking on the error message in the Output Window, which will place the cursor on the program line containing the error. By providing this compile only menu command you can save time by not running the download process, if all you want to do is check the basic syntax of your program.

Build

The other option, which is known as Build, will in addition to compiling your program also give you the option to download it to the BASIC Commander®.

Tools Menu

Here you can find the settings of Editor, Fonts, Colors and Terminal Window. Please click on the tags for their relevant settings.

Fonts Setting

The fonts option in the tools menu, is as the name suggests, a means to control the size and style of the fonts used in your program. This option is for the fonts in the Program Editing Windows.

Print Fonts Setting

This option allows you to choose the style and size of the fonts for printing.

Preferences

The Preference option in the Tools Menu allows you to customise the operation of the innoBASIC Workshop to your own liking. The Editor preferences offer you various choices such as whether you wish to display line numbers on your program and if you would like a color bar to indicate the present cursor line. The Fonts preferences offer you a means to control the size and style of the fonts used in the Program Editing Windows. The Colors preferences allow you to customise your Workshop Window in your own color scheme. The Sample Box will enable you to preview your color choices before finalising your choice by clicking on Enter. The Set Default option will allow you to return to the colors chosen when the system was first installed. The Terminal Window preferences allow you to set the maximum display lines and characters in each line.
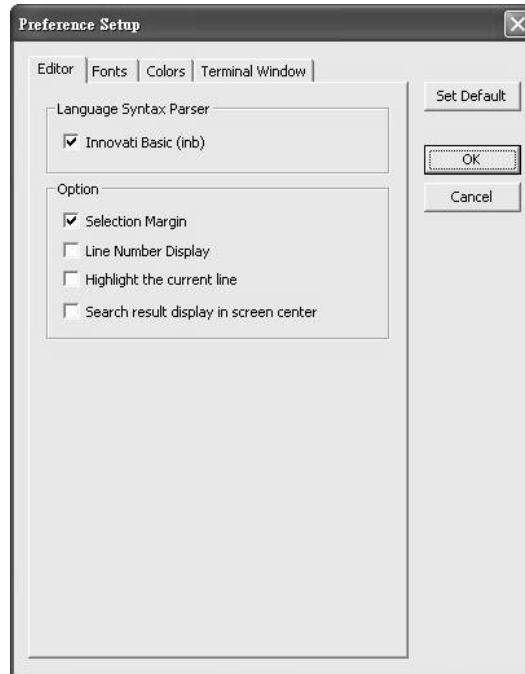
Figure 3-4 Tools Menu Preference Option

Window Menu

The conventional window menu functions are also provided here. You can cascade or tile the windows to help you reference among different files. You can use the new window function to open another identical file or use the split function to split the file into windows, which helps you to reference the context in the same file. You may also select "Tabbed MDI" to display the file in a tabbed format, which saves space when many files are opened.

Help Menu

In the Help Menu, you can use the Help Topics function to find the terms or explanations about the functions that you may encounter when working with the InnoBASIC™ Workshop. It is meant to be provided as a quick reference, if topics are either not mentioned or not explained clearly, please refer to the user's manual for more detailed information. You can use the About function to check the version of the InnoBASIC™ Workshop you are currently using.

4

Hardware Description

Introduction

Here you find information on the Innovati® system hardware, the most important part of which is the BASIC Commander®. This is what is known as a Single Board Computer or SBC for short. In addition to this is the Education Board, into which the BASIC Commander® can be plugged and to which additional electronic components can be connected to expand your program possibilities. And then there are the modules, which are the feature packed hardware units that offer greatly enhanced intelligent control to your projects.

BASIC Commander® Module

The BASIC Commander® is a complete and fully self-contained Single Board Computer. Often known by its abbreviated name of SBC, it is the unit that provides control for your total project. At its heart is a high quality industrial grade commercial microcontroller complete with memory, clock circuits, power control and USB interface. Its outline is shown in the figure.
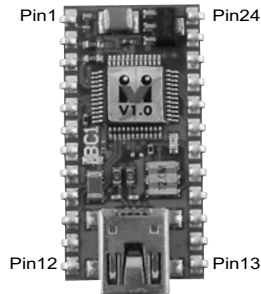
Figure 4-1a 24-pin BASIC Commander®
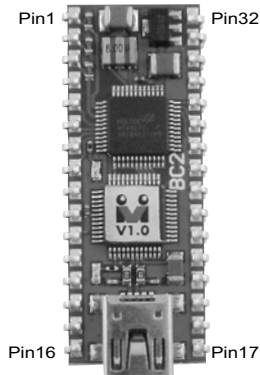
Figure 4-1b 32-pin BASIC Commander®

When the PC is connected to the BASIC Commander® using the supplied USB cable, its power supply will be sourced from the PC.

BASIC Commander® Pinout

The BASIC Commander® comes in two forms; 24-pin and 32-pin modules. The width between the pins are the standard 0.1 inch pitch, which means the modules can be conveniently plugged into standard PCB sockets and standard breadboarding circuit boards. Each pin has its own individual functions, please see the table below.

24-Pin	32-Pin	Pin Name	Pin Function
1	1	SDA	cmdBUS™ Data Pin
2	2	SCL	cmdBUS™ Clock Pin
3	3	EVT	cmdBUS™ Event Pin
4	4	SYN	cmdBUS™ Synchronization Pin
5~12	5~12	P7~P0	General Purpose I/O pins
-	13~16	P23~P20	General Purpose I/O pins
-	17~20	P16~P19	General Purpose I/O pins
13~20	21~28	P8~P15	5V Power Pin (or regulated 5V of VIN or USB)
21	29	VCC	5V Power Pin (or regulated 5V of VIN or USB)
22	30	RES	Reset Pin of the BASIC Commander®
23	31	GND	Ground Pin-Common Ground
24	32	VIN	External Power Supply (unregulated)

If you use a single BASIC Commander® in your application, be sure to connect the unregulated 6~12 VDC power to pin VIN or connect a regulated 5V power supply to the VCC pin. The SDL, SDA, EVT, SYN along with the VCC and GND pins constitute the cmdBUS™. Keep the cable or wiring length as short as possible, otherwise the extra capacitance introduced by long cables or wiring may eventually slow down the cmdBUS™ performance. A maximum of 32 Peripheral Modules can be connected to the cmdBUS™.

The BASIC Commander® hardware has two LEDs, which are off during normal operation. When the green LED is lit, it indicates that communication between the PC and the BASIC Commander® is taking place, while if the yellow LED is lit, it indicates that communication between the BASIC Commander® and a Peripheral Module is taking place.

Education Board

For most projects, users will require to add some of their own peripheral components. For experimental or learning purposes or perhaps for project designs at an early stage in their development, users may find it convenient to use the supplied Education Board. Here a small breadboard is supplied to provide a convenient means of connecting external components to the BASIC Commander. Components such as external switches, LEDs, resistors, capacitors, and potentiometers can be conveniently connected here eliminating the need for soldering and allowing easy debugging and adjustments. Here the breadboard supplied can be used to easily interconnect external components to the BASIC Commander and for quick and easy changes to be made without the inconvenience of soldering or re-soldering. The external components connect to the BASIC Commander through the female headers next to the breadboard.
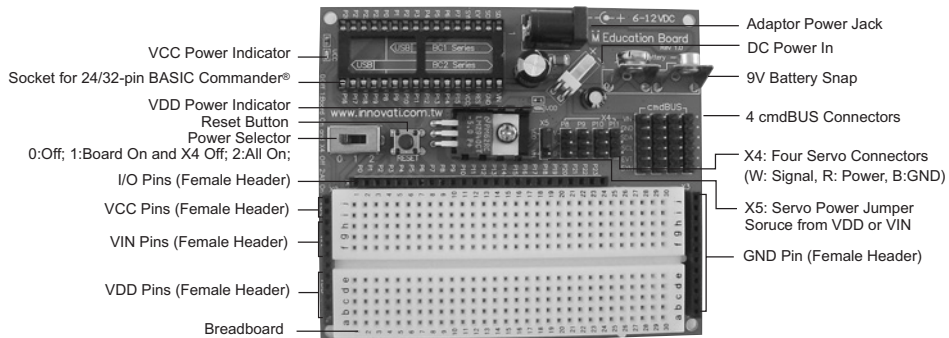
Figure 4-2 Education Board

The BASIC Commander® should be inserted to the 32-pin socket before the power is turned on. Please note that the BASIC Commander® comes in two forms, 24-pin and 32-pin. When inserting the BASIC Commander®, check the drawing on the board for correct direction and alignment.

Take exceptional care to ensure that the BASIC Commander® is inserted correctly into the Education Board socket, otherwise irreparable damage may occur. Only insert the BASIC Commander® with the power off and double check for correct insertion before applying power.

Power Supply

The power supply can come from one of three sources. First, it can come from the USB port, where the power is denoted as VCC, which you can find by the pin header next to the breadboard. However, if the power is supplied through the USB port directly, a maximum of 500 mA can be drawn from the USB Port after they are configured by the host system software, but must not draw more than 100mA until they are configured. Therefore, for more power consumptive applications, power should be supplied externally. When the VCC from the USB port appears, an LED indicator next to the BASIC Commander® socket will illuminate.

Secondly, power can be sourced from an external power supply connected to pin VIN, which may come from a direct power input to the white 2-pin connector, or from a power adapter with a rating of 6-12VDC with central positive, or a 9V Battery connected to the battery snap connector. No matter which external power resource is used, it is regulated to 5 Volts and denoted as VDD. The maximum rating current supplied by the regulated VDD is 1 Amp. When the VIN voltage is sourced from an external source, an LED indicator near the middle of the Education Board will illuminate.

The GND pin is, as the name suggests, the common ground of the system and is common to all of the different power supply sources.

The X4 Connector has 4 Servo Headers, where you may plug in up to 4 standard servos. The control signals are from Pins 8~11. The power to the servo may come from either VDD or VIN by inserting a jumper at Header X5. To reduce the power

consumption of the 9V battery and to eliminate the need for frequent plugging-in and plugging-out of the servo cables, you may set the Slide Switch from Position 2 to 1, which will turn off the power supply to the servos.

The slide switch on the Education Board is used to control the external power supply. When on Position 0, VDD/VIN/VCC are not available, when on Position 1, the VDD/VIN/VCC are available on the female headers around the breadboard, but the VDD or VIN on connector X4 is not available and when in Position 2, both the VDD and VIN on connector X4 are available. When VDD and/or VCC is available, their respective LED indicator will light. Note that the VCC power source may come either from VIN, which will be regulated to 5V on the BASIC Commander® board or directly come from USB power. If the USB cable is connected to the BASIC Commander®, even the slide switch is in position 0, the VCC on Female Header is still available.

Position	VDD/VIN/VCC (on Female Headers)	VIN or VDD (on Connector X4)
0	Off	Off
1	On	Off
2	On	On

The RESET push button is used to reset the BASIC Commander®, which will restart the program execution.You can also find pin-headers labelled cmdBUS, where you may connect up to four of Innovati® Peripheral Modules through the 6-wired cmdBUS™ cables.The following table gives a summary of the connectors or switches you may find on the Education Board.

Connector	Function
32-pin Socket	For 24-/32-pin BASIC Commander?
Power Jack	For 6~12 VDC adapter, center positive
White 2-pin Connector	For 6~12 VDC, power supply
Battery Snap	For 9V Battery
Slide Switch	Power On/Off and Servo Power On/Off
Push Button	Reset Button
X4	Four Servo Connectors
X5	Servo Power Selection
cmdBUS	Four Module Connectors

The Education Board has two LEDs. The one by the BASIC Commander® socket will light when the USB power bus is on and when the USB cable is connected. The other one near the center of the Education Board will light when external power is applied. This power may come from a 9 Volt Battery, power adapter or a power supply from the white 2-pin connector.

Peripheral Modules

No longer is it necessary to select and purchase individual electronic peripheral components and get involved in the time consuming task of circuit board construction and complex interfacing. Innovati® has taken care of these technical related issues by pre-building these peripheral modules and providing you with sophisticated control commands for their operation. Let's take the LCD2X16A Peripheral Module shown in the diagram as an example.
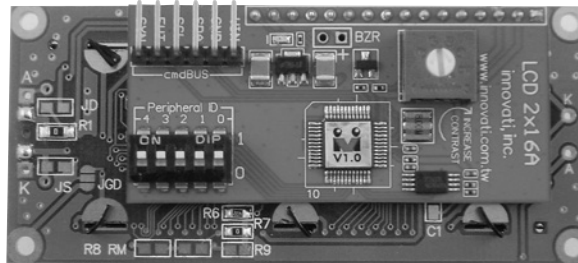
Figure 4-3 LCD2X16A Peripheral Module

Simply setup the DIP switch on the small add-on board, which is used to setup the Module ID. The address can range from 0 to 31, and each peripheral module should have its own unique address. Therefore no two modules should have the same DIP switch ID address on the cmdBUS™. On each Peripheral Module, there is a cmdBUSTM, and a 6-pin pin header labelled cmdBUS™. Connect the module through the flat cable provided to the cmdBUSTM on the Education Board.

VIN (RED)
GND (BLACK)
SDA
SCL
EVT
SYN

Figure 4-4 6-wired cmdBUS™ Cable

The VIN pin is the unregulated external power supply, rating 6~12 VDC, which will also be regulated down to 5 VDC on each Peripheral Module for its internal power requirements. Note that great care should be taken when using this cable as a wrong cable insertion may seriously damage the devices which are connected to the cmdBUS™. After the modules are declared in the program and if the module ID in the program is the same as the hardware switch, it will then be ready for immediate use. Of course, each module comes with its own instruction manual and software driver for installation.

Handling Precautions

When handling the BASIC Commander®, it is important to take the proper anti-static precautions. This of course applies to most other ICs which can be damaged by the static charge built up in certain environments, such as those of low humidity or between certain materials. The BASIC Commander® is supplied in anti-static packaging, which should be used for transportation or storage. Also ensure that you are working in a grounded area when developing your project, which will eliminate the building up of any damaging stray charge. The best method is by using a grounded mat and grounded straps such as seen in professional electronic assembly areas, otherwise if this is not possible then ensuring that you are personally grounded perhaps by touching some grounded object before handling the BASIC Commander® would do much to protect from anti-static damage. When the BASIC Commander® is connected to your PC via the USB cable or after it is inserted into your actual project's hardware and is part of an overall working circuit, then anti-static issues should not be a problem.

InnoBASIC™ Programming Language

5

Introduction

The innoBASIC™ is a high-level programming language for Innovati's BASIC Commander® system. Although it is designed to be an easy-to-learn language, it is also powerful enough to meet the requirements of experienced users. For those not familiar with programming languages, this is the place to start learning from the beginning. Firstly, we'll show you an example program to welcome you into the innoBASIC™ world.

```
Sub Main()
Debug"Hello World!"
End Sub
```

Statements

A program consists of statements, which give instructions to the compiler to generate the final executable code accordingly. Statements may consist of constants, variables, operators and functions to define constants, declare variables, perform arithmetic and logical operations and execute program control transfer and declare subprograms.

Normally, each line contains one statement, which can either be a single or compound statement. However, when statements are short and for reading convenience, multiple statements can be placed in one physical line, however each statement should be separated with its preceding statement by a colon ":" symbol. If a statement is long to the extent that it may cause reading inconvenience, a single underscore "_" preceded with a white-space character is used for line continuation, which allows you to span a logical line to multiple physical lines. Line continuations are treated as if they were space. The following program shows these two statement formats.

```
Sub Main()
Debug _
"Hello World!"
End Sub
```

Note that line continuation is not applicable to comments and string literals.

Comments

An apostrophe is used to denote a comment, for example:

```
Dim a As Short              'Here is the Comment!
```

All characters to the right of the apostrophe are regarded as comments and as such will be ignored by the compiler, unless the apostrophe is part of a string literal. Note that comments cannot span to multiple lines by using line continuations, and no block comment command is available.

Identifiers

An identifier is a name. InnoBASIC™ identifiers must start with a letter, and followed with characters, digits or underscores. An identifier can be up to 31 characters long. InnoBASIC™ is not case sensitive, therefore identifiers xyz, Xyz and XYZ are all equivalent. Keywords cannot be used as an identifier.

Keywords

A keyword is a word that has special meaning in the innoBASIC™ language. Keywords are reserved and may not be used as identifiers. Please refer to Appendix B where all the keywords reserved in the innoBASIC™ language are listed.

Labels

Label declaration statements must appear at the beginning of a logical line and labels should be a legal identifier and must always be followed by a colon. It marks the start of statements the program execution may branch to from a GOTO statement.

Labels have their own declaration space and do not interfere with other identifiers. The scope of a label is the body in which the label is declared

Constants, Variables and Data Types

A constant is a constant value, which never changes during program execution. It is declared with the keyword CONST. For example;

CONST *Constname* As *Type* = *value*

The following examples demonstrate the usage.

```
Const DaysofMay As Byte = 31
Const JAN As String *7 = "January"
Const Scores(4) As Byte = {70,75,80,85,90}
```

The value given should be a literal appropriate for the given type. Constants are implicitly accessible to all program space, so it must be declared as global. You may declare a constant array, but the elements cannot be another const string or array.

In contrast to the constant is the variable, which conveys a value that may be changed during program execution. All the variables are declared with the key word DIM, for example;

DIM *Variablename* As *Type* [= *value*]

Before any variable is used in your program, the system must first be told that it exists as well as what kind of variable it is and in some cases its size. The variables can be initialized when it is declared. If not, Default 0 value or null string for the variables are initialized. Conventionally, the variable primitive data types are Boolean, Byte, Short, Word, Integer, DWord, Long and Float type which are assigned within the RAM data memory. If DIM is used within any procedure, the variables declared are local variables which mean they are visible only inside the variables which can be seen only inside the procedure where they are declared. On the contrary, if DIM is used outside of all procedures, the variables are global, which means they are visible throughout the entire program.

Note that if the variables are declared as global, their initial value cannot be assigned when declared, they must be given in the program. If they are declared as local variables, they don't have such constraints.

Variable Type	Size	Description
BOOLEAN	1 Byte or 1 Bit	Variable commonly used as a status flag or Boolean result. When declared as local variable, 1 Byte is used, when declared as global variable, 1 Bit is used.
BYTE	1 Byte	Unsigned variable for values 0~255.
SHORT	1 Byte	Signed variable for values -128~+127.
WORD	2 Bytes	Unsigned variable for values 0~65535.
INTEGER	2 Bytes	Signed variable for values -32768~+32767.
DWORD	4 Bytes	Unsigned variable for values 0~4294967295.
LONG	4 Bytes	Signed variable for values -2147483648~+2147483647.
FLOAT	4 Bytes	Floating point variable for value -3.4E+38~+3.4E+38.
STRING	N Bytes	A sequence of zeros or more ASCII code characters enclosed by the ASCII double-quote character at the beginning and end. The size N of the string is assigned by the user and is limited by the available RAM size. In a string, a double-quote character should be expressed in 2 double-quote characters.
PERSISTENTBYTE PERSISTENTSHORT	1 Byte	Variable stored in non-volatile EEPROM. Due to the limited operations, the two data types are actually the same.
PERSISTENTWORD PERSISTENTINTEGER	2 Bytes	Variable stored in non-volatile EEPROM. Due to the limited operations, the two data types are actually the same.
PERSISTENTDWORD PERSISTENTLONG	4 Bytes	Variable stored in non-volatile EEPROM. Due to the limited operations, the two data types are actually the same.
PERSISTENTFLOAT	4 Bytes	Floating point variable stored in non-volatile EEPROM for value -3.4E+38 ~ +3.4E+38.

Type Conversions

The general rule for type conversion is to convert a narrower operand into a wider one, promotion, without losing information, such as converting a Short into an Integer or a Long type variable. If conversion is in the opposite direction, clipping, extra high bytes will be dropped out. Cautions must be taken when this kind of type conversion is utilized.

The subsequent subsections shows all the variable types that are supported in the InnoBASIC™ programming language along with the size occupied and brief introductions to them.

Literals

A literal is a textual representation of a particular value of a type. Literal types include Boolean, integral number, floating point, character and string.

Boolean Literals

True and False are literals of the Boolean type that map to the true and false state. They have the value 1 and 0, respectively.

Integral Literals

Integral literals can be decimal (10-based), hexadecimal (16-based), octal (8-based) or binary (2-based). A decimal literal is a string of decimal digits (0-9) and no prefix is needed. A hexadecimal literal is &H followed by a string of hexadecimal digits (0-9, A-F). An octal literal is &O followed by a string of octal digits (0-7). Binary literal is &B followed by a string of binary digits (0 or 1). Decimal literals directly represent the decimal value of the integral literal, whereas octal and hexadecimal literals represent the binary value of the integer literal.

Floating-Point Literals

A floating-point literal is an integer literal followed by an optional decimal point (the ASCII period character) and mantissa, and an optional base 10 exponent. Due to the limitation of memory and computational speed, a floating-point number is represented only in 4 bytes. The valid number of digits is 5, therefore the floating-point operations are sufficient for common practices, but not recommended for high precision calculations.

String Literals

A string literal is a sequence of zeros or more ASCII code characters beginning and ending with an ASCII double-quote character. Within a string, a sequence of two double-quote characters is an escape sequence representing a double quote in the string. In consideration of the limited RAM resources, a proper string size should be given when declared.

Character Literals

There is no specialized data format for a character. It is represented by a single ASCII code character. To distinguish a character literal and a single-character string, a letter c postfix, which means a character, is required.

```
myChar ="H"c    'stands for a single character H
myString ="H"   'stands for a string with one character H
```

Array

An array contains variables of the same type with the exception of bit (Boolean), string types or another array. Each of them is accessed through indices, and is called the elements of the array. If an array has more than one index, say two indices, it is called a two-dimensional array. Note that when declaring, the number(s) in the

parenthesis denotes the maximum index number, instead of the number of elements. The index starts from 0. The array can be initialized when declared as shown in the following example.

```
Dim myarray(5) As Short = {1,2,3,4,5,6}
```

The following example shows how to use an array to make a 9 by 9 multiplication table.

```
Sub Main()

    Dim m(8,8) As BYTE
    Dim I As BYTE
    Dim J As BYTE

For I=0 to 8
For J=0 to 8
    m(I,J)=(I+1)*(J+1)
    Debug m(I,J)," "
Next J
    Debug CR
Next I

End Sub
```

The Terminal Window will show a multiplication table. Note that the size of a single array, no matter its dimensions, is limited to 100 elements.

Operators

There are two kinds of operators. Unary operators take one operand and use a prefix notation. Binary operators take two operands and use an infix notation. Note that for unary operators, they are right-associative whereas the binary operators are all left-associative, which means that operations are performed from left to right. When an expression contains multiple operators, the precedence of the operators controls the order in which the operators are evaluated. Nevertheless, precedence and associativity can be altered by using parentheses.

The following table lists the binary operators in descending order of precedence

Category	Operators
Unary plus and minus	+, -
Multiplication and division	*, /,\ , MOD
Add and Subtract	+,-
Shift	<<, >>
Relational	=, <>, <, >, <=, >=
Bitwise AND, OR, XOR and Complement	AND, OR, XOR , ~
Logical NOT, AND and OR	NOT, AND, OR

The different types of operators are introduced in the subsequent sections.

Arithmetic Operators

There are eight arithmetic operators,

> \+ addition
>
> \- subtraction
>
> * multiplication
>
> / division (the floating-point division)
>
> \ division (the integral division)
>
> MOD modulus (the remainder of integral division)
>
> << shift left (analogue to multiplication of 2's)
>
> >> shift right (analogue to division of 2's)

Relational Operators

The relational operators compare two values and return a TRUE (1) or FALSE (0) result according to the relational comparison.

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

Bitwise Operators

There are four operators for manipulating bit-operations, namely, AND, OR, XOR and ~ (complement). The unary ~ yields the 1's complement of a value by converting every bit. The AND, OR and XOR operators execute a bitwise operation of two operands:

Logical Operators

The logical operators support the logical operations AND, OR and NOT. They create a TRUE or FALSE value. Expressions connected by AND and OR are evaluated from left to right. The evaluation stops as soon as the result is known. The numeric value of a relational or logical expression is 1 if the relation is true, and 0 otherwise. The unary negation operator NOT converts a non-zero operand into 0 and a zero operand into 1.

Assignment Operators

There are 6 assignment operators for expression statements. For simple assignment, the equal sign is used with the value of the expression replacing the variable, in the left operand. The remaining 5 are compound assignment operators. Taking A += B for example, it is equal to the expression A = A + B, and so on.

```
=
+ =
- =
* =
/ =
\ =
```

Program Control Flow

Program statements are executed in the order that they appear in the program source file. Changing this natural execution sequence, can be achieved by conditional statements, unconditional statements and subprogram invocations. The conditional statements allow conditional execution of statements based on expressions evaluated at run time. There are four kinds of statements in this category, IF...THEN...ELSE Statements, SELECT...CASE Statements, DO...LOOP and FOR...NEXT statements. The unconditional statements are the GOTO and invocation statements.

Conditional Statements

IF...THEN...ELSE Statements

The IF...THEN...ELSE statement is one of the most basic flow control statements. It evaluates an expression which must be implicitly convertible to Boolean. If the expression in the IF statement is True, the statements enclosed by the IF block are executed. If the expression is False, then the statements in the ELSE block are executed. More conditions can be evaluated by using the ELSEIF statement, when each of the ELSEIF expressions is evaluated. If one of the ELSEIF expressions evaluates to True, the corresponding block is executed. Once a block has been executed, the execution passes to the end of the IF...THEN...ELSE statement.
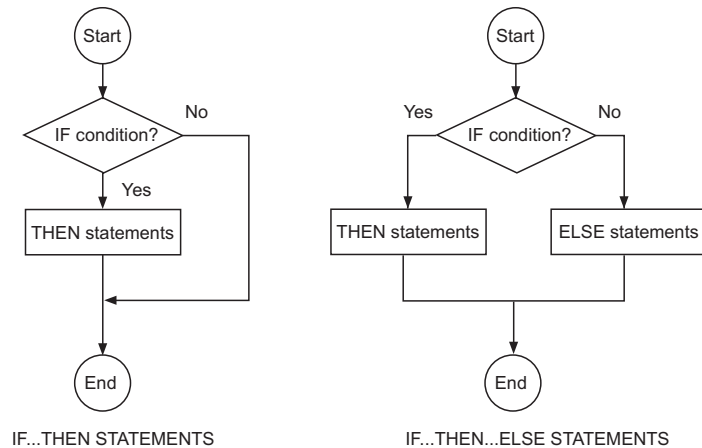
Figure 5-1 IF...THEN...ELSE Statements Flow Diagram

IF...THEN...ELSE Statements has a single statement in the IF block and a single statement in the optional ELSE statement, then the line version of the IF statement is applicable. For example:

```
Sub Main()

Dim a,b,Max As Integer

Debugin "Enter the First Number:", a
Debugin "Enter the Second Number:", b

If a < b Then
    Max = b
Else
    Max = a
End If

Debug "Max is:", Max

End Sub
```

SELECT...CASE Statements

The SELECT...CASE command is an advanced compound decision-making structure of IF...THEN...ELSE structures, which have the same comparison expression, to execute one of several possible actions. When a Select command is executed, the value is compared with the Case constant in the textual declaration order. If the first Case const meets the evaluated value then its block will be executed. If no Case const meets the evaluated value and the optional Case Else statement exists, that Case Else block will be executed. Once a block has finished executing, execution passes to the end of the SELECT...CASE command.
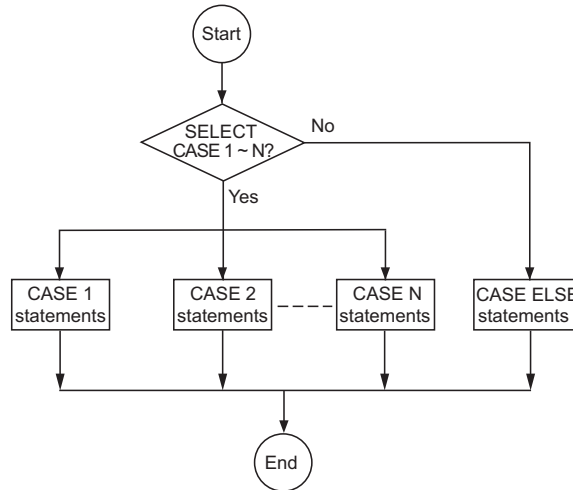
Figure 5-2 SELECT...CASE Statements Flow Diagram

Note that the optional CASE can come with SELECT in the conventional style. The following example illustrates this behaviour:

```
Sub Main()
    Dim x As Byte
    Do
    Debugin"Enter a 1 to 7 to find the nth day of a week.", x, CR
    Select Case x
      Case 1
         Debug "It's Sunday.", CR
      Case 2
         Debug "It's Monday.", CR
```

```
      Case 3
        Debug "It's Tuesday.", CR
      Case 4
        Debug "It's Wednesday.", CR
      Case 5
        Debug "It's Thursday.", CR
      Case 6
        Debug "It's Friday.", CR
      Case 7
        Debug "It's Saturday.", CR
      Case Else
        Debug "Wrong Number!", CR
      End Select
      Loop
End Sub
```

DO...LOOP Statements

During programming, if it is required to execute a program block repeatedly, then one of the most efficient ways of doing so is to use the DO...LOOP command.

DO [{WHILE | UNTIL} *condition*]
[*statements*]
[EXIT DO]
[*statements*]
LOOP [{WHILE | UNTIL} *condition*]

The basic DO...LOOP command will constitute an infinite loop, with the commands enclosed by DO and LOOP to be executed forever. You may add the WHILE qualifier at the beginning or the end of the loop, but not after both, whereas the Boolean condition will be tested on each iteration. As long as the condition is true,

the loop statement block will be executed. If the WHILE is placed at the end of the loop, the loop statement block will be executed at least once, then the Boolean condition will be tested. The UNTIL qualifier is similar to the WHILE qualifier, except that the loop is terminated rather than continuing when the Boolean condition is true. The EXIT DO command may be placed in the loop body which exits the current loop immediately before the loop limit test is executed.
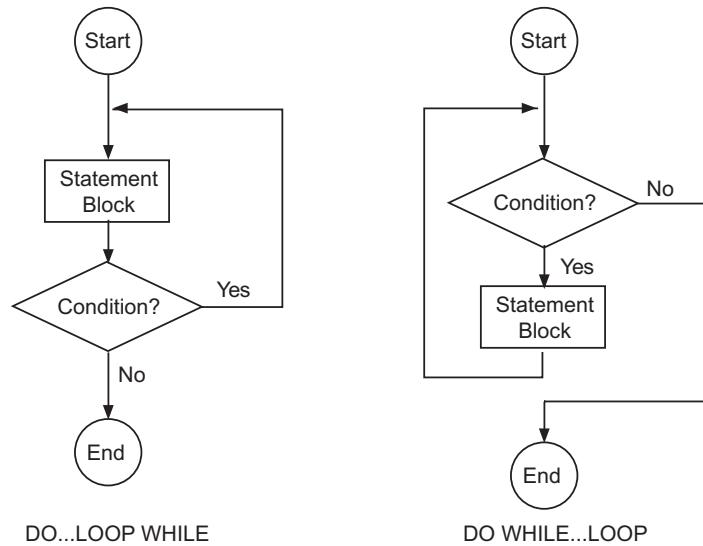
Figure 5-3a DO...LOOP Statements Flow Diagram
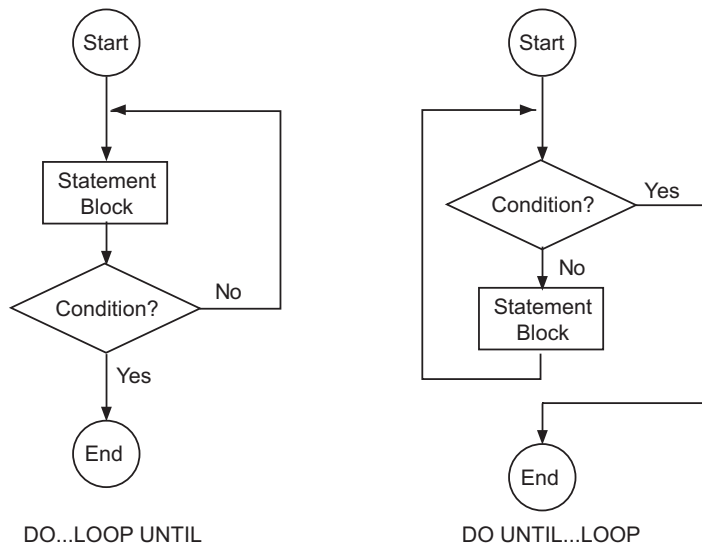
Figure 5-3b DO...LOOP Statements Flow Diagram

The following example demonstrates this behaviour:

```
Sub Main()
    Dim x As Short
    x = 1
    Do While x<5
      Debug "*"
      x+=1 'display 4 asterisks
    Loop
    Debug CR
```

```
x = 1
Do
   Debug "*"
   x+=1 'display 4 asterisks
Loop While x<5
Debug CR
x = 5
Do Until x=0
   Debug "*"
   x-=1 'display 5 asterisks
Loop
Debug CR
x = 5
Do
   Debug "*"
   x-=1 'display 5 asterisks
Loop Until x=0
End Sub
```

FOR...NEXT Statements

You can use a FOR...NEXT statement to execute a block of codes, when you know how many repetitions you want. You can use a loop control variable that increases or decreases with each repetition of the loop.

A For statement specifies a loop control variable, a start value, an end value, and an optional step value. At the beginning of the loop, the lower bound value is assigned to the control variable. The statements enclosed by FOR...NEXT are executed, and then the program returns to the beginning of the loop while the loop control variable increases or decreases with the value in the STEP clause. If the step value is omitted, it is implicitly the literal 1. The control variable is now checked with the end value, if

it is past the end value, the loop will be terminated and the program branches to the statement after the NEXT statement. If you want the loop control variable to increase, then the STEP value must be a positive value and the end value should be no less than the start value. Contrarily, if you want the loop control variable to decrease, then the STEP value must be a negative value and the end value should be no greater than the start value. If the values are not consistent in direction, the loop will not be executed. The following diagram shows the FOR...NEXT operation.
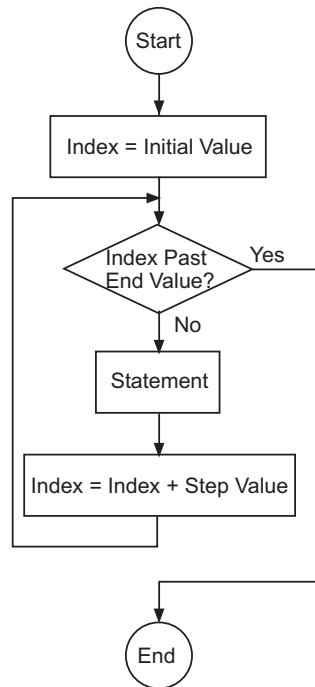
Figure 5-4 FOR...NEXT Statements Flow Diagram

When you use a loop control variable, be careful of its value domain, for example, for a SHORT type variable, its value domain is from -128 to +127, while for a BYTE type variable, its value domain is from 0 to 255. In addition, if the value overflows or underflows due to the STEP value increment or decrement, the value is no longer what you would expect.

The loop control variable is specified in advance. The loop control variable of a For statement must be of a primitive numeric type (Byte, Short, Word, Integer, DWord, Long).

```
Sub Main()
    Dim Index As Byte
    Dim Sum As Byte

    Sum=0
    For Index = 1 To 10 Step 1
    Sum+=Index
    Next
    Debug "1+2+...+10=",sum
End Sub
```

A loop control variable cannot be used by another enclosing FOR...NEXT statement. A FOR statement must be closed by a matching NEXT statement. A NEXT statement always matches the innermost open FOR statement. Usually you may omit the loop control variable, except for program reading clarity. However, if you add a loop control variable which does not match the innermost open FOR statement, a compile-time error results.

You can exit a FOR...NEXT statement with the EXIT FOR keyword, but it is not recommended to branch into a FOR LOOP from outside the loop, which may result in loop control errors.

GOTO Statements

The GOTO command will force the program to jump to a user specified location. This location is given by the label name, which follows the GOTO command. The statement following the GOTO command will not be executed. The next command after the GOTO command to be executed will be the one at the label as specified. The GOTO command is therefore used to give the user a direct means of program control.

Invocation Statements

An invocation statement invokes a SUB or FUNCTION preceded by the optional keyword CALL. The program execution is passed to the SUB or FUNCTION procedure and the program execution is returned to the statement following the invocation statement. The incidence of an EVENT procedure can be deemed as an invocation like SUB and FUNCTION except when it is invoked automatically if a certain event occurs. Please refer to the following "SUB, FUNCTION and EVENT" paragraphs for details.

SUB and FUNCTION

A program is made up of at least one Sub procedure with the name "main". The "Sub Main()" statement denotes the procedure where the program starts and the program terminates when it passes to the end of the Main() procedure. Besides the Main() procedure, you may add more Sub procedures, Functions, which makes your program more structured and more code efficient.

The SUB and FUNCTION subprograms may contain arguments, but only the FUNCTION subprograms return values. Usually, the repeatedly used program paragraphs are written in an independent SUB subprogram, which help reduce program space and increase program readability. Meanwhile, the FUNCTION subprogram executes a calculation, in which the result of the calculation is meant to be returned to the program where it was invoked.

Sub Procedures

Sub procedures are methods which do not return a value. Each time when the Sub procedure is called, the statements within it are executed until the matching End Sub is encountered. Sub Main(), the starting point of the program itself, is a sub procedure. When the application starts execution, control is automatically transferred to the Main Sub procedure, which is called by default.

```
Sub Main()
    Display()
End Sub

Sub Display()
    Debug "Sub Procedure Display() has executed."
End Sub
```

Functions

Function is a method which returns a value. Functions are used to evaluate data, make calculations or to transform data. Declaring a Function is similar to declaring a Sub procedure. Functions are declared with the Function keyword. The following code is an example of how to use Functions:

```
Function Max(I As Integer, J As Integer) As Integer
    If I>J Then Return I Else Return J
End Function

Sub Main()
    Dim X, Y, Z As Integer
    Do
    Debugin "Enter the First Number:" , X
    Debug X, CR
```

```
    Debugin "Enter the Second Number:", Y
    Debug Y, CR
    Z = Max(X,Y)
    Debug "The Maximum value is ", Z, CR
    Loop
End Sub
```

Parameters

A parameter is an argument that is passed to the SUB or FUNCTION procedure. Parameters are enclosed in parentheses after the method name in the method declaration. You must specify the types for these parameters. Note that String and an Array cannot be used as parameters to Sub or Function.

Only the procedure Function may pass parameters when invoked and there are two ways to pass them, ByRef and ByVal. If not explicitly expressed, the ByVal is set as default.

A reference parameter is a parameter declared with a ByRef modifier. A reference parameter does not create a new storage location. Instead, a reference parameter represents the variable given as the argument in the invocation. Modifications of a reference parameter directly and immediately impact the corresponding argument. The following example Swap shows how the two reference parameters work:

```
Sub Swap(ByRef a As Integer, ByRef b As Integer)
    Dim t As Integer = a
    a = b
    b = t
End Sub


Sub Main()
    Dim x As Integer = 1
    Dim y As Integer = 2
```

```
    Debug "before: x = ", x, ", y = ", y, CR
    Swap(x, y)
    Debug "after: x = ", x, ", y = ", y, CR
End Sub
```

The program output is:

Before: x = 1, y = 2

After: x = 2, y = 1

Peripheral Module Programming Features

Peripheral Modules are one of the very special features of the Innovati® system. To employ the Peripheral Modules features, several points associated with software programming should be understood. Here is the section where the exclusive usages are introduced.

Declaration of Peripheral Modules

Each type of Peripheral Module has its product name, which is released with the products by Innovati®, Inc. For instance, the 2x16 LCD Module has the LCD2X16A product name. Users can declare a module name as the LCD2X16A type, along with the ID address that appears on the module in question. The ID should be unique ranging from 0 to 31 for each Peripheral Module. Note that this statement should be written outside of any SUB, FUNCTION and EVENT procedure, which makes it a global declaration.

Invocation of Peripheral Module Commands

As the featured function commands differ from module to module, it is therefore necessary to refer to their relevant documentation for instructions on their use. The command is invoked with a simple suffix of the declared name with a dot in between.

```
Peripheral myLCD As LCD2X16A @ 0
Sub Main()
    myLCD.Display("Hi there!")
End Sub
```

EVENT Procedures

This is used to manage real world events whose occurrences are not predictable and also to reduce the inefficient polling which would take up valuable computing resource. To handle such situations, an EVENT feature is provided, which removes the need for users to take care of the events by frequent polling of the Peripheral Modules. To implement this just write down the event handling procedure, which is enclosed in the EVENT and END EVENT statements, and enable the EVENT ability in the main program. After this the programming effort can be used with other more important tasks, however, when the event that you have enabled occurs, your program will then branch automatically to the event procedure.

Declaration of Peripheral Modules EVENT

An event declaration consists of a module name and a valid event name for the module. It is a little different from the convention for SUB and FUNCTION. The Event name consists of two parts conjugated with a dot. The first part is the name of the user-defined module. The second part is the name of the EVENT that is supported by the Module. Users should check the relevant document for the event available. Note that the EVENT procedure has the highest priority when the program is executing. Therefore, when the EVENT procedure is being executed, no other tasks can be executed, a situation that will remain in place until the EVENT procedure has

ended. For this reason, not to remain in the EVENT procedure is a good habit to cultivate in program management. For the EVENT procedure, parameter passing and value returning are not permitted which can be seen from the SUB and/or FUNCTION.

```
Event MyKeypad.KeypressedEvent()
    Dim KeyID as Byte
    MyKeypad.GetKey(KeyID)                'To get the Key ID
    Debug "Key", KeyID,"is Pressed!", CR  'To display
End Event
```

Sample Project Using the Peripheral Modules

Illustrated here is a simple program, which makes use of the Peripheral Modules. With just a few lines of program and in a very short space of time, you can be up and running for both hardware and software. Working on this example by yourself, you will quickly see the power of the overall system and hopefully be impressed by its capabilities and efficiency.
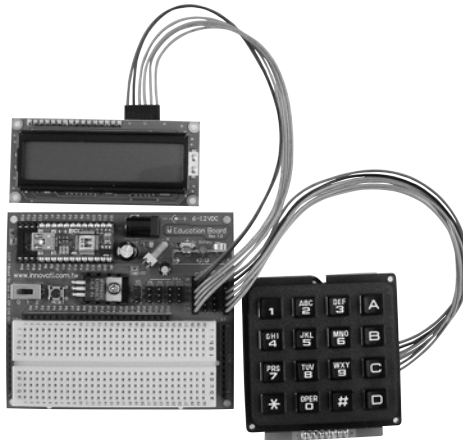
Figure 5-5 Using the LCD2X16A and KEYPADA Peripheral Modules

```
Peripheral myLCD As LCD2X16A @ 0      'set ID switch to 00000 (binary)
Peripheral myKeypad As KEYPADA @ 1    'set ID switch to 00001 (binary)
Sub Main()
    Dim KeyID As Byte
    Dim Status As Byte
    Do
       Status = myKeypad.GetKeyID(KeyID)      'read key
       If Status>0 Then myLCD.Display(KeyID) 'if not pressed, 0 is read
    Loop
End Sub
```

Additional modules as well as discrete components can be added to the breadboard to develop your own project in your own unique way. The possibilities here are only limited by your own creativity and imagination.

Command Set

6

Introduction

In this section you can find the reference material giving full technical details on the innoBASIC™ programming language. The individual commands can be located by using their alphabetical listings.

Types of Commands

Basically there are two kinds of commands that you can write in your program, known as conventional BASIC language commands and the featured system commands. For the conventional BASIC language commands section, our unique featured commands are added which integrates the peripheral hardware modules perfectly seamlessly into the conventional BASIC language. For the system commands section, the featured commands are provided and because they are not common to the conventional BASIC language, they differ from one BASIC language to another. In consideration of experienced users, featured commands from other popular BASIC languages are supported, however minor changes may have been made due to different system considerations. Unless otherwise mentioned, these two types of commands are not distinguished intentionally for simplicity.

Programming Command Conventions

The software commands described in the following section are written in a certain way for which careful attention must be given. The software commands will naturally always contain an essential word describing the basic function. However, following on from this essential word may be other words, which may be essential or optional. By examining the way in which the commands are described in the reference manual their usage can be understood.

Essential words in the command will be written with CAPITAL letters in bold type and italics in bold type. The CAPITAL bold type word must be written exactly as shown, not case-sensitive, whereas the italic bold type words must be replaced with the user values. Non-essential words in the command, or words that the user can decide to add or omit, will be enclosed in curly brackets { }. Only what is contained in the brackets need be written, the actual bracket symbols must not be written. Other square brackets [ ] and parentheses ( ) must be typed in the position of the given syntax. Additionally, a final word must be said about the " | " symbol which represents mutually exclusive elements.

The above conventions of course only apply to the style of writing in the reference manual, since in the actual program, which is a simple ASCII text file, it is not required to write with formatted capitals, bold or italic letters.

Categories

All the commands available in InnoBASIC™ language can be categorized into three kinds. Firstly, the fundamental commands, which constitute the scheme of every program, including declaration, flow control, decision making etc. Secondly, the I/O commands, which use the BASIC Commander® on board I/O pins for various

functions, including basic I/O operation, counter, pulse measurement, communications and other featured commands. Lastly are some mathematical and conversion operation commands. The following table shows the various commands in their respective categories.

Fundamental Commands
CALL
DIM
DO...LOOP
ENUM...END ENUM
EVENT...END EVENT
FOR...NEXT
FUNCTION...END FUNCTION
GOTO
IF...THEN...ELSE
PERIPHERAL
RETURN
SELECT...CASE
SUB...END SUB

I/O Commands
BUTTON
CHECKMODULE
COUNT
DEBUG
DEBUGIN
GETDIRPORT
HIGH
IN
INPUT
LCDCMD
LCDIN
LCDOUT
LOW
OUTPUT
PAUSE
PULSEIN
PULESOUT
PWM
RANDOM
RCTIME
READPORT
RESETMODULE
REVERSE
SETDIRPORT
TOGGLE
WRITEPORT

Mathematical and Conversion Commands	
ABS	FLOOR
ACOS	INTEGER2FLOAT
ASIN	LCASE
ATAN	LEFT
ATAN2	LEN
BYTE2FLOAT	LOG
CEIL	LOG10
COS	LONG2FLOAT
DWORD2FLOAT	MID
EXP	RIGHT
EXP10	SGN
FLOAT2BYTE	SHORT2FLOAT
FLOAT2DWORD	SIN
FLOAT2INTEGER	SQRT
FLOAT2LONG	STRING2FLOAT
FLOAT2REALSTRING	STRREVERSE
FLOAT2SHORT	UCASE
FLOAT2STRING	WORD2FLOAT
FLOAT2WORD	

Command Summary

The following lists all of the software commands in alphabetical order, showing all the details behind each command. This section should form the main programming reference for your application program and the place to consult for issues regarding programming commands.

ABS

Syntax
Result = **ABS***(Argument)*

Operation
To return the absolute value of a floating-point value.
- ●*Argument* ─ the floating-point operand of the ABS function.
- ●*Result* ─ a floating-point variable that receives the result of the ABS function.

Description
The ABS command returns the absolute value of a floating-point value. The result of the ABS is a non-negative value.

Example

```
Sub Main()
    Dim Result As Float

    Result = ABS(2.0)
    Debug "ABS(2.0)=", Result, CR
    Result = ABS(-2.0)
    Debug "ABS(-2.0)=", Result, CR
End Sub
```

ACOS

Syntax
Result = **ACOS**(*Argument*)

Operation
To execute a mathematical inverse cosine function.

- *Argument* — the floating-point operand of the inverse sine function with a range from -1 to 1
- *Result* — a floating-point variable to receive the result of the inverse cosine function. The result ranges from π to 0 radians.

Description
The ACOS function returns the inverse cosine (arccosine) value of a floating-point argument ranging from -1 to 1. The result is in units of radians ranging from π to 0. If converting to degrees, note that 360 degrees is equal to 2π radians.

Example

```
Sub Main()
    Dim Result As Float


    Result = ACOS(0.5)           'the Result is 1.0472
    Debug "ACOS(0.5) = ", Result,"in radians.", CR
End Sub
```

ASIN

Syntax
Result = **ASIN(*Argument*)**

Operation
To execute a mathematical inverse sine function.
- *Argument* — the floating-point operand of the inverse sine function with a range from -1 to 1
- *Result* — a floating-point variable to receive the result of the inverse sine function. The result ranges from $-\pi/2$ to $+\pi/2$ radians.

Description
The ASIN function returns the inverse sine (arcsine) value of a floating-point argument ranging from -1 to 1. The result is in units of radians ranging from $-\pi/2$ to $+\pi/2$. If converting to degrees, note that 360 degrees is equal to 2π radians.

Example

```
Sub Main()
    Dim Result As Float
    Result = ASIN(0.5)          'the Result is 0.52360
    Debug "ASIN(0.5) = ", Result,"in radians.", CR
End Sub
```

ATAN

Syntax
Result = **ATAN(*Argument*)**

Operation
To execute a mathematical inverse tangent function.
- *Argument* — the floating-point operand of the inverse tangent function with a range from -infinity to +infinity
- *Result* — a floating-point variable to receive the result of the inverse tangent function. The result ranges from $-\pi/2$ to $+\pi/2$ radians.

Description
The ATAN function returns the inverse tangent (arctangent) value of a floating-point argument ranging from negative infinity to positive infinity. The result is in units of radians ranging from $-\pi/2$ to $+\pi/2$. If converting to degrees, note that 360 degrees is equal to 2π radians.

Example

```
Sub Main()
    Dim Result As Float
    Result = ATAN(0.5)            'the Result is 0.46365
    Debug "ATAN(0.5) = ", Result,"in radians.", CR
End Sub
```

ATAN2

Syntax
Result = **ATAN2(***ArgumentY, ArgumentX* **)**

Operation
To execute a mathematical inverse tangent function.

- *ArgumentY* — the first floating-point operand of the inverse tangent function which represents the Y coordinate with a range from -infinity to +infinity
- *ArgumentX* — the second floating-point operand of the inverse tangent function which represents the X coordinate with a range from -infinity to +infinity
- *Result* — a floating-point variable to receive the result of the inverse tangent function. The result ranges from $-\pi$ to $+\pi$ radians.

Description
The ATAN2 function returns the inverse tangent (arctangent) value of a pair of floating-point arguments which represents the Y and X coordinates respectively, ranging from negative infinity to positive infinity. The result is in units of radians ranging from $-\pi$ to $+\pi$. If converting to degrees, note that 360 degrees is equal to 2 π radians.

Example

```
Sub Main()
Dim Result As Float

    Result = ATAN2(1,2)              'the Result is 0.46365
    Debug "ATAN2(1,2) = ", Result,"in radians.", CR
End Sub
```

BUTTON

Syntax

BUTTON *Pin, Onstate, Delay, Rate, LoopCounter, TargetState, Address*

Operation

Maintains control over external buttons, and provides options regarding branching and delays.

- *Pin* — a constant or variable (0~23) that specifies the pin number to which the external pushbutton is connected. For the 24-pin BASIC Commander®, the Pin value ranges from 0~15.
- *Onstate* — a constant or variable (0 or 1) that specifies the logical value of the input when the button is pressed. If the input to which the button is connected is normally high and the button pulls it low, the value here should be set to 0. For inputs normally held low and pulled high by the button, the value should be set to 1.
- *Delay* — a constant or variable (0~255) that specifies a time delay until which the button auto-repeat function will be activated. The delay is measured in cycles of the BUTTON routine. If the value here is specified as 0 then no delay or auto-repeat function will be provided. If a value of 255 is specified then a debounce will be provided but no auto- repeat will be provided. This parameter can be used to eliminate the effects of button bounce.
- *Rate* — a constant or variable (0~255) that specifies the number of BUTTON command iterations that will occur between every two auto-repeat operations.
- *LoopCounter* — a byte variable used by the BUTTON command as a loop counter among iterations. Its value should be cleared to 0 before being used by the BUTTON command for the first time and should not be altered by the user thereafter.
- *TargetState* — a constant or variable (0 or 1) that specifies the state of the pin, upon which a branch will occur. If the value is 0, then a branch will occur if the

button in not pressed. If the value is 1, then a branch will occur if the button is pressed.

- *Address* 一 a label that specifies where to branch if the button conforms to the target state.

Description

External buttons are a common feature of most projects and it is this command which provides control over how the button is setup and what should happen when any externally connected buttons are pressed.

Buttons are mechanical devices and when the button is first pressed the internal contacts will bounce back and forth for a few milliseconds before a final and reliable contact is made. During this debounce time, both low and high signals will be detected, which could mean missing valid button presses. To prevent this, usually a small delay of around 20 ms is normally added before the button value is actually read. In the BUTTON command this is achieved using the delay parameter, which after detecting the first button press signal will wait for a specified period, which should be greater than the button bounce period, before looking for other button signals. This time period is specified by the delay parameter which will count down from the specified value each time a BUTTON command is executed. If the button remains in the active state and when the delay counts down to zero, another branch action will take place. At this time the rate counter will start to count down and when zero another branch action will be executed. In this way repeat key functions can be implemented, similar to the way in which a standard computer keyboard operates for repeat key functions. The diagram shows the bouncing action of a mechanical switch when switching from high to low.
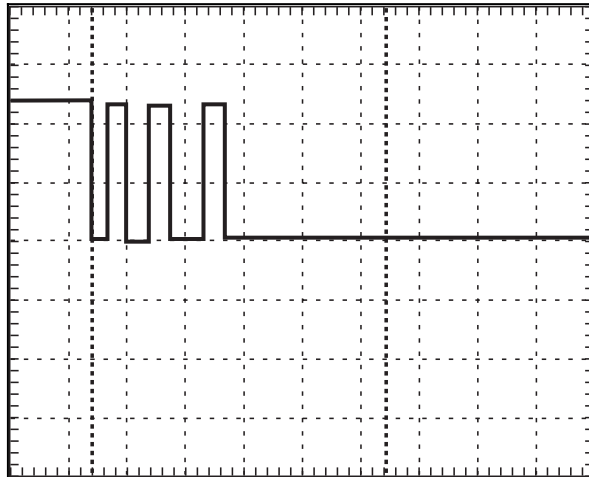
Figure 6-1 Mechanical Switch Bouncing Action from High to Low

Input buttons can be connected so as to bring the input pin to a low condition or to a high condition when pressed. The following two diagrams show both ways of doing this.
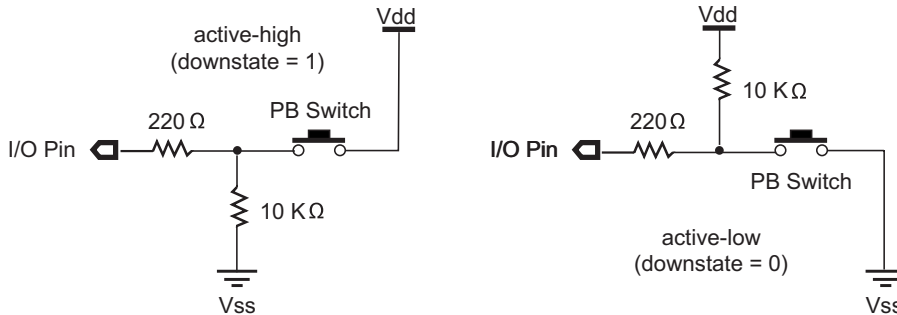
Figure 6-2 Circuitry of Active Low and Active High Buttons

Example

Connect an active-low push button circuit to Pin P0 of the BASIC Commander®. This demo program will display an asterisk "*" on the Terminal Window when you press the button for the first time, then delays for about two seconds before auto-repeat starts. The auto-repeat function continuously sends key pressing signal at a rate of approximately 200 ms (20 x 10 ms PAUSE).

```
Sub Main()
    Dim PIN As Byte
    Dim LoopCounter As Byte=0          'cleared before BUTTON
                                       'command is used
Start:
    Pause 10
    BUTTON 0, 200, 20, LoopCounter, 1, Display
    Goto Start
Display:
    Debug "*"
    Goto Start
End Sub
```

As an exercise try to change the Delay value to different values to see the effect in different modes.

With "0" indicating no delay time, the auto-repeat function starts immediately; the value range "1" to "254" is for various delays before the auto-repeat starts; the value "255" indicates a no auto-repeat function, which is only one action for each button press.

BYTE2FLOAT

Syntax
Result = **BYTE2FLOAT**(*Argument*)

Operation
To convert a Byte value into its floating-point format.
- *Argument* ─ the Byte operand of the BYTE2FLOAT function.
- *Result* ─ a floating-point variable that receives the result of the BYTE2FLOAT function.

Description
The BYTE2FLOAT command converts a Byte value into its floating-point format. The floating-point result will be an integral value ranging from 0.0 to 255.0.

Example

```
Sub Main()
    Dim MyByte As Byte
    Dim MyFloat As Float
    MyByte = 0
    MyFloat = BYTE2FLOAT(MyByte)
    MyByte = 255
    MyFloat = BYTE2FLOAT(MyByte)
    Debug "MyFloat = ", MyFloat, CR
End Sub
```

CALL

Syntax
{CALL} *Name***({***Arglist***})**

Operation
To invoke a procedure with an optional argument list.
- *Name* ― the name of the procedure contains a sequence of letters, digits and underscore. The leading character must be a letter.
- *Arglist* ― a list of the arguments required in the procedure. The argument is preceded with either a byval or byref modifier for argument passing. If no argument is required, the parenthesis can be omitted.

Description
The keyword CALL is optional. When this command is executed the program will branch to the procedure specified by Name. The program will later return to the statement following the invocation statement when it encounters an END SUB or RETURN command in the Procedure in question.

Example

```
Sub SayHello()
    Debug "Hello! ", CR
End Sub

Sub Main()
    Call SayHello()
End Sub
```

CEIL

Syntax

Result = **CEIL**(*Argument*)

Operation

To return the nearest integer that is not smaller than the floating-point argument value.
- *Argument* — the floating-point operand of the CEIL function.
- *Result* — a floating-point variable to receive the result of the CEIL function.

Description

The CEIL command returns the nearest integer (floating-point value) that is not smaller than the floating-point argument value. Usually this is used to round a floating-point value into an integer. Another pairing function is the FLOOR function, which returns the nearest integer (floating-point value) that is not greater than the floating-point argument value.

Example

```
Sub Main()
    Dim Result As Float

    Result = CEIL(2.3)        ' the result is 3.0
    Debug "CEIL of 2.3 is ", result,CR
    Result = CEIL(-2.3)                ' the result is -2.0
    Debug "CEIL of -2.3 is ", result,CR
End Sub
```

CHECKMODULE

Syntax
Status = **CHECKMODULE()**

Operation
To check the Peripheral Module through the cmdBUS™ status
- *Status* — a variable to receive the status. If the Peripheral Module communication is normal, 0 is returned. If the Peripheral Module does not respond within the required period of time, Value 1 is returned. If the cmdBUS™ fails to execute its protocol, Value 2 is returned.

Description
If peripheral Modules are used in the application to enhance system operation reliability, users may use this command to monitor the Peripheral Modules as well as the cmdBUS™ status for malfunctions. This might be due to unpredictable interference from the environment, either electrically or electromagnetically.

If the Peripheral Module communication is normal, a value of 0 is returned. If the Peripheral Module does not respond within the required period of time, a value of 1 is returned. This usually points to the fact that the Peripheral Module in question has failed to operate normally. If the cmdBUS™ fails to execute its protocol, a value of 2 is returned. This is a serious system malfunction on the cmdBUS™, which blocks out the communication scheme of all Peripheral Modules. The reason may be caused by any one of the Peripheral Modules connected to the cmdBUS™ or even by the BASIC Commander® itself. Note that the status checking command always returns the latest status of access to the Peripheral Module, so it should be located immediately after a Peripheral Command. Once a system malfunction is detected, you should take the necessary steps to handle the malfunction, as no built-in recovery methods are available.

Example

```
Peripheral myLCD As LCD2X16A @ 0

Sub Main()
    Dim Status As Byte

    myLCD.Display("Hi")
    Status = checkmodule()

    If Status = 1 Then
    Debug "Module Timeout!",CR
    Elseif Status = 2 Then
    Debug "cmdBUS Error!",CR
    Else
    Debug "Command executed successfully!"
    End If
End Sub
```

COS

Syntax
Result = **COS(*Argument*)**

Operation
To execute a mathematical cosine function.
- *Argument* — the floating-point operand of the cosine function with a range from 0 to 2π
- *Result* — a floating-point variable to receive the result of the cosine function.

Description
The COS function returns the cosine value of a floating-point argument ranging from 0 to 2π. If the argument is out of the range, it is recommended to reduce the argument to fit in the range, otherwise an accumulated error will be introduced. Note that the argument is in units of radians. If converting to degrees, note that 360 degrees is equal to 2π radians.

Example

```
Sub Main()

    Dim myArg As Float
    Dim result As Float

    myArg = pi/4
    result = cos(myArg)                'the result is 0.707107
    Debug "cos(pi/4)=",result,CR
End Sub
```

COUNT

Syntax

COUNT *Pin*, *Duration*, *Variable*

Operation

Counts up the number of edge transitions that appear on a particular pin within a specified time duration and places the value into the indicated variable.

- *Pin* — a constant or variable (0~23) that specifies the pin number where the edge transitions will be counted. For the 24-pin BASIC Commander®, the Pin value ranges from 0~15.
- *Duration* — a constant or variable (1~65535) that specifies the time duration within which the edge transitions will be counted. The unit of Duration is 1 ms.
- *Variable* — a variable of WORD type, in which the count value will be stored.

Description

This command counts up any edge transitions that appear on the pin specified and could be useful for counting up a number of external changing events. Any pin used with the COUNT command will automatically be setup as an input upon execution. The Duration specifies the time duration in unit of 1 ms. Every high-to-low and low-to-high transition will be counted. The minimum width of two adjacent transitions of the input signals should be greater than $10\,\mu$s. In other words, the maximum input signal frequency should be no more than 50kHz for an equal duty square waveform, otherwise, some transitions will not be counted. If the high and low duty are not equal, the shorter duty should be greater than 10 ms. In other words, the maximum input frequency is limited by the shorter duty and will be much less than 50kHz. Note that if the count is greater than 65535, it will overflow to 0 and continue counting. Precautions should be taken for such cases.

Example

This program shows how to use the COUNT command to make an interesting and simple reaction counter game to see how fast a button can be pressed. Remember to connect a push button as shown below.
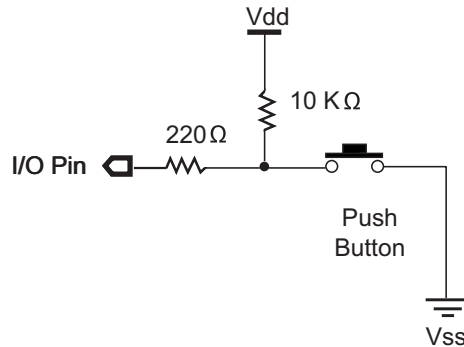
Figure 6-3 Push Button Connection

```
Sub Main()
    DimPushBtn As Byte=0       'Push button on P0
    DimCycles As Word=0        'Counted cycles
    Dim m As Byte=0

Do
    Debug CLS,_
    "How fast can you press within 5 seconds?", CR
    Pause 1000
    Debug "Ready!",CR
    Pause 1000
    Debug "Set!"   ,CR
    Pause 2000
    Debug "Go!"    ,CR
```

```
     COUNT PushBtn, 5000, Cycles

     Debug CR, "Your score is ", Cycles, CR, CR
     Pause 2000
     Debug "Press button to try again!"
     Do
        m=in(Pushbtn)
     Loop Until m=0
Loop
End Sub
```

DEBUG

Syntax
DEBUG *Item* {, *Item*}

Operation
The DEBUG command allows the BASIC Commander® to communicate with the user by displaying a message, a control code or a numeric value on the Debug Window within the innoBASIC™ Workshop.

- *Item* － a message, a control code or a variable to be displayed on the Debug Window. If there is more than one item, they should be separated by a comma.

Description
Some means has to be provided for the BASIC Commander® to communicate and talk to the user and the method of doing this is via the Debug Window in the innoBASIC™ Workshop. This is done by inserting the DEBUG commands into the program. Debugging programs is an essential part of any application development as it is very rare to get things right the first time when programming. Therefore by placing DEBUG commands into a program, you will know exactly where you are in the program and what the contents are of variables in question. For example,

```
DEBUG "This is the wrong path, and the variable I is ", i
```

This command could be placed at an unexpected path in the program and when it is branched to and executed, will show the related variable i which can help with the diagnosis of malfunctions.

Control Code

In addition to debugging purposes, the DEBUG command is very useful as an interactive human-machine interface. To assist in this matter and to provide a more user-friendly console display, additional control codes are provided. These special control codes are summarized in the following table:

Function	Description
CLS	Clear Screen
CR	Carriage Return
TAB	Place a tab
CSRL	Cursor move left
CSRR	Cursor move right
CSRU	Cursor move up
CSRD	Cursor move down
BKSP	Cursor move backward destructively
CLREOL	Clear from cursor to end of line
CLREOS	Clear from cursor to end of screen
CSRXY (x,y)	Move cursor to position column x row y
CSRX (x)	Move cursor to position column x
CSRY (y)	Move cursor to position row y
BELL(n)	Generate a Windows built-in number n sound effect

Data Formatters

Numerical data can be displayed in various formats in the Debug Window. If no specific way is specified then the value will be displayed in decimal format. Therefore for the previous examples, this was not a problem and the values were correctly displayed. However, say the value was required to be displayed in binary or hex format, then the percentage symbol "%" followed by the desired format is used to do so. The following table shows the various options:

Formatting Symbols	Description
?	If the optional "?" formatter is used, an extra string "symbol = " will be added before the displayed value and a carriage return after the displayed value. The symbol stands for a user-defined variable name.
%DEC{n{L|R}}	Displays data in signed decimal format, the optional n value stands for column width. If the given n value is smaller than the actual digits, the width will be automatically expanded to fit the actual number width. Optional values L and R stand for left or right alignment. If L or R is omitted, the default value is aligned left. Leading 0's are omitted in the display.
%BIN{n{L|R}}	Displays data in an unsigned binary format, the optional value n stands for column width. If the given n value is smaller than the actual digits, 8 for 8-bit variables and 16 for16-bit variables, the width will be automatically expanded to fit the actual number width. Optional values L and R stand for left or right alignment. If L or R is omitted, the default is aligned left. Leading 0's of 8- or 16-digit data format are displayed.

Formatting Symbols	Description
%HEX{n{L|R}}	Displays data in unsigned hexadecimal format, optional n stands for column width. If the given n value is smaller than the actual digits, 2 for 8-bit variables and 4 for16-bit variables, the width will be automatically expanded to fit the actual number width. Optional L and R stand for left or right alignment. If L or R is omitted, the default is aligned left. Leading 0's of 2- or 4-digit data format are displayed.
%CHR	Displays data in ASCII character format
%FLOAT{n.m{L|R}}	Displays the floating-point data in scientific format, optional n stands for width of column and m stands for the number of valid digit(s) from 1 to 5. If the given n value is smaller than the actual digits, the width will be automatically expanded to fit the actual number width. Optional L and R stands for left or right alignment. If L or R is omitted, the default is aligned left.
%REAL{n.m{L|R}}	Same as %FLOAT{n.m{L|R}} mentioned above, except that the floating-point number is displayed in real number format.
%REP{n}	Displays the const or variable repeatedly n times. If optional n is not given, a default 1 is assigned.

Therefore in the case of the above example the DEBUG command could be modified to display the variable in binary format as follows:

```
Debug "The value for i is ", %BIN i
```

The Debug command requires a certain amount of time for communication between the BASIC Commander® and the PC through the USB interface. For this reason, in time-sensitive applications, precautions should be taken to avoid using the Debug command within any program time critical path. When using the Debug command in the development stage during debugging, there might be a timing difference when the Debug command is later removed for formal operation.

Example

```
Sub main()

Dim X As Byte = 100
    Debug ? X           'shows "X=100" with carriage return
    Debug "Column 10, aligned to the left",%DEC10L X,CR
    Debug "Column 10, aligned to the right", %DEC10R X,CR

End Sub
```

DEBUGIN

Syntax
DEBUGIN *Item* { , *Item*}

Operation
This command allows users to feed data to the BASIC Commander® through the Debug Window in run-time.

- *Item* — a variable that receives data from the debug console. For operational convenience, the Debugin command also supports the message and control code that is displayed on the Debug Window which helps the user to understand more about what kind of information is waiting to be inputted. Otherwise, the user has to place another Debug command to provide such information. If there is more than one item, they should be separated by a comma.

Description
Some means has to be provided for the BASIC Commander® to communicate and talk to the user. The method of doing this is via the Debug Window in the innoBASIC™ Workshop, by inserting DEBUGIN commands into the program which receives data from the Debug console. This method can either be used for debugging programs or as a human-machine interface for receiving data from the user. For example,

```
DEBUGIN "Please enter your lucky number.", num, CR
```

Other items are the same as a DEBUG command, please refer to the DEBUG command for their usages.

Note that DEBUGIN command requires a certain amount of time for communication between the BASIC Commander® and the PC through the USB interface. For this reason, for time-sensitive applications, precautions should be taken

to avoid using the DEBUGIN command in any program time critical path. While using the DEBUGIN command in the development stage during debugging, there might be a timing difference when the DEBUGIN command is later removed for formal operation. If the DEBUGIN command is not removed for stand-alone operation, when the USB interface is not connected to the PC, the program will enter a dead loop as it continues to wait for data entry.

Example

```
Sub Main()

    Dim yourname As String * 20
    Dim Key As Byte

    Debugin "Please enter your name.", yourname, CR
    Debug "Hi ",yourname,"!"
    Do
      Debugin CR,"Enter in DEC: ", Key, CR              'say, 100
      Debug "The number in DEC is: ",Key, CR

      Debugin CR,"Enter in BIN: ", %BIN Key, CR       'say, 100
      Debug "The number in BIN is: ",%BIN Key, CR

      Debugin CR,"Enter HEX: ", %HEX Key, CR           'say, FA
      Debug "The number in Hex is: ",%HEX Key, CR

      Debugin CR,"Enter a letter: ", %CHR Key, CR
      Debug "The letter is ", %CHR Key,", ASCII Code is ", Key, CR
    Loop
End Sub
```

DIM

Syntax
DIM *Variable* **AS** *Type* **{* *Size*}**

Operation
To declare local or global variables.
- *Variable* — the defined variable name where the value will be stored
- *Type* — one of the legal variable type names, including Boolean, Byte, Integer, Word, Long, Float, Persistentbyte, Persistentinteger, Persistentword, Persistentlong and Persistentfloat.
- *Size* — a constant to specify the size of a string type variable and the size of a string is confined by the RAM available.

Description
All variables must be declared in advance. Conventionally, the variable data types are of Boolean, Byte, Short, Word, Integer, DWord, Long, Float and String type which are assigned with the RAM data memory, yet other featured variable types supported by innoBASIC™ are the persistent type variables which are assigned with the persistent EEPROM data memory. If DIM is used within any procedure, the variables declared are local variable which means they are visible only inside the variables which can be seen only inside the procedure where they are declared. On the contrary, if DIM is used outside of all procedures, the variables are global, which means they are visible throughout the entire program. Note that the persistent type variables are global in nature, which means you must declare persistent type variables outside of all procedures.

Unlike fundamental variables, the string literal can store non-numerical values of more than one single character. Yet, due to limited RAM resources, a proper string size should be given when declared.

Example

```
Dim G As Byte                    'global variable, no initializer

Sub Main()

    Dim X,Y As Byte
    Dim Z As Short =-1           'local variable, optional initializer
    Debug ? G
    Debug ? X
    Debug ? Y
    Debug ? Z

End Sub
```

DO... LOOP

Syntax
DO {*Modifier Condition*}
{*statements*}
LOOP {*Modifier Condition*}

Operation
Will setup a program loop where commands will be either repeatedly executed or not, depending upon user defined conditions.

- *Modifier* — is an optional modifier of WHILE or UNTIL, placed after DO or LOOP, but not both. If a WHILE modifier is used, it will allow the loop to continue if the condition is true. If an UNTIL modifier is used, it will allow the loop to continue until the condition is true.
- *Condition* — Boolean expression
- *Statements* — optional legal statements, including the optional EXIT DO or CONTINUE command(s).

Description
The basic DO... LOOP command will constitute an infinite loop, the commands enclosed by DO and LOOP will be executed repeatedly. You may add the WHILE qualifier at the beginning or the end of the loop, whereas the Boolean condition will be tested. As long as the condition is true, the loop body will be executed. If the WHILE is placed at the end of the loop, the loop body will be executed at least once. The UNTIL qualifier is similar to the WHILE qualifier, except that the loop is terminated rather than continuing when the Boolean condition is true. The statements may contain the optional EXIT DO or CONTINUE command. The EXIT DO command may be placed in the loop body, which exits the current loop immediately before the loop limit test is executed. The CONTINUE command transfers execution to the end of the containing block loop and begins the next iteration. Refer to Chapter 5 for further detailed explanation.

Example

Refer to Chapter 5 for further detailed examples.

DWORD2FLOAT

Syntax
Result = **DWORD2FLOAT**(*Argument*)

Operation
To convert a DWord value into its floating-point format.

- *Argument* — the DWord operand of the LONG2FLOAT function.
- *Result* — a floating-point variable that receives the result of the DWORD2FLOAT function.

Command Description
The DWORD2FLOAT command converts a DWord value into its floating-point format. The floating-point result will be an integral value ranging from 0 to 4294967295. Due to the single precision floating point employed, a DWord variable may not be represented precisely. The nearest integral floating point value will be returned instead. When using this command in your application program, care must therefore be taken.

You can tell whether a DWord value can be represented precisely by examining the number of bits in its binary format. Excluding the leading and trailing 0's of its binary format, if the number of remaining bits is greater than 24, then it cannot be precisely represented.

Example
Due to the single precision floating point employed, the value 4294967295 cannot be represented precisely. The nearest value 4294967296 will be returned instead.

```
Sub Main()

    Dim MyDWord As DWord
    Dim MyFloat As Float

    MyDWord = 4294967295
    MyFloat = DWORD2FLOAT(MyDWord)
    Debug"MyDWord =", MyDWord,CR, "MyFloat = ",MyFloat,CR

End Sub
```

ENUM...END ENUM

Syntax
ENUM *Identifier*
EnumeratorList
END ENUM

Operation
Declare an enumeration
- *Identifier* ─ the enumeration name.
- *EnumeratorList* ─ the list of all the enumerators with optional initial value(s).

Description
The ENUM command is a way to declare a constant. Enumeration must be declared outside of any procedure, which means they are global declarations and have public access only. An enumeration member with "=" is given the value of the constant expression. When constant values are omitted, the order of enumeration member declarations is significant. If the first enumerator value definition in the enumeration has no initial value, the value of the enumeration starts from 0. If an enumerator has no initial value, the previous enumerator value increased by 1 will be given. The following example shows how to use the ENUM command. The enumeration members are accessed by using the dot "." operator between the Identifier and the Enumerator. The following example shows how to use the ENUM command.

Example

```
Enum Color
    Red
    Yellow = 3
    Blue = 1
    Green

End Enum

Sub Main()
    Debug "Enumerator Red = ", Color.Red, CR
    Debug "Enumerator Yellow = ", Color.Yellow, CR
    Debug "Enumerator Blue = ", Color.Blue, CR
    Debug "Enumerator Green = ", Color.Green, CR
End Sub
```

EVENT...END EVENT

Syntax

EVENT *ModuleName.EventName*()
{*statements*}
END EVENT

Operation

Declare an Event procedure.

- *ModuleName* — the name of the module that the user declared which owns the event in question.
- *EventName* — the name of the Event in question.
- *Statements* — any valid innoBASIC™ statement.

Description

The EVENT command declares a procedure, which will be invoked by a peripheral module when a specified event occurs. The parentheses cannot be omitted even although there is no argument applicable. Note that when declaring the Event procedure, the Event name is composed of two parts. The first part is the module name and the second part is the event name. The two names are connected with a dot. The following example shows an event declaration and how to enable the Event function.

Example

```
Peripheral MyKeypad As KeypadA @ 0

Event MyKeypad.KeyPressed()
    Dim KeyID As Byte
    MyKeypad.GetKeyID(KeyID)
    Debug "Key ", KeyID, "is Pressed!", CR
End Event

Sub Main()
    MyKeypad.EnableKeypadEvent()      'Enable the Event
    Debug "Press Key Pad.", CR
    Do:Loop
End Sub
```

Note that when the program is in the EVENT procedure, other events will be blocked out until the EVENT procedure is completely executed. It is highly recommended that as few statements as possible are implemented within an EVENT procedure and that as many handling statements as possible are implemented within the main program.

EXP

Syntax
Result = **EXP(*Argument*)**

Operation
To return the natural exponent value of a floating-point argument.

- *Argument* ─ the floating-point operand of the EXP function.
- *Result* ─ a floating-point variable that receives the result of the EXP function.

Description
The EXP function returns the natural exponent value of a floating-point argument. The result is always a positive value. In mathematics, it is denoted as $y = e^x$, where e is Euler's number $2.71828\cdots$. Its inverse function is the natural logarithm LOG function.

Example

```
Sub Main()

    Dim Result As Float

    Result = EXP(2.0)
    Debug "EXP of 2.0 is ", result, CR
    Result = EXP(-2.0)
    Debug "EXP of -2.0 is ", result, CR
End Sub
```

EXP10

Syntax

Result = **EXP10(*Argument*)**

Operation

To return the base 10 exponent value of a floating-point argument.

- ● *Argument* — the floating-point operand of the EXP10 function.
- ● *Result* — a floating-point variable that receives the result of the EXP10 function.

Description

The EXP10 function returns the base 10 exponent value of a floating-point argument. The result is always a positive value. In mathematics, it is denoted as $y = 10^x$. Its inverse function is the base 10 logarithm LOG10 function.

Example

```
Sub Main()

    Dim Result As Float

    Result = EXP10(2.0)        '100
    Debug "EXP10 of 2.0 is ", result, CR
    Result = EXP10(-2.0)       '0.01
    Debug "EXP10 of -2.0 is ", result, CR
End Sub
```

FLOAT2BYTE

Syntax
Result = **FLOAT2BYTE(*Argument*)**

Operation
To convert a floating-point value into a Byte.
- *Argument* — the floating-point operand of the FLOAT2BYTE function.
- *Result* — a Byte variable that receives the result of the FLOAT2BYTE function.

Description
The FLOAT2BYTE command converts a floating-point value into a Byte. The decimal part of the number will be rounded. If the value exceeds the Byte value range, a value of 0 or 255 will be provided.

Example

```
Sub Main()
    Dim MyFloat As Float
    Dim MyByte As Byte

    MyFloat = 2.4
    MyByte = FLOAT2BYTE(MyFloat)      'the result is 2
    Debug "FLOAT2BYTE of 2.4 :", MyByte, CR

    MyFloat = 2.5
    MyByte = FLOAT2BYTE(MyFloat)      'the result is 3
    Debug "FLOAT2BYTE of 2.5 : ", MyByte, CR
```

```
    MyFloat = -1.0
    MyByte = FLOAT2BYTE(MyFloat)      'the result is 0
    Debug "FLOAT2BYTE of -1.0 : ", MyByte, CR

    MyFloat = 256.0
    MyByte = FLOAT2BYTE(MyFloat)       'the result is 255
    Debug "FLOAT2BYTE of 256.0 :", MyByte, CR
End Sub
```

FLOAT2DWORD

Syntax
Result = **FLOAT2DWORD(*Argument*)**

Operation
To convert a floating-point value into a DWord.
- *Argument* — the floating-point operand of the FLOAT2DWORD function.
- *Result* — a DWord variable that receives the result of the FLOAT2DWORD function.

Description
The FLOAT2DWORD command converts a floating-point value into a DWord. The decimal part of the number will be rounded. If the value exceeds the DWord value range, a value of 0 or 4294967295 will be assigned.

Example

```
Sub Main()
    Dim MyFloat As Float
    Dim MyDword As Dword

    MyFloat = 2.4
    MyDword = FLOAT2DWORD(MyFloat)    'the result is 2
    Debug "FLOAT2DWORD of 2.4 : ", MyDword, CR

    MyFloat = 2.5
    MyDword = FLOAT2DWORD(MyFloat)     'the result is 3
    Debug "FLOAT2DWORD of 2.5 : ", MyDword, CR
```

```
    MyFloat = -1.0
    MyDword = FLOAT2DWORD(MyFloat)    'the result is 0
    Debug "FLOAT2DWORD of -1.0 : ", MyDword, CR
    MyFloat = 4.3E9
    MyDword = FLOAT2DWORD(MyFloat)    'the result is 4294967295
    Debug "FLOAT2DWORD of 4.3E9 : ", MyDword, CR
End Sub
```

FLOAT2INTEGER

Syntax
Result = **FLOAT2INTEGER(*Argument*)**

Operation
To convert a floating-point value into an integer.
- *Argument* ― the floating-point operand of the FLOAT2INTEGER function.
- *Result* ― an integer variable that receives the result of the FLOAT2INTEGER function.

Description
The FLOAT2INTEGER command converts a floating-point value into an integer. The decimal part of the number will be rounded. If the value exceeds the integer value range, a value of +32767 or -32768 will be assigned.

Example

```
Sub Main()
        Dim MyFloat As Float
        Dim MyInteger As Integer

        MyInteger = 22
        Debug "MyInteger : ", MyInteger, CR

        MyFloat = 2.4
        MyInteger = FLOAT2INTEGER(MyFloat)      'the result is 2
        Debug "FLOAT2INTEGER of 2.4 : ", MyInteger, CR

        MyFloat = 2.5
        MyInteger = FLOAT2INTEGER(MyFloat)      'the result is 3
        Debug "FLOAT2INTEGER of 2.5 : ", MyInteger, CR
```

```
    MyFloat = -1.0
    MyInteger = FLOAT2INTEGER(MyFloat)    'the result is -1
    Debug "FLOAT2INTEGER of -1.0 : ", MyInteger, CR

    MyFloat = 32768.0
    MyInteger = FLOAT2INTEGER(MyFloat)     'the result is 32767
    Debug "FLOAT2INTEGER of 32768.0 : ", MyInteger, CR

    MyFloat = -32769.0
    MyInteger = FLOAT2INTEGER(MyFloat)     'the result is -32768
    Debug "FLOAT2INTEGER of -32769.0 : ", MyInteger, CR
End Sub
```

FLOAT2LONG

Syntax
Result = **FLOAT2LONG(*Argument*)**

Operation
To convert a floating-point value into a LONG type
- ● *Argument* ─ the floating-point operand of the FLOAT2LONG function.
- ● *Result* ─ a LONG variable that receives the result of the FLOAT2LONG function.

Description
The FLOAT2LONG command converts a floating-point value into a LONG. The decimal part of the number will be rounded. If the value exceeds the long integer value range, a value of +2147483647 or -2147483648 will be assigned.

Example

```
Sub Main()
    Dim MyFloat As Float
    Dim MyLong As Long

    MyFloat = 2.4
    MyLong = FLOAT2LONG(MyFloat)              'the result is 2
    Debug "FLOAT2LONG of 2.4 : ", MyLong, CR

    MyFloat = 2.5
    MyLong = FLOAT2LONG(MyFloat)              'the result is 3
    Debug "FLOAT2LONG of 2.5 : ", MyLong, CR
```

```
    MyFloat = -1.0
    MyLong = FLOAT2LONG(MyFloat)            'the result is -1
    Debug "FLOAT2LONG of -1.0 : :, MyLong, CR

    MyFloat = 4.3E9
    MyLong = FLOAT2LONG(MyFloat)            'the result is 2147483647
    Debug "FLOAT2LONG of 4.3E9 : ", MyLong, CR

    MyFloat = -4.3E9
    MyLong =FLOAT2LONG(MyFloat)             'the result is -2147483648
    Debug "FLOAT2LONG of -4.3E9 : ", MyLong, CR
End Sub
```

FLOAT2REALSTRING

Syntax
FLOAT2REALSTRING(*Argument, StringVar*)

Operation
To convert a floating-point value into an ASCII character string in real number format.
- *Argument* — the floating-point operand of the FLOAT2REALSTRING function.
- *StringVar* — a string variable that receives the result of the conversion.

Description
The FLOAT2REALSTRING command converts a floating-point value into an ASCII character string in real number format, which has 5-digit valid mantissa with radix point. For example +0.31416. Note that the trailing zeros will be omitted, except the first one after the radix point. For instance, 2 will be converted to +2.0.

Example

```
Sub Main()
    Dim MyFloat As Float
    Dim MyString As String * 11

    MyFloat = 0.31416
    FLOAT2REALSTRING(MyFloat, MyString)
    Debug "FLOAT2REALSTRING of 0.31416 : ", MyString, CR
End Sub
```

FLOAT2SHORT

Syntax

Result = **FLOAT2SHORT(*Argument*)**

Operation

To convert a floating-point value into a SHORT variable.

- ● *Argument* — the floating-point operand of the FLOAT2SHORT function.
- ● *Result* — a SHORT variable that receives the result of the FLOAT2INTEGER function.

Description

The FLOAT2SHORT command converts a floating-point value into a SHORT. The decimal part of the number will be rounded. If the value exceeds the SHORT value range, a value of +127 or -128 will be assigned.

Example

```
Sub Main()
    Dim MyFloat As Float
    Dim MyShort As Short

    MyFloat = 2.4
    MyShort = FLOAT2SHORT(MyFloat)              'the result is 2
    Debug "FLOAT2SHORT of 2.4 : ", MyShort, CR

    MyFloat = 2.5
    MyShort = FLOAT2SHORT(MyFloat)              'the result is 3
    Debug "FLOAT2SHORT of 2.5 : ", MyShort, CR
```

```
    MyFloat = -1.0
    MyShort = FLOAT2SHORT(MyFloat)                    'the result is -1
    Debug "FLOAT2SHORT of -1.0 : ", MyShort, CR

    MyFloat = 128.0
    MyShort = FLOAT2SHORT(MyFloat)                    'the result is 127
    Debug "FLOAT2SHORT of 128 : ", MyShort, CR

    MyFloat = -129.0
    MyShort = FLOAT2SHORT(MyFloat)                    'the result is -128
    Debug "FLOAT2SHORT of -129 : ", MyShort, CR
End Sub
```

FLOAT2STRING

Syntax
FLOAT2STRING(*Argument, StringVar*)

Operation
To convert a floating-point value into an ASCII character string in floating-point format.

- *Argument* — the floating-point operand of the FLOAT2STRING function.
- *StringVar* — a string variable that receives the result of the conversion.

Description
The FLOAT2STRING command converts a floating-point value into an ASCII character string in floating-point format which is composed of a sign character, a 5-digit mantissa with radix point, an exponent sign character E, the exponent sign character and 2-digit exponent. For example +3.1416E-01.

Example

```
Sub Main()
    Dim MyFloat As Float
    Dim MyString As String * 13
    MyFloat = 0.31416
    FLOAT2STRING(MyFloat, MyString)
    Debug "FLOAT2STRING of 0.31416 : ", MyString, CR
End Sub
```

FLOAT2WORD

Syntax
Result = **FLOAT2WORD(*Argument*)**

Operation
To convert a floating-point value into a Word.
- *Argument* ─ the floating-point operand of the FLOAT2WORD function.
- *Result* ─ a Word variable that receives the result of the FLOAT2WORD function.

Description
The FLOAT2WORD command converts a floating-point value into a Word. The decimal part of the number will be rounded. If the value exceeds the Word value range, a value of 0 or 65535 will be assigned.

Example

```
Sub Main()
    Dim MyFloat As Float
    Dim MyWord As Word

    MyFloat = 2.4
    MyWord = FLOAT2WORD(MyFloat)               'the result is 2
    Debug "FLOAT2WORD of 2.4 : ", MyWord, CR

    MyFloat = 2.5
    MyWord = FLOAT2WORD(MyFloat)               'the result is 3
    Debug "FLOAT2WORD of 2.5 : ", MyWord, CR
```

```
    MyFloat = -1.0
    MyWord = FLOAT2WORD(MyFloat)                    'the result is 0
    Debug "FLOAT2WORD of -1.0 : ", MyWord, CR

    MyFloat = 65536.0
    MyWord = FLOAT2WORD(MyFloat)                    'the result is 65535
    Debug "FLOAT2WORD of 65536 : ", MyWord, CR
End Sub
```

FLOOR

Syntax
Result = **FLOOR**(*Argument*)

Operation
To return the nearest integer that is not greater than the floating-point argument value.
- *Argument* — the floating-point operand of the FLOOR function.
- *Result* — a floating-point variable that receives the result of the FLOOR function.

Description
The FLOOR command returns the nearest integer (floating-point value) that is not greater than the floating-point argument value. Usually this is used to round a floating-point value into an integer. Another pairing function is the CEIL function which returns the nearest integer (floating-point value) that is not smaller than the floating-point argument value.

Example

```
Sub Main()
    Dim Result As Float
    Result = FLOOR(2.3)                        'the result is 2.0
    Debug "FLOOR of 2.3 is ", result,CR
    Result = FLOOR(-2.3)                        'the result is -3.0
    Debug "FLOOR of -2.3 is ", result,CR
End Sub
```

FOR...NEXT

Syntax
FOR *Index* = *StartValue* **TO** *EndValue* {**STEP** *StepValue*}
{*Statements*}
NEXT {*Index*}

Operation
Will establish a repeatable loop between the FOR and NEXT commands.

- *Index* — this is a variable which is used to store a numerical value which controls the number of times the loop is run. For program clarity considerations, placing the optional Index after NEXT is recommended.
- *StartValue* — a constant or variable which defines the initial value of the Counter
- *EndValue* — a constant or variable which defines the end value of Counter. If the value of the Counter exceeds this value, then when a NEXT command is encountered, the program will leave the FOR...NEXT loop and execute the command following the NEXT command.
- *StepValue* — a constant or variable which defines the amount by which Index is increased each time the loop is run. If the STEP command is not used, a default step Value 1 will be employed.
- *Statements* — optional legal statements, including the optional EXIT FOR or CONTINUE command(s).

Description
The FOR and NEXT commands will establish a repeatable loop, with any number of statements placed within the loop body. Each time the NEXT command is encountered the program returns to the FOR command. The first time the FOR command is encountered, the variable Index will be loaded with StartValue and for the following iterations, Index is incremented by the StepValue. If the new value of Index does not exceed EndValue, the loop body will be executed again, otherwise the

program will proceed to the consequent statement after the NEXT command. Other loops can be nested within a FOR...NEXT loop, with the maximum number of nestings limited by the program space. The statements may contain the optional EXIT FOR or CONTINUE command. The EXIT FOR command may be placed in the loop body which exits the current loop immediately before the loop limit test is executed. The CONTINUE command transfers execution to the end of the containing block loop and begins the next iteration.

Example
Refer to Chapter 5 for further detailed examples.

FUNCTION...END FUNCTION

Syntax
FUNCTION *FunctionName*(**{***Arglist***}**) **AS** *ReturnType*
{*Statements***}**
END FUNCTION

Operation
Declare a function with an optional argument list.
- *FunctionName* — the name of the function contains a sequence of letters, digits and underscore. The leading character must be a letter.
- *Arglist* — is a list of optional arguments required in the function. Each argument is preceded with either a byval or byref modifier to indicate the argument passing method. The parenthesis cannot be omitted even though no argument required.
- *ReturnType* — specifies the return data type, which may be Byte, Integer, Word, Long or Float.
- *Statements* - any valid innoBASIC™ statement.

Description
The FUNCTION command declares a Function which can be invoked by its FunctionName to execute some user-defined function. Unlike Sub, there is a value returned to the caller.

Example

```
Function Sum(X As Short,Y As Short) As Integer
    Return X+Y
End Function

Sub main()

    Dim X, Y As Short
    Dim Z As Integer

    X=1
    Y=2
    Z=Sum(X,Y)
    Debug "X=", X, CR, "Y=", Y, CR, "Z=X+Y=", Z, CR

End Sub
```

GETDIRPORT

Syntax
Result* = GETDIRPORT *Port

If the port number is a constant, you may also use one of the following formats instead.

Result = GETDIRPORT0
Result = GETDIRPORT1
Result = GETDIRPORT2

Operation
Obtains the I/O direction settings of the specified port.
- ● *Port* — a constant or variable (0 ~2) that specifies the port number. Port 0 consists of pins P0~P7, Port 1 consists of pins P8~P15 and Port 2 consists of pins P16~P23. For the 24-pin BASIC Commander® which has two ports, the Port value is 0 or 1.
- ● *Result* — a Byte variable to receive the port direction setting.

Description
The GETDIRPORT command is used to read the current I/O direction settings of the specified port for applications in which the I/O directions may switch between input and output mode during program execution. Data 0 stands for output and 1 stands for input mode. For pins which with a smaller number are of the lower bit order of a data byte. For example, the P0 setting will appear in the Bit 0 of the data read. The I/O directions default to input after the program starts.

Example

The following example changes the I/O direction of Port 0, which is connected to seven LEDs through resistors. The LEDs will turn on and off accordingly.

```
Sub Main()

    Dim KeyAs Byte
    Dim PortStatus As Byte

Start:                  'P0 ~ P7 are default input mode.
      WRITEPORT0 &H00    'Write low to output buffers
    Do
      Debugin "Input any key to turn on LED 0, 2, 4, 6: ",%CHR Key, CR
      SETDIRPORT 0,&HAA   'Switch P0, P2, P4, P6 to OUTPUT mode
      PortStatus=GETDIRPORT0
      Debug "Port 0 status is: ", %BIN PortStatus, CR,CR

      Debugin "Input any key to turn on LED 1, 3, 5, 7: ",%CHR Key, CR
      SETDIRPORT 0,&H55   'Switch P1, P3, P5, P7 to OUTPUT mode
      PortStatus=GETDIRPORT0
      Debug "Port 0 status is: ", %BIN PortStatus,CR,CR
    Loop
End Sub
```

GOTO

Syntax
GOTO *LabelName*

Operation
The GOTO command will cause the program to move to a specified program location, labeled with the user-given LabelName.

- *LabelName* — this is a label that specifies the point where the program will branch to.

Description
The GOTO command will force the program to jump to a user specified location, which contains a LabelName followed with a semicolon. This location is given by the label name which follows the GOTO command. The statement following the GOTO command will not be executed. What will actually be executed is the next command after executing the GOTO command as specified by the label. The GOTO command is therefore used to give the user a direct means of program control. Users are not recommended to use the GOTO statement in their programs, which usually will cause difficulties in reading the program. However, for a nested statement structure, to branch to the most external loop, it is convenient to use the GOTO statement.

Example

The following demonstrates how to use the GOTO command. You can always find more structured statements instead.

```
Sub Main()

Start:
    Debug "Tick!",CR
    Pause 1000
    Goto Start

End Sub
```

HIGH

Syntax
HIGH *Pin*

Operation
Sets the specified pin to a logic high level

- *Pin* — a constant or variable (0~23) that specifies the pin that the high level is to be applied to. For the 24-pin BASIC Commander®, the Pin value ranges from 0~15.

Description
This command will set the specified pin to a high level of around 5 volts. Usually, a pin must be previously changed in advance to input or output mode before executing the corresponding input or output operations. However, as only one pin is involved in the HIGH operation, the pin will be changed to the output mode automatically by the system. It is not necessary for the user to change the mode manually to an output pin before executing this instruction.

The following shows a simple circuit to illuminate a small LED using the HIGH and LOW command. When the HIGH command is executed on the pin as shown, the LED will be turned off and turned on when LOW command is executed. Each I/O pin has a typical sink current capacity of around 20 mA and a source capacity of around 10mA which means the device is able to consume more power when the pin is low.
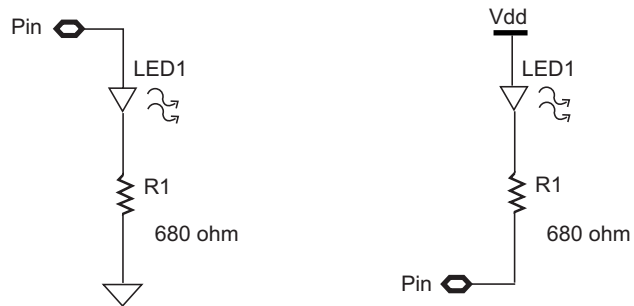
Figure 6-4 Illuminating a small LED using HIGH and LOW commands

Note that as the pins have current drive limitations, driving heavier loads such as light bulbs or relays, which draw much more current than LEDs, will require a transistor switch to be connected to the output terminal. Some simple circuits to implement this are shown below.
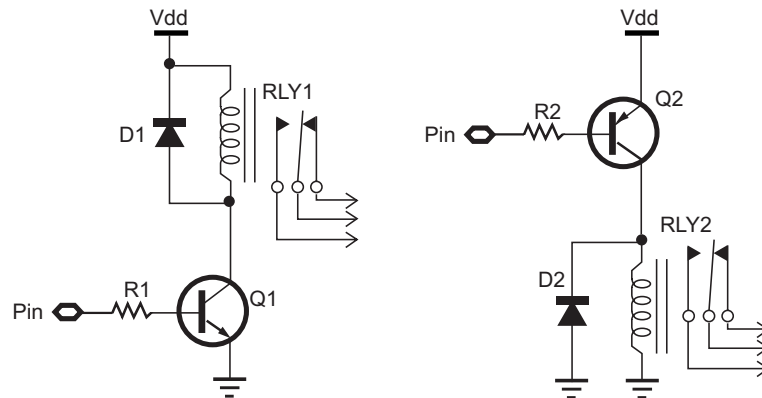

Figure 6-5 Using transistors to drive heavier loads

It is important to check the relay manufacturer's datasheet to determine the relay coil current requirements and to ensure that the transistor can handle this current.

Example

For the LED circuit shown above the following program will cause the LED connected to Pin 0 to flash at a fixed rate of 1 Hz.

```
Sub Main()
    Do
      HIGH 0          'LED on
      Pause 500       '0.5 second delay
      LOW 0           'LED off
      Pause 500       '0.5 second delay
      TOGGLE 0        'LED on
      Pause 500       '0.5 second delay
      TOGGLE 0        'LED off
      Pause 500       '0.5 second delay
    Loop
End Sub
```

IF...THEN...ELSE

Syntax

Two forms of IF...THEN...ELSE commands are available, namely block and line version of IF...THEN...ELSE commands.

Line Version:
IF *Condition* THEN *Statements* {ELSE *Statements*}

Block Version:
IF *Condition* THEN
Statements
{ELSEIF *Condition* THEN
***Statements*}**
{ELSE
***Statements*}**
END IF

Operation

The IF...THEN...ELSE command executes commands based on expression evaluation.

- • *Condition* — an expression of condition(s) that can be evaluated as a Boolean value.
- • *Statements* — any valid innoBASIC™ statement.

Description

The IF...THEN...ELSE command is the basic conditional command. Each expression in an IF...THEN...ELSE command must be convertible to Boolean. If the expression in the IF command is True, the statements enclosed by the THEN block are executed. If the expression is False, each of the ELSEIF expressions is evaluated. If one of the

ELSEIF expressions evaluates to True, the corresponding block is executed. If no expression evaluates to True and there is an ELSE block, the ELSE block is executed. Once a block finishes executing, the program execution passes to the END IF command. The line version of IF...THEN...ELSE command is available for program simplicity when there is only one statement after THEN and ELSE. In the line version you may place more than one statement after THEN and ELSE by adding a colon ":" between two statements.

Example

Refer to Chapter 5 for further detailed examples.

IN

Syntax
Result = **IN** *Pin*

Operation
Reads the external logic input status of the specified pin.

- *Pin* ─ a constant or variable (0~23) that specifies the pin to be read. For the 24-pin BASIC Commander®, the Pin value ranges from 0~15.
- *Result* ─ a Byte variable to receive the external logic level on Pin.

Description
This command will read the external logic level status from the specified pin. Usually, a pin must be changed in advance to input mode. However, as only one pin is involved in the IN operation, the pin will be changed to the input mode automatically by the system. It is not necessary for the user to change the mode manually to an input pin before executing this instruction. Note that after power-on, all the I/O pins will be in an input mode.

Example
The following example shows how to use IN command to read an external logic level.
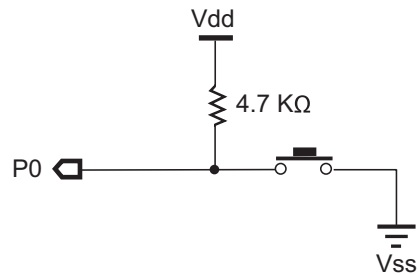
Figure 6-6 Reading an external logic input status from P0

```
Sub Main()
  Do
Wait:
    If IN(0) = 1 Then
       Pause 10
       Goto Wait
    Else
       Debug "Button is pressed.", CR
Release:
    If IN(0) = 0 Then Goto Release
    Debug "Button is released.", CR
    End If
  Loop
End Sub
```

INPUT

Syntax
INPUT *Pin*

Operation
Configures the specified Pin to the input mode.

- *Pin* — a constant or variable (0~23) that specifies which pin will be set to the input mode. For the 24-pin BASIC Commander®, the Pin value ranges from 0~15.

Description
If you would like to read a digital signal from the outside world, then first the pin on which the data is to be read should have been changed to an input mode beforehand. Due to the simplicity of the pin-related input output commands, the mode change implementation is automatically carried out by the system. The user need only simply use the input output commands without worrying about the direction of the pins in question. However, this INPUT command is still provided and note that if you use the INPUT command to change a pin from an output mode to an input mode, the high/low state of the pin will disappear and the pin will be placed into a high impedance state. Therefore the addition of a pull-high resistor on the pin is recommended to avoid uncertain logic states existing which might result in unwanted errors logically or electrically.

For commands that rely on input pins, like PULSIN and SERIN, the mode corresponding pins will be changed to the input mode automatically. The default mode after a program start (reset) is the input mode.

Example

The following example changes the I/O directions of P0, which connects an LED through a resistor, and the LED will turn on/off accordingly.
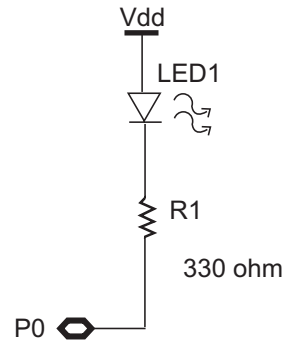

Figure 6-17 Input/Output Circuit

```
Sub Main()
    Dim key As Byte

Start:
    WRITEPORT0 &H00 'Write low to output buffers
    Do
      Debugin "Input any key to turn on LED.", %CHR key, CR
      OUTPUT 0 ' Switch P0 to OUTPUT mode, turn on LED0
      Debugin "Input any key to turn off LED.", %CHR key, CR
      INPUT 0 ' Switch P0 to INPUT mode, turn off LED0

    Loop
End Sub
```

INTEGER2FLOAT

Syntax
Result = **INTEGER2FLOAT**(*Argument*)

Operation
To convert an Integer value into its floating-point format.
- *Argument* ─ the Integer operand of the INTEGER2FLOAT function.
- *Result* ─ a floating-point variable that receives the result of the INTEGER2FLOAT function.

Description
The INTEGER2FLOAT command converts an Integer value into its floating-point format. The floating-point result will be an integral value ranging from -32768.0 to +32767.0.

Example

```
Sub Main()
    Dim MyInteger As Integer
    Dim MyFloat As Float

    MyInteger = -32768
    MyFloat = INTEGER2FLOAT(MyInteger)
    Debug "INTEGER2FLOAT of -32768 : ", MyFloat, CR

    MyInteger = 32767
    MyFloat = INTEGER2FLOAT(MyInteger)
    Debug "INTEGER2FLOAT of 32767 : ", MyFloat, CR
End Sub
```

LCASE

Syntax
LCASE(*TargetString*)

Operation
Changes the specified string to lower case letters.
- ● *TargetString* ─ the string to be converted.

Description
The LCASE command changes the specified string to lower case letters.

Example

```
Sub Main()
    Dim MyString As String * 12

    MyString = "Hello World!"
    Debug "Original string: ", MyString, CR

    LCASE(MyString)
    Debug "All in lower case: ", MyString, CR

    UCASE(MyString)
    Debug "All in upper case: ", MyString, CR
End Sub
```

LCDCMD

Syntax
LCDCMD *Pin, Command*

Operation
Send command to an LCD Module.

- ● *Pin* ─ a constant or variable (1, 9 or 17) that specifies the first pin of seven contiguous pins where the LCD Module is connected. 1 stands for Pin 1 to Pin 7, 9 stands for Pin 9 to 15 and 17 stands for Pin 17 to 23. For the 24-pin BASIC Commander®, the Pin value will be 1 and 9 only.
- ● *Command* ─ const or variable (0~255) that specifies the LCD command to send.

Description
The three LCD commands (LCDCMD, LCDIN and LCDOUT) allow the BASIC Commander® to interface directly to a standard LCD display that employs a Hitachi 44780 or compatible LCD controller. There are several 1x16, 2x16 and 4x20 character LCD modules available. To minimize the pins required, the 4-bit interface to the LCD is employed. Therefore a total of seven I/O pins are required. To provide flexibility to users, you may specify the first Pin Number of 1, 9 or 17, which means you are to use P1~P7, P9~P15 or P17~P23. The following drawing shows the wiring when P1~P7 are used.
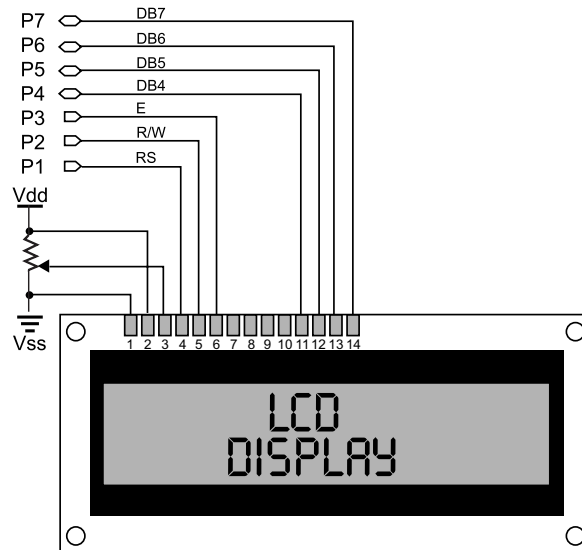
Figure 6-8 Wiring connection when using P1~P7

When the LCD is first powered-up, it defaults to an 8-bit interface and must be configured as a 4-bit bus before sending commands. This process is known as initializing the LCD and is the first thing your program should do upon starting up.

Refer to the Example section for the LCD initialization code The Hitachi 44780 LCD controller provides some special instructions for initializing the display, moving the cursor, changing the default layout, etc. Refer to the table below.

Action	Command (in Decimal)	Description
Do Nothing	0	No operation
Clear display	1	Clear the display and move the cursor to home position
Home display	2	Move the cursor and to home position
Inc Cursor	6	Set the cursor direction to the right, without a display shift
Display off	8	Turn off the display (display data is retained)
Display On	12	Turn on the display without the cursor (display is restored)
Blinking Cursor	13	Turn on the display with blinking cursor
Underline Cursor	14	Turn on the display with underlined cursor
Cursor left	16	Move the cursor one character to the left
Cursor right	20	Move the cursor one character to the right
Scroll Left	24	Scroll the display one character to the left
Scroll right	28	Scroll the display one character to the right
Move to CGRAM Address	64 + address	Move the pointer to the Character RAM location
Move to DDRAM Address	128 + address	Move the cursor to the Display Data RAM location

For most users, the commands shown above are should be sufficient, however, for more advanced applications you may need the commands shown below, which will allow even more powerful control of the LCD module.

Action	Command (in Binary)								Description
	7	6	5	4	3	2	1	0	
Clear Display	0	0	0	0	0	0	0	1	Clear the entire display and move the cursor home (Address 0)
Home Display	0	0	0	0	0	0	1	0	Move the cursor home and return the display to home position
Entry Mode	0	0	0	0	0	1	M	S	Set the cursor (M: 0=left, 1=right) and display scrolling (S: 0=no scrolling, 1=scroll)
Display/Cursor	0	0	0	0	1	D	U	B	Set the cursor direction to the right, without a display shift
Scroll Display / Shift Cursor	0	0	0	1	C	M	0	0	Shift the display or the cursor (C: 0=cursor, 1=display) to the left or to the right (M: 0=left, 1=right)
Function Set	0	0	1	B	L	F	0	0	Set the bus size (B: 0=4bits, 1=8 bits), number of lines (L: 0=1 line, 1=2 lines) and font size (F: 0=5x8, 1=5x10)
Move to CGRAM Address	0	1	A	A	A	A	A	A	Move the pointer to character RAM location specified by Address (A)
Move to DDRAM Address	1	A	A	A	A	A	A	A	Move the cursor to display RAM location specified by Address (A)

The following figure shows the most common DDRAM mapping, which you need to know for detailed operation.
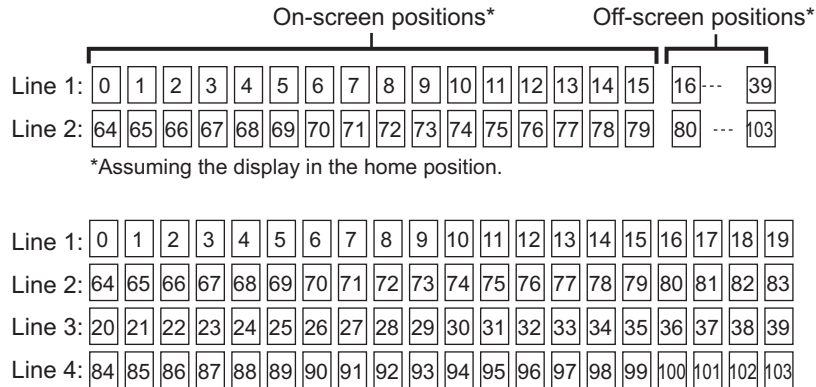
Figure 6-9 DDRAM Mapping of 2X16 and 4X20 Display

Example

```
Sub Main()
    Dim I, DataIn As Byte
    Dim MyString(8) As Byte = "Hi There!"
    PAUSE 1000          'allow LCD to self-initialize first
    LCDCMD 1, 32        'set data bus to 4-bit mode
    LCDCMD 1, 40        'set to 2-line mode with 5x8 font
    LCDCMD 1, 15        'display on with blinking cursor
    LCDCMD 1, 6         'auto-increment cursor
    LCDCMD 1, 1         'clear display
```

```
    For I=0 To 8
       LCDOUT 1, &H80+I, [MyString(I)] 'display "Hi There!"
    Next

    Debug "Read from LCD : "

    For I=0 To 8
       LCDIN 1,&H80+I, [DataIn]          '&H80, DDRAM address of Line 1

    Debug %CHR DataIn
    Next

End Sub
```

LCDIN

Syntax
LCDIN *Pin*, *Command*, [*DataIn*]

Operation
Receive data from an LCD Module.
- *Pin* ─ a constant or variable (1, 9 or 17) that specifies the first pin of seven contiguous pins where the LCD Module is connected. 1 stands for Pin 1 to Pin 7, 9 stands for Pin 9 to 15 and 17 stands for Pin 17 to 23. For the 24-pin BASIC Commander®, the Pin value will be 1 and 9 only.
- *Command* ─ a const or variable (0~255) that specifies the LCD command to send.
- *DataIn* ─ a list of variables that receive the incoming data.

Description
The three LCD commands (LCDCMD, LCDIN and LCDOUT) allow the BASIC Commander® to interface directly to a standard LCD display that employs a Hitachi 44780 or compatible LCD controller. The LCDIN command is used to send one instruction and then receive at least one data byte from the LCD's Character Generator RAM or Display Data RAM. Please refer to the LCDCMD section for more detailed interfacing and wiring information.

When the LCD is first powered-up, it defaults to an 8-bit interface and must be configured as a 4-bit bus before sending commands. This process is known as initializing the LCD and is the first thing your program should do upon starting up. Please refer to LCDCMD section for more detailed LCD initialization information.
The LCDIN command is used to receive ASCII character values, decimal, hexadecimal and binary translations and string data from the LCD's Character Generator RAM or Display Data RAM.

Example

Refer to the LCDCMD command for the usage.

LCDOUT

Syntax
LCDOUT *Pin*, *Command*, [*DataOut*]

Operation
Send data to an LCD Module.
- ● *Pin* — a constant or variable (1, 9 or 17) that specifies the first pin of 7 contiguous pins where the LCD Module is connected. 1 stands for Pin 1 to Pin 7, 9 stands for Pin 9 to 15 and 17 stands for Pin 17 to 23. For the 24-pin BASIC Commander®, the Pin value will be 1 and 9 only.
- ● *Command* — a const or variable (0~255) that specifies the LCD command to send. Usually, it is the Character RAM address plus an offset of 64 or the Display RAM address plus and offset of 128, to where the ASCII code is sent.
- ● *DataOut* — a list of const or variables that carry the data to be displayed.

Description
The three LCD commands (LCDCMD, LCDIN and LCDOUT) allow the BASIC Commander® to interface directly to a standard LCD display that employs a Hitachi 44780 or compatible LCD controller. The LCDIN command is used to send one instruction and then receive at least one data byte from the LCD's Character Generator RAM or Display Data RAM. Refer to the LCDCMD section for more detailed interfacing and wiring information.

When the LCD is first powered-up, it defaults to an 8-bit interface and must be configured as a 4-bit bus before sending commands. This process is known as initializing the LCD and is the first thing your program should do upon starting up. Refer to LCDCMD section for more detailed LCD initialization information.
The LCDOUT command is used to send one instruction followed by at least one data byte to the LCD. The data that is output is written to the LCD's Character Generator RAM or Display Data RAM.

Example

Refer to the LCDCMD command for usage.

LEFT

Syntax
LEFT(*TargetString, length*)

Operation
Retains the leftmost length letters of the specified string TargetSring with other letters truncated.

- *TargetString* — the string operand of the LEFT function.
- *Length* — a const or variable that specifies the length of the string to be kept.

Description
The LEFT command retains the leftmost letters specified by the Length and truncates the remaining letters of the string. If the specified length is longer than the size of the target string, the extra length will be ignored and the string remains unchanged.

Example

```
Sub Main()

    Dim MyString As String * 12

    MyString = "Hello World!"
    LEFT(MyString, 5)
    Debug "Leftmost 5 letters: ", MyString, CR
    MyString = "Hello World!"
    RIGHT(MyString, 5)
    Debug "Rightmost 5 letters of MyString: ", MyString, CR
```

```
    MyString = "Hello World!"
    MID(MyString, 3, 5)
    Debug "Middle 5 letters from the third letter ", MyString, CR

End Sub
```

LEN

Syntax
Length = **LEN(*StringVar*)**

Operation
To return the length of a string.
- *StringVar* ─ the string operand of the LEN function.
- *Length* ─ a variable that receives the length of the given string.

Description
The LEN command returns the length of a string. For an empty string, which contains no ASCII characters, the length is 0. The maximum length of a string is the size it declares.

Example

```
Sub Main()
    Dim MyString As String * 12
    Dim Length As Byte

    MyString = ""
    Length =LEN(MyString)
    Debug "Length of a null string is ", Length, CR
    MyString = "Hello!"
    Length = LEN(MyString)
    Debug "Length of MyString is ", Length, CR

End Sub
```

LOG

Syntax

Result = **LOG**(*Argument*)

Operation

To return the natural logarithm value of a floating-point argument.

- *Argument* ─ the floating-point operand of the LOG function.
- *Result* ─ a floating-point variable that receives the result of the LOG function.

Description

The LOG function returns the natural logarithm value of a floating-point argument. In mathematics, it is denoted as $y = \ln(x)$. Its inverse function is the natural exponent EXP function.

Example

```
Sub Main()

    Dim Result As Float
    Result = LOG(2.7183)
    Debug "LOG of 2.7183 is ", Result, CR
    Result = LOG(7.3891)
    Debug "LOG of 7.3891 is ", Result, CR

End Sub
```

LOG10

Syntax
Result = **LOG10** (*Argument*)

Operation
To return the base 10 exponent value of a floating-point argument.
- *Argument* ─ the floating-point operand of the LOG10 function.
- *Result* ─ a floating-point variable that receives the result of the LOG10 function.

Description
The LOG10 function returns the base 10 logarithm value of a floating-point argument.
The Argument is always a positive value. In mathematics, it is denoted as $y = \log(x)$.
Its inverse function is the base 10 exponent EXP10 function.

Example

```
Sub Main()
    Dim Result As Float

    Result = LOG10(10.0)
    Debug "LOG10 of 10.0 is ", Result, CR
    Result = LOG10(100.0)
    Debug "LOG10 of 100.0 is ", Result, CR

End Sub
```

LONG2FLOAT

Syntax
Result = **LONG2FLOAT**(*Argument*)

Operation
To convert a Long value into its floating-point format.
- *Argument* — the Long operand of the LONG2FLOAT function.
- *Result* — a floating-point variable that receives the result of the LONG2FLOAT function.

Description
The LONG2FLOAT command converts a Long value into its floating-point format. The floating-point result will be an integral value ranging from +2147483647 to -2147483648. Due to the single precision floating point employed, a LONG variable may not be represented precisely. The nearest integral floating point value will be returned instead. When using this command in your application program, care must therefore be taken.

You can tell whether a LONG value can be represented precisely by examining the number of bits in its binary format. Excluding the leading and trailing 0's of its binary format, if the number of remaining bits is greater than 24, then it cannot be precisely represented.

Example
Due to the single precision floating point employed, the value of 2147483647 cannot be represented precisely. The nearest value 2147483648 will be returned instead.

```
Sub Main()
    Dim MyLong As Long
    Dim MyFloat As Float

    MyLong = -2147483648
    MyFloat = LONG2FLOAT(MyLong)      'the result is -2147483000
    Debug "LONG2FLOAT of -2147483648 : ", MyFloat, CR

    MyLong = 2147483647
    MyFloat = LONG2FLOAT(MyLong)      'the result is 2147484000
    Debug "LONG2FLOAT of 2147483647 : ", MyFloat, CR

End Sub
```

LOW

Syntax
LOW *Pin*

Operation
Sets the specified pin to a logic low level

- *Pin* — a constant or variable (0~23) that specifies the pin that the low level is to be applied to. For the 24-pin BASIC Commander®, the Pin value ranges from 0~15.

Description
This command will set the specified pin to a low level close to 0 volts. Usually, a pin must be changed to an input or output mode in advance to execute the corresponding input or output operations. However as only one pin is involved in a LOW instruction operation, the pin will be changed to the output mode automatically by the system. It is not necessary for the user to change the mode manually by executing an OUTPUT command beforehand. Refer to the HIGH command for other related information.

Example
Refer to the HIGH command for usage.

MID

Syntax
MID(*TargetString, Start, Length*)

Operation
Remains the middle Length letters starting from Start position of the specified string TargetSring with other letters truncated.

- ● *TargetString* ― the string operand of the MID function.
- ● *Start* ― a const or variable that specifies the starting position of a string
- ● *Length* ― a const or variable that specifies the length of the string to be copied.

Description
The MID command remains the middle Length letters starting from Start position of the specified string. If Start is greater than the number of characters in the target string, the string will become a null string. If the specified length is longer than the remaining size of the target string counting from position Start, the extra length will be ignored.

Example
Refer to the LEFT command for usage

OUTPUT

Syntax
OUTPUT *Pin*

Operation
Configures the specified Pin to the output mode.

- ● ***Pi*n** — a constant or variable (0~23) that specifies which pin will be set to the output mode. For the 24-pin BASIC Commander®, the Pin value ranges from 0~15.

Description
If you want to send a digital signal to the outside world, the pin that is used to send the signal should be changed to an output mode beforehand. Due to the simplicity of the pin-related input output command, the mode change is executed automatically by the system. The user need only simply use the input output commands without being concerned about the direction of the pins in question. However, this OUTPUT command is still provided and note that if you use the OUTPUT command to change a pin from an input mode to an output mode, the last write for the high/low state will ensure that there is no electrical conflicts on the pin that may cause damage.

Example
Refer to the INPUT command for usage.

PAUSE

Syntax
PAUSE *Duration*

Operation
This command will force the program to wait for the time specified.

- *Duration* ─ a variable or a constant that specifies the number of times and the duration for which the pause is active. The duration unit is 1 millisecond.

Description
This command will insert a delay between the previous statement and the next statement. Using a PAUSE command gives the user some control over the program execution speed by allowing delays to be inserted at any point in the program. The duration unit is 1 millisecond.

Example
The following program demonstrates how to make an LED flash with different pause time.
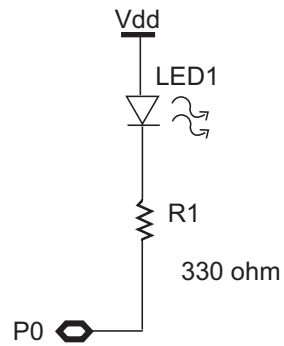
Figure 6-10 LED Flasher circuit

```
Sub Main()
    Dim Period As Word

START:
    Debugin "Set pause (0~65535 ms) between LED ON/OFF.", Period, CR

    Do
      HIGH 0
      Pause(Period)
      LOW 0
      Pause(Period)
    Loop
End Sub
```

PERIPHERAL

Syntax
PERIPHERAL *Name* **AS** *TypeName* **@** *ID*

Operation
Declare a peripheral module.
- *Name* — a name declared by user, similar to a variable name.
- *TypeName* — one of the peripheral module types, conceptually similar to a data type.
- *ID* — a const (0~31) that indicates the peripheral ID set by the DIP switch on the peripheral module.

Description
This command is used to declare a peripheral module by giving it a unique module name and a TypeName to specify what kind of module it is. The ID is a unique address through which all the modules attached to the cmdBUS™ can be addressed individually. The TypeNname belongs to an ever-growing list of Innovati® developed modules, where all newly-developed modules will be assigned a new type name. The Peripheral Module should be declared outside of all procedures. In other words, the Peripheral Modules act like a global variable, and can therefore be recognized from any place within the whole program.

Not like other commands, the Peripheral command is associated with the Peripheral Module Hardware. The documents accompanying the module hardware, will supply the module type name and associated dedicated functions will be provided. Note that more than one identical module can be connected to the cmdBUS™, if required, as long as they are assigned with different IDs.

Example

The following program demonstrates how to declare and give commands to the peripheral modules.

```
Peripheral myLCD As LCD2x16a @ 0        'a 2x16 LCD module with ID 0

Sub Main()
    myLCD.Display("Hi There!")
End Sub
```

PULSEIN

Syntax
PULSEIN *Pin, State, Variable*

Operation

This command will measure the pulse width of a pulse that appears on the specified pin.

- ● *Pin* — a constant or variable (0~23) that specifies the pin where the pulse width is to be measured. For the 24-pin BASIC Commander®, the Pin value ranges from 0~15.
- ● *State* — a constant or variable (0~1) that specifies whether the pulse is a positive or negative pulse. If a (0) is written here, then a negative pulse will be measured and if a (1) is written, then a positive pulse will be measured.
- ● *Variable* — a variable of WORD type where the measured pulse width value will be placed. The unit of measurement is of $5\,\mu$s. If the pulse measured is longer than the maximum of 65535 units or the pulse does not appear, a value of 0 will be returned, which indicates an invalid measurement.

Description

This command is used to measure the width of a pulse presented on a specified pin. It is useful in cases where pulse widths are used by some external hardware to express an external measurement. An example of this might be some external ICs which measure temperature, pressure etc, but whose output is expressed as a pulse width rather than a voltage. Such ICs could interface directly to the BASIC Commander® and the PULSEIN command used to measure the IC output. The pulse to be measured can be either a high pulse or a low pulse. It must be noted that when the command is executed, it will start to measure the pulse width when it receives the first pulse edge. This will be a high going edge for a high pulse, with State variable set to 1 or a low going edge for a low pulse, with State variable set to 0. The units measured will be 5

μs. If the pulse measured is longer than the maximum of 65535 units, it overflows to 0. However, the system will only wait for a specified time for this first edge to appear. If it does not appear then a zero value will be loaded into Variable and the program will continue and execute the next command. This prevents the program from hanging up at this command in the event of no pulse appearing. When the PULSEIN completes the measurement, either successfully or nor, the program will continue and execute the next command.

Example

The following example shows how to use the PULSEIN command to measure an external low pulse period.
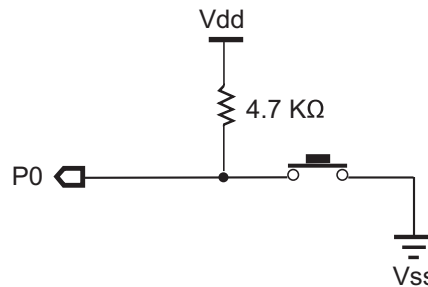
Figure 6-11 Measuring a Pulse Width on P0

```
Sub Main()

    Dim Result As Word

    Do
      Debug "Press the key.", CR
wait:
      Pulsein 0,0,Result      'measure low pulse on P0
      If Result=0 Then Goto wait 'not pressed or longer than 327 ms


      Result\=200             'convert 5us unit to 1ms unit
      Debug "The key pressed for ", Result," ms.",CR
    Loop
End Sub
```

PULSEOUT

Syntax
PULSEOUT *Pin, Duration*

Operation
This command will generate a pulse that appears on the specified pin.

- ● *Pin* 一 a constant or variable (0~23) that specifies the pin where the pulse will be generated. For the 24-pin BASIC Commander®, the Pin value ranges from 0~15.
- ● *Duration* 一 a constant or variable (0~65535) that specifies the length of the pulse width. The unit of measurement is 5 μs.

Description
The PULSEOUT command will generate a pulse of user specified width on a specified pin. The type of pulse is dependent upon the high low status of the pin that the pulse is to be generated on. If the pin is in a low state when the PULSEOUT command is executed, then a high pulse will be generated and if the pin is in a high state, then a low pulse will be generated.

Example
The following program shows how to use the PULSEOUT command to drive a servo. The servo position is usually controlled by a 0.5 ms to 2.5 ms high pulse within each 20 ms period. In this program we set two positions of 1 ms and 2 ms. The servo will rotate back and forth every 4 seconds. Connect the power and ground to the servo power and ground lines and P0 to the servo control line.

```
Sub Main()
    Dim b As Byte
    Low 0               'initialize Pin 0 to low to have a high pulse
    Do
      For b=0 To 100    '2 seconds servo rotation
      PULSEOUT 0,200    'move to position of 1ms pulse width
      Pause 19          'constitute a 20 ms cycle
Next
    For b=0 To 100      '2 seconds servo rotation
      PULSEOUT 0,400    'move to position of 2ms pulse width
      Pause 18          'constitute a 20 ms cycle
      Next
    Loop

End Sub
```

PWM

Syntax
PWM *Pin, Duty, Cycles*

Operation
This command will generate an output in pulse-width modulation format.

- ● ***Pin*** ― a constant or variable (0~23) that specifies the pin where the pulse width modulation signal is to be generated. This pin will be set to output mode initially then set to the input mode after the command has executed. For the 24-pin BASIC Commander®, the Pin value ranges from 0~15.
- ● ***Duty*** ― a constant or variable (0~255) that specifies the duty cycle of the output waveform.
- ● ***Cycles*** ― specifies the number of cycles (about 1.15 ms per cycle) for which the PWM output will be generated which in turn is actually specifying the amount of time for which the PWM output will operate. Specifying a zero value will not generate any PWM signal output. Its value can be a constant, variable or an expression and must have a range between 0 and 255.

Description
The PWM command allows the BASIC Commander® to generate an analogue voltage output on its digital pins. When you set an output pin high, the voltage of the pin will be close to 5V and set an output pin low, the voltage of the pin will be close to 0V. If you switch the pin rapidly between high and low, then you will get a voltage of the pin in between. The actual voltage that you can get depends on the time ratio of high to low, which is called the duty cycle. For example, if the Duty is 150, $(150/255) \times 5V = 2.94V$, the PWM command outputs a train of pulses whose average voltage is 2.94V. The PWM command has a fixed period of 1.15 ms as shown in the following figure.
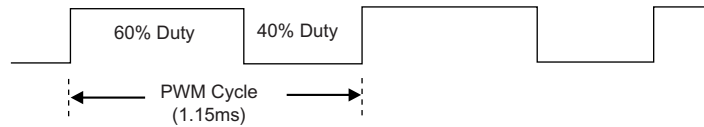
Figure 6-12 PWM Waveform with 60% Duty Cycle

The following low pass resistor/capacitor circuitry filters out the pulses and maintains the analogue voltage after the command has finished. The analogue voltage that will be held depends on how much current is drawn from it by external circuitry, including the capacitor current leakage. In order to hold the voltage, a periodical PWM command execution to charge the resistor/capacitor circuit is needed.
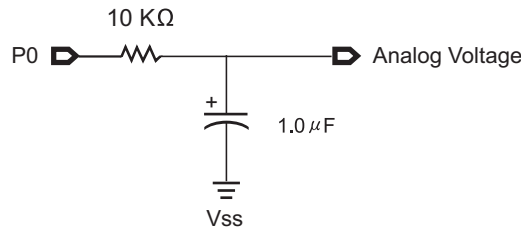

Figure 6-13 Adding a Low Pass Filter circuit to the PWM output

It takes time to charge a capacitor to the desired voltage in the beginning. You may use the rule of thumb formula: Charge time $= 5 \times R \times C$, to estimate at least how many cycles should be given for charging. For instance, computation of the charge time is shown in the following formula.

Charge time $= 5 \times 10 \times 10^3 \times 1 \times 10^{-6} = 50 \times 10^{-3}$ seconds, or 50 ms.

For the BASIC Commander®, each cycle is about a 1.15 ms, it would therefore take at least 44 cycles to charge the capacitor. Assuming Pin 0 is used, then the command will be:

```
PWM 0, 150, 44                          'charge to 2.94 V, on Pin 0
```

Example

```
Sub Main()
    Dim f As Float
    Dim duty As Byte
    Dim b As Byte

Start:
    Debugin "Enter desired voltage(0~5V) : ",f,CR
    If f<0 Or f>5 Then
       Debug "Invalid value.",CR
       Goto Start
    Else
       f=f*255/5
       duty=float2byte(f)
       Debug "duty=",duty,CR
       For b=1 to 100     'Hold the voltage for 5 seconds
       PWM(0,duty,44)
       Next b
    End If

    Goto Start
End Sub
```

RANDOM

Syntax
RANDOM *Variable*

Operation
This command will generate a random number.

- *Variable* ― the generated random number will be placed in this previously defined variable

Description
This command will generate pseudo-random number and place it in the defined variable. Pseudo-random means that the number is not strictly random as a sequence will be formed which will repeat itself after a period of time. However this method of random number generation is usually suitable for most application purposes. To generate a true random number some other externally controlled event must be factored into the number generation. The random number sequence will also depend upon the initial number chosen to initiate generation. This can be controlled by setting up an initial value for the variable. This value will then be taken as the basis for the random number generation. This is known as the random number generation seed value.

Example
The following program demonstrates how to combine an external push button and RANDOM command to make your own lottery number generator.
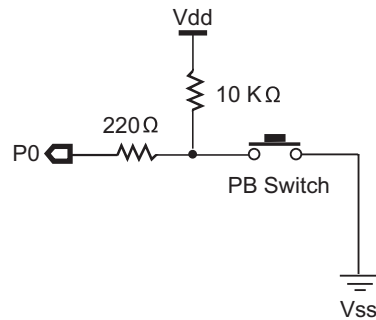
Figure 6-14 Generating Random Numbers with Push Button & RANDOM command

```
Sub Main()
    Dim Key As Byte
    Dim Seed As Dword =10
    Dim X,Y As Byte
    Dim Temp As Dword
    Dim NumArray(5) As Dword
    Do
       Debug "Press push button to get your lucky numbers.", CR
Wait:
       RANDOM(Seed)
       If IN(0)<>0 Then Goto Wait
       Debug "The lucky numbers are "
       For Y=0 To 5
GenRandom:
          RANDOM(Seed)
          Temp = Seed Mod 47
          Temp += 1
          For X = 0 To 5
          If Temp = NumArray(X) And X <> Y Then
```

```
            Goto GenRandom
          End If
        Next
        NumArray(Y) = Temp
        Debug NumArray(Y), " "
      Next
      Debug CR
    Loop
End Sub
```

RCTIME

Syntax
RCTIME *Pin, State, Variable*

Operation
Measure the charge/discharge time of resistor/capacitor (RC) circuit.

- ● *Pin* － a constant or variable ranging from 0 to 23 that specifies the I/O pin to use. This pin will be configured as input mode. For the 24-pin BASIC Commander®, the Pin value ranges from 0~15.
- ● *State* － a constant or variable of 0 or 1 to specify the desired measure state. Once Pin is not in State, the command ends and stores the result in Variable.
- ● *Variable* － a variable, usually a word, in which the time measurement will be stored. The units will are 5 μs.

Description
RCTIME can be used to measure the charge or discharge time of an external resistor-capacitor circuit. This allows you to measure the resistance or capacitance value of external R or C type sensors such as thermistors, potentiometers or capacitive humidity sensors.

When RCTIME executes, it configures the Pin as an input, then starts a counter. It stops this counter as soon as the specified pin is no longer in State (0 or 1). If the pin is not in the state when the command executes, RCTIME will return a 1 in Variable. If the pin remains in State longer than 65535 timing cycles, RCTIME returns a 0.

The following two figures show the circuit needed for the RCTIME command. You may choose either of them with an approximate 0.7 VDD voltage span before the RCTIME command stops the timer.
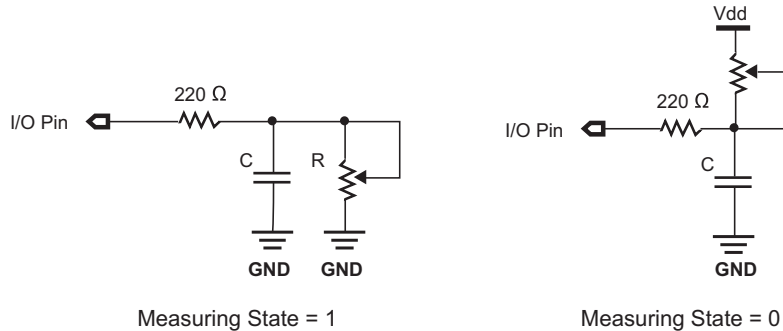
Figure 6-15 Circuits needed for RCTIME command

Once the timer has stopped, the counter value is stored in the variable. However there must be a method of obtaining the R or C value? The RC time constant, or tau (τ) for short, the multiplication of R and C, represents the time required for a given RC circuit to charge or discharge 63% of the initial voltage.

The general RC charging/discharging formula:

$$\text{time} = - \tau \times (\ln (\text{Vfinal} / \text{Vinitial}))$$

Let's take a 10 kΩ resistor and 0.1 μF capacitor for instance. Since for State 1 or 0, the Vfinal / Vinitial is fix at 0.4, we can estimate the time:

$$\text{time} = 9.163 \times 10^{-4}$$

As the units for the timer counting is 5 μs, so therefore the time is approximately 183 counts.

Hence, we can use the rule of thumb to calculate the R or C value in question.

$$\text{RCTIME counts} = 183 \times \text{R (in k}\Omega) \times \text{C (in } \mu\text{F)}$$

Before RCTIME executes, the capacitor must be charged to 5V for State 1 or discharged to 0V for State 0. You may use the rule of thumb formula to estimate at

least how much time should be given for charging.

$$\text{Charge time} = 5 \times \tau$$

For instance, assuming the capacitor is 0.1uF and the resistor is 220 ohm, as depicted in the figure, the charging time should be at least:

$$\text{Charge time} = 5 \times 220 \times 0.1 \times 10^{-6} = 0.11 \text{ ms.}$$

Example

The following program demonstrates how to use the RCTIME command to measure and display the resistance value of an externally connected potentiometer, which is connected to Pin 0 via an RC circuit as shown above.

```
Sub Main()
    Dim result As Word
    Dim resultf As Float
    Do
      High 0
      Pause 1
      RCTIME 0,1,result
      resultf=Word2float(result)
      resultf=resultf/183*10
      Debug "The potentiometer value (KΩ) : ", resultf, CR
    Loop
End Sub
```

READPORT

Syntax
Result = **READPORT** *Port*

If the port number is a constant, you may also use one of the following format types instead.

Result = **READPORT0**
Result = **READPORT1**
Result = **READPORT2**

Operation
To read the specified I/O port.
- *Port* — a constant or variable (0 ~2) that specifies the port number. Port 0 consists of pin P0~P7, while Port 1 consists of pin P8~P15, and Port2 consists of pin P16~P23. For the 24-pin BASIC Commander®, the Port value is 0 or 1.
- *Result* — a Byte variable to receive the external logic level on Port.

Description
The READPORT command is used to read digital signals from the outside world. Unlike the IN command, the I/O mode of the pin will not be changed to the input mode automatically. For the READPORT command, there are eight pins involved and each pin may be set to the input or output mode. You have to configure each bit explicitly. For pins which with smaller index are of the lower bit order of a data byte. Note that when the I/O pins are configured in the input mode, it is recommended that an external 10 KΩ pull-high resistor for each I/O pin is connected. Otherwise due to the high impedance floating state of these pins, the I/O pin could be randomly read as either a 1 or 0 which would not reflect the true condition of the digital signal on the pin

For those pins that are set in the output mode, reading data from those pins will not read the actual status of the I/O pin, as only the data previously written to the output pin is read.

Example
Connect the eight I/O pins of Port 1 as inputs and the eight pins of Port 0 as outputs as shown below. The push button status will be shown on the LEDs.
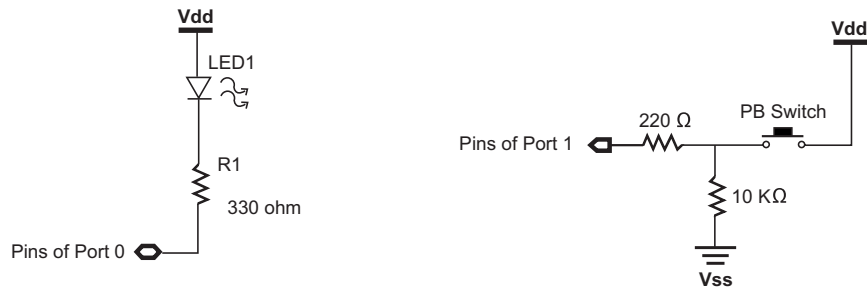

Figure 6-16 Port 1 I/O pins as inputs connected to Port 0 pins as outputs

```
Sub Main()
    Dim IOStatus as Byte

    SETDIRPORT1(&B11111111)          'set Port 1 as input
    SETDIRPORT0(&B00000000)          'set Port 0 as output
    Do
      IOStatus=READPORT1()
      WRITEPORT0(IOStatus)
    Loop
End Sub
```

RESETMODULE

Syntax
RESETMODULE

Operation
Rest all the peripheral modules that connected to the cmdBUS™.

Description
This command is used to reset all the modules that connected to the cmdBUS. Usually this command is employed when the BASIC Commander® fails to communicate with the peripheral module(s) due to the unpredictable peripheral module failures. Note that this command pulls low the SYN line of the cmdBUS™, even the normal operating modules will be forced to restart, too.

RETURN

Syntax

RETURN {*ReturnValue*}

Operation

Returns from a Sub, Function or Event. If returning from a Function, a ReturnValue must be provided.

- ● *ReturnValue* ─ optional return value which may appear only in the Function body.

Description

The Sub, Function or Event body will end when the END SUB, END FUNCTION or END EVENT is encountered respectively. However, if returns are required under other situations, the RETURN command can be used. The ReturnValue is only used in the Function body to pass the result of the executed function. If the RETURN command is encountered before any Sub, Function or Event is invoked, it will result in a branch to the end of the main program.

Example

```
Function Max(Z1 As Integer, Z2 As Integer) As Integer
    If Z1>Z2 Then
       Return Z1
    Else
       Return Z2
    End If
End Function
Sub Main()
```

```
    Dim X, Y As Integer
    Debugin "Enter X = ",X
    Debug X,CR
    Debugin "Enter Y = ",Y
    Debug Y,CR
    Debug "Max(X,Y)=", Max(X,Y), CR
End Sub
```

REVERSE

Syntax
REVERSE *Pin*

Operation
Reverses the specified pin's direction whether in the input or output mode.

- ● *Pin* — a constant or variable (0~23) that specifies which pin will be set to the opposite mode of either input or output mode. For the 24-pin BASIC Commander®, the Pin value ranges from 0~15.

Description
If you want to read or write a digital signal from or to the outside world, the pins must be changed to the corresponding mode beforehand. Due to the simplicity of the pin-related input output command, the mode change job is automatically executed by the system. This is done by simply using the input output commands without worrying about the direction of the pins in question. In addition to the INPUT and OUTPUT command, the REVERSE command is also provided, which reverses the current input or output mode. Caution must be taken with your application circuit to prevent error paths that may result due to unexpected input output direction changes causing damage to your external circuit. Please check the INPUT and OUTPUT command to see the notes and examples on using them.

Example
The following example changes the I/O directions of P0, which connects an LED through a resistor. The LED will turn on/off accordingly.
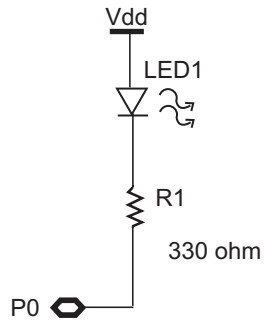
Figure 6-17 Input/Output Circuit

```
Sub Main()
    Dim key As Byte

Start:
    WRITEPORT0 & H00  ' Write low to output buffers

    Do
      Debugin "Input any key to toggle LED On/Off.", %CHR key, CR
      REVERSE 0       ' Switch P0 to OUTPUT mode, turn On LED0
    Loop

End Sub
```

RIGHT

Syntax

RIGHT(*TargetString*, *length*)

Operation

Retains the rightmost length of letters of the specified string TargetSring with other letters truncated.

- ● *TargetString* — the string operand of the RIGHT function.
- ● *Length* — a variable that specifies the length of the string to be kept.

Description

The RIGHT command retains the rightmost letters specified by the Length and truncates the rest of the letters of the string. If the specified length is longer than the size of the target string, the extra length will be ignored and the string remains unchanged.

Example

Refer to the LEFT command for usage.

SELECT... CASE

Syntax

SELECT {CASE} *Expression*
{ CASE *Const*
Statements}
{ CASE ELSE
Statements }
END SELECT

Operation

The SELECT command executes commands based on the value of an expression.

- *Expression* — a variable, a constant, or an expression.
- *Const* — a constant to be compared to Expression, if it is the same, the commands in this CASE will be executed.
- *Statements* — any valid innoBASIC™ statement.

Description

The SELECT command is an advanced decision-making structure using the compound IF...THEN...ELSE structure to execute one of several possible actions based on the value of a single expression. When a SELECT command is executed, the SELECT expression is evaluated first, and then compares it with the CASE constant in the textual declaration order. If the first CASE constant meets the evaluated value then its related instruction block will be executed. If the no CASE constant meets the evaluated value and there is a CASE ELSE command, that block will be executed. Once a block has finished executing, execution passes to the END SELECT command. The EXIT SELECT command may be placed in the loop body, which exits the current loop immediately before the loop limit test is executed. Note that the optional CASE can come after the SELECT to support the conventional command format.

Example

Refer to Chapter 5 for further detailed examples.

SETDIRPORT

Syntax
SETDIRPORT *Port*, *Dir*

If the port number is a constant, you may also use one of the following format types instead.

SETDIRPORT0 *Dir*
SETDIRPORT1 *Dir*
SETDIRPORT2 *Dir*

Operation
To set the I/O direction settings of the specified port.

- *Port* — a constant or variable (0 ~2) that specifies the port number. Port 0 consists of pins P0~P7, while Port 1 consists of pins P8~P15, and Port 2 consists of pins P16~P23. For the 24-pin BASIC Commander®, the Port value is 0 or 1.
- *Dir* — a byte specifies the I/O directions. Each bit of the data byte specifies the direction of each pin, 0 is output and 1 is input.

Description
If you want to read digital signals from the outside world by using the READPORT command, the corresponding pins must be first configured to the input mode. The SETDIRPORT command is used to configure the I/O direction settings of the specified port. Each pin of a port can be configured independently. Data 0 stands for output and 1 stands for input mode. For pins which with smaller index are of the lower bit order of a data byte. For example, the P0 setting will appear in Bit 0 of the data read. The I/O directions default to input after the program starts.

Note that when the I/O pins are configured to be in the input mode, it is recommended that an external 10KΩ pull-high resistor is connected to each I/O pin. Otherwise due to the high impedance floating state, the I/O pin may be randomly read as 1 or 0, which does not reflect the actual signal value on the pin.

Example

Refer to the GETDIRPORT command for usage.

SGN

Syntax
Result = **SGN(***Argument***)**

Operation
To return the sign value of a floating-point value.
- *Argument* — the floating-point operand of the SGN function.
- *Result* — a SHORT variable that receives the result of the SGN function.

Description
The SGN command returns the sign value of a floating-point value. If the floating-point value is positive then 1 is returned; If the floating-point value is negative then -1 is returned; If the floating-point value is of value 0 then 0 is returned.

Example

```
Sub Main()
    Dim Result As Short

    Result = SGN(-0.1)
    Debug "SGN(-0.1) = ", Result, CR
    Result = SGN(0)
    Debug "SGN(0) = ", Result, CR
    Result = SGN(0.1)
    Debug "SGN(0.1) = ", Result, CR
End Sub
```

SHORT2FLOAT

Syntax
Result = **SHORT2FLOAT(***Argument***)**

Operation
To convert a SHORT value into its floating-point format.
- ● *Argument* — the SHORT operand of the SHORT2FLOAT function.
- ● *Result* — a floating-point variable that receives the result of the BYTE2FLOAT function.

Description
The SHORT2FLOAT command converts a SHORT value into its floating-point format. The floating-point result will be an integral value ranging from -128.0 to +127.0.

Example

```
Sub Main()
    Dim MyShort As Short
    Dim MyFloat As Float
    MyShort = -128
    MyFloat = SHORT2FLOAT(MyShort)     'MyFloat has the value -128.0
    Debug "SHORT2FLOAT of -128 : ", MyFloat, CR
    MyShort = 127
    MyFloat = SHORT2FLOAT(MyShort)     'MyFloat has the value 127.0
    Debug "SHORT2FLOAT of 127 : ", MyFloat, CR
End Sub
```

SIN

Syntax

Result = **SIN(*Argument*)**

Operation

To execute a mathematical sine function.

- *Argument* ─ the floating-point operand of the sine function with a range from 0 to 2π
- *Result* ─ a floating-point variable to receive the result of the sine function.

Description

The SIN function returns the sine value of a floating-point argument ranging from 0 to 2π. Note that the argument is in units of radians. If converting to degrees, note that 360 degrees is equal to 2π radians.

Example

```
Sub Main()
    Dim MyFloat As Float
    Dim Result As Float

    MyFloat = pi/4
    Result = SIN(MyFloat)     'the Result is 0.707107
    Debug "SIN(pi/4)=", Result, CR
End Sub
```

SQRT

Syntax
Result = **SQRT(*Argument*)**

Operation
To return the square root of a floating-point argument value.

- *Argument* ─ the floating-point operand of the SQRT function.
- *Result* ─ a floating-point variable that receives the result of the SQRT function.

Description
The SQRT command returns the square root of a floating-point value. The result of the SQRT is a non-negative value.

Example

```
Sub Main()
    Dim MyFloat, Result As Float

    MyFloat=100
    Result = SQRT(MyFloat)    'the result is 10
    Debug "SQRT of 100 : ", Result, CR
    MyFloat=-100
    Result = SQRT(MyFloat)     'the result is NaN
    Debug "SQRT of -100 : ", Result, CR
End Sub
```

STRING2FLOAT

Syntax
FloatVar = **STRING2FLOAT**(*StringVar*)

Operation
To convert an ASCII character string into a floating-point value.
- *StringVar* — the ASCII character string operand of the STRING2FLOAT function.
- *FloatVar* — a Float type variable that receives the result of the conversion.

Description
The STRING2FLOAT command converts an ASCII character string into a floating-point value. The value expressed into the ASCII character string must be a floating-point format. The floating-point format is composed of a sign character, a 5-digit mantissa with radix point, an exponent sign character E, the exponent sign character and 2-digit exponent. For example +3.1416E-01.

Example

```
Sub Main()
    Dim MyString As String * 12
    Dim MyFloat As Float
    MyString = "9500"
    MyFloat =STRING2FLOAT(MyString)          'incorrect format
    Debug "STRING2FLOAT of ", MyString, ":", MyFloat, CR

    MyString = "+3.1416E-01"
    MyFloat =STRING2FLOAT(MyString)          'the result is +3.1416E-01
    Debug "STRING2FLOAT of ", MyString, ":", MyFloat, CR
```

```
    MyString = "0.031416"
    MyFloat =STRING2FLOAT(MyString)              'incorrect format
    Debug "STRING2FLOAT of ", MyString,": ", MyFloat, CR
End Sub
```

STRREVERSE

Syntax
STRREVERSE(*TargetString*)

Operation
Returns a string with all characters in reversed order of the given string.

- *TargetString* — the string to be converted.

Description
The STRREVERSE command reverses all the characters of the given string.

Example

```
Sub Main()
    Dim TargetString As String * 12

    TargetString = "Hello World!"
    Debug "TargetString is ", TargetString, CR
    STRREVERSE(TargetString)            'MyString contains "!dlrow olleH"
    Debug "TargetString has been reversed to ", TargetString, CR
End Sub
```

SUB...END SUB

Syntax
SUB *SubName(ArgList)*
{*Statements*}
END SUB

Operation
Declares a procedure with an optional argument list.
- *SubName* — the name of the procedure contains a sequence of letters, digits and an underscore. The leading character must be a letter.
- *ArgList* — is a list of the arguments required in the procedure. The argument is preceded with either a byval or byref modifier for argument passing. The parenthesis cannot be omitted even if there is no argument required.
- *Statements* — any valid innoBASIC™ statement.

Description
The SUB command declares a procedure which can be invoked by its SubName to execute some specific program.

Example

```
Sub SayHi()
    Debug "Hi!",CR
End Sub
```

TOGGLE

Syntax
TOGGLE *Pin*

Operation
Toggles the state of an output pin

- *Pin* — a constant or variable (0 ~23) that specifies which pin the high or low level is to be applied to. For the 24-pin BASIC Commander®, the Pin value ranges from 0~15.

Description
This command will toggle the specified pin to a high level of 5 volts or a low level of 0 volts. Usually, a pin must be changed to either an input or output mode in advance to execute the corresponding input or output operations. However, because only one pin is involved in the TOGGLE operation, the pin will be changed to the output mode automatically by the system. It is not necessary to first execute an OUTPUT command beforehand. Please refer to the HIGH command for other related information.

Example
Refer to the HIGH command for usage.

UCASE

Syntax
UCASE(*TargetString*)

Operation
Changes the specified string to upper case letters.

- *TargetString* ― the string to be converted.

Command Description
The UCASE command changes the specified string to upper case letters.

Example
Refer to the LCASE command for usage.

WORD2FLOAT

Syntax
Result = **WORD2FLOAT**(*Argument*)

Operation
To convert a Word value into its floating-point format.
- *Argument* ― the Word operand of the WORD2FLOAT function.
- *Result* ― a floating-point variable that receives the result of the WORD2FLOAT function.

Description
The WORD2FLOAT command converts a Word value into its floating-point format. The floating-point result will be an integral value ranging from 0.0 to 65535.0.

Example

```
Sub Main()
    Dim MyWord As Word
    Dim MyFloat As Float

    MyWord = 0
    MyFloat = WORD2FLOAT(MyWord)
    Debug "MyWord = ", MyWord, "MyFloat = ", MyFloat, CR

    MyWord = 65535
    MyFloat = WORD2FLOAT(MyWord)
    Debug "MyWord = ", MyWord, "MyFloat = ", MyFloat, CR
End Sub
```

WRITEPORT

Syntax
WRITEPORT *Port, Data*

If the port number is a constant, you may also use one of the following format types instead.

WRITEPORT0 *Data*
WRITEPORT1 *Data*
WRITEPORT2 *Data*

Operation
To write data to the specified I/O port without changing the direction settings.
- ***Port*** — a constant or variable (0 ~2) that specifies the port number. Port 0 consists of Pins P0~P7, while Port 1 consists of Pins P8~P15, and Port 2 consists of Pins P16~P23. For the 24-pin BASIC Commander®, the Port value is 0 or 1.
- ***Data*** — a constant or variable which specifies the data to be written to the specified Port.

Description
The WRITEPORT command is used to write digital signals to the I/O port. Unlike the OUT command, for which the I/O mode of the pin will be changed to the output mode automatically, for the WRITEPORT command, there are 8 pins involved and each pin may be set to either input or output mode. You have to configure each bit explicitly. Data 0 will set the corresponding pin to a low state and 1 to a high state. For pins which with smaller index are of the lower bit order of a data byte.

For those pins that are setup in the input mode, writing data to these pins will not affect the I/O pins. If these pins are reconfigured to the output mode at a later time, the previous output data stored in the buffer will then take effect.

Example

Refer to the READPORT command for usage.

Appendix

Appendix A 一 ASCII Table

The following table lists the first 128 ASCII characters. Note that the first 32 codes are control codes. They don't have standardized screen symbols. Their symbols are represented with names used in referring to these codes. For example, to move the cursor to the beginning of the next line, the LF (Line Feed) and CR (Carriage Return) control codes are used.

Dec	Hex	Char	Name / Function	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	NUL	Null	32	20	space	64	40	@	96	60	`
1	01	SOH	Start of Heading	33	21	!	65	41	A	97	61	a
2	02	STX	Start of Text	34	22	"	66	42	B	98	62	b
3	03	ETX	End of Text	35	23	#	67	43	C	99	63	c
4	04	EOT	End of Transmission	36	24	$	68	44	D	100	64	d
5	05	ENQ	Enquiry	37	25	%	69	45	E	101	65	e
6	06	ACK	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	BEL	Bell	39	27	'	71	47	G	103	67	g
8	08	BS	Backspace	40	28	(	72	48	H	104	68	h
9	09	HT	Horizontal Tab	41	29	)	73	49	I	105	69	i
10	0A	LF	Line Feed	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	Vertical Tab	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	Form Feed	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	Carriage Return	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	Shift In	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	Data Link Escape	48	30	0	80	50	P	112	70	p
17	11	DC1	Device Control 1	49	31	1	81	51	Q	113	71	q
18	12	DC2	Device Control 2	50	32	2	82	52	R	114	72	r
19	13	DC3	Device Control 3	51	33	3	83	53	S	115	73	s
20	14	DC4	Device Control 4	52	34	4	84	54	T	116	74	t
21	15	NAK	Negative Acknowledge	53	35	5	85	55	U	117	75	u
22	16	SYN	Synchronous Idle	54	36	6	86	56	V	118	76	v
23	17	ETB	End of Trans. Block	55	37	7	87	57	W	119	77	w
24	18	CAN	Cancel	56	38	8	88	58	X	120	78	x
25	19	EM	End of Medium	57	39	9	89	59	Y	121	79	y
26	1A	SUB	Substitute	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	File Separator	60	3C	<	92	5C	\	124	7C		
29	1D	GS	Group Separator	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	Record Separator	62	3E	>	94	5E	^	126	7E	~
31	1F	US	Unit Separator	63	3F	?	95	5F	_	127	7F	DEL

Appendix B - InnoBASIC™ Keywords

The following table lists all the keywords reserved in the innoBASIC™ language.

ABS	ACOS	AND	AS
ASIN	ATAN	ATAN2	BELL
BKSP	BOOLEAN	BUTTON	BYREF
BYTE	BYTE2FLOAT	BYVAL	CALL
CASE	CEIL	CHECKMODULE	CLREOL
CLREOS	CLS	CONST	CONTINUE
COS	COUNT	CSRD	CSRL
CSRR	CSRU	CR	DEBUG
DEBUGIN	DEFAULT	DIM	DIRPIN0 ~ 31
DIRPORT0 ~ 2	DO	DWORD	DWORD2FLOAT
ELSE	ELSEIF	END	ENUM
EVENT	EXIT	EXP	EXP10
FALSE	FLOAT2BYTE	FLOAT2DWORD	FLOAT2INTEGER
FLOAT2LONG	FLOAT2REALSTRING	FLOAT2SHORT	FLOAT2STRING
FLOAT2WORD	FLOOR	FOR	FUNCTION
GETDIRPORT	GETDIRPORT0 ~ 2	GOTO	HIGH
HOME	IF	IN	INPUT
INTEGER	INTEGER2FLOAT	LCASE	LCDCMD
LCDIN	LCDOUT	LEFT	LEN
LOG	LOG10	LONG	LONG2FLOAT
LOOP	LOW	MID	MOD
NEXT	NOT	OR	OUTPUT
PAUSE	PERIPHERAL	PERSISTENTBYTE	PERSISTENTDWORD
PERSISTENTFLOAT	PERSISTENTINTEGER	PERSISTENTLONG	PERSISENTSHORT
PERSISTENTWORD	PIN0 ~ 31	PORT0 ~ 2	PULSEIN
PULSEOUT	PWM	RANDOM	RCTIME
READPORT	READPORT0 ~ 2	RESETMODULE	RETURN
REVERSE	SELECT	SETDIRPORT	SETDIRPORT0 ~ 2
SGN	SHORT	SHORT2FLOAT	SIN
SQRT	STEP	STRING	STRING2FLOAT
STRREVERSE	SUB	TAB	THEN
TO	TOGGLE	TRUE	UCASE
UNTIL	WHILE	WORD	WORD2FLOAT
WRITEPORT	WRITEPORT0 ~ 2	XOR	