Legal and License Information

Manual Version and accuracy (revision E)

This is manual revision E. All information in this manual is believed to be correct at the time of publication. However, as with any product documentation, there may be unintentional errors or omitted information. Protean will periodically update this manual and updates are available on-line at the Protean Logic Inc. web site. Feel free to download future copies. Printed versions of this and future manuals are available as well for modest cost.

Single User License Agreement

The FBASIC™ Language, Compiler, and associated Tools are protected under United States copyright law. Protean Logic grants the single user license holder the right to use this software on one or many computers, provided that not more that one person is using this software AT THE SAME TIME. Separate licensing agreements with Protean Logic will supersede this single user license agreement. Contact Protean Logic for information regarding possible site licensing or educational licensing.

Limited Warranty

Protean Logic warrants the disks and materials contained in the development kit free from defects in materials or workmanship for a period of 30 days from the date of purchase. If, in this time the disks are found to be defective, they may be returned to Protean Logic for replacement. Protean will refund the purchase price of complete and undamaged development kits at the customers demand if such demand is made within 30 days from the date of purchase.

Protean Logic makes no representations or warranties as to the merchantability or fitness of this product to a particular purpose. Products developed with the development kit should not be used in a life support application without express written agreement with Protean. Protean makes no other warranty, either expressed or implied.

Technical Support

Protean Logic maintains an internet web site for all customers. Software updates are available from the Protean to all customers. Simply e-mail "support@protean-logic.com" and provide the invoice number and date of purchase in your message. We will e-mail you a reply with an attachment of the latest software. The URL for the Protean Logic web site is, "http: //www.protean-logic.com".

Protean Logic can be reached directly at (303) 828 9156. Protean Logic also responds to FAX messages daily. The FAX number is (303) 828-9316.

Properties

FBASIC and Protean Logic are trademarks owned by Protean Logic. All other trademarks contained in this manual are the property of their respective holders.

1 Getting Started

1.1 The Goal Here is "Instant Gratification"

In this chapter you will learn how to connect the TICkit hardware to a console computer, use the FBASIC compiler program to compile a sample program, use the TICkit download program to copy the compiled program into the TICkit's EEprom, and execute the program on the TICkit. To do this you will need a TICkit circuit board, an IBM compatible computer with at least 500K of available memory, one free serial port, and a special serial cable for downloading from the IBM computer to the TICkit. A diagram of this cable is shown in Appendix A of this manual if you need to make another for some reason.

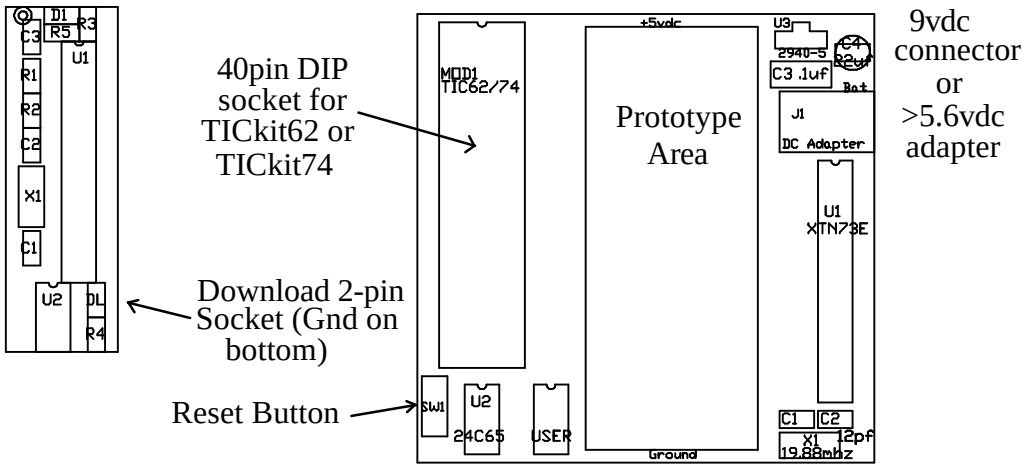
Throughout this manual, the IBM computer is referred to as the "Console". Downloading refers to the process of copying a program from the Console to the TICkit EEprom.

Debugging refers to the process of watching the TICkit execute a program. Debugging is accomplished by running the debugging program on the console, which is connected to the TICkit via a two wire cable, and performing special debug commands that the TICkit understands.

1.2 Overview of the process

1. Connect The download cable between a serial port on your computer and the two pin download socket on the TICkit Module or similar connector on your custom circuit.
2. Connect power to the TICkit. If you purchased a project board, simply plug in the wall adapter in it's socket. If you are using the module in your own prototype, apply a regulated 5 Vdc to the appropriate pins on the TICkit. +5 Vdc (vdd) connects to pin 30 of the module and Gnd (vss) connects to pin 29 or pin 10. If you are using the 28 pin processor IC in your own design pin 20 is +5 (vdd) and pin 8 and 19 are the ground pins.
3. Install the TICkit software by placing the supplied disk in a drive and typing: a:install or b:install (depending on which floppy drive the disk is in).
4. Run the TICkit debugger on the console computer by changing to the directory where the software is installed and typing: debug 1 or debug 2 or debug 3 or debug 4 (depending on which serial port you plugged the download cable into).
5. Reset the TICkit by pressing the button on the T62-PROJ board or by removing and re-applying power to the TICkit module. If everything is working, Some additional information will be displayed in the dialog box on the console computer that looks something like:
   TOKEN: E0 PC: 01FA Command:
6. Quit the debug program by pressing 'Q'.
7. Compile the example program by typing: fbasic first62
8. Download the program using the debugger. Type: debug 1 first62 (the port number may not be one, use the same number as you used in step 4). Reset the TICkit and then press 'D' at the command prompt on the console computer. Answer 'Y' when asked if you wish to download.
9. Execute the downloaded program by pressing 'E' at the command prompt on the console computer.

1.3 Step One: Connect the cables



The TICkit 62 is the current version of the TICkit. It is a small PCB module approximately the same size as a 40 pin DIP package. Only 32 of these pins are actually used by the TICkit 62, the rest are reserved for possible use in the future TICkit products. The TICkit 62 can be plugged into a 40 pin DIP socket, a solderless breadboard, or into Protean's T62-PROJ project board. The idea here is to allow projects to be built on inexpensive carrier boards and then to move the processor modules from project to project. The download socket for the TICkit62 module is a vertical 2-pin socket located at the bottom of the module next to the socketed EEprom. The ground pin is the lower pin but no damage is done by reversing the polarity. The power connection is made through the DIP pins of the module consult the pin-out diagram for connection information. The download connection is also available through the DIP pins. If a T62-PROJ carrier board is used, simply plug the module into the 40pin DIP socket and apply power at the adapter jack on the left side of the board or solder the supplied 9volt battery plug in the holes provided and connect a battery.

Connect the Download cable to a free serial port on the Console. The download cable connector has a 9 pin D connector for the console serial port. If your computer has only a 25 pin connector, a 9 to 25 pin adapter will work fine. Also a 25 pin female connector (like Radio Shack # 276-1548) may be wired up according to the download circuit shown in appendix 'A' of this manual. Plug the two pin connector of the download cable into the two pin socket labeled, "DL" on the TICkit. The "DL" socket is not polarized in any way, so there is a possibility the download cable will be inserted incorrectly into the TICkit. The ground pin (the wire with the markings) should be to the left or bottom. If the cable is inserted incorrectly, no damage will occur, simply unplug and then re-plug the Download cable with the correct polarity so the download software will connect with the TICkit.

Every time the TICkit is reset, it tests for a reasonable response to a small message that is sent out the DL port. If there is a correct response to the message, the TICkit assumes it is connected to a Console and enters the debugging mode. If there not a correct response, or there is not a correct idle state voltage on the DL port, the TICkit will simply start executing the program that is contained in its EEprom.

Once the power and download cables are properly connected, the console needs to establish communication with the TICkit. On the console computer, go to the directory where the software is located. This may be located on a floppy disk if you did not copy the files from the distribution disk onto your hard disk drive. You can install the files to your hard disk by running the install.exe program on the release disk (type a:install at the DOS prompt). Once the software is installed, change the directory to where your TICkit software was placed. At the DOS prompt, type:

```
DEBUG62 <serial_port_number>     (com2 example: DEBUG62 2)
```

Or, if you are using a TICkit57 use the command line that follows.

```
DEBUG57 <serial_port_number>     (com2 example: DEBUG57 2)
```

All aspects of DEBUG are the same between the two programs. However, internal communication offsets differ for each device and the proper program must be used for correct memory information to be displayed.

The "serial_port_number"should be the number of the COM port that the download cable was plugged into. The screen of the Console will contain a large, divided box in the lower half. The left side of this box is called the "debug dialog" area and will display information about the debug session. When the TICkit and the Console connect, a message indicating connection will display along with certain information like the current token, PC, MP, and SP. Do not worry about the exact meaning of these registers at this point. Usually the TICkit will require resetting to cause it to connect to the Console. Reset the TICkit by either pushing the reset button in the middle left of the TICkit or by removing then re-applying power to the TICkit.

At this point, the debugging program on the Console should display a message indicating it has connected with the TICkit. If this is not so, verify that the cable is installed correctly. Check that the two pin connector is correctly plugged into the TICkit. If this connector is reversed, the TICkit will not connect. Verify that the debug program was started on the correct serial port. If, after checking all these possibilities, the TICkit still will not connect, contact Protean via voice at (303) 828-9156, FAX (303) 828-9316, or e-mail: support@protean-logic.com.

1.4 Step two: Compiling a program

At this point, the TICkit and the Console are successfully connected. Quit the debugger program by pressing 'Q' or <cntrl-Z>. We will come back to using the debugger later.

Some sample programs were included with the compiler. One of these programs, called first.bas is what we will use to demonstrate how to compile, download, and run a program. The compiler will need the program to be contained in an ASCII text file in the current DOS directory. The compiler is invoked simply by typing FBASIC and then the name of the source file to be compiled. In our example, the program is in an ASCII text file called "first.bas", so type:

```
fbasic first62
```

The compiler reports a few lines of progress while the program is compiled and then returns to the DOS prompt. If the program compiled successfully, two files will have been made by the compiler. These files are "first.tkn" which is the file to download to the TICkit, and "first.sym" which is a file for debugging purposes that tells the debugger where lines of code are and where global variables are.

When you write your own programs, simply prepare a source file using any editor. The DOS edit command will work just fine. Then follow the procedure above to compile your program. It is normal to have errors reported while compiling. If your program causes errors during compilation, re-edit and compile your program until no errors are produced. Warning lines during compilation are not technically errors, rather they inform the programmer of a possibility of incorrect program operation. The user may choose to ignore the warnings, or re-write the program to eliminate the lines that generated the warnings.

At this point, the tokens generated by the sample program are ready to be downloaded. Start the debugging program as you did in step one above, but this time add the name of the sample program. Type:

```
debug62 <serial_port> <name_of_file>
```

In this case, for example, assume the TICkit is connected to serial port COM2 and type:

```
debug62 2 first62
```

Once again, the debugger should report that the TICkit is connected (If not press the TICkit reset button). This time, the name of the program "first" should display at the bottom of the dialog box and the word "SYMBL" indicates that there is symbolic information available for this file.

1.5 Step three: Getting the program inside the TICkit

At this point, the Console computer is running the debugger and talking to the TICkit, but the TICkit has not been programmed. This "programming" process is referred to as downloading. The debugger is used to download the token file generated by the compiler to the TICkit. The TICkit will write the tokens into it's EEprom for permanent storage (or until a different program is downloaded). The debugger is instructed to start downloading by pressing the letter 'D' at the debugger's command prompt. The debugger will then ask if you really want to download to the TICkit. Press 'Y' to initiate the transfer. The debugger will read the token file, transfer the tokens to the TICkit, then verify the transfer.

One thing which we have assumed is that the debugger knows which token file to use. It will, provided the debugger was started with a file name on the command line. If that is not the case, use the "file" command of the debugger to specify which file to use by pressing 'F' at the debugger's command prompt.

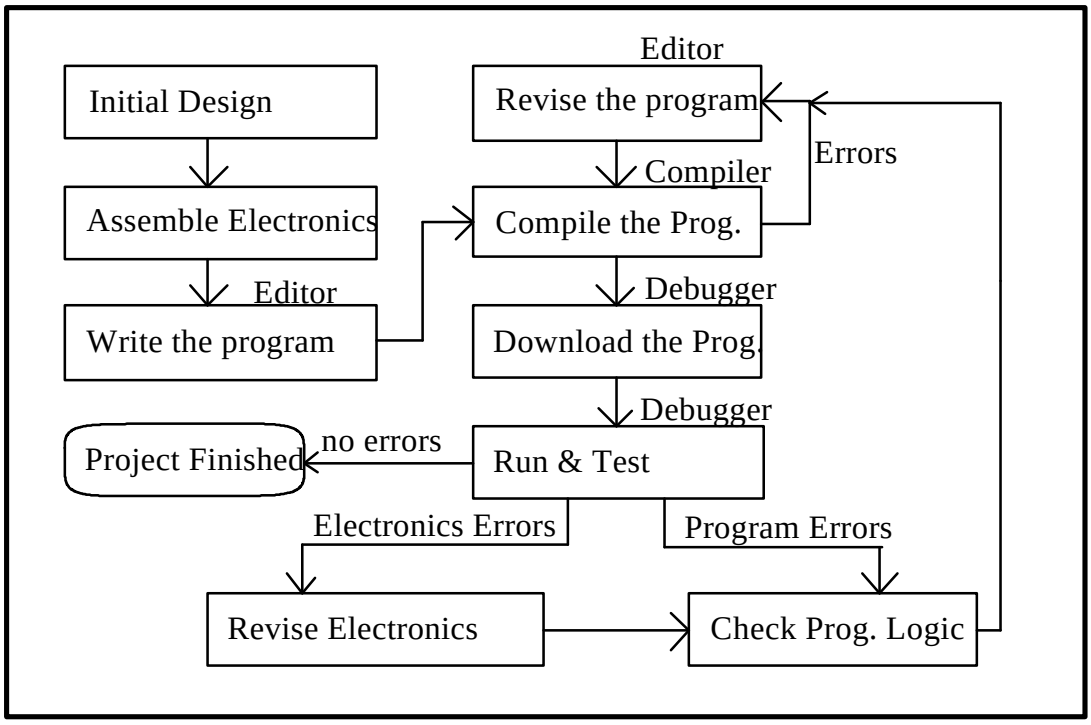
If all went well with the download, there will be a message indicating the file was downloaded and verified. The PC (program counter) register will be pointing to the first token of the downloaded program, and a command prompt will display. Now press the letter 'E' which is the debugger command for execute. The program will run and "Hello world..." will display above the debug dialog box. Congratulations! The TICkit program placed that message there. Your first program has been compiled, debugged, downloaded, and executed. You can reset the TICkit and press 'T' or 'S' to watch the TICkit execute the program a source line or token at a time.

1.6 If you are having trouble

If your TICkit does not seem to be responding, or the console computer is not executing as this manual says it should, follow the steps below to attempt to remedy the problem. If none of these things work, contact Protean Logic at: (303) 828 9156.

1. Verify that the power and download connections are not reversed. The plugs are not polarized, so try plugging the cables in every orientation. Press the reset switch (or remove and repaly power) after every change.
2. Run the programs from DOS. If you are in windows, exit to DOS. Verify that no mouse drivers or other items are using the required serial port. Disable any TSRs which might interfere with transfer timing.
3. Run all programs from the same directory where the software was installed (\tickit). Make sure that the DOS debug program is not being run instead of the TICkit's because of a path search.

1.7 The TICkit development cycle: The standard routine



The "first" example is a very simplified version of the steps required to get a pre-written program compiled and installed into a TICkit device. The routine for initially writing, compiling and debugging a program is not any more difficult. The diagram above graphically illustrates the steps involved for developing a program and what software tools are used for each step.

The very first step is setting up the development configuration. The supplied development integration tool is called the TICkit launcher. Setting up a development configuration is an optional step, but a very fruitful step if the project is large or difficult in any way. The chapter on the TICkit launcher explains the specifics required to build a

configuration. In general, the common commands required to perform each step of development are entered into a special configuration file. This file acts as a type of menu to easily select each step of the development cycle with just a few keystrokes and frees the user from having to remember specific command line parameters.

The next step is to type in a new program or to copy an existing program which will be modified for a new application and make initial modifications. The tool used to do this, a text editor, is not supplied with the TICkit package, but every version of DOS has a text editor. There are also special editors available just for program development. For the sake of discussion, this manual assumes you will be using an MS-DOS version 5.x and 6.x program called "edit" to enter and modify programs. Refer to your MS-DOS documentation for instructions for the text editor, or use whatever editor you are most familiar with. Most professional programmers prefer to continue to use whichever editor they have been using in the past. This saves learning a new tool. Some people even use word processors to make their programs and simply store the files as ASCII text files which are readable by the compiler.

After an ASCII text file is prepared using an editor or word processor, the next step is to compile the program. The supplied program called "fbasic.exe" reads the ASCII text file and generates two additional files as output. One file is the machine representation of the program called a token file, and the other file is a collection of information used by the debugger called a symbol file. The compilation step usually produces AN ERROR LIST. As annoying as a list of errors is, it really is a great time saver. The compiler can detect many types of errors as it is generating the token file. Because the entire file is scanned, most errors in a program can be detected even before the program ever runs. The fact that errors are common is the reason for the smaller loop in the development diagram. After the compiler reports errors, the programmer runs the editor again to correct the reported problems, then re-runs the compiler. This small sequence is repeated until the compiler reports no errors.

The next phase of the development cycle is the debug cycle. The tool used here is the supplied "debug.exe" program (actually DEBUG62.exe or DEBUG74.exe). This program is run and the token and symbol file for the program are loaded into the debug program. Then the tokens are downloaded into the TICkit hardware using the download command of the debugger. At this point any of many types of debugging techniques are used to verify that the program actually does do what it is intended to do. The program can be executed and run at full speed, or the programmer can interactively step through each line of the program and watch the results a line of the program at a time. Watching the program execute a line at a time is called "source level debugging" and is a very effective way for finding bugs in programs. The debugging phase of the development cycle is used to find "run-time" or "logical" errors in a program where as the compiler can only catch "syntactical" or "grammatical" errors. Usually there will be at least a few errors of logic in a program. This fact generates the larger loop in the development diagram. When an error in logic is detected while debugging the program, the programmer must go back to the editing stage of the development cycle to edit the program source code, re-compile the program, re-download the program, and test again until no errors of logic remain. At this point the program is complete. This scenario assumes that any circuitry created for the task works properly also. Often changes in the user's hardware interface will require changes in the program which requires re-editing, re-compiling, downloading, and debugging once again.

1.8 What next?

The following chapter talks about the TICkit launcher. This program is the integrated interface for programming the TICkit. Using the launcher saves lots of typing while developing a program. It allows the various command lines, like the ones we just used to compile and download the sample program, to be entered into a special configuration file. Then the compiling, downloading, editing, etc. for a program can be started with just a few keystrokes instead of typing a whole command line each time.

After the Launcher is explained, the next chapter will take a closer look at the sample program. This time the emphasis is on programming, not just using the tools to get the program into the TICkit. In this chapter, the fundamentals of the FBASIC language are discussed. After this chapter, many programmers will be ready to get to their project. The remainder of the manual can be used as a reference.

The next three chapters deal with the programming language in more detail. Chapter 4 talks about expressions, types, and other issues that more complex FBASIC programs can use to produce better programs. Some philosophy of why FBASIC is the way it is appears here. Chapter 5 talks about the Keywords used in FBASIC. Listing keywords in alphabetical order enables this chapter to be used as a reference. Chapter 6 contains a list of functions. This chapter is

organized by what the functions do. Most programmers will want to spend some time reviewing this list to see what is available and what sort of arguments the functions need.

The final chapters talk about the Debugger, the Compiler and the Console program. The Console program may be used instead of the debugger once a program is operational. The Console program uses the full screen to display information from a TICkit. In this case, the console computer becomes an input/output device for the TICkit. Review these chapters to find more advanced techniques to employ with these tools. The Debugger instructions will be especially useful.

Be sure and become aquainted with the Protean Web Site at: //www.csn.net/Protean. This site contains many applications notes, product update information and links to other useful data sources. Also, spend some time to explore the sample programs on the release disk. Information about the TICkit changes quickly and often there are new libraries and other resources which have yet to be documented which are contained on the release disk.

2 The TICkit Launcher

2.1 What is a launcher? How will it help when programing?

A launcher is simply a type of menu program. Because the use of the editor, compiler, and downloader is cyclical in nature, a convenient way to repeatedly execute each of these tools on the required files is a real time saver. The launcher does just this.

Entering the command line for each source file to be edited in a program, the compile command, and the debug command into the launcher allows the programmer to repeat any of these commands with a few key strokes.

The TICkit launcher will hold up to ten command lines. The list of command lines is displayed in the center of the screen plus three other options used to load different configurations, edit the current configuration, make a new configuration based on the current one, and exit the launch program.

The launcher is started by typing:

```
tickit [<configuration_file>]
```

The configuration_file is optional. Each list of files is referred to as a launch configuration. Commonly, there is a separate configuration for each program under development. These configurations can be named to correspond with the name of the primary source file. For example, with the program "first.bas", a configuration named "first.tic" could contain the following three command lines:

```
0 edit first.bas
1 fbasic  first
2 debug62 2 first
```

These three commands would be repeated frequently if the program "first" were complex and required a lot of debugging and compiling during its development. The edit of libraries and other source files could also be added to this list. The process of starting the launcher for this configuration would be to type:

```
tickit first
```

Each one of the commands on the list is given a number. Pressing the number while the launch menu is displayed will cause that command to be executed. The commands can also be selected by using the arrow keys to highlight the desired command and pressing the <enter> key.

Some programs, like the FBASIC compiler, produce relevant information immediately before they terminate. (The error list). For this reason, the launcher can be made to wait for a key press after each command on the list. This allows the programmer to examine the data on the screen before the launch menu is re-displayed.

2.2 How to configure the TICkit launcher for a program.

Essentially, every program requires a separate launcher configuration. A new configuration is created by starting the launcher with an existing configuration, or the default configuration if no other configurations exist in the working directory, and modifying it to fit the new program. The new configuration is given a new name and is saved during the edit process.

To modify a configuration, simply select the Configure Launcher (C) option on the launch menu. A box with the list of command lines will display in the upper left of the launch screen. Use the arrow keys to move to the lines to change and modify each line as required. Change the configuration name to be the name of the new configuration. Press the <esc> button to end the configuration edit session. If a file exists with the same name as the configuration name given, the launcher will ask if it is OK to overwrite it. You may press 'Y' to overwrite the old file, 'N' to re-edit configuration information, or 'C' to cancel the edit session and return to the main launch menu without saving any changes made to the configuration.

Often a configuration will be identical to ones already created. In this case call up the existing configuration using the (L) option then use the "New Configuration" (N) option to change only the name of the program in the configuration.

The user may also select a different configuration while remaining in the launcher by selecting the Load Configuration (L) option on the launch menu. The launcher will ask for a name of the configuration to load. If the file name given exists, the configuration will be loaded. Only the root name need be given. The ".tic" suffix will be added to the configuration name automatically. If the specified configuration file does not exist, a pick list of existing configurations will be displayed to choose from. Arrow up or down to select the configuration name you wish to load. Press enter to load the highlighted configuration. The user may wish to press the <tab> key to clear the configuration name when loading a configuration. This has the effect of calling up the pick list immediately. If the user presses the <esc> key while the load name is being entered, the old configuration is continued and the user is returned to the launch menu.

To exit the launcher, select the eXit launcher (X) command.

3 FBASIC Anatomy

3.1 Dissecting the sample program, "first.bas"

```
DEF tic62_c
LIB fbasic.lib


GLOBAL word eeprom_pntr
GLOBAL byte each_byte


FUNCTION none main
BEGIN
    rs_param _set( debug_pin )
    =( eeprom_pntr, "Hello World..." )
    =( each_byte, ee_read( eeprom_pntr ))
    WHILE <>( each_byte, 0b )
        con_out_char ( each_byte )
        ++( eeprom_pntr )
        =( each_byte, ee_read ( eeprom_pntr ))
    LOOP

    REP
    LOOP
ENDFUN
```

The sample program "first.bas", which is included in the Development Kit, is shown above. This program places the string "Hello World..." on to the console screen. This program is typical of a program written in FBASIC. LIBRARIES are usually referenced at the beginning of a program, the GLOBAL variables are declared and DEFINITIONS are listed. Finally the program ends with the FUNCTION blocks that make up the procedural part of the program.

This example only has one FUNCTION, but usually there will be many functions in a program. The order of FUNCTIONs is important in FBASIC. FUNCTION names, like all symbols, must be defined or declared before they are referenced. This means that a FUNCTION block for a function name must be placed before any code which calls that function.

The beginning execution point for all FBASIC programs is the FUNCTION main. The FUNCTION main will almost always be the last function block in a program because it will reference all the other functions in a program, if any others exist. The FUNCTION main must have no parameters and no return value. Another interesting point illustrated in the example, is that there is really nothing for the TICkit to do when "main" finishes. So, it is a good idea to simply place the TICkit in an infinite loop instead of allowing the TICkit to execute random code when main finishes.

3.2 A word about libraries

The first two lines of the example are a DEFINE directive and a reference to the library, "fbasic.lib". These two lines work together to inform the compiler about the device that the program will eventually operate within. The define line informs the fbasic.lib which version of TICkit hardware it is dealing with. The fbasic.lib file contains special instructions that inform the compiler about keywords, available variable sizes, and what built-in hardware functions are available in the TICkit. Virtually every FBASIC program will reference this library. This library, and its component library "token.lib" are good sources of information about the standard library in the TICkit. By editing the file "token.lib", the calling definitions for internal routines can be examined along with any notes or special definitions for the routines. This information is as accurate as possible because this is what the compiler actually uses to make the program. A little later in this chapter, our example will be modified to use another library that comes with the development kit that makes the program even simpler.

The next few lines are GLOBAL lines. These statements define symbolic names and sizes to variable storage areas. In our example, a 16 bit word is associated with the symbolic name "eeprom_pntr" and an 8 bit word is associated with

the symbolic name "each_byte". The programmer never needs to know the physical location of these variables since the compiler will always know where they are on the basis of their symbolic names.

The next lines create a procedure block for the symbol "main". As mentioned before, the FUNCTION main is the starting execution point for any FBASIC program. Every FUNCTION in FBASIC must be given a name and a type for any value that it returns. "Main" will never return a value (it has no place to go), so it is defined as type "none". The FUNCTION "main" never has any parameters either, but if it did have parameters or local values, they would be defined for the duration of the FUNCTION block and would appear between the FUNCTION and BEGIN statements.

BEGIN is the statement which marks the beginning of code generation. All the statements between a BEGIN and an ENDFUN are code generating statements. In our example, two assignments, two loops, some math, some EEprom functions, and a console output function are referenced.

FBASIC has expression evaluation, but it has no "operators". This means that all arithmetic is performed using function calls. Even assignment, (=) is accomplished using functions. This "limitation" makes the language very simple, but possibly a bit unfamiliar. To reference a function simply use the function's name followed by a left parenthesis "(". This tells the compiler that the program is to execute the code contained in the procedure block or operation which has that name. If any parameters are to be used, they would be placed after the left parenthesis, but before the matching right parenthesis ")". Parameters in function calls can be variable references, parameter references, constants, or other functions with return values.

In our example, the first assignment line will assign a value to the variable "eeprom_pntr". The value it assigns is a 16 bit pointer to the string "Hello World...". The string "Hello World" appears to the compiler as a constant. This may seem mystical but it really is quite simple. When the compiler sees a quote (") it understands that a string constant is being defined. All characters that appear in the string will be placed at the end of the program code and a pointer to beginning of that EEprom location will be used as the value of the constant. Our example places the EEprom address of the place where "Hello World..." is stored into the variable "eeprom_pntr".

The next line reads a byte from the EEprom at the location given by the variable "eeprom_pntr" and places that byte into the variable "each_byte". The function "ee_read", which is contained in the standard library, is what actually does this operation. The byte that is returned from that function is placed in "each_byte". Assignment operators in the standard library copy the contents of the second variable (ee_read) into the memory area of the first variable (each_byte).

The next line is a WHILE statement. This statement marks the beginning of a structured loop in FBASIC. An expression follows that tests for a looping condition. The body of the loop will be executed only while the expression evaluates to a non-zero (true) value. The first LOOP statement ends this WHILE block. The expression for this WHILE statement tests the variable "each_byte" against the byte constant 0. If they are not equal, the "<>" function returns a value of 255 (all 8 bits are one). If "each_byte" is equal to 0, the "<>" function returns a 0 indicating that the comparison failed. All relational functions in the standard library return either a 0 or 255.

The body of the loop contains three function calls. The first call is to a function which outputs one byte to the Console. This function will cause the contents of the variable "each_byte" to appear as an ASCII character on the Console display. The second function call is a 16 bit increment function. This function returns no value, but increments the argument by one. The third function in the loop is like the function which preceded the loop. It simply reads a byte from the EEprom at the specified address and places it in the variable "each_byte". These three statements will be executed until a 0 is read from the EEprom. The zero will be there because FBASIC always terminates string constants with a single 0 byte.

The last two statements form an infinite loop. The REP statement starts a structured looping block and the LOOP statement ends the block. Since both the top and bottom of the loop are unconditional, the TICkit will simply loop in this location until it is reset.

All loops in FBASIC have one of two starting statements and one of two ending statements. Loops can be started with either a REPEAT or a WHILE statement. The WHILE statement establish a condition for entering and continuing the loop. REPEAT causes repetition with no condition. Loops can be ended with either a LOOP or an UNTIL statement. The UNTIL statement establishes an exit condition for exiting the loop. The LOOP statement will never cause an exit,

but simply causes the body of the loop to repeat. The body of the loop must use some other means, like a WHILE or STOP, to exit. Any combination of beginning and ending statements forms a valid structured loop in FBASIC.

Two other statements are associated with loops in FBASIC. The STOP statement will cause the loop to be exited, while the SKIP statement will cause execution to jump to the LOOP statement.

3.3 A more elegant "first.bas"

```
DEF tic62_c                    ; version 62A of TICkit
LIB fbasic.lib

FUNC none main
BEGIN
    rs_param_set( debug_pin )
    con_string( "Hello World..." )

    REP
    LOOP
ENDFUN
```

This version of "first.bas" uses a library which has a pre-written routine for doing string output to the console. The function "con_string" is contained in the library "constrin.lib". This general purpose routine uses a pointer into EEprom as the pointer to the beginning of a ASCII string. The contents of the string will be output to the Console until an 0 character is encountered in the EEprom. The "con_string" library file contains:

```
; Generic function to output a string of characters from
; EEprom to the Console

LIB fbasic.lib           ; This will be ignored if the root
                         ; program referenced fbasic.lib

FUNCTION none con_string
    PARAM  word pointer

    LOCAL  byte each_byte
    LOCAL word temp_pntr
BEGIN
    =( temp_pntr, pointer )
    =( each_byte, ee_read( pointer ))
    WHILE <>( each_byte, 0b )
        con_out_char( each_byte )
        ++( temp_pntr )
        =( each_byte, ee_read( temp_pntr ))
    LOOP
ENDFUN
```

This Library function is quite similar to the original "first.bas" except that it uses local values and a parameter to make it a more general purpose function. The PARAMETER statement informs the compiler that a symbol of the given type or size is going to be coming from the calling reference. The statements in the function can have access to this data by referencing the parameter name. The LOCAL statements are just like GLOBAL variable definitions except that they exist only to the statements contained in the function. This saves on memory space and also prevents accidental symbol name conflicts in programs that use this library. A temporary copy of the pointer passed to the "con_string" function is made so that the calling value is not modified.

The lines at the beginning of the file that begin with ";" are comments. Any part of a line that follows a ";" is treated as a comment and is ignored by the compiler. Therefore a ";" as the first character of a line is equivalent to the REMARK statement.

Examination of other libraries contained in the FBASIC Development Kit will illustrate other programming concepts for the FBASIC language.

3.4 FBASIC line syntax (labels, remarks, conditionals)

FBASIC is a line oriented language. This means that there is really only one statement per line. There are quite a few additional things a programmer can do with a line though, besides just putting a statement on it. For example, a line may be blank, or it may have a comment, or it may have a label, or a conditional compilation directive, or it may even be extended onto the next line. The sample program above used blank lines to keep things a bit easier to read, and the library routine above used the ';' on a few lines to place text messages to the programmer for future reference. The code sample below shows a few more things that can be done:

```
; Code fragment to illustrate line syntax

:again1 con_string( "hello ~
                     ~again...\x0d" ) ; repeat this

IFDEF exit_capable IF ==( con_in_char( 0 ), 23b )
IFDEF exit_capable     GOTO done1
IFDEF exit_capable ENDIF

GOTO again1

:done1
```

This code fragment does not exemplify good programming practice, but it does illustrate some of the trickier things that can be done with lines in FBASIC. The first line is simply a comment line to explain what the code does. The next line uses the ":" to associate the label "again1" with this line in the program. All labels in FBASIC are local, so only other lines in the same function can reference "again1". This same line uses con_string to output a string of characters to the console. The literal string is a bit peculiar looking, however. The "~" character is used to extend a line onto a following line. Therefore, this string is actually, "hello again...\x0d". Using line extension can make a program easier to read when lines get long. Another element of this line that is a bit odd is the "\x0d" in the string. The '\' character is an escape character. The escape character is used whenever something unusual is to be done with the character, or characters, that follow. In this case the '\x' informs the compiler to insert a byte with the value of the following two hexadecimal digits. In this example, a value of 0d is used which is an ASCII return character. The following table summarizes the escapecharacters and their meanings:

Escape seq.	Sequence Meaning
\R	ASCII return character
\L	ASCII line feed character
\\	\ character (no escape)
\"	" character (doesn't terminate literal)
\'	' character (doesn't terminate literal)
\~	~ character (doesn't extend line)
\xnn	character of hexadecimal value nn (2digits)
\dnnn	character of decimal value nnn (3digits)

A few lines further into this code fragment are three lines with IFDEF directives. IFDEF is a compiler directive. The lines that follow the IFDEF <symbol_name> will only be compiled if the <symbol_name> has been defined. In our example, the symbol "exit_capable" is tested to see if it has been defined previously in the program. If it has, the three lines comprising the IF statement will be included in the compile. Otherwise, the three lines are ignored. The IFDEF directive is used by the fbasic.lib file to include only the appropriate version of the token.lib. This is how DEF tickit_2 at the beginning of the program causes the proper code to be generated for the 2.x version of the TICkit interpreter.

3.5 Constants, constants, and more constants

The rules regarding constants are often hidden or overlooked aspects of programming languages. FBASIC allows for different sizes of constants, different radix of constants, and some special types of word constants which are actually pointers into EEprom storage. Why all the different types? By expressing constants in the proper size and in the proper way, the program executes faster and more efficiently. At the same time, the programmer can easily understand what the constants mean. For example, the decimal number 128 may not seem structurally significant, but the binary

representation of that number, 10000000, clearly indicates that the 7th bit is set high. Constants are not too difficult to learn provided that their basic structure is understood.

For numeric constants, the structure always starts with a numeral. Often a leading zero is used to ensure that any non-numeric elements of the constant (like radix or hexadecimal characters) do not fool the compiler. An optional radix indicator may follow the leading zero in the second character, then one or more digits of the constant, and ends with an optional size indicator. For example, 0x0fa8L is a hexadecimal constant as indicated by the 'x', and it is a LONG size as indicated by the trailing 'L'.

Radix indicators are: Y=binary, D=decimal, X=hexadecimal.

Size indicators are: B=byte, W=word, L=long.

```
; Examples of constants

=( var1, 0y00010011b )   ; the y makes it binary (base 2)
                         ; the b makes it a byte
=( var2, 0xff04w )       ; the x makes it hexadecimal (base 16)
                         ; the w makes it a word
=( var3, 0d12345678l )   ; the d makes it decimal (base 10)
                         ; the l makes it a long
```

In addition to the numeric constants, there are also ASCII constants. The ASCII constants allow for strings of values. Therefore, they are indicated with quotes. The "" is used to indicate a word constant which points into EEprom memory where the string of ASCII constants will be stored. The ' ' is used to indicate a byte constant or multiple byte constants in an INITIAL statement. Usually only the first byte of a ' ' string is used as the byte value of the constant, but the INITIAL statement is able to use all of the byte constants and place them in allocations that can use more than one byte value. An example of these string constants is: "hello world", used in the first.bas program. The byte constant 'hello world', would actually evaluate to 104, which is the ASCII code for a lower case H character. Unless in an INITIAL statement the ' ' will usually only have one character in them. For example:

```
con_out_char( 'H' )      ; an alphanumic value byte constant
```

3.6 Using DEFINES and Constant Operators

Larger programs often have many references to the same constants. To prevent typing errors and to provide for easy modification of the constants involved, symbols are used in place of numbers throughout the program. This is accomplished using the DEFINE directive. The example below shows how the constant, "temp_offset", is used in place of the number 103b. First the symbol is defined, then later in the program, the symbol is used instead of the number. Imagine a program that has 45 lines of code that refer to temp_offset.

```
DEF temp_offset 103b
.
.
.

IF >( in_val, temp_offset )
    con_out( +( temp_offset, in_val ))
ELSE
    con_string( "Reading out of range" )
ENDIF
```

Now imagine that while debugging you decide the temperature offset of your device needs to be changed from 103 to 121. A program that used the DEFINE, requires only on change. A program that used the number in every reference would need all 45 lines changed, assuming you could find every occurance.

Another useful tool to use with symbolic constants is the '|' compile time operator. The "vertical bar" operator performs a bit-wire OR of the constants adjacent to it. The example below is very common in TICkit programs that us RS232. It uses DEFINED constants with the '|' operator to build up the format, baud rate, and pin number used in the rs_param_set() function. This notation is much clearer than a binary number.

```
rs_param_set( rs_invert | rs_4800 | pin_d5 )
```

instead of:

```
rs_param_set( 0y11000101b )
```

3.7 String constants and implicit allocation

Very commonly, a program needs to output a sequence of alphanumeric bytes for display. These sequences are called strings. FBASIC supports this common requirement by utilizing the double quotes " " to generate a special string constant. The string constant performs two distinct operations. First, it causes the compiler to place the contents of the quoted string into the EEprom. Second, the " " string produces a word constant that is the EEprom address of the first character of the string. This has the net effect of both allocating and initializing memory as well as producing a way to keep track of the constant.

The string information is placed in the EEPROM immediately following the program tokens and a \0 (byte of value zero) is appended to the end of each string. The appended \0 at the end of the " " string can be used to determine the end of the string and is a common convention refered to as "null termination".

```
con_string( "Have a nice day\r\l" ) ; a word constant which
                                    ; is a pointer into EEprom

                  ; It points to the beginning of the string,
                  ; where the compiler placed it in EEprom.
```

3.8 Allocation Constants and Field Names

The last type of constants have to do with EEprom allocations. These constants provide a means of working with EEprom storage on a record or array basis. Some of these issues may not be clear immediately, unless you are familiar with upper level languages which have structure capability like C or Pascal. But these concepts are not difficult, just keep in mind that these values are not the storage, but simply word pointers to the storage and can be manipulated like any other word size number. Look in the Keyword section of the manual under ALLOCATE, RECORD, FIELD, and INITIAL for more information.

First, let's look at the structure of a FIELD name. Fields are the part of a record that actually hold information. Records can be thought of as a way to collectively refer to more than one data value. FIELDs usually hold simple bytes, words, or longs, but they can also refer to previously defined RECORDs. This creates a tree system. Not only can a FIELD refer to single data items, it can also refer to more than one. So, by using a "count", an array of data items can be referred to in a FIELD. For example:

```
RECORD product
    FIELD byte prod_name[25]
    FIELD word prod_code
ENDREC

RECORD demo
    FIELD long demo_no
    FIELD word demo_time
    FIELD byte name[30]
    FIELD product demo_prod
ENDREC

ALLOCATE demo demos[50]

INITIAL prod_code@demo_prod@demos[0] 1001
INITIAL prod_name@demo_prod@demos[0] 'TICkit Assemblies'
```

Don't be concerned if this all seems a little foreign right now. Remember, we are concerned with understanding constants at this point. All the record and allocation stuff can come a bit later. From our example though, there are six

FIELD lines. The first FIELD line and the fifth FIELD line all use a "count" to indicate that the field contains 25 and 30 bytes respectively.

The sixth FIELD line shows how a FIELD in one record can refer to a previously defined record.: FIELD product demo_prod

The ALLOCATE line is what actually reserves the space in the EEprom. In this case it will reserve enough room to hold all the fields for the record demo. The allocation is named "demos". Whenever we refer to "demos" in the program, we are actually referring to the EEprom address of the first 8 bit location of this allocation. Therefore, simply using the word "demos" in an expression is using a constant. The more information that is attached to demos, for example the demos[0], the further the constant is pointing into the allocation. Records are also constants. Records named in expressions refer to offsets within an allocation. Also, fields are simply offsets from the beginning to the record in which they appear. The '@ is used to add up all these offsets at compile to refer to individual fields within an allocation. For Allocations or Records where more than a single count of an item exists, a numeric constant can be used with a @ to get to the correct individual storage element. All this works out nicely as a way to refer to EEprom storage symbolically.In the above structure example, the following line outputs the product name to the console:

```
con_string( prod_name@demo_prod@demos[0] )
```

There is one more issue related to the FIELD, RECORD, ALLOCATION scheme, however. Occasionally you may want to know what the size of a storage element is. By using just the record name in an expression, the size of the storage element is used in the expression. This constant is very useful to calculate the location of a particular count storage element using variables at run time.

The '!' is often used in this sort of calculation. The '!operator lets the compiler know that you intended to use a partial field name. Without the '!' operator, the compiler would report an error if a partial field name is used in an expression. This basically boils down to an array offset. Therefore, in the following example, a 16 bit corrected value is returned from an 8 bit input that represents an A/D reading.

```
RECORD each_entry
    FIELD word adj_value
ENDREC

ALLOC each_entry A_D_correct 256

FUNCTION word A_D_adjust
    PARAMETER byte ad_inval
BEGIN
    =( exit_value , ee_read_word(  ~
        ~ +( !a_d_correct, *( ad_inval, each_entry ))))
ENDFUN
```

The assignment statement uses a standard array calculation of an offset plus a size times the array index to come up with the EEprom address of the correct word for the given 8 bit a_d_value.

Ok, this last section got pretty deep. Just remember that there are constants for both the initial offset of an EEprom allocation as well as the size of an Allocation element. When you start using the EEprom as a storage medium, these types of constants will come in quite handy. They will also eliminate the need to remember a bunch of numbers. Once you get a good handle on the ALLOCATE statements, take a look at the SEQUENCE statement. It is just like ALLOCATE but does not use EEprom space.

3.9 Variables, Global vs Local and precious RAM space

The discussion above dealt with constants, but the real issue in programing is utilizing variable space efficiently. Computers of all sizes have limited resources. Small computers and controllers, like the ones that implement FBASIC, have particularly harsh limitations in terms of RAM memory. The TICkit 57 has only 48 bytes of RAM total while the TICkit 62 has 96 bytes of RAM. The current FBASIC TICkit token scheme limits the maximum available RAM in any processor to 128 bytes. RAMs used to store variable's information which changes quickly, and stack-based data such as program flow information.

Because this type of memory is so scarce, FBASIC has provided many features to optimize its use and organization. The issues of data size have already been discussed in reference to constants, but using only as much space as is required for any given variable is probably more important in the discussion of RAM than constants. Besides choosing the smallest size of variable, another option exists for limiting the scope of a variable.

Variable scoperefers to how long, or for what section of a program, space is allocated to a variable. GLOBAL variables have global scope. This means that space is allocated to the variable name for the entire time the program is executing. LOCAL variables have local scope sometimes called function scope. LOCAL variables only have space allocated during the short period that the program is executing in the function the variable was defined within.

LOCAL variables offer several advantages. First, they allow different functions to share the same RAM space for variables. Second, they limit where a variable name can be referenced. This provides the compiler with an ability to check the programmer's work. If a variable is defined only within a function, any reference to that variable outside of the function can be assumed to be an error.

GLOBAL variables can be used by any function and actually operate a little bit faster than LOCAL variables. The main drawback to GLOBALs, however, is that they occupy scarce RAM space even if the information is not being used, or is no longer needed.

Now, there is an obvious question that arises out of this discussion. What happens when the memory space is exceeded? If there are too many GLOBALs, the compiler will report an error. However, the more common situation is that the memory is exceeded dynamically while the program is running. This occurs because the compiler can not forsee how the local variables will be used and when they will allocate memory. As the program is running and executing functions and nested functions, the local memory stack may grow to the point that it starts overwriting the GLOBAL area. This will usually result in strange program results.

If a program is using a lot of LOCAL variables and there is a possibility of a stack overflow, the programmer should execute the program with the debugger connected in monitor mode. The debugger continuously monitors both the stack pointer and memory pointer and alerts the user if an overflow occurs. THIS IS A TRICKY SOURCE OF UNEXPLAINABLE BUGS, and is a good thing to check if a program mysteriously stops functioning properly.

The TICkit62 implements a stack overflow vector call. This was unavailable in the TICkit57. Basically, a vector call is simply a function that gets called by something other than the lines of your program. In the case of the stack overflow vector, the function called "stack_overflow" will be executed whenever the interpreter runs out of memory. You can not return from this function, so this function is typically used either to inform the programmer of something which needs attention, or is designed in a final product to perform a controlled shutdown. For example, the maker of an elevator controller assumes the stack will never overflow, but if it does due to some unforseen circumstance, he may program the elevator to apply brakes, turn off motors, and allert the security system.

3.10 Variable Arrays and Indirection

Most of the time, variables are simply named locations in the computers memory used for storing discrete information. Sometime, though, arrays are used to allow run-time distinction between variables. When a variable or data item is refered to by name it is called a direct reference. There are times when a generic piece of a program is to operate on data items which are to be determined by the execution of the program not just the position in a program. In this case, we need a way to change the reference to data under program control. This is most commonly accomplished using one direct variable to "point" to another data element. This reference is refered to as indirect. FBASIC allows explicit pointers with ALLOCATIONs but not with variables. Variable indirection can only be accomplished implicitly with Arrays. Array variables look just like any other variables except that they use the "[ ]" characters to indicate an array index. This index can be a constant or another byte size, variable expression. The "[ ]" are also used in the array definition like a GLOBAL or LOCAL statement to indicate how many elements will be in the array of that name. Arrays can be viewed as a finite number of similar sized storage elements lined up in a row in memory. The entire row is referred to by the name of the array, and the individual elements are refered to by a combination of the name of the array and a number index that indicates which element, from the beginning of the array, to use. An index starts at 0 for the first element and continues up to the size of the array less one.

There are actually two types of arrays in FBASIC. There are variable arrays and allocation arrays. Both allow indirect reference to memory, but the variable arrays are used to access the internal RAM of the processor and are very fast. The allocation arrays are used to conveniently calculate offsets in EEprom or some other off-processor memory resource. These array elements have to be de-referenced (read or writen) explicitly with read and write functions and are typically a lot slower to access than variable arrays.

Arrays are used most commonly to refer to elements that are handled the same for one purpose, but differently for another. For example, we might have a routine that manipulates dates and times that are read from a clock device. In this case, we will want to read all the clock information in at once so there is no minute, second, or hour roll-over between consecutive reads from the device. A single routine reads 16 bytes of information in from the clock IC into an array of values and treats all of the bytes the same. The display routine is only concerned with certain array elements and treats each element differently. The example below demonstrates this:

```
; program fragment to illustrate the use of arrays

GLOBAL byte read_vals[16]       ; define 16 element array of
                                ; byte values

FUNC none read_ic               ; reads all 16 bytes from the
                                ; device
    LOCAL byte val_numb 0b
BEGIN
    read_ic_init()       ; gets the IC ready to xmit all regs
    REP
        =( read_vals[ val_numb ], read_ic_byte())
                    ; the above line assumes that a function
                    ; called read_ic_byte will return the
                    ; next consecutive register of the clock
                    ; IC internal memory
        ++( val_numb )
    UNTIL == ( val_numb, 16b )
ENDFUN


FUNC none display_time
BEGIN
    lcd_string( "The time is: " )
    lcd_write_num( read_vals[ 5 ] )   ; 5th element is hours
    lcd_send( ':' )
    lcd_write_num( read_vals[ 6 ] ) ; 6th element is mins
ENDFUN
```

3.11 Functions, parameters, and exit value

The discussion of variables above suggests that functions have some special significance besides just being subroutines. This is exactly the case in FBASIC. Functions are used extensively in expression evaluation and device driver creation. Functions are just small sections of instructions which act like mini-programs. They can have their own memory variables, their own compile defines, and some special names for input and output.

Functions have some very special local values called parameters and exit_value. These local values are used to get information into the function from the rest of the program and to return values back to the rest of the program.

The exit_value is used as the default method of returning a single value to the rest of the program. It is very common for a section of a program to need to return back one result. This is so common that FBASIC has dedicated a symbol named "exit_value" as a pre-defined local symbol in every function which is declared to return a value. For each function, exit_value will be of the type and size that the function was declared to be and can be assigned and manipulated just like any other local variable. When an EXIT or ENDFUN is encountered, the data contained in the exit_value is sent back to the calling program as the value of the function.

Parameters are the opposite of exit_value, but can be used to return information also. Parameters appear as local variables, but are really just pointers to variables in the calling program. This gives the function the ability to indirectly refer to data the calling program has for varying situations. The function can read and manipulate pointers. Keep in mind that any change to a parameter in a function will be reflected in the corresponding variable of the calling function or program. It is usually good programming practice to avoid modifying parameters.

The following simple example illustrates how an addition function can be made:

```
FUNC word plus
    PARAMETER word val_1
    PARAMETER word val_2
BEGIN
    =( exit_value, +( val_1, val_2 ))
ENDFUN
```

An example of the use of the plus function as we defined it above would be:

```
=( sum_val plus( val_1, val_2 ))
```

This returns with the word length sum of val_1 and val_2 and assigns that value

to the variable sum_val of word length that must have already been defined as a global or local before using it in the call to the plus function.

This is sort of a trivial example, as a '+' is used to implement the 'plus' function. A more likely case would be a keyboard input routine, which might return the ASCII value from a routine that scans keyboard hardware.

Just to make this discussion relevant, the following code sample comes from the file "ltc1298.lib" and shows how a library can be used to make a generic driver for an IC.

3.12 A device driver library for the LTC1298 (12bit A/D)

```
; Functions to control A/D
; These functions rely on three defines to work properly
;   cs   = Chip Select pin  'Must have a separate line  '
;   clk  = Clock control pin  'Can share a data line '
;   data = data pin   'Can share a data line i.e. an LCD'

; Routine to read a data from an LTC1298 or LTC1288 A/D chip
```

```
FUNC word read_ltc1298
    PARAM byte config  ; This value indicates mode and channel
                       ; for the A/D chip.
                       ; bit 7 = mode ( 0=single end,
                                         1=differential)
                       ; bit 1-6 = channel select
                       ; bit 0 = polarity for differential or
                       ;         lsb channel select
    LOCAL byte count 0b
BEGIN
    pin_low ( ltc_clk )
    pin_low( ltc_cs )
    pin_high ( ltc_data )           ; start bit
    pulse_out_high ( ltc_clk, 10w )

    IF b_and(  config, 0y10000000b ) ; differential conversion?
        pin_low( ltc_data )
    ELSE
        pin_high( ltc_data )
    ENDIF

    pulse_out_high( ltc_clk, 10w )
    IF b_and( config, 1b )        ; select channel or polarity
        pin_high( ltc_data )
    ELSE
        pin_low( ltc_data )
    ENDIF

    pulse_out_high( ltc_clk, 10w )
    pin_high( ltc_data )           ; use msb first format
    pin_high( ltc_clk )            ; clock in the msbf bit
    =( count, pin_in ( ltc_data ))  ; make data line an input
    pin_low( ltc_clk )             ; return clock to low state

    =( count, 0b )
    =( exit_value, 0w )            ; get data loop ready
    REP
        pulse_out_high( ltc_clk, 10w )  ; clock for next bit
        =( exit_value , <<( exit_value )); shift exit to left
        IF  pin_in( ltc_data )
            ++( exit_value )
        ENDIF

        ++ ( count )
    UNTIL == ( count, 12b )

    pin_high( ltc_cs )
ENDFUN
```

The example above is a bit lengthy, but is a working example of a device driver using a function with parameters. The parameter is a single byte and tells the device how to configure its 2 input channels. Depending on the level of the 7th bit and the 1st bit this device can do either differential or single ended conversions and it can be programmed to return the level of each channel individually or the difference of the two channels in either polarity. The protocol for sending this information and retrieving the conversion result is not highly complex, but could easily waste a day of time to figure out and debug. If you wanted to use an LTC1298 in your design, you would not need to worry about the communications protocol. As in the program sample below, you would simply include this library routine in your program and call the function. The program below reads the two channels of the LTC1298 and captures the data on a PC using the ACQUIRE.EXE program. The example is complex, but should give you some ideas of what can be done with the TICkit. This program would work with up to 26 TICkits in a small data aquisition network.

```
; This program uses an LTC1298 or LTC1288 (3v version)
; to take 12bit analog voltage readings once a second
; and sends these readings to a PC console
; running the ACQUIRE  program.

; This program is designed so that multiple TICkits can be
; connected to this wire in a multi-drop   configuration.

; Thanks to Scott Edwards for his Jan 1, 1996 "Nuts and Volts"
; article highlighting the use of the LTC1298 with the TICkit.

; Written by: Glenn Clark

DEF tickit_d LIB fbasic.lib

DEF ltc_cs pin_D0       ; pin D0 connects to ltc chip select
DEF ltc_clk pin_D1      ; pin D1 connects to ltc clk line
DEF ltc_data pin_D2     ; pin D2 connects to ltc data line

LIB ltc1298.lib         ; contains routine to drive LTC1298

DEF designation 'a'     ; this is the polling code for the PC
                        ; for multiple TICkits connected to
                        ; the serial wire

DEF net_pin pin_A7     ; this is the network aquisition pin

FUNC none line_sync
    LOCAL  byte match_count 0b
    LOCAL byte rs_errors
BEGIN
    REP
        IF == ( rs_receive ( 0, rs_errors ), designation )
            IF ==( rs_errors, 0b )
                ++ ( match_count )
            ELSE
                =( match_count, 0b )
            ENDIF
        ELSE
            =( match_count, 0b )
        ENDIF
    UNTIL >=( match_count, 2b )

    delay (1)
    rs_send ( designation, 0b )
ENDFUN

FUNC none  rs_out        ; convert word to serial string
    PARAM  word in_val   ; parameter is destroyed
BEGIN
    rs_send( +( 48b, trunc_byte( /( in_val, 1000w ))), 0b )
    =( in_val, %( in_val, 1000w ))
    rs_send( +( 48b, trunc_byte( /( in_val, 100w ))), 0b )
    =( in_val, %( in_val, 100w ))
    rs_send( +( 48b, trunc_byte( /( in_val, 10w ))), 0b )
    =( in_val, %( in_val, 10w ))
    rs_send( +( 48b, trunc_byte( in_val )), 0b )
ENDFUN
```

```
FUNC none main
    LOCAL byte  tic_count
BEGIN
    pin_high ( ltc_cs )
    pin_low ( ltc_clk )
    rs_param_set ( rs_invert | rs_9600 | net_pin )
    rs_stop_chek ()
    rtcc_int_256 ()
    REP
        =( tic_count, 150b )    ; used 150 instead of 156
                                ; to fudge latency time and
                                ; probable xmit delays
        WHILE  tic_count
            rtcc_wait ()
            rtcc_set ( 6b ) ; divide by 250 ( 256 - 250 = 6 )
                            ; enough time for approx 128 tokens
                            ; results in 78.25 readings per sec
            -- ( tic_count )
        LOOP                ; this loop should exit every 2 secs

        line_sync()
        rs_send( ':', 0b )
        rs_out( read_ltc1298( 0b ))
        rs_send( ' ', 0b )
        rs_out( read_ltc1298( 1b ))
        rs_send( 13b, 0b )
    LOOP
ENDFUN
```

This program uses the internal RTCCcounter of the TICkit to take readings approximately every second. There are many libraries supplied with this development kit which are not documented in this book. Use your text editor to look at all the *.lib files to see what is available. Also, check in periodically with the Protean BBS or Protean home page to see if new function libraries are available. Most of the libraries have some documentation in their source and can be used "as-is" to accomplish many interesting things.

3.13 Captain, I think the functions are overload'n!

One last interesting feature of FBASIC is that it can overloadfunction names. This means that different functions can have the same symbol name. This is very useful for generic functions that perform similarly but the data they operate on differs. For example, when adding numbers, different variable precisions can operate more efficiently than others. The "+" sign is still the ideal symbol for all addition functions, though. FBASIC will count the number of arguments in a function reference and consider their types to determine which of the many possible "+" functions to use in each case. Therefore, adding two bytes can use a different routine than two 32 bit longswhile still using the "+" symbol for the function.

In the example of the function "plus" in section 3.10 of this manual, to make it work with byte values and 32 bit long values it would be necessary for the programmer to create functions exactly like "plus" using byte and long types for the PARAMETER definitions. These functions would normally be collected together in a library of similar functions.

Programmers may wish to take advantage of this feature as they write special I/O libraries. Careful use of this feature can make nice general purpose libraries.

3.14 What's Next?

This discussion only begins to cover the FBASIC language. The programmer needs to review the KEYWORD summary and the standard library summary for more information on the FBASIC language. The next chapter gets provides many examples. If this chapter gets boring, simply skip it and start writing some programs. When you need a function or flow control capability, look to the KEYWORD summary or standard library summary to find what you need. Spend some time looking at the sample code and the supplied libraries.

3.15 Check out the the Protean Web Site

The Protean web site (http://www.protean-logic.com) is good source for information and sample programs. Many programs and libraries are posted on the site for users to draw on for their own applications. The message area can be used to ask other users questions, or to share ideas, etc. Leave comments and questions on the web site to the page master. Protean checks these messages periodically and will respond to messages as soon as possible. Enjoy the FBASIC TICkit!

4 The Console Program

4.1 Turning your computer into a dumb terminal.

Often, the TICkit is programmed to run with no need to display or get keyboard information. When this is not the case, however, your console computer can act as a display and keyboard for the TICkit. This convenient little trick is performed by running the "consoleœexe" program on the console computer. From your DOS prompt, type:

```
console <serial_port_number >
```

Now any console functions contained in the program in the TICkit will talk to the Console computer.

When you want your computer back, hold down the Control key and press the letter C (<ctrl-C>). Occasionally, the Console program will be waiting for some handshaking from the TICkit. If the TICkit was physically disconnected or reset at precisely the wrong point, the Console may not respond to <ctrl-C>. Simply re-boot or reset your console computer if this happens.

4.2 The Console Protocols (home brew TICkit I/O)

You can write your own types of Console programs, also. The handshake protocol for the TICkit is quite simple. The timing requirements are a bit fast, so make any loops tight to ensure that the Console commands are all recognized. The protocols for the nine console functions is as follows:

9600 baud - half duplex - 8 bit, 1stop bit, no-parity.
Assumes that the xmit and receive pins are physically connected.
TICkit will wait approx. .5 seconds for response after initial byte.
Most significant bytes are sent first.

test_console:	TIC:7F, con:8A.
8bit_char_disp:	TIC:59, con:90, TIC:val.
8bit_num_disp:	TIC:51, con:90, TIC:val.
16bit_num_disp:	TIC:52, con:90, TIC:val, con:90, TIC:val.
32bit_num_disp:	TIC:54, con:90, TIC:val, con:90, TIC:val, con:90, TIC:val, con:90 , TIC:val.
8bit_char_in:	TIC:69, con:val.
8bit_num_in:	TIC:61, con:val.
16bit_num_in:	TIC:62, con:val, TIC:90, con:val.
32bit_num_in:	TIC:64, con:val, TIC:90, con:val, TIC:90, con:val, TIC:90, con:val.

5 The Debug Program

5.1 What exactly does the debugger do?

The debugger is a program which runs on the Console. It communicates, via the serial port and cable, with the TICkit. The debugging program can download a program to the TICkit, or verify a program contained in the TICkit. The debugger can display information sent by the TICkit as Console output; or it can get information from the keyboard of the console computer and send it to the TICkit program as console input, just like the Console program in the last chapter. The main purpose for the debugger, however, is to aid in the debugging of a program. When debugging, a line-by-line display of what is happening as the program executes appears on the Console screen. Before each line is executed, the source text for that line is displayed followed by a request for a debug command. The user can execute that line, part of that line, or step into a sub-function of the line. The user can also examine the contents of a memory variable or change the value of a memory variable. This cycle is repeated for each line as it is encountered during the execution of a program. By carefully watching what happens as the program executes, sources of error show themselves quite readily.

5.2 The Debugger's Screen Format

```
C:\TICKIT>debug62 2 first
TICkit DEBUG62 program - Protean Logic  - (c) 1995
Console Active, attached  via COM2 to TICkit...

====================|Debug Dialog|============================|Watch Points|===
|                                                             |                |
|                                                             |                |
|                                                             |                |
|                                                             |                |
|                                                             |                |
|                                                             |                |
|                                                             |                |
|*** Reset TICkit for debugging now...                        |                |
|***************** Command:                                   |                |
|Connected to TICkit...                                       |                |
|TOKEN:E0  PC:002E Command:                                   |                |
=================|TKN:first   |=|SYMBL|=======================|MP:  |=|SP:  |==
```

When the debugger starts, the screen is split into two parts. The top part is the Console display area. This area, although smaller, acts just as the screen of the Console program in the last chapter. The bottom half of the screen is a split box. The left side of this box is called the "Debug Dialog" area. The right side of the box is called the "Watch Points" area.

The bottom of the dialog area displays the name of the file which is being debugged, if given, as well as the words TOKEN or SYMBL depending on whether or not a symbol file could be located with the same root name as the token file. Symbolic program line information and variable names are only available when the word SYMBL appears at the bottom of the dialog area. The bottom of the watch points area displays the current value of the "memory pointer (MP)" and the value of the "stack pointer (SP)". These values indicate how much RAM is available for use in the TICkit at each point of program execution. If ever the MP is greater than or equal to the SP, a STACK OVERFLOW error message is reported in the dialog area.

5.3 Debug Commands (doing what you want to do)

```
====================|Debug Dialog|============================|Watch Points|===
|F=specify symbol and token Files                             |                |
|D=Download to TICkit        C=Compare file with TICkit       |                |
|                                                             |                |
|V=memory Value access       W=Watch value manipulation       |                |
|E=Execute and disconnect    M=execute and Monitor program    |                |
|B=Breakpoint manipulation for program monitoring             |                |
|                                                             |                |
|S=Step into function        P=Pass over function             |                |
|T=Trace through the program and display tokens               |                |
|R=Reset the TICkit, restart the program at its beginning     |                |
|Q=Quit debug, return to DOS (TICkit will run program)        |                |
=================|TKN:first   |=|SYMBL|=======================|MP:  |=|SP:  |==
```

The first command to become familiar with is the '?' command. This will display a brief key to the debug commands in the debug dialog box as shown above.

The fourteen, one letter commands are all that are required to debug a TICkit program. The summary of these function follows:

?: Display a summary of commands. This command is useful while becoming familiar with the debug program. This command has no effect on the status of the program being debugged, but simply provides a simple on-line reference for the user and suggests which command might be useful at a given point in debugging a program.

F: Specify a file name to associate with the program in the TICkit. Only a root name is required. When the file name is entered, the debugger will attempt to locate both a token file and a symbol file of the name given. The token file will be used by the Download (D) and compare (C) commands. The symbol file contains all symbolic information like source line information and global variable name and size.

D: Download the token file to the TICkit. This command will ask the user for a Yes (Y) before continuing to prevent an accidental download. After the file downloads, the debugger will automatically do a comparison of the TICkit EEprom with the token file to verify the file was downloaded correctly.

C: Compare the token file against the contents of the TICkit. Only success or failure is reported. To prevent commercial programs from being pirated from a programmed TICkit, the download and compare debug commands only send information to the TICkit. In other words, there is no way to read the contents back from the TICkit.

V: Allow the user to look at and optionally change a value in the TICkit memory. When the command is entered, a line is displayed in the dialog area which asks for the value's address or name. At this point a TICkit RAM address or a symbolic name for a global variable from the source file may be entered. If an address is entered, the user will also be asked for a size of the memory value. Enter 'B' for a byte, 'W' for a word, or 'L' for a long. If a symbol name is entered, the size of the variable will be known automatically. The user may also simply press return when asked for an address or symbol name. This will cause a list of global symbols to be displayed in the dialog area. A variable can be chosen from this list by using the arrow keys and the <return> key. However the variable or address is entered, the debug program will display the current contents of the address followed by a colon. The user may enter a new value or press return to leave the value unchanged.

W: Manipulate Watchpoints. This function is used to maintain a table of up to five variables that the debugger should watch. Each value that is watched will display automatically in the watch point area of the debug screen. When the user asks to manipulate watch points, the debugger will display a line in the dialog area asking for the watch point number. This is a value, one through five, that specifies a watch point. After this number is entered, A list of variables will display in the dialog area. Use the arrow keys and the <return> key to select which variable to watch. At this point the debugger will automatically display the value for the memory location. A watch point can be removed by entering the watch point number preceded with a minus sign.

E: Executes the program contained in the TICkit EEprom from the current program counter (PC) location. The TICkit will stop asking for debug commands, effectively disconnecting from debug, at this point. Console information will continue to be communicated to/from the TICkit. The program within the TICkit may restore connection with the debugger by executing the "debug_on" function. Any breakpoints will be ignored while there is no debug connection to the TICkit. To execute a program but retain the debug connection, use the monitor (M) debug command instead of the execute (E) command.

M: Monitors the TICkit program while it executes. This method of program execution is much slower than normal TICkit execution, but maintains the debug connection between the debugger and the TICkit. This allows the debugger to update memory value watch points (when implemented), and to stop program execution when a break point is detected. An alternative to using the Monitor mode, is to modify the program and place debug_on() and debug_off() function calls in key areas of the program. The debug_on function has the same effect as a breakpoint. By using the monitor function in conjunction with the debug_on method, a program can be debugged much faster and easier.

B: Manipulate Breakpoints. Break points provide a means of interrupting program execution at predefined points in a program. This is often useful in larger programs where only a certain part of a program needs to be debugged. When a program is executing in monitor mode, execution will halt as a source line marked as a break point is about to be executed and the user will be asked for a debug command. Up to 10 break points can be active at one time. Setting breakpoints is very easy. After the user requests to manipulate break points, a list of current breakpoints is displayed in the debug dialog area. The debugger will then ask for the break point command. Enter the number of the break point to modify. At this point a list of source lines will display in the dialog area. Scroll through this list to select the desired line as a break point using the vertical arrow keys and the page up/down keys. Press <enter> to select the desired line or <esc> to cancel the break point selection. Break points can be removed by entering the number of the break point preceded by a minus sign at the break point command. Breakpoints can also be specified using the "default breakpoints within symbol file" method. In this method, the user edits the .SYM file for the program to be debugged. Any line which is to have a break point should have a '+' placed as the first character of the desired source line. Using this method will cause the debugger to automatically load the break points for these lines when the file is selected.

S: Step into subroutine. This command will execute the current line and display the next source line either in a subroutine or the next consecutive line of the program. This command is used to test all levels of the source code.

P: Pass over subroutine. The pass (P) and the step (S) debug commands are almost identical, but differ in the way they handle calls to subroutines. The Pass command will execute the subroutine, but will not display any source of the subroutine. The next source line displayed, and the next opportunity to enter a debug command, will not occur until the source line immediately following the subroutine call is about to be executed.

T: Trace tokens. This command will execute the next token and ask for another debug command. Use this command for debugging programs that do not have an accompanying symbol file, or to see exactly what is happening at each token of a program.

R: Reset the TICkit. Restores all I/O pins of the TICkit to power-on status and starts the program from the initial point.

Q: Quits the debug program. The user will return to the DOS prompt, or other calling program if the debugger was started from a launcher. This will implicitly cause the TICkit to execute when the debug connection times out in the TICkit.

These commands are simple but effective for tracking down run-time bugs. Users will use the Pass (P) and Step (S) commands most frequently. Try out the debugger on the sample program "first" to get a feel for how to trace through a program.

A program can also be modified to include "debug_on" and "debug_off" function calls. This can be useful for speeding up the debugging process. Using these functions in areas of the program that need debugging can be great for skipping larger sections of a program that either do not need debugging, or which must run at full speed for some reason.

The <esc> key or the <ctrl-C> key can also be useful at various points in debugging. They can be used to cancel a command. This might be particularly useful when a request for a debug command is not displayed. For example, the <ctrl-C> key can be used to exit the debug program while the "Execute" command is active and the target processor is running.

6 The Compiler Program

6.1 How to invoke the compiler...

The Compiler is definitely the most complex of all the programs in the FBASIC TICkit package, and yet it is probably the easiest to use. Simply enter the word FBASIC at the command prompt followed by the name of the primary source file to compile.

The source file is called "primary" because there may be multiple source files for a program through the use of the LIBRARY and INCLUDE statements in the primary source file. The primary source file is the only source file in the program not referenced by any other source file. The primary source file references all the other files.

6.2 The FBASIC command line

```
FBASIC <source_file_name>  [<symbol_name>  <symbol_contents>]
```

In our "first.bas" example, the user would type:

```
fbasic first
```

The compiler will start and report the progress of the compile. If the compiler finds any problems, error or warning messages are displayed. In the case of error messages, no final token file or symbolic file will be created. Warnings allow the compile to continue, but put the programmer on notice that a possibility of error in the source file(s) exist. The best programming practice is to write programs that do not generate warnings or errors.

6.3 What do the error messages really mean?

Error messages can be a bit cryptic sometimes. Often this is because the compiler is not able to determine the desired meaning of a line so the report of the error makes little sense to the programmer. However, examination of error message reveals that there are four distinct pieces of information in every error or warning report.

```
ERROR: LCD_FMT.LIB(37) Unknown expression.
```

The above error message is typical of the error reports from the compiler. The first word indicates is the error report is a true ERROR or if it is just a WARNING. The second word is the name of the source file where that the error was discovered. Next is a number enclosed in parenthesis. This number is the number of the errant line in the file named. The line number may be the most useful information in an error report because it allows the programmer to find and examine the line directly with a text editor. The remaining part of an error report gives the programmer some hint of what is wrong with the line. Often, a single error will produce several error reports since the compiler is not really sure what is wrong with the line. After all the source files have been scanned for errors, a final count of error and warning producing lines is displayed. Only the number of lines with errors and warnings are reported, not the number of error reports. This is usually a more accurate indication of the number of actual errors in a program.

6.4 Command line Symbol Definition

The FBASIC command line can also be used to define one symbol within the compile. Defining a symbol from the command line is useful for creating multiple programs from a single source file. For example, a motor control program may be identical for two motors except for the RPM sampling delay of the more powerful motor. A single source file for the two versions of the control program can be used in which the delay is dependent on a symbol's definition. Simply compile each version with a different symbol value. This technique is especially valuable as programs are modified throughout their life. A single source file ensures that all versions of the program get updated with exactly the same modifications. The program fragment and command line below illustrate this technique.

```
.
.
.
   delay( rpm_sample_interval)
.
.

fbasic rpm_sample_interval=3000
```

6.5 The Symbol file: A neat debugging trick

The compiler will produce two files as output. One file is the token file. It will share the same root name as the primary source file but with the extension ".tkn". This is the file which is downloaded to the TICkit. The second file is the symbol file. It also shares the root name of the primary source file but has an extension of ".sym". This file contains a list of all the source lines in the compile that actually produce tokens and the address of the first token of the line where it will reside in the TICkit EEprom. Also, a list of Global data symbols is contained in the symbol file which matches TICkit RAM offsets with symbolic names and types.

All of this information is used by the debugger during tracing. The user may wish to edit this file to place default break points in a complicated debug session. This is accomplished by placing an '+' at the beginning of a line that is to have a break point where it appears in the symbol file. Special care must be exercised when editing a symbol file. If any offsets are changed, or the order of lines is altered, the debugger will become confused.

Default watch points can also be specified in a similar way. Simply place a '+' at the beginning of the line which references the symbol to be watched in the symbol file. Only the first 10 break points will be loaded, and only the first 5 watch points will be loaded using the symbol file method.

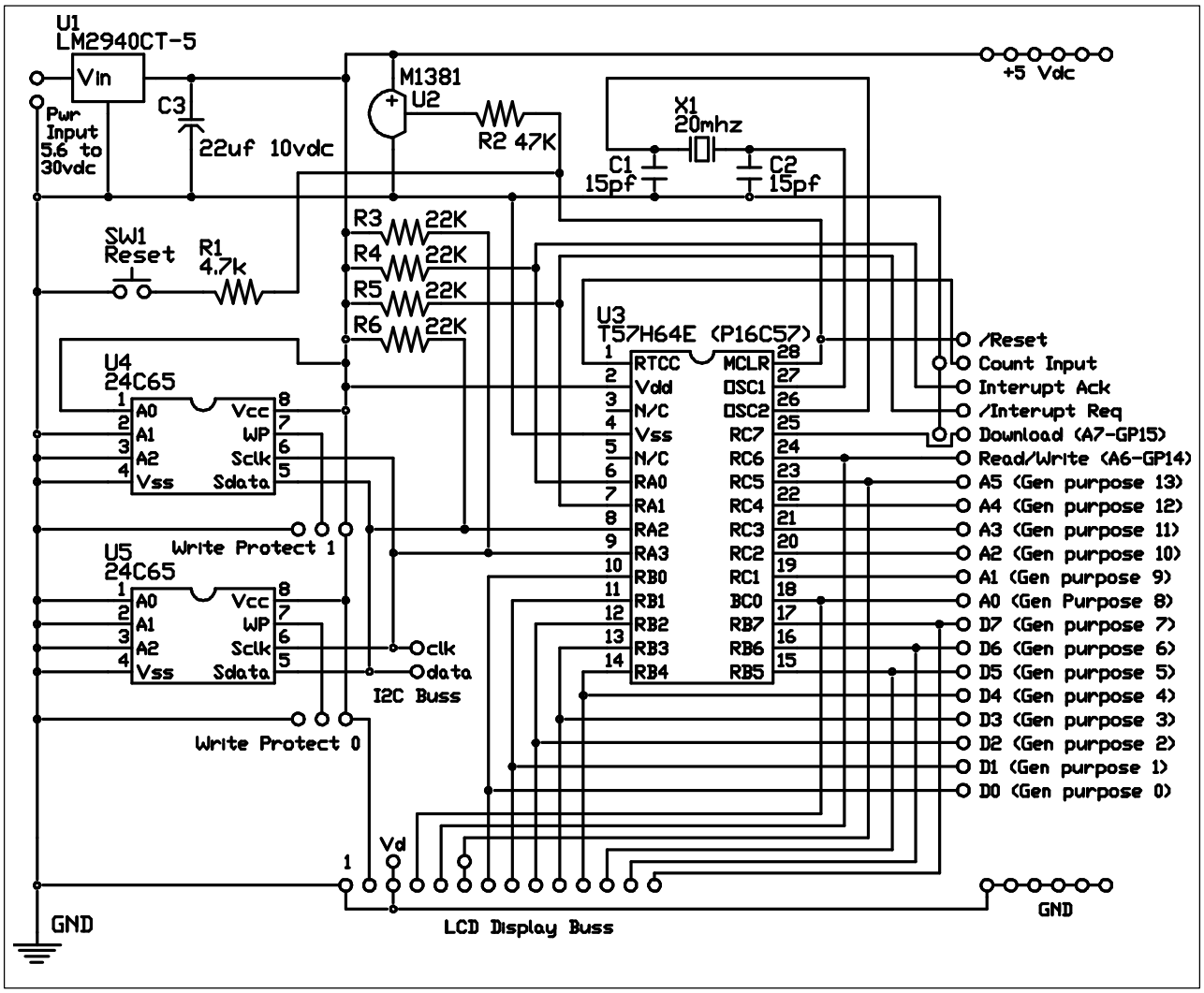
6.6 Compiler Method of Setting Break and Watch Points

The compiler can also set default Breakand Watch points in the symbol file. Use the keyword, "BREAK", at the beginning of any procedural line to associate a break point with that line. This keyword has absolutely no effect on the token file, but places a '+" in the symbol file at that line. The line that the BREAK keyword is used on must be code producing. For that reason, a REP statement or similar statements are not able to trap the BREAK.

Watch points can also be set in the source file. Use the keyword, "WATCH" at the beginning of any GLOBAL or ALIAS statements. At this time, none of the debuggers are capable of watching local values or parameters. Future debuggers may have this capability.

Appendix A: TICkit57 Hardware

A.1 FBASIC TICkit57 schematic diagram



This diagram is the schematic for a 20MHz, 64kbit EEprom TICkit 57. The crystal can be either a 4MHz or a 20MHz, depending on which interpreter program is contained in the preprogrammed PIC. The EEproms may be either 2Kbyte (24LC16) or 8Kbyte (24C65) versions, also depending on the program in the PIC. For the 8Kbyte versions, up to 8 EEprom devices can share the same two lines (SCL and SDA) from the PIC. The combination of the A0, A1, and A2 EEprom select lines determine the addressing of the EEproms. For a single EEpromconfiguration, all lines A0, A1, and A2 should be grounded, as shown.

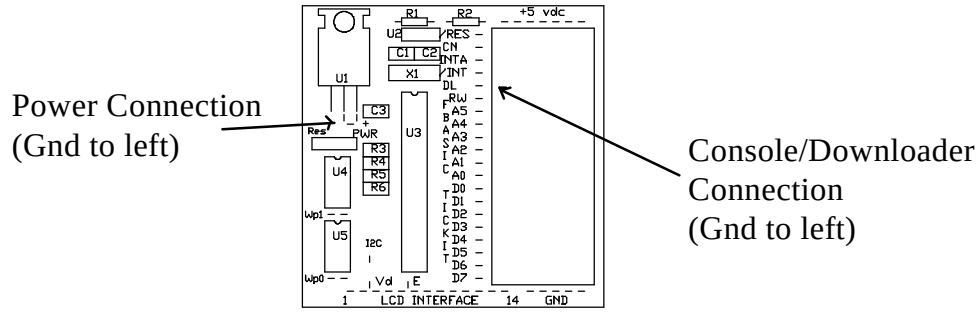
Support for two EEproms addressed in blocks 0 and 1 is provided and each EEprom may be individually write protected.

A.2 TICkit57 Specifications

Physical Dimensions: Overall; 2.5 x 2.5 inches,
Prototype area; 1.0 x 2.5 inches
Power Supply: Input; at least 5.7 volts @ 50ma Output 5.0 volts @ 900ma
Input/output: I/O pins can sink up to 40ma each or 150ma total. I/O pins can source 50ma total.

See the Microchip™ PIC databook for I/O specifications. All PIC16C57 I/O parameters apply to TICkit I/O lines. 4MHz versions use less power and can operate on a lower voltage.

A.3 Component Placement Diagram

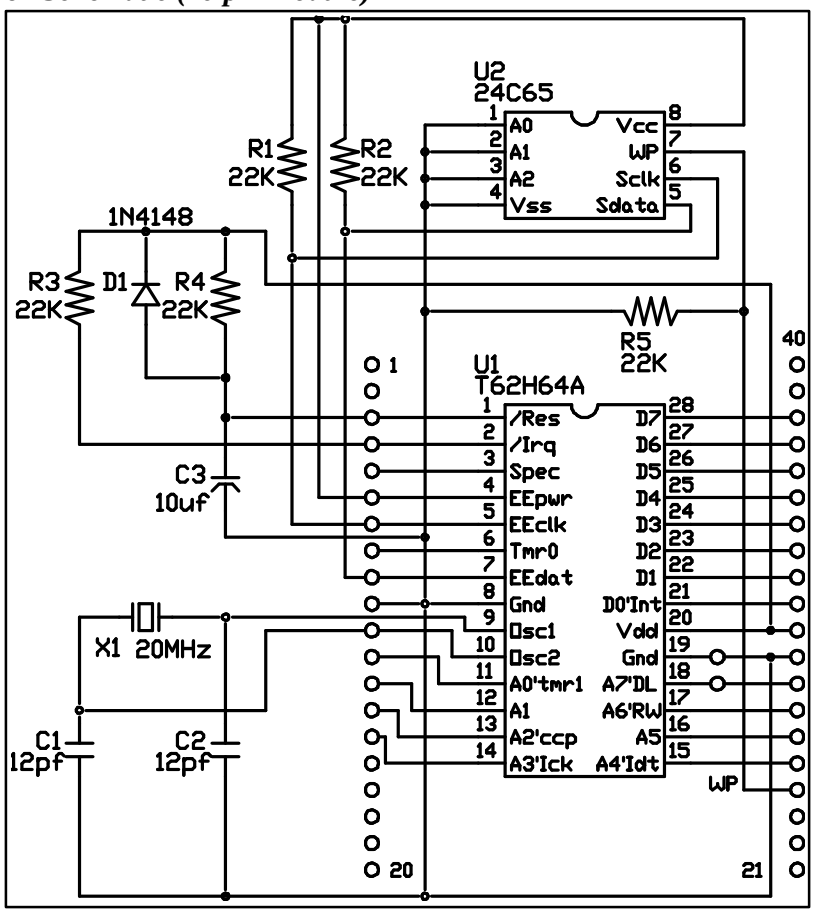


The above diagram shows the locations of components and pins for the TICkit. One important point to notice, is that the data group of outputs is numbered in the opposite order from the address group pins. This is simply a placement issue, but there is a possibility of confusion when wiring components to the TICkit.

Another point to notice is that the power and download connections are non-polarized two pin connectors. The ground pin is always to the left, but the user must exercise caution when applying power or when connecting the Download cable to ensure proper connection polarity. Reverse polarity will not damage the TICkit however - DO NOT PLUG THE POWER INTO THE DOWNLOAD PORT - this will destroy the TICkit interpreter IC.

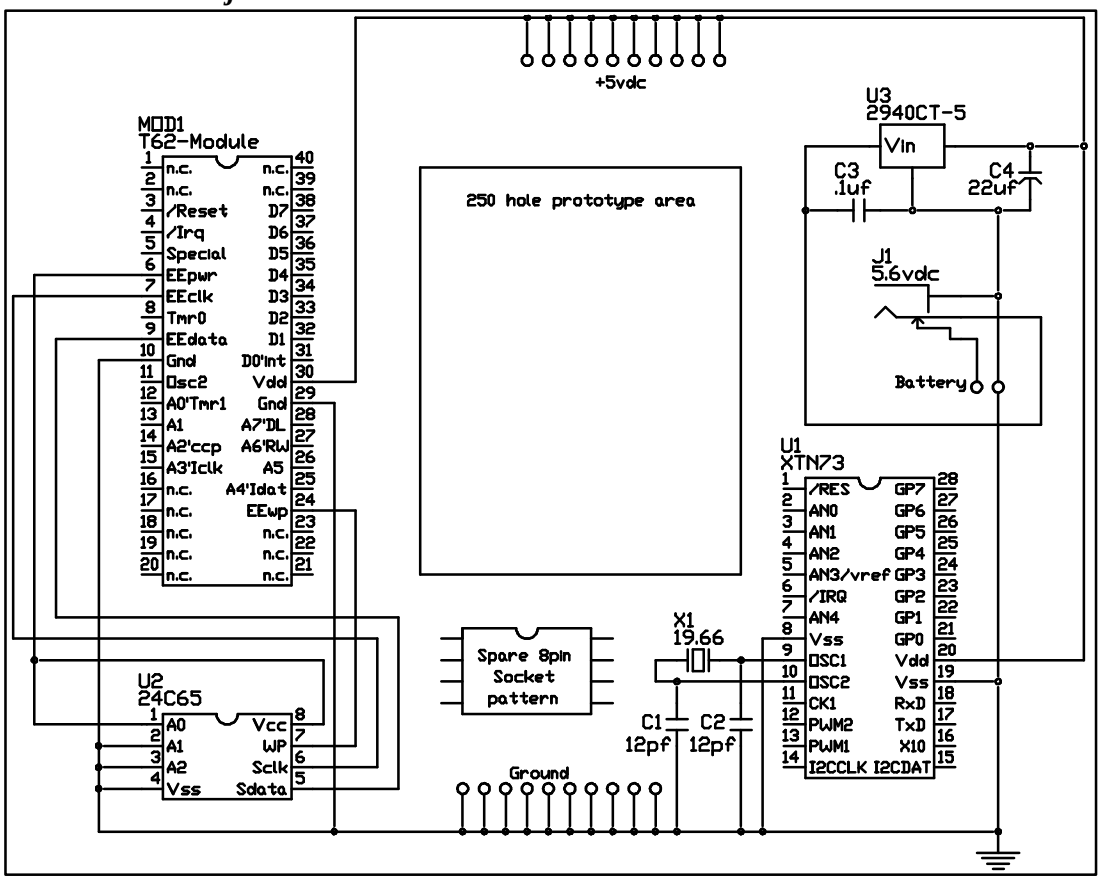
Appendix B: TICkit 62 Hardware

B.1 TICkit 62 Schematic (40 pin module)



The TICkit 62 is available as a single IC or as a 40 pin module. The 40 pin module is a small printed circuit board with a pin pattern and overall size that matches the standard size of a 40 pin DIP. The schematic shown above is the circuit for the module. Notice that some of the top and some of the bottom pins have no connections. Components D1, R4, and C3 form a basic reset circuit which ensures that power is stable before the T62 processor IC begins to run. R3 pulls the /IRQ input high to eliminate any false Interrupts. Interrupting devices connected to this pin should all be open collector (open Drain) to allow wire or-ing of the inputs. R1 and R2 pull the I2C lines high. If you will be using these lines to connect to other I2C devices which are 24 inches or more away from the TICkit, pull the lines stronger with resistors as small as 1.2K ohms.

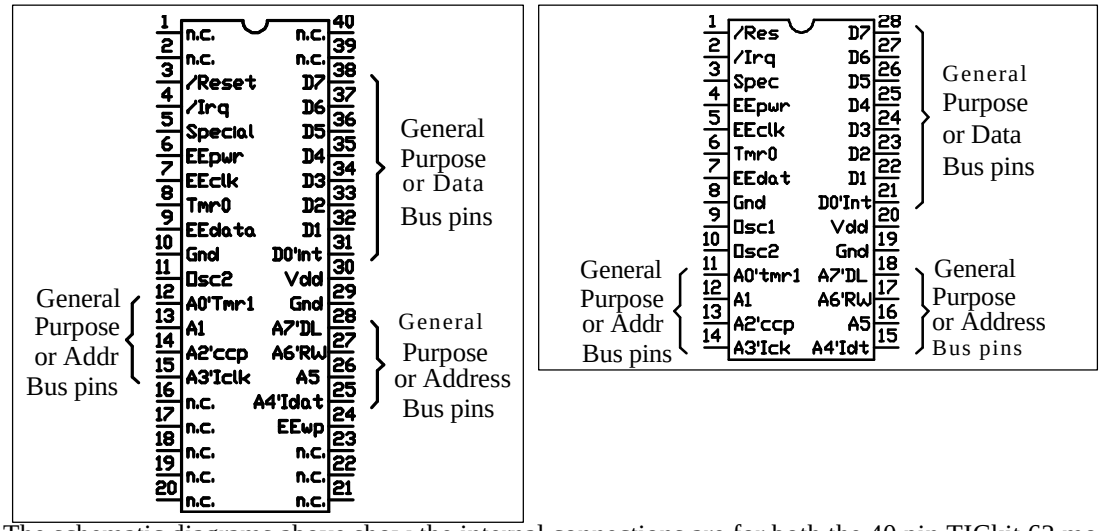
B.2 TICkit 62 Project Board Schematic



The T62-PROJ project board provides a means for wiring up simple TICkit 62 based projects. A 40 pin IC socket accepts the TICkit 62 module. Additionally, a +5 vdc regulated power supply is implemented on the board. Simply connect any DC source between 5.6 and 18 vdc into the coaxial power connector (center -). Notice that input voltages greater than 6 volts will limit the power output of the supply because of all the excess voltage the regulator needs to drop. This will overheat the regulator if a larger current is being drawn and cause the regulator to automatically shut itself off.

A socket for an additional EEprom is supplied which has already been strapped for block 001 (the second 8k block in the TICkit 62's address space). There is also a foil pattern for an Xtender or a second TICkit 62 on the board. Simply solder in the Crystal, IC socket, and capacitors. I2C and other connections will have to be hand wired to complete an Xtender installation.

B.3 The TICkit 62 Module and IC pin diagrams



The schematic diagrams above show the internal connections are for both the 40 pin TICkit 62 module (on the left) and the TICkit 62 interpreter IC (on the right). The interpreter IC is available in both a 28 pin PDIP and 28 pin SOIC package.

B.4 Making your own layout using the 28 pin IC

Using the IC alone is not recommended for your first experience the TICkit. However, for production runs, or for smaller and lighter circuits, you will probably want to use the TICkit interpreter IC instead of the module. There are few special considerations when using the IC alone but the following list will make sure your project goes off without difficulty.

1. Connect both of the IC's ground pins to ground. On the module, only on pin needed to be grounded, but the IC needs both pins to be grounded.
2. Keep the Oscillator wire runs as short as possible. Using a crystal time base is suggested over a resonator to ensure that communication baud rates are as close as possible to the official value. Variations between the sending and receiving communication time bases are sometimes large enough to cause communications errors due to the way in which the async start bit is detected. Even a resonator with an error as small as 1% may result in communication if the sending device has a 1% time base error, and the baud rates are high (9600 and above).
3. The reset circuit used in the module is probably more elaborate than required by most applications. However, the reset pin should never be connected directly to Vdd. A resistor of at least 10K should be used to prevent the IC from sensing a reset voltage greater than Vdd which is the ICs internal programming condition.
4. The pull-up resistor for the EEprom bus (an I2C buss) need to be matched to the overall length of the buss. If the bus length is quite long, pull up resistors should be used at both physical ends of the bus. The 22K ohm resistors used by the module are sufficiently low for lengths up to approximately 24 inches. The pull-up resistance should not be less than 2K. For long EEprom bus lengths some experimentation should be done before a PCB is layed out to ensure that the communications are reliable.
5. The Microchip PIC16Cxx ICs are very resistant to static discharge, but the clampling diodes used for this protection can create problems if your circuit will ever be partially powered down. Because all I/O lines are diode clamped to both Vss and Vdd, any voltage which remains on an I/O line may inadvertantly power the IC. Series resistors or other similar measures may be used to prevent the Interpreter IC from running or drawing power in a power off situation.
6. The 24C65 EEproms used by the TICkit to store the user's program and data generates its own programming voltages and timing. This is convenient from a development point of view, but can be a source of problem when you do not want your program to accidentally be written over. The TICkit has solved this problem by supplying power to the 24C65 from one of its I/O pins. The forces the EEprom into reset during power up and down. Therefore, the EEpower pin can and should be used as the system reset to keep all devices reset until the

controller is stable. You may also wish to use 24LC64 EEproms which have a hardware write protect pin on them.

Using the IC instead of the module creates a more compact and less expensive design, so do not be intimidates to venture into this type of project.