Legal and License Information

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.

Versions and Accuracy

This manual documents features for TICkit interpreters TICkit62 version C. All information contained in this manual is believed to be accurate. However, Protean Logic disclaims any responsibility for incorrect information contained in this manual.

Some features documented in this manual may not be completely implemented in current releases of software or hardware. These features are scheduled to be released in the near future and are documented now to reduce manual printing costs as these features are implemented. Where this is the case, notations indicating the versions containing the new features are contained in the "readme.txt" file supplied with the release disk. Review the readme.txt file prior to program design to ensure that necessary features are implemented in the current release.

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.

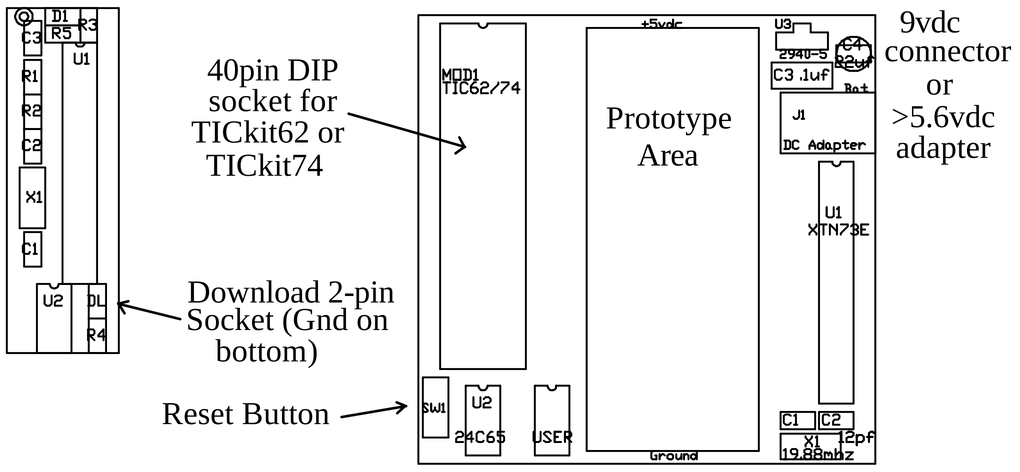
Debugging refers to the process of watching the TICkit execute a program. Debugging is accomplished by running the debugging program on the console, which is connected to the TICkit via a two wire cable, and performing special debug commands that the TICkit understands.

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.

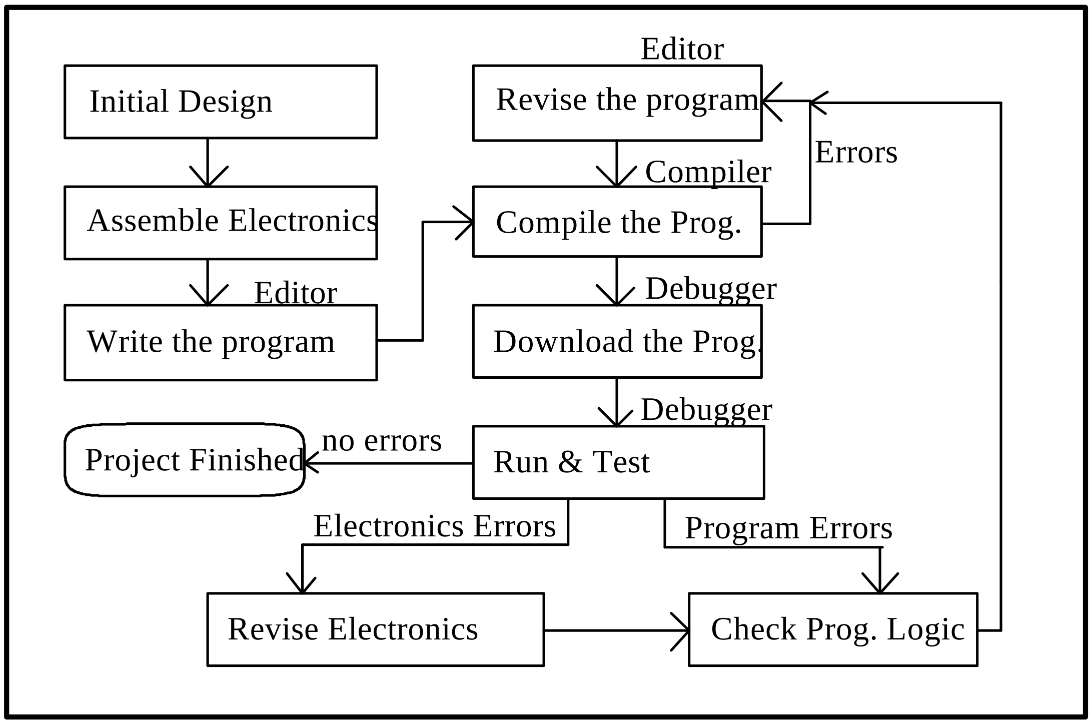
If all went well with the download, there will be a message indicating the file was downloaded and verified. The PC (program counter) register will be pointing to the first token of the downloaded program, and a command prompt will display. Now press the letter 'E' which is the debugger command for execute. 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 expressionevaluation, 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 a 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 librarywhich 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 escape characters 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 arrayof 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 SEQUENCEstatement. 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. RAM is 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 librariessupplied 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 longs while 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.

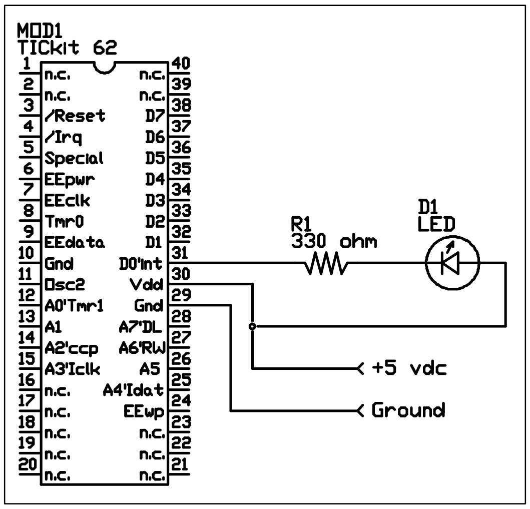
3.15 Check out the the Protean Web Site

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 Simple Examples

4.1 A simple program to blink an LED

After you get your TICkit up and running the "Hello World..." program, a good second program is a simple program to blinkan LED. This assures that you understand basic I/O and how to connect devices electrically to the TICkit. In this example a general purpose I/O pin drives an LED via a current limiting resistor, R1. The output is wired to be low active, which means the LED is lit by outputing a ground level. It is desirable to drive higher current devices at ground level because the internal nature of the TICkit processor can drive higher currents from ground than from +5 vdc. The circuit is shown below.



An LED (Light Emitting Diode) is a special diode fabricated to glow brightly when a current passes through it. Like all diodes, it has a polarity.

In the schematic symbol, the arrow should point to the more negative connection to forward bias the LED. The cathode (the terminal the arrow points to) is usually indicated by a flat side on the LED. The anode (the terminal the arrow points away from) usually has the longest lead.

Because an LED's junction drop is 2 volts, a current limit resistor is required to prevent the LED from burning out in a 5 volt system.

This circuit is very simple. When your program instructs, the TICkit processor will turn on an internal switch that connects the pin labeled D0 to the ground. This completes a circuit in which current flows from the +5 vdc power supply input, through the forward biased LED, through the 330 ohm current limiting resistor and then through the TICkit processor to the ground of the power supply. No other pin of the TICkit module need to be connected. For the +5 vdc input you can use any regulated power supply. Many people have access to a 5 volt supply. If not, you can use one of the circuits shown in the next section as a power supply. Any of the general purpose outputs can be used for this program (pins labeled D0 through D7 or A0 through A7). They all function in the same way. When writing your programs refer to the pins through their cooresponding symbolic names. For example the pin labeled D0 is symbolically refered to as "pin_d0" within a program just as the pin labeled A5 is called "pin_a5". The symbols for the pins are actually numeric constants that evaluate to a number between 0 and 15. Pin D0 is pin number 0, pin D1 is number 1 etc. and pin A0 is pin number 8, pin A1 is pin number 9 and so on. It is usually preferable to refer to constants by symbolic name as it makes the program easier to understand and allows easier modifications later on. Numbers or variables can be used in the pin_high() or pin_low() functions when your application can benifit from a pin reference that is variable.

The program for blinking the LED is equally simple. Most of the program is the required fbasic verbage to inform the compiler of the version of the TICkit and where to start the program. Before showing the final LED blinking program, examine the program below to simply turn on the LED.

```
DEF tic62_c
LIB fbasic.lib

FUNC none main
BEGIN
    pin_low( pin_d0 )    ; this is the same as pin_low( 0 )

    REP
        debug_on()
    LOOP
ENDFUN
```

The first function this program executes is the pin_low(pin_d0) line. This function makes the specified pin an output and switches it to ground. Once this line executes, the LED is on. The lines at the end of the program are there because of the nature of a controller. The TICkit is a controller computer. This means it presumably controls something. In the above program, the last three lines make the program continually ask to connect to the console. If these lines were not there, the TICkit would have no idea what to do when it finished the function, so it would execute random garbage contained in it's eeprom. This could reset the processor or do virtually anything. By putting the loop at the end of our program, we can be sure that the processor is occupied in the loop and the LED stays on for us to observe.

Most control programs are just big loops. They execute the same basic task over and over their entire life. As you write more programs you will see this tendency emerge.

Okay, lets make the light blink. This next program does indeed blink the light, but does not give satisfactory results, see if you can discover why:

```
DEF tic62_c
LIB fbasic.lib

FUNC none main
BEGIN
    REP
        pin_low( pin_d0 )
        pin_high( pin_d0 )
    LOOP
ENDFUN
```

Did you figure it out? The pin_d0 will indeed turn the LED on and off, but at so fast of a rate that it appears to be on constantly. This effect is useful for multiplexing, but not for blinking some lights. The correct program needs some delay for both the on state and the off state. If you have more delay in the off state than the on state, the LED will appear dimmer. If you have more delay in the on state than the off state, the LED appears brighter. This is an important concept called pulse width modulation (PWM) that we will discuss in detail later on. The correct program for a 1 second blink rate is as follows:

```
DEF tic62_c
LIB fbasic.lib

FUNC none main
BEGIN
    REP
        pin_low( pin_d0 )      ; turn LED on
        delay( 500 )           ; leave LED on for 500/1000 of a sec.
        pin_high( pin_d0 )     ; turn LED off
        delay( 500 )          ; leave LED off for 500/1000 second.
    LOOP
ENDFUN
```

The delay() function halts the processor for the specified number of milliseconds (1/1000 second). The delay function expects to see a number between 0 and 65535 (the range for a 16 bit word). Feel free to modify this program. Control more LEDs, or maybe increase the blink rate by lowering the delays. If the blink rate is less than about 1/30 of a second, the LED appears to be on constantly. At this rate, you can alter the relative on and off delays to observe the effects of PWM. The following code produces a continually glowing LED at about 1/2 brightness.

```
DEF tic62_c
LIB fbasic.lib

FUNC none main
BEGIN
    REP
        pin_low( pin_d0 )     ; turn LED on
        delay( 15 )           ; leave LED on for 15/1000 of a sec.
        pin_high( pin_d0 )    ; turn LED off
        delay( 15 )           ; leave LED off for 15/1000 second.
    LOOP
ENDFUN
```

There is another way to turn off the output of a pin besides changing the level of its output. You could use the pin_in() function as shown below.

```
DEF tic62_c
LIB fbasic.lib

GLOBAL byte trash        ; an 8 bit variable used below

FUNC none main
BEGIN
    REP
        pin_low( pin_d0 )       ; turn LED on
        delay( 500 )            ; leave LED on for 500/1000 of a sec.
        =( trash, pin_in ( pin_d0 ))        ; turn LED off
        delay( 500 )           ; leave LED off for 500/1000 second.
    LOOP
ENDFUN
```

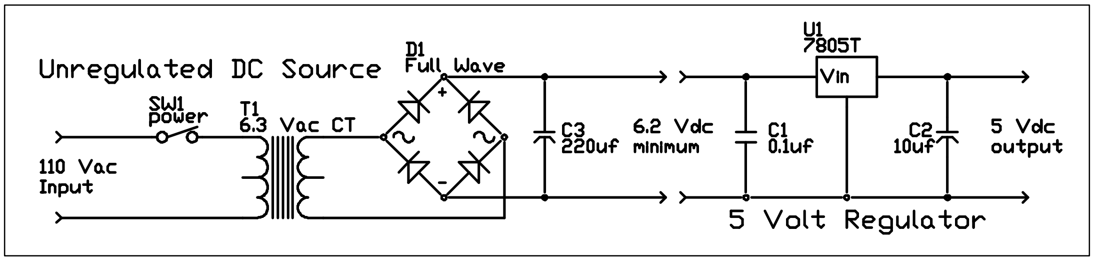
The pin_in function makes the specified pin an input and reads the level on the pin. A 0 is returned if the level is low (<2.5 volts) or 255 if the level is high (>2.5 volts). In our example, we do not care what the level is on the pin, we just want to turn off the output and make the pin an input. The returned value must be assigned to something though, or the compiler will generate an error because it knows the pin_in function returns a number and expects the program to use that value. Examples of the pin_in() function are used extensively in later examples to read button presses.

4.2 Construction techniques and power sources

Lets take a minute and talk about the nuts and bolts of making projects. The TICkit module is designed to easily plug into a solderless breadboard. These are readily available from most electronic parts stores, including Radio Shack. Almost any 6 volt battery can be used as a power source for a TICkit, but make sure you do not use any battery with more than 6 volts output or you will fry the TICkit processor.

There is a power supplyand construction area for a TICkit project on the T62-PROJ project board if you are making a more permenant project. You can just use the power supply on the T62-PROJ board by soldering wires on the +5 and ground buses and plugging these into a solderless breadboard.

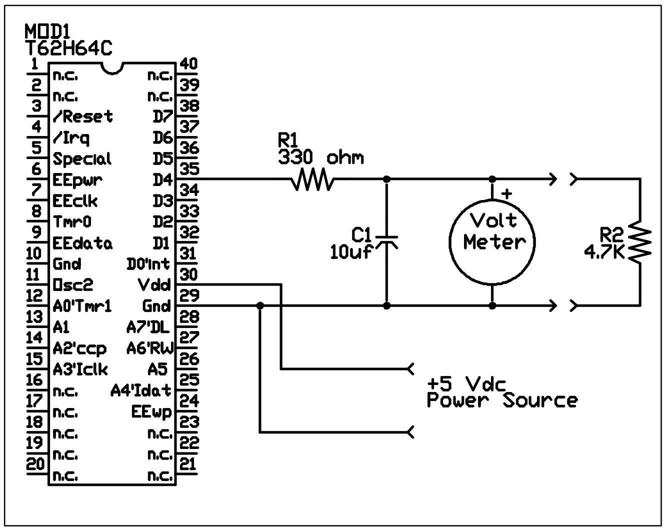
The more reasonable approach is to make or purchase a +5 volt regulated power supply. To make your own, you can use the following circuit based on a 7805 (LM340). All of the parts required for this supply are readily available from parts stores including Radio Shack. The unregulated DC source can also be a wall mount transformer supply.



The circuits shown above are suitable for most typical applications. More advanced projects might require regulators with lower queisent (no load) power consumption to conserve battery power, or you might need more voltages than just +5 volts. There are many, many good texts on power supply design and countless monolythic IC solutions for any of a wide range of power requirements. All the TICkit directly needs is a good 5 volts with a reasonably sharp rise time. The 20 MHz TICkit62 itself consumes less than 30ma not counting loads you place on it. The 4 MHz TICkit 62 uses less than 15ma unloaded.

4.3 A simple PWM circuit for controlling a low voltage DC motor.

We have already touched on the idea of pulse width modulation in our blinking LED example. PWM is a way of producing a variable power/voltage/current output from a switched output. The TICkit 62 has no analog output, only digital (switched) outputs, so PWM is the only direct way to produce a variable (analog) output. The TICkit 62 has two methods of producing PWM directly. The first uses a built in function called "cycles()" to produce a square wave of given duration, frequency, and duty cycle, on any of the general purpose I/O pins. The second method uses some dedicated hardware built into the TICkit 62 processor for continuously producing a PWM output. This method can only produce PWM on the pin labeled "A2'CCP". CCP stands for Counter/Capture/PWM. This second method can actually perform 10 bit PWM.



This program uses the cycles function to produce a ramping voltage between 0 and 5 vdc. The meter can be a voltage meter or an O-scope if you have one.

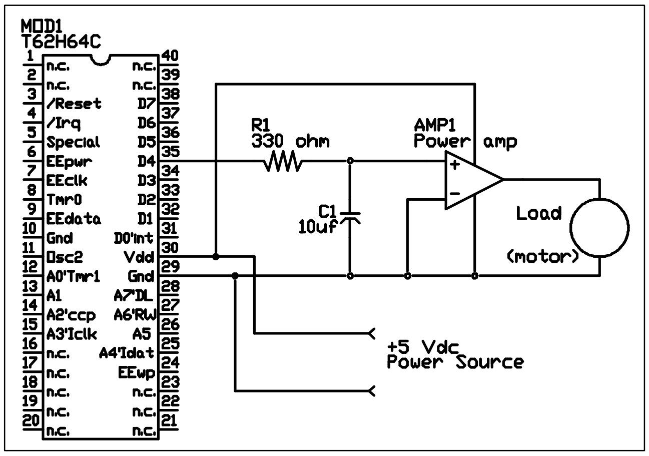
If R2 is disconnected, the voltage repeatedly ramps up to 5 volts then ramps down to zero cleanly. With R2 in circuit, there is a relatively large spike at the end of the ramp and the ramp gets slugish toward 5 volts. These distortions occur because R2 loads the circuit when there are program interruptions in the PWM output.

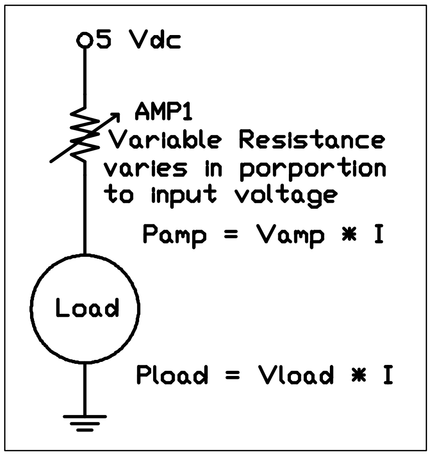
As this circuit demonstrates, the cycles() method of producing PWM is sufficient only if the circuit will not be loaded very heavily. There is a relationship which exists between the driving capability of the PWM device (the TICkit and series resistor), the size of the capacitor, the frequency of the PWM signal, and the load size. If the frequency is high enough, even larger loads can use this method.

Notice in the program that follows, that each time the cycles function executes, only 20 square waves are generated. Between each execution of the cycles function, the program does some math and some flow control. Even though these other program steps take only a small fraction of time, it is enough of a break in the PWM output to create a glitch when there is much load at all on the output. This type of glitch is virtually unavoidable when using a software emulation method to generate PWM.

```
DEF tic62_c
LIB fbasic.lib

GLOBAL word duty_cycle ; make room for a variable and give it a name

DEF wave_length 256    ; produces a pulse frequency of approx 11KHz
DEF per_level 20       ; produces a ramp requency of approx 550Hz

FUNC none main
BEGIN
    pin_low( pin_d4 )    ; make pin D4 an output
    =( duty_cycle, wave_length )
    REP
        REP
            cycles( pin_d4, per_level, duty_cycle, wave_length )
            --( duty_cycle )
        UNTIL ==( duty_cycle, 0 )

        REP
            ++( duty_cycle )
            cycles( pin_d4, per_level, duty_cycle, wave_length )
        UNTIL ==( duty_cycle, wave_length )
    LOOP
ENDFUN
```

One way in which to get a larger driving capacity out of this method of PWM is to connect the unloaded PWM output to some type of linear amplifier like an audio output amp. This works well and eliminates the problem with the limited drive capacity of the TICkit as well as the "glitches" when the TICkit is in between cycles() functions as the program executes. The problem with this method is that it is very power in-efficient. When the output of the amplifier is mid-way between ground and max voltage output, the difference between the max voltage and the output must be dissipated by the amplifier. This generates a lot of heat and wastes a lot of power. The follwing diagrams show the amplifier arrangement and compare it to a variable resistor. The power dissipated by the resistor is equal to the product of the voltage it drops times the current flowing through it. A switch is like a very large value adjustable resistor adjusted to one extreme or the other. So either it drops zero voltage, or it passes zero current. In either extreme the power dissipated is zero because the product of anything multiplied by zero is zero. Now, if this resistor is adjusted mid-way, like our amplifier producing a half voltage output, the power is equal to 1/2 of the max voltage times the current drawn by the load. Just for argument, assume we are dealing with a 5 volt system and a load that draws 1 amp at 2.5 volts. If the amplifier is outputing 2.5 volts it must be droping the remaining voltage (2.5 volts). This means that it is dissipating 2.5 watts (2.5 v * 1 amp). Which is exactly what the load is consuming. Half of our supply energy is wasted and we have a significant heat problem.



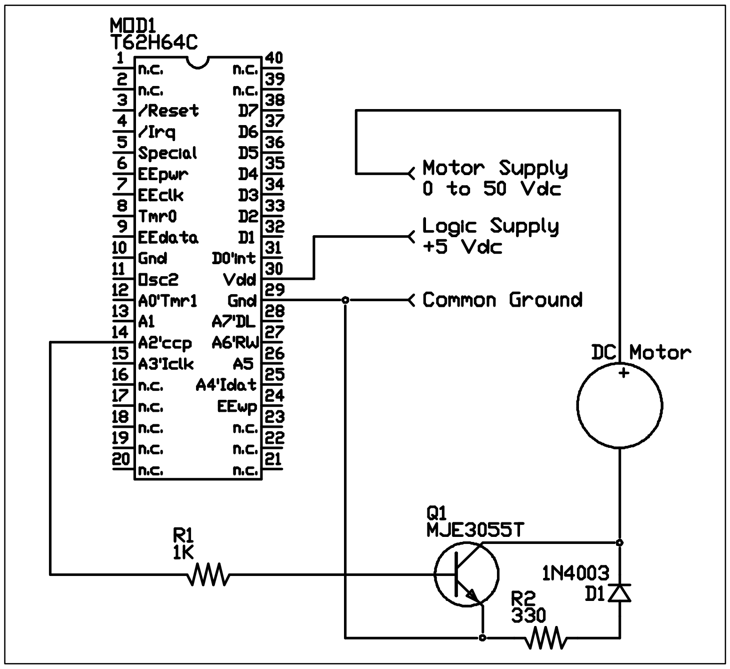


Now lets deal with the actual best way to use the TICkit 62 to control a DC motor. This approach uses the built in hardware to generate continuous PWM. Instead of a built in software routine turning a general purpose pin on and off, the TICkit 62 uses dedicated hardware to turn pin A2'CCP on and off on the basis of values contained in special registers. The TICkit 62 provides functions to set these registers and the hardware does the rest independent of what our program is doing. This is called background functionality.

We control an internal timer, called timer2, to generate the pulse frequency for our PWM. Timer2 has a control, a period, and a count register. These determine the frequency of the PWM. To make the TICkit 62 actually perform PWM, the CCP registers must be configured. These are the control and CCP data registers. Once configured, you write to the CCP data register to control the duty cycle. There are symbolic names for values that can be write to the control register. These constants are defined in the token library. The program looks like this:

```
DEF tic62_c
LIB fbasic.lib

GLOBAL word ccp_reg  ; CCP register is actually a word (16 bit)
                     ; but only the lower byte (8 bits) are used.
                     ; The high byte is used internally as a
                     ; buffer. The Alias statement lets us
                     ; conveniently refer to the low byte
ALIAS byte ccp_duty ccp_reg 0
```

```
FUNC none main
BEGIN
    pin_low( pin_a2 )
    tmr2_cont_set( tmr2_con_on )
    tmr2_period_set( 255b )       ; this produces a pulse frequency
    ccp1_cont_set( ccp_pwm )      ; of 19531 Hz. Clock frequency/256

    =( ccp_duty, 0b )     ; now our CCP unit is set up to do PWM
    REP                   ; this is the main loop
        REP               ; this loop decreases motor speed
            --( ccp_duty )
            ccp1_reg_set( ccp_reg )
            delay( 10 )
        UNTIL ==( ccp_duty, 0b )

        REP               ; this loop increases motor speed
            ccp1_reg_set( ccp_reg )
            delay( 10 )
            ++( ccp_duty )
        UNTIL ==( ccp_duty, 0b )
    LOOP
ENDFUN
```



This circuit controls a relatively large DC motorrunning at a supply voltage of up to 50 volts follows. This circuit can conceivably switch up to 5 amps with this single switching transistor and flyback diode.

Realistically, however, you should only use this circuit for switching 2 amps or less. If you are going to switch higher currents, R1 should be reduced to 150 ohms.

No interface components are shown in the diagram.

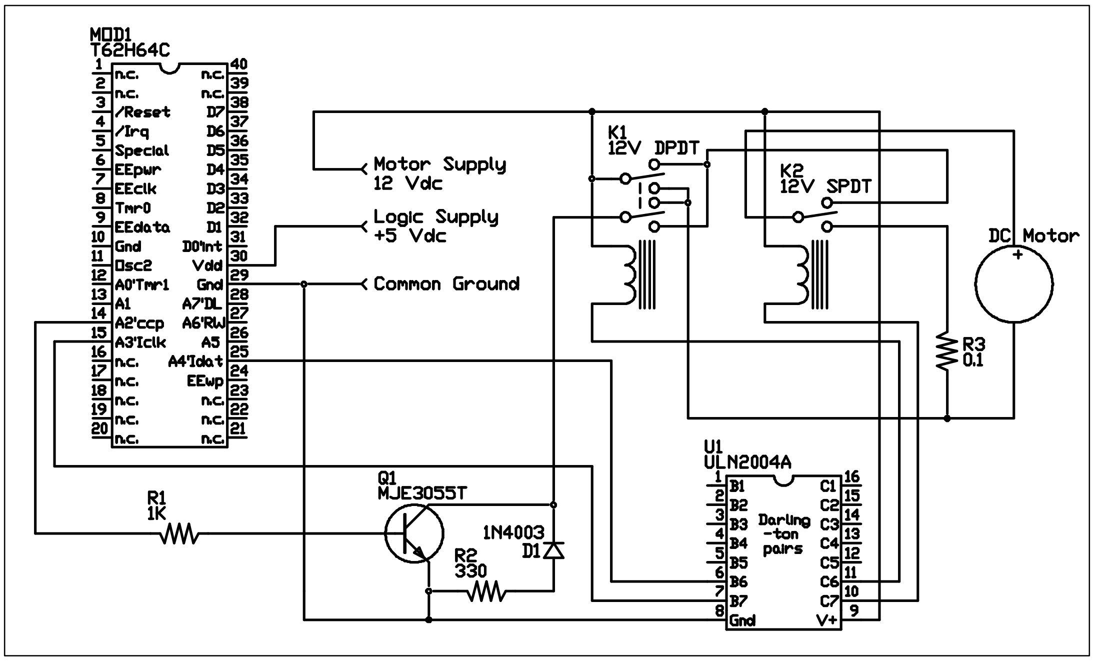
The transistor, Q1 is operated as a saturation switch. This means that when A2 is high, the current allowed to flow through the transistor via R1 is significantly greater than the load current divided by the transistors Gain. Said another way, we are driving the transistorway on. This makes the transistor act like a switch, either it is off and has no current flow, or it is on and has virtually no resistance. The diode D1 and resistor R2 are designed to drain off the parasitic flyback voltage created when current is removed from an inductor (the motor in this case). Bypassing the reverse EMF or flyback voltage safegaurds components and reduces heat load on Q1.

4.4 Controlling relays for motor direction and electric braking

Now that we have a means for varying the drive to a motor or some source like it, we may need to reverse the direction of the motor or provide a means for braking. The easy way to do this is with relays. Driving relaysis actually very easy with the TICkit. Our circuit will use an IC which has several transistors in them arranged as darlington pairs. This single IC will provide buffering for up to 7 relays. We only need to drive two for this example.

Relays are simply magnetically operated switches. When the coil is energized, the switch is thrown. Two different types of relays are used in this example, one is a DPDT (double pole double throw) for motor polarity, and the other is a SPDT (single pole double throw) for braking. The circuit shown below is very similar to the last circuit except that the relays change how the resulting power is applied to the motor. The relay K1 in its un-energized position connects the motor to achieve forward rotation (forward rotation is assigned by convention of the motor's manufacturer . When positive voltage is applied to the motor lead marked as positive the motor rotates forward). When K1 energizes, the positive of the motor is connected to ground and the output of the PWM is connected to the negative of the motor, making it rotate in reverse.

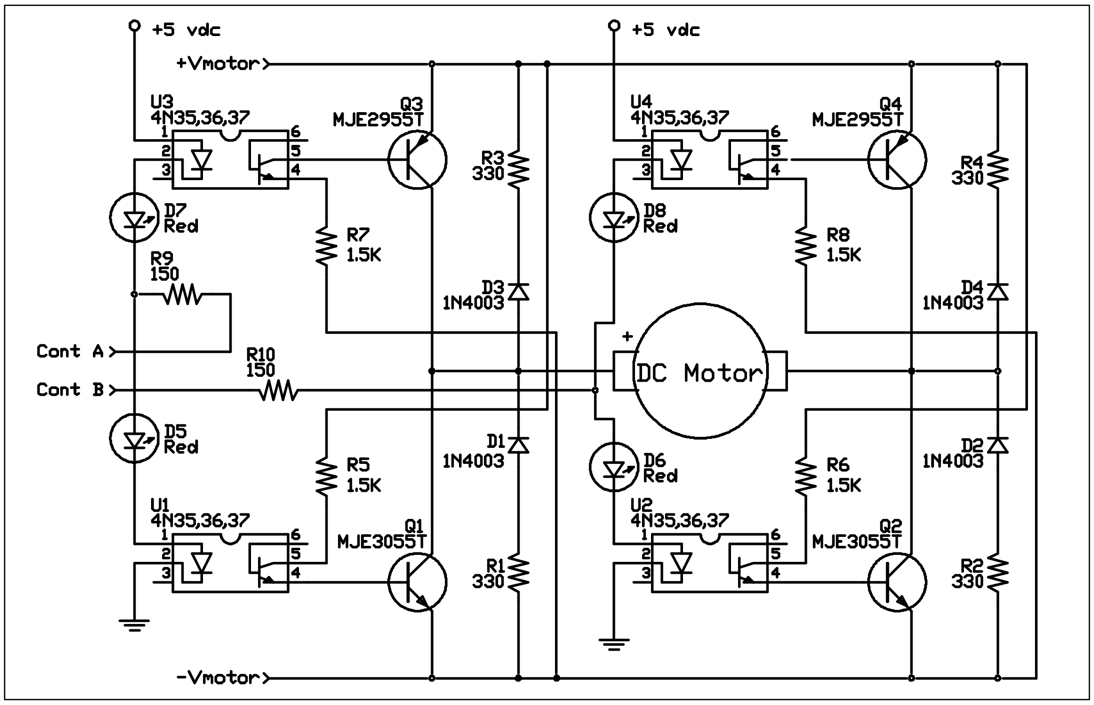
Relay K2 controls whether the positive of the motor connects to K1' output or if it is shorted through R3 to the negative of the motor. When K2 is un-energized, the motor sees power from the PWM and direction control circuits. When K2 is energized, the motor is connected to a resistive load that impedes the rotation of the motor. If the motor is not shorted when power is removed, it simply coasts. If the motor is geared there may be some self braking, but braking capabilities are usually required.



The following program illustrates these types of controls. The program sets up the PWM, turns the motor on at half speed and rotates it forward for 1 second, removes power and brakes the motor for 3 seconds, reverses direction and powers the motor at 1/4 speed for 2 seconds, removes power and lets the motor coast for 6 seconds, then repeats the process. A real control program probably has some type of user interface for setting motor speed and direction instead of a hard coded routine. You can use the console statements with the download cable to make an elementary front end as a further programing exercise. An item that is usually found in this type of program is an acceleration and deceleration routine. If you have delicate instruments or payload handled by the motor, you don't want it damaged by inertial forces as your motor slams on and off. Play with different ideas and see what you come up with. This is the essential electro-mechanical motor control circuit.

As you can see from the program, to energize a relay, perform a pin_high() on the specified I/O pin. In this example we are using a 12 volt supply for both the motor and the relays. If the motor is really large and has large acceleration loads, you might need to separate the supplies to prevent the relays from dropping when the motor starts. Also, you might use a larger voltage on the motors, which either the relay's voltage will need to match, or a separate lower voltage supply will be required for the relays.

```
DEF tic62_c
LIB fbasic.lib

GLOBAL word ccp_reg
ALIAS byte ccp_duty ccp_reg 0

DEF motor_reverse pin_a4  ; use symbolic name for direction I/O
DEF motor_brake pin_a3    ; use symbolic name for braking
                          ; notice that the names imply the
                          ; meaning when the I/O is high

FUNC none main
BEGIN
    pin_low( pin_a2 )
    tmr2_cont_set( tmr2_con_on )
    tmr2_period_set( 255b )       ; this produces a pulse frequency
                                  ; of 19531 Hz. Clock frequency/256

    ccp1_cont_set( ccp_pwm )
    =( ccp_duty, 0b )
    ccp1_reg_set( ccp_reg )       ; turn motor off
    ; now our CCP unit is set up to do PWM
    ; repeat sequence of motor movements.
    REP

        pin_low( motor_reverse )      ; motor in forward dir
        pin_low( motor_brake )      ; motor is under power
        =( ccp_duty, 128b )
        ccp1_reg_set( ccp_reg )     ; power motor at 1/2 speed
        delay( 1000 )                 ; wait 1 second.

        pin_high( motor_brake )  ; remove power and brake the motor
        =( ccp_duty, 0b )
        ccp1_reg_set( ccp_reg )  ; put PWM at 0
        delay( 3000 )            ; wait 3 seconds

        pin_high( motor_reverse )   ; motor is reversed
        pin_low( motor_brake )      ; release the brake
        =( ccp_duty, 64b )          ; power at 1/4 speed
        ccp1_reg_set( ccp_reg )
        delay( 2000 )               ; wait for 2 seconds

        =( ccp_duty, 0b )
        ccp1_reg_set( ccp_reg )  ; put PWM at 0
        delay( 6000 )            ; wait 6 seconds
    LOOP
ENDFUN
```



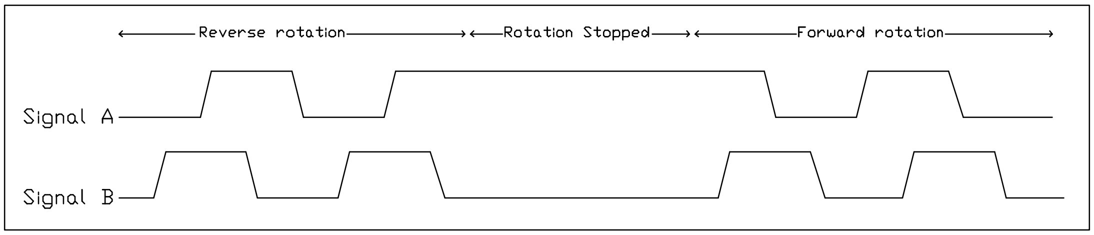
The solid state equivelent of the relay and PWM transistor is called an 'H' bridge. A schematic for a working H-bridgeis shown above. The resistor values were selected for a 12 volt motor @ 2 amp max. If contA and contB are at the same logic level, the motor is not being driven. If contA is low and contB is high, the motor spins forward. If contA is high and contB is low, the motor spins in reverse.

4.5 Closed Loop Circuit Feedback in Control Circuits

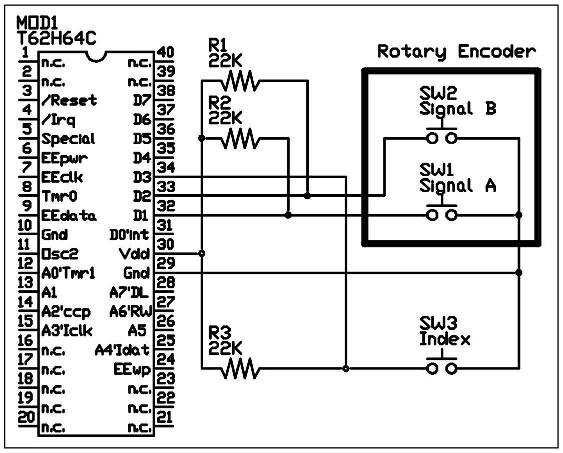
Most control systems, especially those dealing with mechanical control, use a feedback system to see if the desired positioning has indeed taken place. When an action is double checked by a sensor and corrective action is taken the control mechanism is called "closed loop". This is similar to sending a registered letter through the mail with a return receipt requested. You can be sure your letter was indeed delivered. Ordinary mail is "open loop" and you rely on the integrity of the postal system to get your mail delivered, and you tolerate some lost mail.

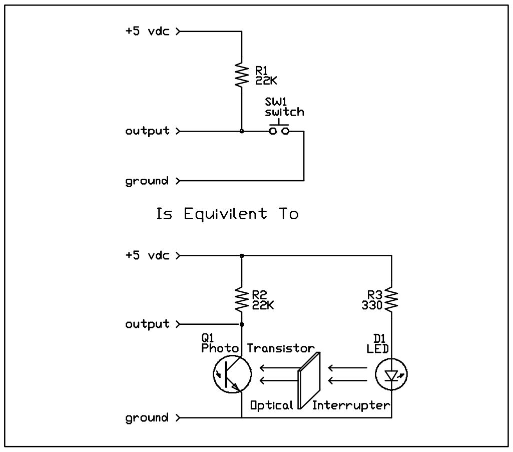
In the next example a quadrature encoding sensor is used with an index sensor to locate the absolute position of a rotating shaft. Although we won't put all the electronics for driving the motor in the schematic, these additional components could easily be incorporated with a motor driving circuit like those talked about earlier.

First, what is a quadrature encodingsensor. An encoding sensor is a collection of switches, either mechanical, optical, or magnetic that indicate the angular position of a shaft. These types of encoders usually have between 16 and 512 positions per revolution. Some encoders produce an absolute binary or Gray's coded position output. The one used in this example is a relative position sensor that produces a quadrature output. The output waveforms look as shown below. When the sensor rotates in one way, signal A's phase leads signal B's, when the sensor rotates the other way, signal B's phase leads signal A's. The sensor electronics need to watch these two signals to increment or decrement a counter. We assume the motor and mechanical inertia of the system prevent the signals from changing too fast. This is a reasonable assumption when the motor's output shaft is geared down. If there are more than 20 phases per second per signal, dedicated electronics are needed to count the position.



Earlier we said the quadrature encoder gives relative position. By this we mean an additional signal, called an "index", is required in the system to reset the position count in the controller. When power is first applied to the system, the controller must turn the motor on in the direction toward the index mark. When the index signal switches, the controller must reset the counter. After this step, the controller has the absolute position of the system mechanics.





The diagrams show an endoder circuit and how an opticalIndex is created. The LED is continuously lit and an optical interrupting fixture is connected to the rotating shaft so that only one position interrupts the beam. When the interrupting is not in place, the light turns on the photo transistor. Because the resistance of the transistor when turned on is so much lower than the 22K pull-up resistor, the output is very nearly at ground level. When the optical interruption blocks the light from the LED, the transistor turns off and has a high resistance relative to the 22K resistor. The output then is very nearly +5 vdc.

The following program fragment for the circuit follows. Notice that this is not a complete program and needs to be integrated into a positioning program, like the ones previously shown, to be a complete servo system.

```
GLOBAL word shaft_pos   ; absolute shaft position
GLOBAL byte prev_sigb   ; previous signal B

FUNC none position_count
    LOCAL byte cur_sigs
BEGIN
    =( cur_sigs, dport_get())      ; read all 8 pins of D port
    IF ==( prev_sigb, b_and( cur_sigs, 0y00000100b ))
        ; no change to count
    ELSE
        IF prev_sigb
            IF b_and( cur_sigs, 0y00000010b )
                --( shaft_pos )
            ELSE
                ++( shaft_pos )
            ENDIF
        ELSE
            IF b_and( cur_sigs, 0y00000010b )
                ++( shaft_pos )
            ELSE
                --( shaft_pos )
            ENDIF
        ENDIF
    ENDIF

    =( prev_sigb, b_and( cur_sigs, 0y00000100b ))
ENDFUN
```

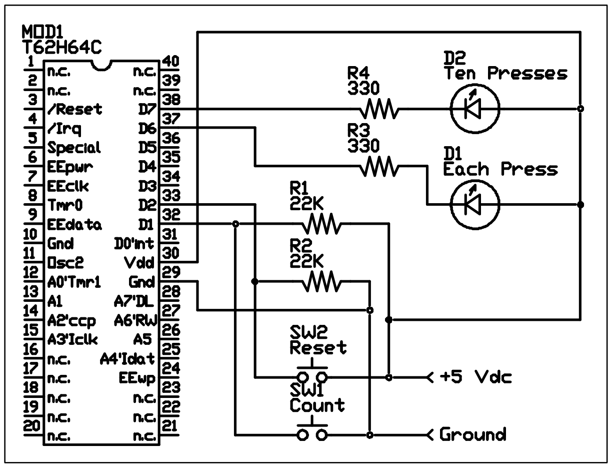
This concludes our discussion on electro-mechanical control. Many other options exist in this arena from driving solenoids, to driving stepper motors, to using self contained servo mechanisms like RC servos. Check the release disk and the Protean web site for sample programs and applications notes. If you are interested in building some of the circuits talked about in this section, Digi-key Corporation and Jameco Electronics are sources for all parts mentioned in these circuits. You can find their contact information at the Protean web site.

4.6 Reading and Debouncing Switches.

No matter what your project is, a simple user interface is often required. A user interface usually constists both of a way to tell the controller what to do, and a way for the controller to tell you what it is doing. We have looked at LEDs as a way for the controller to indicate its status, but how do we tell the controller what to do, aside from changing its program?

The most common answer to this question is a collection of buttons and switches. This can vary from a few push buttons to accomplish a "wrist watch" type of interface, to a full 84 key ASCII keyboard.

We touched on the concepts relating to switch input in the rotary encoder example. The basic electrical problem is to make an SPST button (single pole single throw) produce the voltages required by the digital circuits of the controller. The solution is to use a resistor to either pull up or pull down the voltage when the switch is open. The next circuit example uses two switches and two LEDs. As shown in the schematic below, the switch SW1 is wired so that it connects the pin labeled D1 to ground when it is closed. When SW1 is open, pin D1 sees +5 volts through resistor R1. R1 is called a pull-up resistor because its function is to pull a digital line high when no other component is driving it low. Conversly, SW2 is connected so that when closed, it connects the TICkit pin labeled D2 to +5. R2 pulls pin D2 low when the switch is open, so it is called a pull-down resistor. Both SW1 and SW2 are momentary push buttons, which means they connect only while a being pressed.



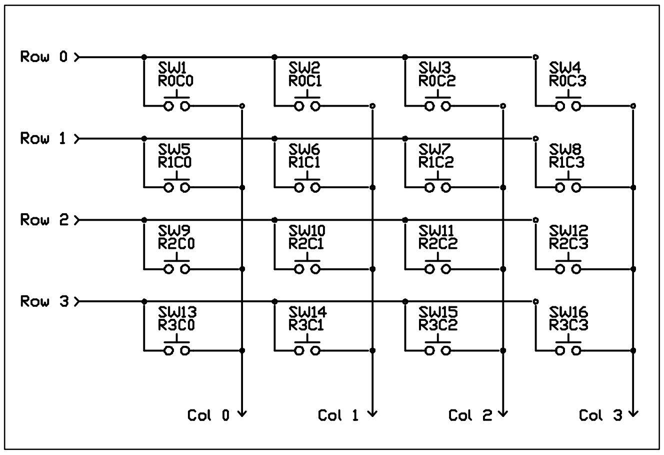
The program shown below uses the circuit above to implement a meaningless program. When SW1 is pressed 10 or more times, LED2 lights. LED1 will light every time SW1 is pressed. Button SW2 resets LED2 if it is on and restores the count of button presses to 0.

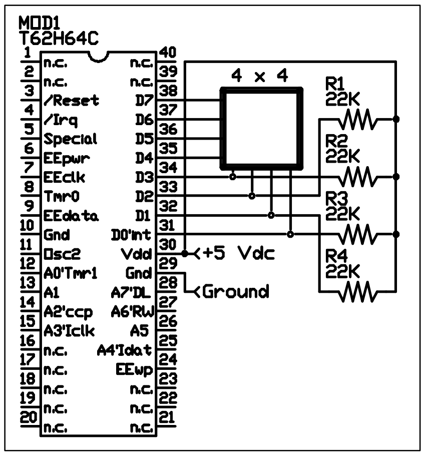
```
DEF tic62_c
LIB fbasic.lib

GLOBAL byte press_count 0b

FUNC none main
BEGIN
    REP
        IF pin_in( pin_d1 )
            ; do nothing the button is not pressed
            pin_high( pin_d6 )
        ELSE
            ; button is pressed
            pin_low( pin_d6 )
            IF <( press_count, 10b )
                ++( press_count )
            ELSE
                pin_low( pin_d7 )
            ENDIF
        ENDIF

        IF pin_in( pin_d2 )
            =( press_count, 0b )
            pin_high( pin_d7 )
        ENDIF

        ; try putting the following in the program later
        ; delay( 20 )
    LOOP
ENDFUN
```

When you type in this program, leave the delay(20) line commented out, and execute the program. You will find the results unsatisfactory. The 10 count LED seems to light too soon, sometimes it lights on the first key press. Why is this?

The reason has to do with the physical nature of a switch. Most switches bounce their contacts due to the mechanical properties of the switch. This means that for a few milliseconds, the contacts are closing and opening for a random number of times. This TICkit processor is fast enough to catch these very fast bounces which look like repeated key presses. Now put the delay(20) line in the program by removing the ';'. The delay of 20 milliseconds makes the program insensitive to key bounce and thus it works just as we expect. Often, there is no need for an extra delay when debouncing keys in a program. Many times there is enough delay associated with the main control function too make the key scanning insensitive to key bounce.

Our next two switch examples involve scanned key matrixIt may seem like a lot of added complexity to scan a matrix of keys when compared to the simplicity of running each switch to an I/O line on the processor. In fact it is more complex, but it uses fewer I/O lines as the number of keys grows, and it requires fewer steps to determine if any keys are pressed. This can save processing time because keyboards spend most of their time with no keys pressed.





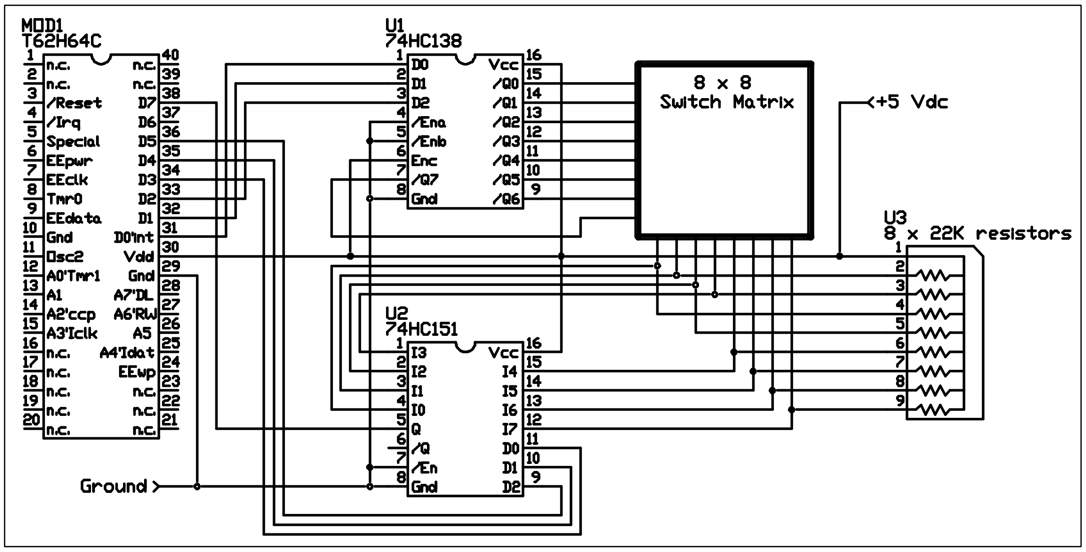
Notice in the first diagram that each key connects a unique combination or row and column wires. It is the combination of row and column that allow the microcontroller to determine which key is pressed. The number of rows or columns may change in different keypads, but the basic idea remains the same. Your program needs to determine the exact meaning of each key. Some keys may produce specific actions, other keys may be converted to ASCII characters for display or for use as data.

The first circuit uses a 16 key matrix arranged as 4 rows of 4 columns. We bring one row of the four low to see if any keys are pressed on that row. The four column inputs are then read to see if there are any lines low, if so, the corresponding key is pressed. It is important that only one row output be low at a time to correctly identify a single key press. The column inputs are all tied high with pull-up resistors to make the inputs high when no key is pressed. If appropriate, however, the program could make all row outputs low and read the column inputs. If all the column inputs are still high, none of the keys are pressed. This can be a useful way to determine if program time needs to be devoted to keyboard scanning. The following program demonstrates the technique used to scan a key matrix directly.

```
DEF tic62_c
LIB fbasic.lib

GLOBAL byte scan_row 0y11111110b
GLOBAL byte scan_col 0y00000001b
GLOBAL byte scan_number 0b

FUNC none main
BEGIN
    dtris_set( 0y00001111b )
    REP
        dport_set( scan_row )
        delay( 1 )
        IF b_and( dport_get(), scan_col )
            ; no key is down go to next scan
            ++( scan_number )
            IF ==( scan_col, 0y00001000b )
                =( scan_col, 0y00000001b )
                IF ==( scan_row, 0y11110111b )
                    =( scan_row, 0y11111110b )
                ELSE
                    =( scan_row, <<( scan_row ))
                    ++( scan_row )
                ENDIF
            ELSE
                =( scan_col, <<( scan_col ))
            ENDIF
        ELSE
            ; key is pressed
            con_out( scan_number )
            REP
                delay( 10 )
            UNTIL b_and( dport_get(), scan_col )
        ENDIF
    LOOP
ENDFUN
```

There are only a few tricks to key scanning. The first is to allow time between when you write the row scan out and when you read the scan result in. The second is to make sure that all keys are released after a key press is detected, before you detect the next key press. If you do not do this, multiple keys depressed accidentally can lead to completely wrong interpretations about key presses. If you need multiple keys to be pressed simultaneously, like a shift or "alt" key, put all those keys on a seperate row. You may even wish to put diodes on these keys.

This key scanning circuit also uses a few CMOS logic IC (integrated circuit). This is to illustrate the use of such circuits and how they can save microcontroller I/O. This circuit can scan up to 64 SPST normally open switches, and uses only 7 I/O lines.



```
DEF tic62_c
LIB fbasic.lib

GLOBAL byte key_value
GLOBAL byte ascii_value ob

FUNC byte key_lookup
    PARAM byte key_in
BEGIN
    =( exit_value, '?' )
    IF <( key_in, 10b )
        =( exit_value, +( key_in, '0' ))
    ELSE
        IF <( key_in, 36b )
            =( exit_value, +( -( key_in, 10b ), 'A' ))
        ENDIF
    ENDIF
ENDFUN
```

```
FUNC none main
BEGIN
    dtris_set( 0y11000000b )
    rs_param_set( debug_pin )
    REP
        =( key_value, 0b )
        =( ascii_value, 0b )
        REP
            dport_set( key_value )
            delay( 1 )
            IF pin_in( pin_d7 )
                IF ==( ascii_value, 0b )
                    =( ascii_value, key_lookup( key_value ))
                    con_out_char( ascii_value )
                ENDIF

                delay( 10 )
            ELSE
                ++( key_value )
            ENDIF

        UNTIL ==( key_value, 64b )
    LOOP
ENDFUN
```

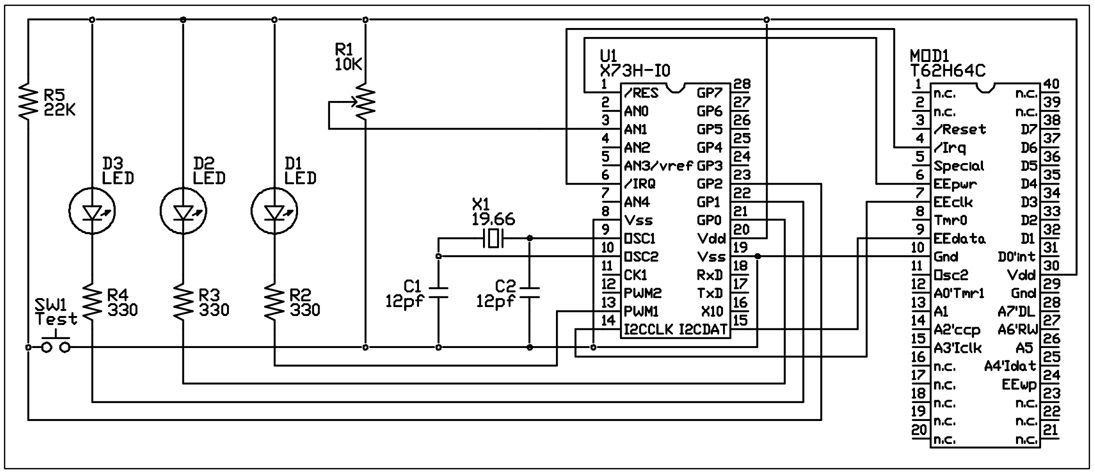
The program above is elementary, but shows how to get from key scan numbers to ASCII output.

4.7 Using Protean's I2C Xtender IC for more resources

A common problem encountered when designing controller applications based on single chip controllers is the lack of I/O or other controller resources. To meet this demand for additional capabilities, a trend has developed toward serially connected peripheral ICs. One example of this is Protean's I2C Xtender IC This device is a specially programmed IC that responds to commands over its Inter-Integrated Circuit (IIC or I2C) bus. This bus connects to a host processor using only two wires. If a TICkit is the host, the connection can use the EEprom bus wires leaving all 16 of the TICkits general purpose I/O lines available. Up to 8 or more Xtenders can be connected to a single host via these two lines.

A single Xtender IC gives the system the following hardware resources: 2 CCP I/O pins, a 32 bit real-time seconds counter, 3 time bases, a unipolar stepper motor controller, 128 bytes of RAM, a buffered RS232 port, a 16 bit counter, 4 100-Hz PWM outputs, and 5 8-bit A/D channels.

A sample connection to an Xtender is shown below. The I2Cclk and I2Cdata lines form the logical connection. In addition to these lines, the /IRQ line, and /RES of the Xtender are connected to the /IRQ, and EEpwr of TICkit respectively. A sampling of I/O components are connected to the Xtender in the diagram to demonstrate the A/D, PWM1 (CCP1), button input, general purpose output, 100 Hz PWM and time base outputs.



In the program that follows, every press of SW1 causes a read from the RTC seconds count and a read of A/D channel 1. You can vary the input to the A/D channel by changing the position of R1. To make the operation of the A/D clearer, you can use a multi-turn version of resistor R1. LED D3 follows the status of SW1 except that it inverted to demonstrate TICkit processing. LED D2 shows the effects of the 100 Hz PWM. LED D1 shows the effects of the PWM1 output which is a hardware generated, higher frequency pwm. LED D1 and D2 bright and dim out of phase with each other so that while one gets brighter, the other gets dimmer.

There are many more things that the Xtender can do and programming the Xtender is a subject in and of itself, but this example shows how simple register write and reads accomplish control of the Xtender. Communication with the Xtender takes place at the same speed as communication with the EEprom on the TICkit (400 Kbps) so it takes commands very quickly.

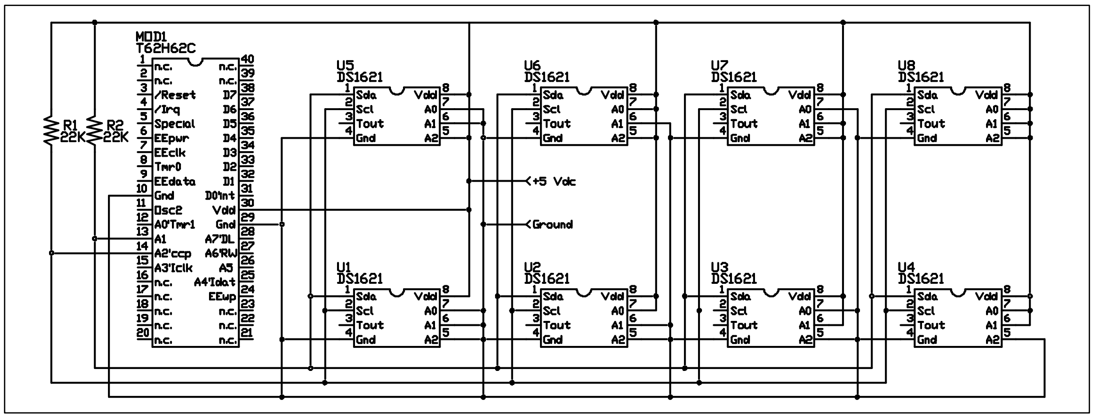
Commands for the Xtender are formed from constants contained in the Xtender's library. Use the '|' vertical bar character to combine the device number with the specific command. This method keeps the code very clean and readable.

```
DEF tic62_c
LIB fbasic.lib

LIB xtn73h.lib

GLOBAL byte duty_temp 0b      ; duty cycle for D1 and D2
GLOBAL byte button_last 0b    ; last status of button
GLOBAL long time_temp           ; used to build up the seconds count
GLOBAL byte temp_val          ; temporary value for reading/writing
```

```
FUNC none main
BEGIN
    delay( 500 )         ; let Xtender get out of power up reset
    ; initialize the Xtender
    IF ==( i2c_read( xtn_dev0 | xtn_reset ), 8b )
        ; this is a version H Xtender
    ENDIF

    i2c_write( xtn_dev0 | xtn_pins_out, 0y00000011b )
    i2c_write( xtn_dev0 | xtn_pins_in, 0y00000100b )
    i2c_write( xtn_dev0 | xtn_gp_cont, xtn_pwme_0 )
    i2c_write( xtn_dev0 | xtn_ad_con, xtn_ad_pwr | xtn_ad_chan1 )
    i2c_write( xtn_dev0 | xtn_tmr2_con, xtn_tmr2_en )
    i2c_write( xtn_dev0 | xtn_tmr2_per, 255b )
    i2c_write( xtn_dev0 | xtn_ccp1_con, xtn_ccp1_pwm )

    REP
        IF b_and( i2c_read( xtn_dev0 | xtn_pins ), 0y00000100b )
            =( button_last, 0b )    ; button is not pressed
            i2c_write( i2c_dev0 | xtn_pins_low( 0y00000010b )
        ELSE
            ; button is pressed
            i2c_write( i2c_dev0 | xtn_pins_high( 0y00000010b )
            IF button_last
                ; get real time seconds count and A/D value
                =( temp_val, i2c_read( xtn_dev0 | xtn_clk_tic ))
                ; above captures 32 bit count
                =( temp_val, i2c_read( xtn_dev0 | xtn_clk_cnt3 ))
                =( time_temp, to_long( temp_val ))
                =( temp_val, i2c_read( xtn_dev0 | xnt_clk_cnt2 ))
                =( time_temp, +( *( time_temp, 256b ), temp_val ))
                =( temp_val, i2c_read( xtn_dev0 | xnt_clk_cnt1 ))
                =( time_temp, +( *( time_temp, 256b ), temp_val ))
                =( temp_val, i2c_read( xtn_dev0 | xnt_clk_cnt0 ))
                =( time_temp, +( *( time_temp, 256b ), temp_val ))
                =( temp_val, i2c_read( xtn_dev0 | xtn_ad_reg ))
                con_string( "Time at press: " )
                con_out( time_temp )
                con_string( "  Analog Level: " )
                con_out( temp_val )
                con_string( "\r\l" )
            ENDIF
            =( button_last, 255b )
        ENDIF

        delay( 10 )
        ; deal with the PWMs
        i2c_write( xtn_dev0 | xtn_ccp1_low, duty_temp )
        i2c_write( xtn_dev0 | xtn_ccp1_high, 0b )
        i2c_write( xtn_dev0 | xtn_pwm_0, com( duty_temp ))
        ++( duty_temp )
    LOOP
ENDFUN
```

4.8 Connecting with Other Resources via I2C

The previous example demonstrated how a TICkit can communicate to an Xtender IC via the TICkit's built in I2C interface. There are many manufacturers of I2C compatible products and not all of them use protocols which are compatible with the TICkit's built in read and write formats. This example deals with one such part. This example connects a TICkit to 8 Dallas DS1621 temperature sensing IC via two TICkit general purpose I/O pins controlled by a TICkit I2C simulating library. This simulated I2C is not nearly as fast as the protocols built into the TICkit, but it will accomplish the communications required fairly quickly, certainly as fast as required by most applications. This program actually implements a complete on-demand temperature acquisition system. A partial schematic for this circuit is shown below. An actual application would probably have additional circuitry connected to the DS1621 ICs.



Notice that the two lines used for the I2C bus are pulled high. This is required by the I2C protocol because multiple sources can drive both the clock and data lines. Also notice how the pins A0, A1, and A2 on each DS1621 are strapped for a different address. This is how each IC knows which I2C address to respond to. The 'A' pins allow the designer to specify 3 of the 7 I2C address pins. The other four are hard coded by the manufacturer of the IC. Some ICs internally specify all 7 address lines and must be ordered with different address (like the Xtender) if more than one will be used on an I2C bus.

The following program is relatively complex for an example. It shows how defines can be used in conjunction with a pre-written library to customize the library for the program. This was done with sim_i2c.lib file to specify which pins to use. We also use the DEF directive to define which pin to communicate with the PC or terminal. In this example, we can use the download socket on the TICkit module with the download cable, except that we use a terminal program like WINTERM instead of the download software. This just makes demonstrating easier. We leave it to the reader to examine the sim_i2c.lib file to see how this all works.

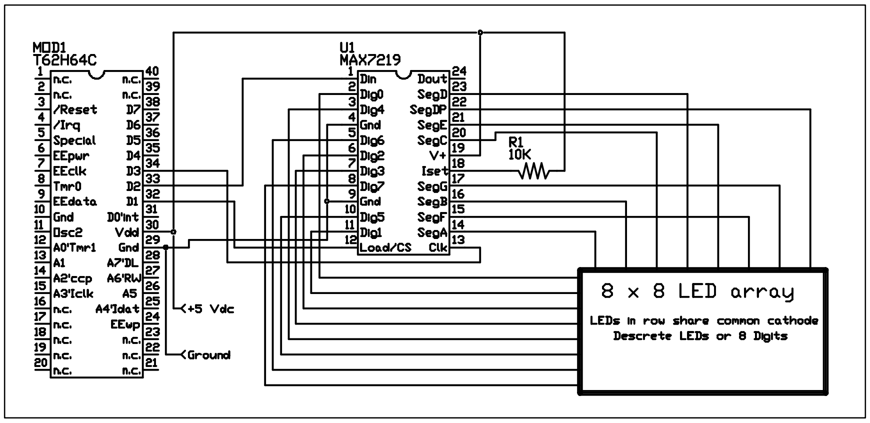
```
; Program to read the 1621 on command and return the value to a PC
; via a serial port. Simulated I2C routines using GP pins are
; used for this routine because the 1621's protocols are too
; complex for the i2c_read and i2c_write functions.

DEF tic62_a
LIB fbasic.lib

DEF si2c_data pin_a1      ; these are the pins to use for I2C
DEF si2c_clk pin_a2
LIB sim_i2c.lib           ; this libarary is un-documented in man.
                          ; but is contained on the release disk.

; The 1621 has three pins for strapping an I2C address.
; This means up to 8 1621 devices can exist on the I2C bus
; and be addressed independently.

; The defines below give the addresses for the 8 devices.
; If a read is to be performed, the lsb of the address must be set.

DEF DS1621_dev0    0xA0b
DEF DS1621_dev1    0xA2b
DEF DS1621_dev2    0xA4b
DEF DS1621_dev3    0xA6b
DEF DS1621_dev4    0xA8b
DEF DS1621_dev5    0xAAb
DEF DS1621_dev6    0xACb
DEF DS1621_dev7    0xAEb
```

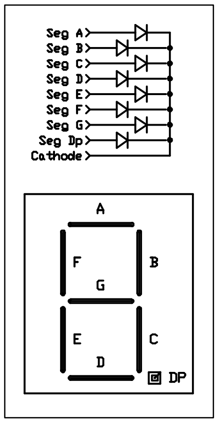
```
; The 1621 has a fairly elaborate command system.
; Following the typical I2C device address/control byte,
; an 8 bit command byte is used to inform the 1621 of the
; nature of the data transfer. The commands and associated
; data are listed below:

DEF DS1621_temp 0xAAb ; This command reads the temperature of the
                      ; last conversion. The 1621 will send 2
                      ; bytes(16 bits) unless a stop bit is sent
                      ; after the first byte. The second byte only
                      ; has information in bit 7 because the 1621
                      ; only converts 9 bits of data.

DEF DS1621_t_high  0xA1b ; This read/write 16bit register is used
                         ; to control the Tout pin of the 1621.
                         ; This is the high value used in
                         ; comparisons with actual temp reading.
                         ; If the actual temperature exceeds this
                         ; value, Tout is driven high.

DEF DS1621_t_low   0xA2b ; This read/write 16bit register is used
                         ; to control the Tout pin of the 1621.
                         ; This is the low value used in comparisons
                         ; with actual temp reading. If the actual
                         ; temperature is less than this value,
                         ; Tout is driven low. This creates a
                         ; hysterisis region to prevent critical
                         ; oscillations around a single temperature
                         ; setpoint.

DEF DS1621_config  0xACb ; This read/write 8 bit register configures
                         ; the 1621 for operation. The bits below
                         ; explain the config options.

DEF DS1621C_done   0y10000000b ; 1= conversion finished.
DEF DS1612C_thf    0y01000000b ; 1= The device has exceeded the
                               ; t_high value. This bit is only
                               ; reset by writing the config
                               ; register. This bit is unaffected
                               ; by the temp falling below TH or TL
DEF DS1621C_thl    0y00100000b ; 1= The device temp has fallen below
                               ; the t_low value. This bit is only
                               ; reset by writing 0 to it in the
                               ; config register. This bit is not
                               ; affected by the temp exceeding TL
                               ; or TH.
DEF DS1621C_NVB    0y00010000b ; 1= 1621 is busy writing data to the
                               ; EEprom. Values written to th or tl
                               ; are stored in non volatile memory
                               ; and writes can require up to 10 ms.
DEF DS1621C_pol    0y00000010b ; Output polarity for Tout. 1= a high
                               ; is Vdd.
DEF DS1621C_single 0y00000001b ; 1= do a single conversion when
                               ; start is commanded. 0= do
                               ; continuous conversion when
                               ; start is commanded.
DEF DS1621_count   0xA8b ; This 8bit register holds the count used
                         ; for temp conversion. This read only
                         ; register can be used for increased
                         ; precision (up to 16bit)
DEF DS1621_slope   0xA9b ; This 7bit register holds the slope count
                         ; used for temp conversion. This read only
                         ; register can be used with the count
                         ; register to calculate a more precise
                         ; temperature (up to 16 bit)
DEF DS1621_start   0xEEb ; A write to this register starts
                         ; conversions. The config register
                         ; determins if a single conversion
                         ; takes place or if continuous conversions
                         ; will follow. A bit of the config register
                         ; indicates when conversion is complete.
DEF DS1621_stop    0x22b ; A write to this register halts continuous
                         ; conversion mode. The current conversion
                         ; will finish then the 1621 will remain
                         ; idle until the next start command.

DEF pc_serial pin_a7 ; this is the pin to use to communicate to PC

GLOBAL byte rs_command   ; this is the command received from PC
GLOBAL byte err_val      ; error on rs receive
GLOBAL byte dev_addr     ; computed I2C address for 1621
GLOBAL byte config_read  ; value of configuration register
                         ; as read from the specified 1621
GLOBAL word temp_result  ; 16 bit result as read from 1621
ALIAS byte temp_high temp_result 1b  ; upper byte of result
ALIAS byte temp_low temp_result 0b   ; lower byte of result

GLOBAL byte trash        ; dummy variable when making pins inputs

FUNCTION none rs_word
   PARAM word rs_data
   LOCAL word place 10000w
   LOCAL word num
BEGIN
    =( num, rs_data )
    REPEAT
```

```
        rs_send( +( '0', trunc_byte( /( num, place ))))
        =( num, %( num, place ))
        =( place, /(place, 10b ))
    UNTIL ==( place, 1b )

    rs_send( +( '0', trunc_byte( num )))
ENDFUN

FUNC none si2c_comm  ; function to start message and send command
    PARAM byte addr
    PARAM byte comm
    LOCAL byte trash
BEGIN
    REP
        si2c_start()
        IF si2c_wbyte( addr )
            STOP
        ENDIF

        si2c_stop()
    LOOP

    =( trash, si2c_wbyte( comm ))
ENDFUN

FUNC none main
BEGIN
    rs_param_set( rs_invert | rs_9600 | pc_serial )
    pin_high( si2c_data )
    pin_high( si2c_clk )

    ; configure 1621 for single conversion on command
    si2c_comm( ds1621_dev0,  ds1621_config )
    =( trash, si2c_wbyte( ds1621c_single ))
    si2c_stop()

    si2c_comm( ds1621_dev1,  ds1621_config )
    =( trash, si2c_wbyte( ds1621c_single ))
    si2c_stop()

    si2c_comm( ds1621_dev2,  ds1621_config )
    =( trash, si2c_wbyte( ds1621c_single ))
    si2c_stop()

    si2c_comm( ds1621_dev3,  ds1621_config )
    =( trash, si2c_wbyte( ds1621c_single ))
    si2c_stop()
    si2c_comm( ds1621_dev4,  ds1621_config )
    =( trash, si2c_wbyte( ds1621c_single ))
    si2c_stop()

    si2c_comm( ds1621_dev5,  ds1621_config )
    =( trash, si2c_wbyte( ds1621c_single ))
    si2c_stop()

    si2c_comm( ds1621_dev6,  ds1621_config )
    =( trash, si2c_wbyte( ds1621c_single ))
    si2c_stop()

    si2c_comm( ds1621_dev7,  ds1621_config )
    =( trash, si2c_wbyte( ds1621c_single ))
    si2c_stop()

    REP
        ; wait for a command from PC ( ignore bogus values )
        =( rs_command, rs_receive( 0b, 0b, err_val ))
        IF err_val
        ELSE
            IF and( >=( rs_command, 'A' ), <=( rs_command, 'H' ))
                ; valid command calc I2c address and get readings
                =( dev_addr, +( 0xA0b, *( 2b, -( rs_command, 'A' ))))
                si2c_comm( dev_addr,  ds1621_start )

                ; repeatedly read config until conversion is done
                REP
                    si2c_comm( dev_addr,  ds1621_config )
                    si2c_stop()
                    si2c_start()
                    =( trash, si2c_wbyte( b_or( dev_addr, ~
                      ~ 0y00000001b )))
                    =( config_read, si2c_rbyte( 0b ))
                    si2c_stop()
                UNTIL b_and( config_read, ds1621c_done )

                ; now read the conversion results
                si2c_comm( dev_addr,  ds1621_temp )
                si2c_stop()
                si2c_start()
                =( trash, si2c_wbyte( b_or( dev_addr, 0y00000001b )))
                =( temp_high, si2c_rbyte( 0xffb ))
                =( temp_low, si2c_rbyte( 0b ))
                si2c_stop()

                ; now compute result into a normalized 16 bit number
                ; and send result to PC with a return at the end.
                =( temp_result, /( temp_result, 128b ))
                rs_word( temp_result )
                rs_send( '\r' )
            ENDIF
        ENDIF
    LOOP
ENDFUN
```

4.9 Using a 3-wire interface to control tons of LEDs

In the last examples, we used the I2C bus to communicate to peripheral ICs. The I2C bus is sometimes called the 2-wire bus. In this example we will use a 3-wire bus, another serial standard, to communicate with a MAXIM IC designed for driving multiplexed numeric LED displays. The IC is the MAX72198-Digit LED Display Driver. This IC drives a matrix of LEDs so that 256 individual LEDs can be driven from a single 24 pin IC. The magic of this technique is called multiplexing (time multiplexing to be exact). This means that at any given point of time, only 8 LEDs are being driven, but each 8 LEDs is driven in quick succession over time. Our eyes interpret this blur as the desired pattern; all LEDs which received any drive appear to be on continuously. This is similar to our first example where two LEDs blinked alternately, when there was no delay in the loop, both LEDs appeared to be on continuously. This IC is called a digit driver because 7 segment LED digits contain 8 LEDs (7 segments and a decimal point) that share a common cathode or anode. By connecting all the same segments together and calling them rows, and using each of the 8 digits common cathodes at columns, an 8 x 8 matrix of diodes is created. If you just want to control LEDs and not digits, you can electrically arrange your diodes in groups of 8 that share a common cathode. This has been done often for Christmas displays. The circuit for this example is shown below. There are no real surprises here, multiple 7219s can be daisy chained for more LED drivers yet. The program simply lowers the "load /CS" line, shifts 16 bits of data into the Din pin using the clk pin, and the communication is complete. A resistor is used with the Iset pin to Vdd. This sets the maximum drive for any segment





The program for interfacing to the 7219 is also elementary. Each communication sends 16 bits which is comprised of 8 data bits and 4 bits of register address. The remaining 4 bits are unused. The 16 bits are sequentially shifted out the D2 pin and clocked into the 7219 using the D3 pin. The 16th bit shifts out first and each bit is latched in on the rising edge of clk. A subroutine takes care of shifting out the 16 bits. DEF statements define constants used to refer to each of the registers in the 7219. The program simply lights every LED in each row then every LED in each column in succession.

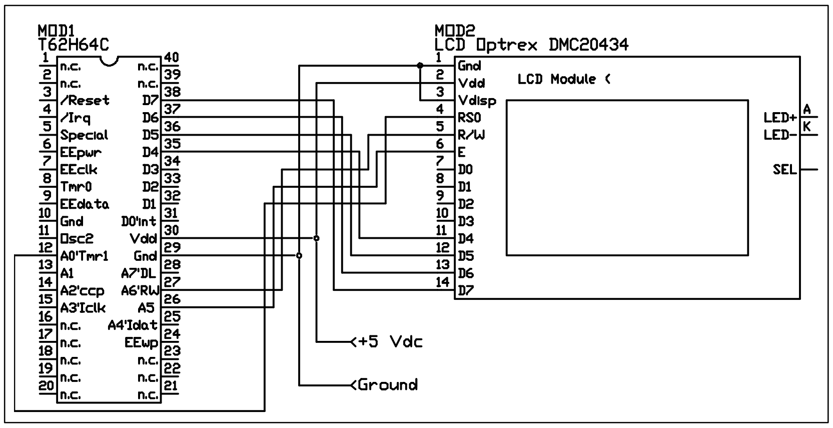
```
DEF tic62_c
LIB fbasic.lib

DEF max7219_data pin_d2
DEF max7219_clk pin_d1
DEF max7219_load pin_d3

DEF max7219_dig0 0x0100w
DEF max7219_dig1 0x0200w
DEF max7219_dig2 0x0300w
DEF max7219_dig3 0x0400w
DEF max7219_dig4 0x0500w
DEF max7219_dig5 0x0600w
DEF max7219_dig6 0x0700w
DEF max7219_dig7 0x0800w
DEF max7219_decode 0x0900w
DEF max7219_intens 0x0A00w
DEF max7219_limit 0x0B00w
DEF max7219_shutdn 0x0C00w
DEF max7219_test 0x0F00w

GLOBAL word cur_row
GLOBAL byte cur_col
```

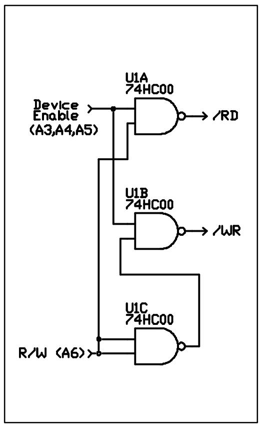
```
FUNC none max7219_send
    PARAM word max_comm
    PARAM byte max_data
    LOCAL word max_result
    LOCAL byte bit_counter 16b
BEGIN
    =( max_result, +( max_comm, max_data ))
    pin_low( max7219_clk )
    pin_low( max7219_load )
    REP
        IF b_and( max_result, 0x8000w )
            pin_high( max7219_data )
        ELSE
            pin_low( max7219_data )
        ENDIF

        pin_high( max7219_clk )
        --( bit_counter )
        pin_low( max7219_clk
    UNTIL ==( bit_counter, 0b )

    pin_high( max7219_load )
ENDFUN
```

```
FUNC none main
BEGIN
    ; start by initializing the display
    max7219_send( max7219_decode, 0y00000000b ) ; numeric decode
    max7219_send( max7219_intens, 0y00001111b ) ; full brightness
    max7219_send( max7219_limit, 0y00000111b ) ; all rows (digits) on
    max7219_send( max7219_shutdn, 0y00000001b ) ; normal operation
    max7219_send( max7219_test, 0y00000001b ) ; test in progress
    delay( 500 ) ; one half second of LED test
    max7219_send( max7219_test, 0y00000000b ) ; no test in progress

    REP
        ; test rows independently
        =( cur_row, max7219_dig0 )
        REP
            max7219_send( cur_row, 0y11111111b ) ; All 8 LEDs on
            delay( 250 )                         ; wait 1/4 second
            max7219_send( cur_row, 0y00000000b ) ; all 8 LEDs off
            =( cur_row, +( cur_row, 0x0100w ))   ; next row
        UNTIL ==( cur_row, max7219_dig7 )

        ; test columns independently
        =( cur_col, 0y00000001b )
        REP
            max7219_send( max7219_dig0, cur_col )
            max7219_send( max7219_dig1, cur_col )
            max7219_send( max7219_dig2, cur_col )
            max7219_send( max7219_dig3, cur_col )
            max7219_send( max7219_dig4, cur_col )
            max7219_send( max7219_dig5, cur_col )
            max7219_send( max7219_dig6, cur_col )
            max7219_send( max7219_dig7, cur_col ) ; turn of col LEDs
            delay( 250 )
            max7219_send( max7219_dig0, 0b )  ; turn off col LEDs
            max7219_send( max7219_dig1, 0b )
            max7219_send( max7219_dig2, 0b )
            max7219_send( max7219_dig3, 0b )
            max7219_send( max7219_dig4, 0b )
            max7219_send( max7219_dig5, 0b )
            max7219_send( max7219_dig6, 0b )
            max7219_send( max7219_dig7, 0b )

            =( cur_col, <( cur_col ))
        UNTIL ==( cur_col, 0b )
    LOOP
ENDFUN
```

4.10 Using the Bus Routines to Control an LCD module

The example for controlling LEDs is similar to this example in that the goal is to display visual information to the user of the application. In this example we are connecting the TICkit 62 to an LCD module based on the popular Hitachi 44780 chip set. Most of the LCD alpha-numeric displays on the market use this chip set and it has become the dominant defacto standard for displays up to 4 lines by 40 characters. These modules are usually available for $10 or less from surplus outlets like B.G. Micro.





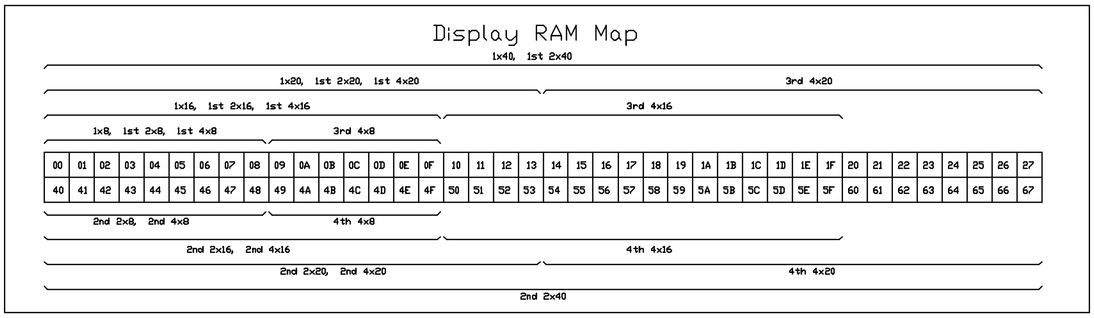
These modules are available with special electronics made by Scott Edwards Electronics and others which allow them to be interfaced serially. This example does not use any additional electronics and interfaces to the TICkit via a 4 bit serial bus connection. The direct connection is a more versatile in that it allows reading of the modules memory as well as writing to it. This is handy for scrolling and other effects. These modules can be connected by either a 4 bit or an 8 bit bus, but to conserve I/O we sacrifice some speed of bus transfer to retain pins D0 through D3 for other uses. The circuit for this example is shown above.

The key to this type of module is understanding its internal archetecture and command format. We will discuss that next, but first we need to discuss the TICkit bus emulation routines. There are three routines available in the TICkit 62 for using the general purpose pins in a bus simulation. As you might imagine, the D-pins are used for the data lines of the bus, and the A pins are used for address lines. The function buss_setup() is used to tell the TICkit which of the address lines will be used for bus functions and which are free for general purpose use. The buss_setup() function also specifies which data lines are used, either all 8 pins or only the top 4 are used by the bus simulation. An additional option is available, if only 4 bits of data lines are used. That option is a single or double nibble. The LCD module can use a 4 bit double nibble bus method instead of an 8 bit bus to transfer 8 bits worth of data.

The bus simulation can only use address lines 0 through 5 for actual address lines, pin A6 is used as a Read/Write control line and pin A7 is reserved for download purposes. There is also a subtle difference between pins A0-A2 and pins A3-A5. When the bus is idle or between operations, all the address lines go to zero. To prevent false writes or reads to multiple registers in a single device, pins

A3-A5 go to zeros first. This causes any selected device to become unenabled, then pins A0-A2 can change with no effect. This method of bus interface is similar to Rockwell's or Motorola's method. It means that there is a device enable and reading/writing is controlled by a single read/write line. The other common type of businterface is the Intel or Zilog method where a seperate line is used for /write and for /read. These pins can be derived from the select logic and the R/W pin to generate the desired signals. A logic diagram for this is shown above with this example's schematic.

Now lets take a closer look at the LCD module'selectronics and archetecture. Whether the display is 2 x 40 or 4 x 16 or any line-column configuration in between, the internals of the module are the same. The display memory is organized as one line at addresses x00 to x27 and the second line at 0x40 to 0x67. If the display has 4 lines, the display memory for the first line is split between the first and third line, while the display memory for the second line is split between the second and forth line. This makes writing specific display positions and scrolling the display arcane, but it is doable none the less.


Display RAM Map
1x40, 1st 2x40
1x20, 1st 2x20, 1st 4x20
3rd 4x20
1x16, 1st 2x16, 1st 4x16
3rd 4x16
1x8, 1st 2x8, 1st 4x8
3rd 4x8
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27
40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64 65 66 67
2nd 2x8, 2nd 4x8
4th 4x8
2nd 2x16, 2nd 4x16
4th 4x16
2nd 2x20, 2nd 4x20
4th 4x20
2nd 2x40

In addition to the 160 bytes of display data RAM (DD RAM), there are 64 bytes of user programmable character generator RAM. If the display is programmed for 5x7 characters, this comes out to 8 custom characters. If the display is programmed for 5x10 characters, there are only 4 custom characters available. The table that follows shows the mapping of the character generator RAM (CG RAM) when in 5x7 character mode.

Only the pattern for two of the eight possible custom characters is shown, but the method should be clear from these two examples. The two characters are programed for an 'H' and the small letter 'g' to demonstrate a descender.

Character Codes (as in DD RAM) 7 6 5 4 3 2 1 0	CG RAM addresses (for programming) 7 6 5 4 3 2 1 0	Character Pattern (as in CG RAM) 7 6 5 4 3 2 1 0
0 0 0 0 * 0 0 0	0 0 0 0 0 0 0 0	X X X **1** 0 0 0 **1**
	0 0 0 0 0 0 0 1	X X X **1** 0 0 0 **1**
	0 0 0 0 0 0 1 0	X X X **1** 0 0 0 **1**
	0 0 0 0 0 0 1 1	X X X **1 1 1 1 1**

	0 0 0 0 0 1 0 0	X X X **1** 0 0 0 **1**
	0 0 0 0 0 1 0 1	X X X **1** 0 0 0 **1**
	0 0 0 0 0 1 1 0	X X X **1** 0 0 0 **1**
	0 0 0 0 0 1 1 1	X X X 0 0 0 0 0
0 0 0 0 * 0 0 1	0 0 0 0 1 0 0 0	X X X 0 0 0 0 0
	0 0 0 0 1 0 0 1	X X X 0 0 0 0 0
	0 0 0 0 1 0 1 0	X X X 0 **1 1 1 1**
	0 0 0 0 1 0 1 1	X X X **1** 0 0 0 **1**
	0 0 0 0 1 1 0 0	X X X **1** 0 0 0 **1**
	0 0 0 0 1 1 0 1	X X X 0 **1 1 1 1**
	0 0 0 0 1 1 1 0	X X X 0 0 0 0 **1**
	0 0 0 0 1 1 1 1	X X X 0 **1 1 1** 0

There are two addressable registers in a 44780 based module. These are the command register and data register. As you would expect, the data register is used to read or write data to the DDRAM or CGRAM. The control register is less obvious. Reads from the control register return a busy flag in bit 7 and the current address counter (DDRAM pointer) in bits 0 thru 6. The table that follows summarizes the command structure:

Instruction name	Control RS R/W	Data Bits 7 6 5 4 3 2 1 0	Description
Clear Display	0 0	0 0 0 0 0 0 0 1	Clears display and returns cursor to home position (address 00)
Return Home	0 0	0 0 0 0 0 0 1 X	Places cursor at address 00. Also un-shifts display
Entry Mode	0 0	0 0 0 0 0 1 I S	Sets the cursor movement direction. I=1 inc, I=0 dec, S=0 no shift, S=1 shift display.
Display Control	0 0	0 0 0 0 1 D C B	Turn Display on (D), Turn Cursor on (C), Blink Cursor on (B).

Cursor & Display Shifting	0 0	0 0 0 1 D R X X	Controls shifting and cursor movement. D=1 shift display, D=0 cursor move, R=1 shift right, R=0 shift left.
Interface & Format	0 0	0 0 1 D L F X X	Controls data bus width and display format. D=1 8 bit bus, D=0 4 bit bus, double lines (L), Large Font (F)
Set CG RAM Address	0 0	0 1 A A A A A A	Sets the address for CG RAM reading and Writing. Subsequent read and writes to data register affect CG RAM contents.
Set DD RAM Address	0 0	1 A A A A A A A	Sets the address for DD RAM reading and Writing. Subsequent read and writes to data register affect DD RAM contents.
Read Status	0 1	B A A A A A A A	Reads 44780 status. B=busy processing, AAAAAAA = address count; either DD or CG RAM address.
Write Data	1 0	Data to Write	Either DD or CG data
Read Data	1 1	Data Read	Either DD or CG data

The program which follows follows a specific sequence of commands to initialize the display. A specific command write timing pattern is necessary to ensure the display initializes properly.

```
DEF tic62_c
LIB fbasic.lib

DEF xbuss_mask 0y00100001b   ; These are the address lines used
DEF lcd_data_reg 0y00100001b ; Address of data register
DEF lcd_cont_reg 0y00100000b ; Address of control register
```

```
FUNC none lcd_init
BEGIN
    buss_setup( +( xbuss_mask, buss_4bit ))  ; setup buss for 4bit
    delay( 15 )                              ; wait 15ms
    buss_write( lcd_cont_reg, 0y00110000b )
    delay( 5 )
    buss_write( lcd_cont_reg, 0y00110000b )
    delay( 1 )
    buss_write( lcd_cont_reg, 0y00110000b )
    lcd_cont( 0y00100000b )             ; turn it into 4two

    buss_setup( +( xbuss_mask, buss_4two ))
    lcd_cont( 0y00101000b )             ; assumes 2 line 5x7 font
    lcd_cont( 0y00001111b )
    lcd_cont( 0y00000001b )
    lcd_cont( 0y00000110b )
ENDFUN


FUNC none lcd_cont_wr
    PARAM byte in_val
BEGIN
    WHILE >=( buss_read( lcd_cont_reg ), 0y10000000b )
    LOOP         ; delay until not busy

    buss_write( lcd_cont_reg, in_val )
ENDFUN

FUNC none lcd_data_wr
    PARAM byte in_val
BEGIN

    WHILE >=( buss_read( lcd_cont_reg ), 0y10000000b )
    LOOP         ; delay until not busy

    buss_write( lcd_data_reg, in_val )
ENDFUN
```

```
FUNC none lcd_string
    PARAM word in_ptr
    LOCAL word temp_ptr
    LOCAL byte temp_val
BEGIN
    =( temp_ptr, in_ptr )    ; don't affect calling value
    =( temp_val, ee_read( temp_ptr ))
    WHILE temp_val
        lcd_data_wr( temp_val )
        ++( temp_ptr )
        =( temp_val, ee_read( temp_ptr ))
    LOOP
ENDFUN

FUNC none lcd_out
    PARAM word lcd_data
    LOCAL word place 10000w
    LOCAL word num
BEGIN
    =( num, lcd_data )
    REPEAT
       lcd_data_wr( +( '0', trunc_byte( /( num, place ))))
       =( num, %( num, place ))
       =( place, /( place, 10b ))
   UNTIL ==( place, 0b)
ENDFUN

FUNC none main
    LOCAL word lcount 0
BEGIN
    delay( 500 )       ; wait for 1/2 second for power to settle
    lcd_init()
    lcd_cont_wr( 0y00000001b )   ; Reset the LCD for good measure
    lcd_string( "Hello World..." )
    lcd_cont_wr( 0y11000000b )   ; position to first char on line 2
    lcd_string( "Loop Count: " )
    REP
        lcd_cont_wr( 0y11001100b ) ; 12th char on line 2
        lcd_out( lcount )
        ++( lcount )
    LOOP
ENDFUN
```

This program just initializes the display, says "Hello World..." and counts loops on the second line of the display. A simple program, but a good basis for working with these very versatile LCD modules. There are additional LCD functions contained on the release disk for you to look over.

4.11 Fixed Point Arithmetic.

It is fairly common to deal with fractional results when designing controllers. This is a display or calculation restraint because the units of measure are directly dictated by the sensors in the controller. A controller with an LCD screen or other, more elaborate output capabilities should be able to present data in an expected format. The TICkit 62 does not have a floating point library in its current version, but it still can display numbers with fractional components.

The TICkit 62 has a signed LONG type number which is a 32 bit integer variable type. This size of integer can display 9 digits of accuracy.Assume we are dealing with numbers from our sensors which are no greater than 4096 (12 bit). This number only requires 4 digits to represent its full range. If we use a LONG type to represent this number during calculation or display, we can scale the meaning of this number by as much as 100000. In other words, we define one as being 100000 and we display our numbers with a decimal point 5 places to the left. The number one will display as 1.00000 which is exactly what you expect.

The key to making this easy is the format versions of the long numeric output functions. There are three versions of this on the TICkit release disk. Function lcd_fmt() is for displaying formatted longs on an LCD, function con_fmt() is for displaying formatted longs on the debug console, and function rs_fmt() is for displaying formatted longs to an rs232 device. To make clear how these functions work and how they are used, a copy of the con_fmt function is shown below. The meanings of the format characters are explained in the source for the function.

```
;Routine to output a Long Number to the LCD display (Signed)

; Meanings of format string characters
; '$' print a $
; '.' print a .
; '#' print a number ( leading zeros will not be printed )
; '0' print a number ( leading zeros will print from this digit on )
; 'X' hold a place but to not print the number
```

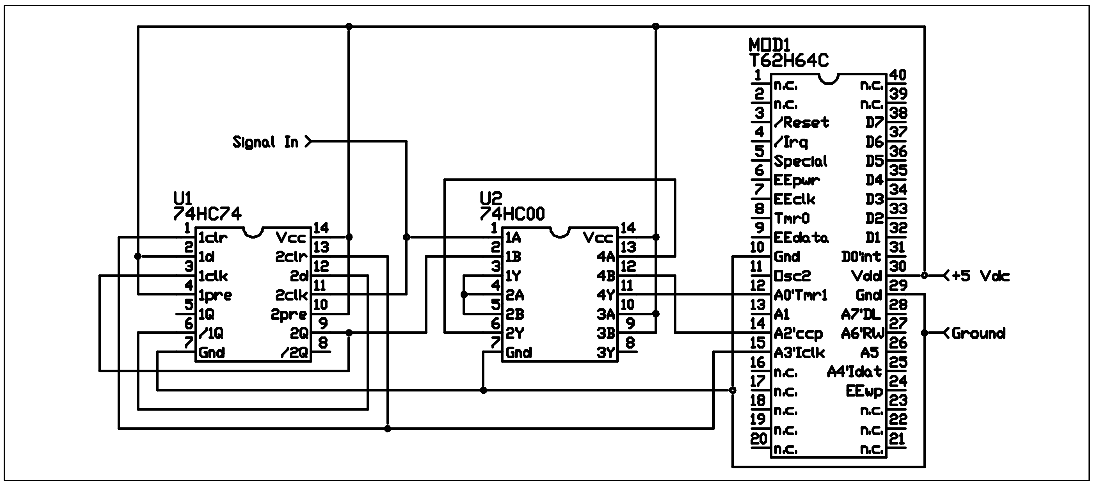
```
FUNCTION none con_fmt
    PARAM long in_data    ; Data to print
    PARAM word pointer     ; Data format string
    LOCAL long tempnum     ; Copy of print data
    LOCAL word hpointer    ; Copy of string pointer
    LOCAL long divisor 1l  ; Divisor , used by routine
    LOCAL byte tempchr      ; Data hold variable
    LOCAL byte first 0b    ; flags register
    LOCAL byte tempdig     ; temporary digit
BEGIN
    ; this section counts the number of digits and determines what
    ; the most significant digit's divisor will be as a result.
    =( tempnum, in_data )
    =( hpointer, pointer ) ; Store format string start
    =( tempchr, ee_read( hpointer ))    ; Read format string
    WHILE tempchr
        IF ==( tempchr, '#' )
            =( divisor, *( divisor, 10b ))
        ELSEIF ==( tempchr, '0' )
            =( divisor, *( divisor, 10b ))
        ELSEIF ==( tempchr, 'X' )
            =( divisor, *( divisor, 10b ))
        ENDIF

        ++( hpointer )
        =( tempchr, ee_read( hpointer ))    ; Read format string
    LOOP

    ; Check for negative: displays sign and
    ; make number positive for conversion
    IF <( tempnum, 0b )
        con_out_char( '-' )
        =( tempnum, -( 0l, tempnum ))
    ENDIF

    ; Check for overflow of number: If divisor too large,
    ; write an * to indicate
    ; Then do conversion on remaining modulus of divisor
    IF >( /( tempnum, divisor ), 0b )
        con_out_char(  '*' )
        =( tempnum, %( tempnum, divisor ))  ;
    ENDIF
    ; Begin actual conversion and display loop here
    =( divisor, /( divisor, 10b ))
    =( hpointer, pointer )        ; Store format string start
    =( tempchr, ee_read( hpointer ))
    WHILE >=( divisor, 1b )
        IF ==( tempchr, '.' )
            con_out_char( tempchr )
        ELSEIF ==( tempchr, '$' )
            con_out_char( tempchr )
        ELSEIF ==( tempchr, 'X' )
            =( tempnum, %( tempnum, divisor ))
            =( divisor, /( divisor, 10b ))
        ELSEIF ==( tempchr, '0' )
            =( tempdig, trunc_byte( /( tempnum, divisor )))
            =( tempdig, +( tempdig, '0' ))
            =( first, 0xffb )
            con_out_char( tempdig )
            =( tempnum, %( tempnum, divisor ))
            =( divisor, /( divisor, 10b ))
        ELSEIF ==( tempchr, '#' )
            =( tempdig, trunc_byte( /( tempnum, divisor )))
            =( tempdig, +( tempdig, '0' ))
            IF <>( tempdig, '0' )
                =( first, 0xffb )
            ENDIF

            IF first
                con_out_char( tempdig )
            ENDIF

            =( tempnum, %( tempnum, divisor ))
            =( divisor, /( divisor, 10b ))
        ELSE
            con_out_char( tempchr )
        ENDIF

        ++( hpointer )
        =( tempchr, ee_read( hpointer ))
    LOOP
ENDFUN
```

A typical application for this sort of thing would be to display the output of a 12 bit ratiometric A/D reading in volts. Rather than show the whole program, only a fragment which relates to this discussion is shown. The variable ad_in is a word variable that contains the value read for an LTC1298 12 bit A/D in a 5 volt system. This means that 0 is 0 volts and 4095 is 5 volts and all values in between are assumed to be linearly related.

```
    ; 12 bits can display three decimal points but needs 4 to
    ; completely hold the number. Therefore lets assign one to
    ; be the value 10000. To produce the number of volts from
    ; the value read we need to divide 5 times the reading by
    ; 4096.

    GLOBAL long conv_result

    =( conv_result, *( 50000, ad_in ))
    =( conv_result, /( conv_result, 4096 ))
    con_fmt( conv_result, "#.000X" )
    con_string( " Volts" )
```

4.12 Using the CCP Input to Measure a Pulse.

The TICkit has a pulse_in function which works very well for measuring pulses provided you know when they are coming. The TICkit does not need to be doing anything else while it waits for the pulse to occur. This generally is not the case in the real world. This next application demonstrates how the CCP output can be used in conjunction with timer1 and some discrete logic to make a very precise pulse measurement system that measures in background while the TICkit continues its other tasks. The CCP output is configured for PWM output like in previous examples. This time, however, we are not as interested in the duty cycle as the period. Lets say we are interested in pulses that are very fast and we want a resolution of 1 microsecond. The oscillator on a 20MHz TICkit produces a period of 0.2 micro seconds. This means we want to divide this by a factor of 5 to produce a period of 1.0 micro second. This is accomplished by loading the Timer2 period register with a value of 4. The CCP register is loaded with 2 (for a 50% duty cycle) and the output on the CCP pin will be 1 MHz or have a period of 0.1 us. We then gate this signal with an and gate and some trigger logic which feeds the Tmr1 pin (pin_a0). Now reset the trigger circuit and examine the contents of timer1 as soon as it remains constant at any value other than 0, we have measured a pulse. The contents of timer1 is the count of micoseconds the pulse was high. The circuit for this follows:

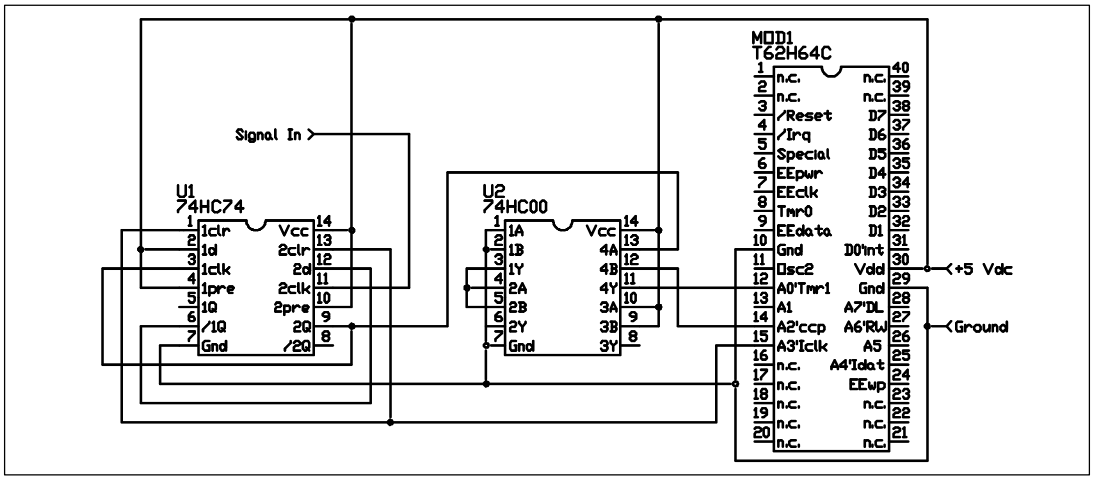


The circuit for gating the time base uses two flip flops(special logic components that hold their state until reset). The TICkit arms the circuit by bringing pin_a3 low and then high. The next rising edge on U1-2clk will turn U1-2 on and allow the time base to get through to the input of timer1. As soon as U1-2 turned on, U1-1 is clocked and turns on as well. Because the D input of U1-2 is connected to /Q of U1-1, the next pulse on the signal will turn U1-2 off permanently before any of the time base can be counted. So, at this point, the count in the TICkit's timer1 represents the amount of time that the signal was high. The program for this circuit follows:

```
DEF tic62_c
LIB fbasic.lib

GLOBAL word last_count 0
GLOBAL byte count_done 0b

FUNC none main
BEGIN
    rs_param_set( debug_pin )
    pin_low( pin_a2 )
    tmr2_cont_set( tmr2_con_on )
    tmr2_period_set( 4b )      ; set for a period of 1.0 us
    ccp1_cont_set( ccp_pwm )
    ccp1_reg_set( 2w )         ; set for approx 50% dutycycle

    ; time base is now operational
    pin_low( pin_a3 )          ; reset trigger circuit
    tmr1_reg_set( 0 )          ; clear timer1
    pin_high( pin_a3 )         ; arm the trigger circuit
    ; pulse measurement circuit is now active
    REP
        IF <>( last_count, 0 )
            IF ==( last_count, tmr1_reg_get())
                ++( count_done )
            ENDIF
        ENDIF

        ; do whatever during the body of the loop.
        ; timing is not critical.
        =( last_count, tmr1_reg_get())
    UNTIL count_done

    con_string( "Pulse Width = " )
    con_out( last_count )
    con_string( "us" )
    REP
        debug_on()
    LOOP
ENDFUN
```

4.13 Using Timer1 to calculate RPM.

Measurement of RPM or the time between repetitive events is simple. In the last example, the timebase could only be counted while the signal input was high and during the first cycle following the rising edge of that signal. By eliminating one of the AND gates, the time base is counted durring the entire first cycle following arming. By taking the reciperocal of the time, we have the RPM. Doing the reciperocal requires a bit of mathematic manipulation, but nothing too hard for the TICkit. First, the revised circuit:



The following program shows fixed point arithmetic used to scale the results for 1000 RPS (rotations per second). Realistically, we should slow our time base, but this shows how sensitive this method can be. We choose a scale where 1000000 represents the number 1.000000. When we divide 1 by the number of microseconds the result is the number of millions of events that took place in one second. Due to our scale, we can simply move our imagined decimal point to the right to see how many thousands of events took place per second.

Use the con_fmt() function detailed earlier to display the scaled results on the debug console. The result is shown in thousands of rotations per second with 2 decimal places of precision. Examine the code to see how this is done:

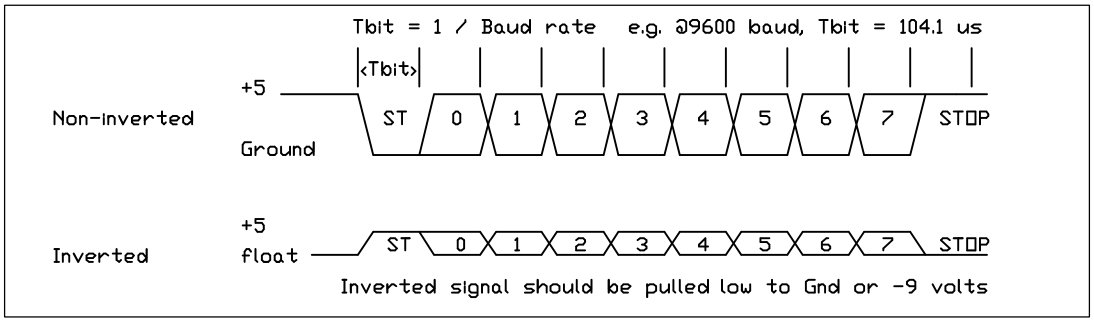
```
DEF tic62_c
LIB fbasic.lib

GLOBAL word last_count 0
GLOBAL byte count_done 0b

FUNC none main
BEGIN
    rs_param_set( debug_pin )
    pin_low( pin_a2 )
    tmr2_cont_set( tmr2_con_on )
    tmr2_period_set( 4b )     ; set for a period of 1.0 us
    ccp1_cont_set( ccp_pwm )
    ccp1_reg_set( 2w )        ; set for approx 50% dutycycle

    ; time base is now operational
    pin_low( pin_a3 )         ; reset trigger circuit
    tmr1_reg_set( 0 )         ; clear timer1
    pin_high( pin_a3 )        ; arm the trigger circuit

    ; pulse measurement circuit is now active
    REP
        IF <>( last_count, 0 )
            IF ==( last_count, tmr1_reg_get())
                ++( count_done )
            ENDIF
        ENDIF

        ; do whatever during the body of the loop.
        ; timing is not critical.
        =( last_count, tmr1_reg_get())
    UNTIL count_done

    con_string( "Pulse Width = " )
    con_out( last_count )
    con_string( "us" )
    =( rps_result, /( 1000000L, last_count ))
    con_string( "Thousands of Rotations Per Second = " )
    con_fmt( rps_result, "####.00X" )
    REP
        debug_on()
    LOOP
ENDFUN
```

4.14 Interfacing to RS232 devices.

Another very common use for TICkit type processors is to "glue" together various electronic instruments using RS232 format serial connections. Many such instruments are available as a result of Marine use of GPS, Compas and LORAN. NEMA standards for communications as well as serial interfaces for LCD displays and countless data acquisition instruments, make the RS232 format one of the most essential controller interfaces.

Even though RS232 is so wide spread, it is a standard which was not initially intended for many of the uses it now performs. This leads to a rather loose interpretation of the signal names and meanings. Generally, the only standard part of the RS232 standard is the bit timing of the serial data stream. The voltages, polarities, pin assignments, and connectors all vary by application. Therefore, when we refer to RS232 in respect to the TICkit, we are refering to the bit timing of the stream. The TICkit can produce TICkit output that is either intended for standard RS232 drivers like the MAX232 or 1489 driver ICs, or it can produce an open drain inverted output that can, in most cases, be connected directly to RS232 sockets via a resistor.

The following diagrams illustrate RS232 timing and how inverted or non-inverted signals appear on output pins.


Tbit = 1 / Baud rate e.g. @9600 baud, Tbit = 104.1 us
<Tbit>
+5
Non-inverted
Ground
ST
0
1
2
3
4
5
6
7
STOP
+5
Inverted
float
ST
0
1
2
3
4
5
6
7
STOP
Inverted signal should be pulled low to Gnd or -9 volts

There are 5 basic functions associated with serial communications on the TICkit. There are complex functions are available that build on these functions.

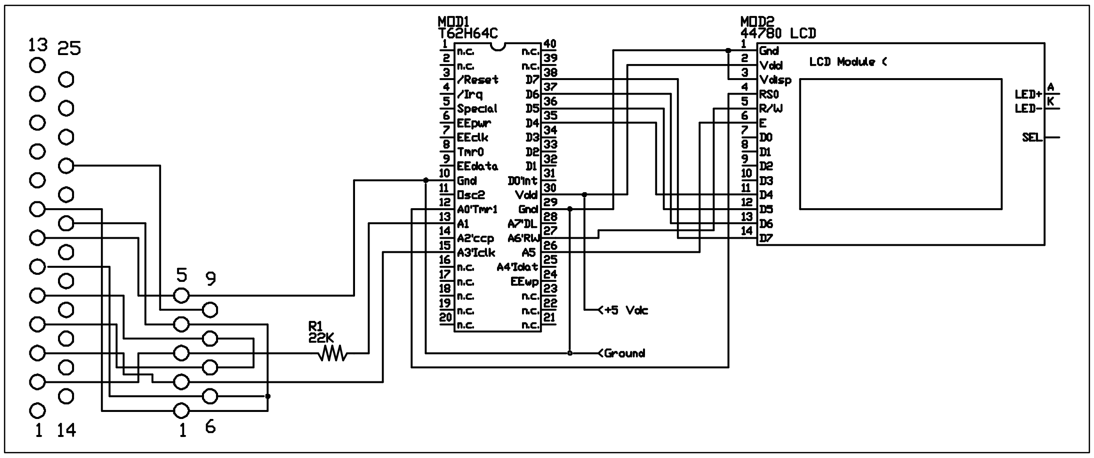
The first functions are the rs_param_set() and rs_param_get() function. These two functions are used to setup subsequent serial communcations. These functions set the baud rate, pin number for data, and determine if the stream is inverted or not.

The third function is the rs_send() function. This function sends the specified byte out using the format and pin set by the rs_param_set() function. The function, rs_break() can be used to send a byte with a forced framing error, but this is seldom required. This is used for advanced serial protocols.

The fourth function is the rs_receive() function. This function receives a single byte of specified format from the specified pin. This function can have a timeout or wait indefinately to receive a byte. It also returns error information when an error in format is detected in the input stream. A special control parameter allows a handshake pin to be used in addition to the data stream pin specified with the rs_param_set() function.

The fifth function, rs_recblock() is similar to the rs_receive() function except that it can receive more than one byte. This function also has the control parameter and can be instructed to ignore data until a matching byte or a break is detected in the input stream. These special conditions can be useful when creating a network of controllers that are linked by a shared serial line.

For this manual, we are only going to deal with serial transfer to and from a PC. We use a normal communications program like WINTERM or PROCOMM to send and receive serial data over a standard COM port. The cable we use is shown below. Also included are the standard pin assignments for 9 and 25 pin PC connectors.



Pin Function	DB9	DB25
Frame Ground	-	1
Transmit data (TD)	3	2
Receive data (RD)	2	3
Request to Send (RTS)	7	4
Clear to Send (CTS)	8	5
Data Set Ready (DSR)	6	6
Signal Ground (SG)	5	7
Data Carrier Detect (DCD)	1	8

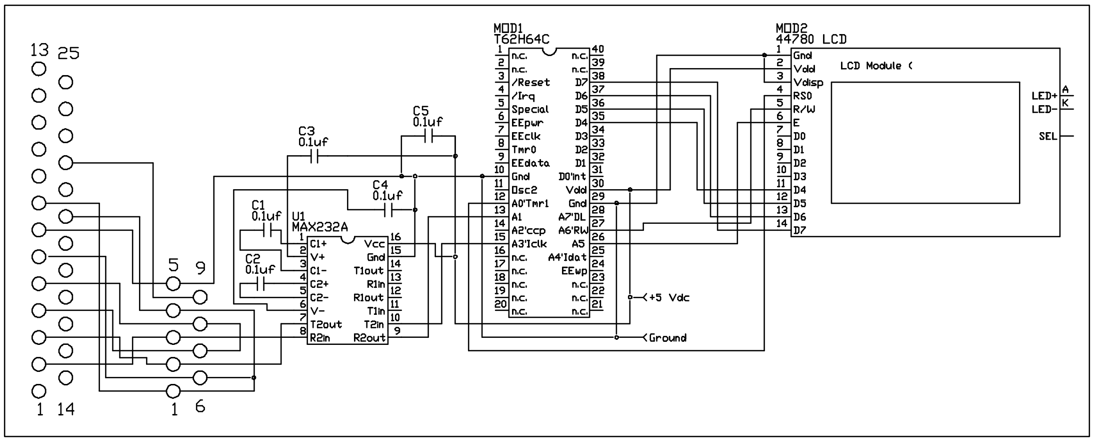
Data Terminal Ready (DTR)	4	20
Ring Indicator (RI)	9	22

The demonstration program is as simple as the circuit. The program initializes the LCD display then displays 20 characters it receives with the rs_receive() function. The the rs_send() function is used by the rs_string() function to send a message to the PC saying, "send a block beginning with 'A'". At this point, the program uses rs_recblock() to get a block of 10 characters which are prefaced with the letter 'A'. When all 10 characters are received, the string is displayed on the LCD. The process is repeated indefinately

```
DEF tic62_c
LIB fbasic.lib

; These defines used by the LCD libraries
DEF xbuss_mask 0y00100001b   ; These are the address lines used
DEF lcd_data_reg 0y00100001b ; Address of data register
DEF lcd_cont_reg 0y00100000b ; Address of control register

LIB lcdinit4.lib
LIB lcdsend.lib
LIB lcdstrin.lib

FUNC none rs_string
    PARAM word in_string
    PARAM word temp_ptr
    PARAM word temp_chr
BEGIN
    =( temp_ptr, in_string )
    =( temp_chr, ee_read( temp_ptr ))
    WHILE temp_chr
        rs_send( temp_chr )
        ++( temp_ptr )
        =( temp_chr, ee_read( temp_ptr ))
    LOOP
ENDFUN
```

```
FUNC none main
    LOCAL byte in_count
    LOCAL byte temp_chr
    LOCAL byte in_array[ 10b ]
BEGIN
    delay( 500 )         ; delay 1/2 second
    lcd_init()
    rs_param_set( rs_invert | rs_9600 | pin_a3 )
    =( in_count, 0b )
    REP
        lcd_data_wr( rs_receive( 0b, 0b, 0b ))
        ++( in_count )
    UNTIL ==( in_count, 20b )

    rs_param_set( rs_invert | rs_9600 | pin_a1 )
    rs_string( "Send a block beginning with 'A'" )

    rs_param_set( rs_invert | rs_9600 | pin_a3 )
    =( temp_chr, rs_recblock( 0b, rs_cont_addr, 'A',~
     ~ in_array[ 0b ], 10b )

    rs_param_set( rs_invert | rs_9600 | pin_a1 )
    rs_string( "Block was: " )
    =( in_count, 0b )
    REP
        rs_send( in_array[ in_count ] )
        ++( in_count )
    UNTIL ==( in_count, 10b )

    reset()
ENDFUN
```

The PC's communication program must be set to the following settings: 9600 baud, the com port number that the cable is plugged into, no handshake, 8 bits, no parity, 1 stop bit. Play with this program and circuit to get a feel for how things work. Some people may find that when the TICkit sends data nothing is received by the PC or possibly garbled data is received. This is because the voltages generated by the TICkit are in the range of 0 to 5 volts. True RS232C states that the voltages should range between +3 and +9 volts for a "space" (low) and -3 to -9 volts for a "mark" (high). The following circuit accomplishes an official interface to a PC. The only program change required for this circuit is to remove the "rs_invert" word from the rs_param_set function calls. The circuit for PC communication, with conforming drivers is shown here.

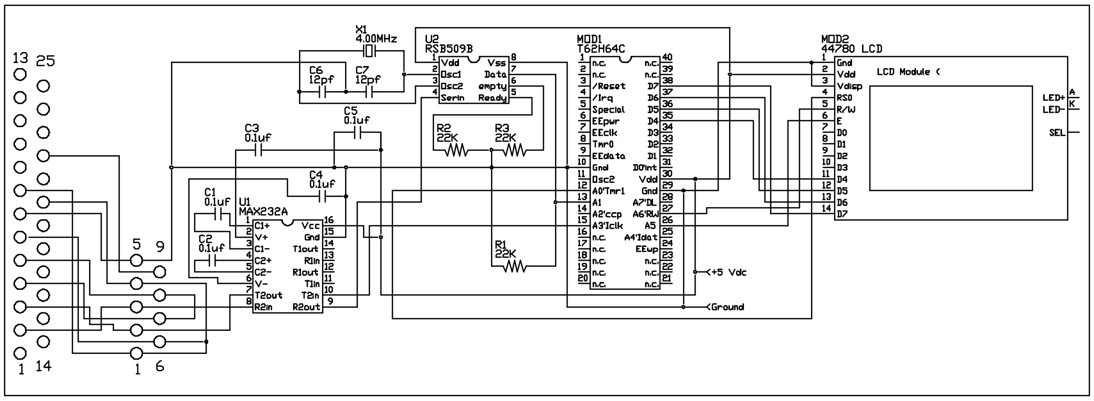


4.15 Using the RSB509 to Receive RS232 in Background.

In the previous example, you may notice that there are times when data sent to the TICkit from the PC is lost or garbled. This is not caused by a driver problem like transmission in the other direction. The cause for this problem is rooted in the fact that the rs_receive() and rs_recblock() functions are RS232 emulations. This means that there is no internal hardware dedicated to monitoring the input. The only time the pin is being monitored for RS232 input is while the function(s) is executing. Therefore, for the time that the program is dealing with a received byte, it is not listening for the next byte from the sender.

This problem can be solved in one of three ways. The first is to establish a special protocol so that the PC, or whatever device is sending to the TICkit, transmits only when the TICkit is ready. An example of such a protocol is used by the console functions and the debug program. This works well, but is often not possible when using existing designs for transmitting. Another example is to use the handshake lines in conjunction with the TICkit's receive functions. Unfortunately, very few RS232 sending devices monitor the handshake lines on a byte by byte basis. They typically assume that the receiver can take a byte or two more even after the handshake line indicates busy. The only sure way to receive an asynchronis data stream is to use dedicated receiving hardware.

Protean has created the RSB509 for this purpose. This 8 pin IC works with the TICkit's receive functions, but buffers received data and only sends to the TICkit when signaled. The circuit for interfacing to the RSB509 follows. Notice that only one general purpose I/O line is used to connect to the RSB509. The TICkit sends a quick pulse out the interface pin to signal the RSB509 its readiness. The RSB509 then sends one byte if it has buffered data to send.



The program fragment for this is shown below. This is similar to the previous example except that the pulse protocol has been included for controlling the RSB509.

```
LIB rsb509b.lib

FUNC none main
    LOCAL byte in_count
    LOCAL byte temp_chr
    LOCAL byte in_array[ 10b ]
    LOCAL byte in_err
BEGIN
    delay( 500 )         ; delay 1/2 second
    lcd_init()
    rs_param_set( rs_invert | rs_9600 | pin_a3 )
    pin_high( pin_a3 )
    delay( 10 )
    =( temp_chr, pin_in( pin_a3 ))  ; end command pulse
    rs_send( 'A' )                  ; program RSB509 for an A
    rs_send( rsb509_baud1 )         ; program for 9600
    delay( 100 )                    ; give RSB509 0.1 sec to reset

    =( in_count, 0b )
    REP
        pin_high( pin_a3 )            ; ask RSB509 for data
        =( temp_chr, rs_receive( 0b, 0b, in_err ))
        IF in_err
            ; no RSB data
        ELSE
            lcd_data_wr( temp_chr )
            ++( in_count )
        ENDIF
    UNTIL ==( in_count, 20b )
    pin_high( pin_a3 )
    delay( 10 )
    =( temp_chr, pin_in( pin_a3 ))  ; end command pulse
    rs_send( 'A' )                  ; program RSB509 for an A
    rs_send( rsb509_baud1 | rsb509_addr )  ; program for 9600
    delay( 100 )                    ; give RSB509 0.1 sec to reset

    rs_param_set( rs_invert | rs_9600 | pin_a1 )
    rs_string( "Send a block beginning with 'A'" )

    rs_param_set( rs_invert | rs_9600 | pin_a3 )
    =( in_count, 0b )
    REP
        pin_high( pin_a3 )            ; ask RSB509 for data
        =( in_array[ in_count ], rs_receive( 0b, 0b, in_err ))
        IF in_err
            ; no RSB data
        ELSE
            ++( in_count )
        ENDIF
    UNTIL ==( in_count, 10b )


    rs_param_set( rs_invert | rs_9600 | pin_a1 )
    rs_string( "Block was: " )
    =( in_count, 0b )
    REP
        rs_send( in_array[ in_count ] )
        ++( in_count )
    UNTIL ==( in_count, 10b )

    reset()
ENDFUN
```

Notice that the rs_recblock function is eliminated and rs_receive() is used in the loop instead. This is because the RSB509 performs the address detection and is, therefore, easier to interface using the byte by byte method.

4.16 Example Summary

This concludes our examples section of the manual. This is the first manual printing to include this chapter, so there may be errors. Please let Protean know if you find mistakes with the examples given. All programs and circuits are based on working counterparts, but many examples in this book were modifyed for simplicity and could contain simplification errors or transcription errors.

As you build these examples and design your own circuits keep in mind the following list of suggestions. It might save you some time, aggravation, and money.

1. Whenever you apply power to a circuit for the first time, verify the power connections with an ohm meter. Apply power briefly to check for shorts, hot components, or smoke. When you are confident your circuit is not damaging itself, then power the circuit for extended periods.
2. When you have a design worked out, program all unused general purpose pins to be outputs. Or, tie all unused inputs to either ground or +5 vdc. This prevents floating inputs from oscillating internally and conserves power and reduces heat.
3. When debugging your program, make use of all the debugging tools. This means writing a stack overflow routine so that the TICkit informs you in some way (A console message or turning on an LED) that the TICkit's stack has been exceeded. Single step through your program, or use the debug_on() function in areas of your program that may contain bugs and trace through them. If an area of problem in your code must run at full speed, wire in extra LEDs and temporarily modify your code to have it show via the LEDs what is happening.
4. Develop good revision techniques. This means putting comments at the beginning of your program file every time you make a modification. Make a copy of your program every time you start a new series of modifications so you can revert back to the last working version if

   clear which modification is the source of the problem.
5. Use the Protean Web site and Email extensively. This is the most economical and effective way to get support from Protean and the best way to get new ideas and learn about new ways of solving old problems.
6. If it seems like the TICkit is just on the edge of being fast enough to accomplish a task, consider putting some or all of that task into a dedicated peripheral IC. Often adding a few dollars of silicon to a project saves tremendous costs in software development and product support.

We hope these suggestions are helpful to you. Enjoy your TICkit. We always like to hear how our customers are using our products, so send us an Email about your projects if you get a chance.

5 FBASIC Keywords

5.1 Keywords, Are they commands or what?

Keywords are words that are the basic building blocks of a language. Unlike variables or function names, they cannot be renamed or created by the programmer. This means they are the quintessential character of the language.

In FBASIC, keywords are used to control compiling of source files, define other data symbols, define procedure symbols, and to explain the flow control of the finished program. These groupings are referred to as compile directives, definition or declaration directives, and flow control directives. Statements, Commands and Directives are all synonymous terms in FBASIC. Keywords also inform the compiler about specifics of the host processor like memory limitations or special internally generated operations like array dereferencing.

The keywords of FBASIC are summarized in the four groupings that follow:

Compile directives:

```
ANOTE = places a note in the compiler output.
BREAK = places a break point in debugger symbol file.
WATCH = places a watch point in debegger symbol file.
KEYWORD = informs compiler that symbol is a keyword.
VECTOR = informs the compiler about an interupt vector.
DEFINE = assign a symbol to a textual meaning.
INCLUDE = compile a component source file at this point.
INTERNALS = inform compiler about internal token generation.
LIBRARY = compile a unique source file at this point.
MEMORY = inform compiler about memory limits.
IFDEFINED = affirmative conditional line in compilation.
IFNOTDEFINED = negative conditional compilation line.
```

Data Definition and Declaration directives:

```
SIZE = assigns a symbol to a physical data size.
TYPE = assigns a symbol a logical meaning of a data size.
GLOBAL = allocate RAM for a global variable.
LOCAL = allocate RAM for a local variable.
PARAMETER = define parameter for a FUNCTION or OPERATION.
ALIAS = rename a global RAM location as another symbol.
ALLOCATE = allocate EEprom space for data storage use.
INITIAL = define initial contents of an ALLOCATE.
RECORD = define a data structure block for an ALLOCATE.
SEQUENCE = defines a sequence for external structures.
FIELD = define a component field of a RECORD.
ENDRECORD = ends a RECORD block definition.
```

Procedural Declaration and Definition directives:

```
      FUNCTION = define a function block.
      ENDFUNCTION = ends a FUNCTION block definition.
      PROTOTYPE = Declares a function symbol with no procedure.
      OPERATION = define an operation block.
      ENDOPERATION = ends an OPERATION block definition.
      EQUIVALENT = define a function to be equivalent to another.
```

Flow Control directives:

```
      IF = mark the start of a conditional program path.
      ELSEIF = mark the start of a alternate conditional path.
      ELSE = mark the opposite condition program path.
      ENDIF = mark the end of conditional program paths.
      REPEAT = mark start of an unconditional loop construct.
      WHILE = mark start of a loop and define the looping test.
      UNTIL = mark end of a loop and define the exit test.
      LOOP = mark end of an unconditional loop construct.
      SKIP = define condition to skip to the end of a loop.
      STOP = define condition to exit a loop.
      CALL = calls another function. (default keyword)
      EXIT = exit current function and return to calling function.
      GOTO = execute at specified label.
      GOSUB = execute at specified label and RETURN here.
      RETURN = return to line following prior gosub.
```

Flow Control directives can only appear in special blocks called "procedure blocks". The blocking concept is used by FBASIC to keep things neat in a source file. Procedure blocks are started using the FUNCTION directive and end with the ENDFUN directive.

The other type of blocking structure in FBASIC is RECORD. This is used to collectively refer to a group of data items by one symbolic name and assign the initial values to be contained in this group when the program starts. This is useful for data storage applications such as lists.

Elements of a language that are not keywords are the standard libraries. Libraries are simply a set of pre-written functions and definitions that are assumed to be useful to programmers in that language. Library functions can be overridden by the programmer for any specific task. The standard libraries of functions and data types for FBASIC are summarized later in this manual.

Detailed information on each keyword directive follows.

ALIAS

ALIAS: Alias declaration of internal RAMstorage.

ALIAS <size_or_type> <variable_name> <overlay_name> (<overlay_offset>)

This directive is used outside of procedure blocks. Use this directive to refer to a previously allocated RAM storage location by a name different than that used for the initial allocation. Use of aliases can conserve RAM space in conditions where the programmer knows that there will be no conflict between the two names for the location. ALIAS can also be used to overlay variables of smaller size over a variable of larger size. This is useful for building up or deconstructing larger types. (This syntax is dated, use the GLOBAL or LOCAL directives with the ALIAS option instead).

ALLOCATE

ALLOCATIONs: Used to reserve program memory locations

```
ALLOC(ATE) <initial_offset>
ALLOC(ATE) <size_type_or_record> <allocation_name> ( '[' <count> ']')
```

This statement can only appear outside of procedure blocks. Allocate directs the compiler to reserve sufficient program memory to store count items of the size given. The symbolic name will be a constant that points to the first address of this reserved area. This is useful for symbolic representation of stored data. The ALLOCATE directive can also be used to indicate to the compiler where to begin the data storage in EEprom. Normally the compiler places all allocations and strings immediately following the program code in the EEPROM. The initial offset form of ALLOCATE can be used to force the storage to begin at a different location. This can be useful if part of the EEPROM is write protected.

See RECORD and FIELD for more information on structures in EEPROM

SEQUENCE is a similar directive but does not effect EEPROM allocation at all. Use SEQUENCE instead of ALLOCATE if symbolic addresses are being assigned to memory other than the EEPROM of the TICkit.

ANOTE

ANOTEs: Used to place text in the compiler output

```
ANOTE <any single line of text>
```

This statement can only appear outside procedure blocks. ANOTE directs the compiler to place the following text in the status output of the compiler. This action has no effect on the output token file. This directive can be used to indicate which libraries or include files are compiled for any given program.

BREAK

BREAKs: Used to indicate a default break point to the debugger

```
BREAK <any procedural keyword and statement>
```

This statement can only appear inside procedure blocks. BREAK directs the compiler to place a break point symbol in the symbol file. This action has no effect on the output token file, but instructs the debugger that this line is a break point when the debugger starts. Only the first 10 default break points can be recognized by the debugger.

CALL

CALL: Evaluates an expression.

(CALL) <expression_with_no_return_value>

This statement is procedural and can appear only after the BEGIN statement in a procedure block. The CALL keyword is optional because any first word on a line which is not a keyword will be interpreted as a CALL statement. This statement will cause the following expression, of SIZE none, to be evaluated. If the first function or operation of the expression has a return value, an error will be returned.

DEFINITION

DEFINITIONS: Textual equates in the source code.

DEF(INITION) <symbol_name> <any_text>

This directive is valid anywhere, but is a global equate only when it appears outside a procedure block. This is used to make code more readable and to eliminate arcane numeric references. DEFINED symbols can also be used to conditionally compile certain sections of a program. See IFDEFINED and IFNOTDEFINED for more details on conditional compilation.

EQUIVALENT

EQUIVALENT: Defines a function prototype that uses a procedural section of an existing function

```
FUNC(TION) <size_or_type> <function_name>
    parameters....
    locals...
EQUIV(ALENT) <existing_function_name>
```

This directive is not currently implemented for the TICkit57 and TICkit62. The Equivalent directive is used when special versions of existing functions are required that use more specific parameter and return value types.

EXIT

EXITs: Returns to the line CALLing function.

EXIT

This statement can only appear within a procedural block. EXIT will cause the execution to resume at a point immediately following the reference to the function that EXIT appears in. If the function has a return value, the value contained in the variable "exit_value" will be passed back to the calling reference as the value of the function. Therefore, to return a value for a function, assign the desired return value to the variable "exit_value" immediately before executing an EXIT statement. An implicit EXIT occurs whenever ENDFUNCTION is encountered.

FIELD

FIELDs: Defines subordinate elements of a RECORD or ALLOCATION.

```
FIELD <type_or_record> <symbol_name> ( '[' <count> ']' )
```

This directive is used in RECORD blocks to define data elements inside the larger structure. The "count" is optionally used to make an array out of the field. The default count is one.

FUNCTION

FUNCTIONs: External function code definition.

```
FUNC(TION) <size_or_type> <function_name>
    parameters....
    locals...
BEGIN
    procedural statements....
ENDFUN(CTION)
```

This directive can only be used outside of procedure blocks and defines a function. Any PARAMETERs, LOCALs, and DEFINEs defined within the FUNCTION block are only defined to procedure statements within that FUNCTION block. If the function has a return value, a local variable called "exit_value" of the function's type will exist for the duration of the function. Assign the desired return value "exit_value" immediately before EXIT or ENDFUNCTION.

GLOBAL

GLOBALs: Global internal RAM storage allocation.

```
GLOBAL <size_or_type> <variable_name>  ( '[' <count> ']'  ) (<initial_value(s)>)
  or
GLOBAL <size_or_type> <variable_name> ALIAS <variable_name> ( <offset> )
```

This directive is used outside of procedure blocks. This directive allocates Global data from the bottom of memory as opposed to the internal RAM stack which grows down from the top of memory. Therefore, usage of global values reduces the available stack for subroutines and local values. Because local allocations are returned to the RAM pool after a function finishes executing, local variables are often more memory efficient than global values. Global values execute faster however, and may even be more space efficient if the variable can be safely re-used in multiple functions. An array of elements can be defined by using the '[]' characters and an element count. Exercise care when defining arrays not to exceed the memory capacity of the device. The "WATCH" directive may be used at the beginning of a GLOBAL statement to place a watch point symbol in the symbol file. Upon startup, the debugger will automatically watch up to five GLOBAL values with WATCH directives.

The ALIAS option can be used to make the defined variable overlay an existing variable. This can prove useful for building up larger types or for creating special combined types.

GOSUB

GOSUB: Executes a sub-section of a function as a sub-routine.

```
(GOSUB) <local_line_label>
```

This statement is procedural and can appear only after the BEGIN statement in a procedure block. Program execution will shift to the line beginning with a label that matches "local_line_label". This line must be in the same function as the GOSUB. Execution continues from that line until a RETURN statement is encountered, and resumes immediately following the last GOSUB statement executed. Care must be exercised when using GOSUB. If GOSUB and RETURN are not matched properly, the program's stack could be destroyed, leading to unpredictable results. The preferred method for performing subroutines within FBASIC is to use multiple functions with CALL statements.

GOTO

GOTO: Causes the flow of the program to alter.

```
GOTO <local_line_label>
```

This statement is procedural and can appear only after the BEGIN statement in a procedure block. This directive causes execution within the program to jump to the location specified by "local_line_label". The line_label must be in the same procedure block as the GOTO.

IF

IFs: Creates alternations and branches in the program flow.

```
IF <logical_expression>
    procedural statements....
( ELSE or ELSEIF <logical_expression> )
    procedural statements....
ENDIF
```

The IF statement is procedural and can only appear after the BEGIN statement in a procedural block. Use this directive to change the flow of a program based on a condition, usually a variable comparison. The ELSE or ELSEIF are optional extensions to this directive. This directive can be lexically nested.

IFDEFINED

IFDEFINEDs: Conditionally compile a line.

    IFDEF[INED] <define_symbol> [any other directive]

The IFDEF statement is a compiler directive that and can appear anywhere. The directive which follows on the same line as the IFDEF will only be executed if the referenced <define_symbol> exists. If the symbol does not exist, the line will not be compiled. This line can be used in conjunction with the INCLUDE or LIBRARY directives to conditionally compile large sections of a program.

IFNOTDEFINED

IFNOTDEFINEDs: Conditionally compile a line.

    IFN[OT]DEF[INED] <define_symbol> [any other directive]

The IFNOTDEFINED statement is a compiler directive and can appear anywhere. The directive which follows on the same line as the IFNOTDEFINED will only be executed if the referenced <define_symbol> does not exist. If the symbol exists, the line will not be compiled. This line can be used in conjunction with the INCLUDE or LIBRARY directives to conditionally compile large sections of a program.

INCLUDE

INCLUDEs: Compile directive to include a subordinate file at this point.

    INCLUDE <source_file_name>

The INCLUDE directive is used to merge another source file in the compilation of the program. This is useful when organizing large programs or for special methods of repeating code in a program. The INCLUDE directive differs from the LIBRARY directive in that the file will be included regardless of whether or not the file was included in the compile previously. Using INCLUDE with IFDEFINED and IFNOTDEFINED statements creates a powerful conditional compilation capability.

INITIAL

INITIALs: Set an initial value for EEprom Allocations.

    INIT(IAL) <full_field_name> <initial_value> (<additional_values>...)

The INITIAL statement is used to place initial values into EEprom allocations. This can be very useful for creating tables, etc. The "full_field_name" must include the name of the allocation, all records, and the field name to completely identify the field. At least one initial value is required. If the field has a count greater than one, then additional initial values may be included up to the count of the field. For byte type field, the special constant format ' ' may be used to specify a string of initial values. This format differs from the " " constant format which evaluates to a word. The ' ' format evaluates to multiple bytes.

INTERNALS

INTERNALS: Species token code for internal operatoins

```
INTERNALS <token codes> ...
```

This directive appears only in the token library for the TICkit interpreter. The list of tokens instructs the compiler how to generate array dereferences and other token references generated automatically by the expression generator.

KEYWORD

KEYWORDs: This directive is used to inform the compiler that a symbol is reserved.

```
KEYWORD <reserved_symbol>
```

This directive appears only in the "fbasic.lib". A user may wish to use this directive to reserve symbols that will eventually appear in a program. Normally, this directive will not be used except in the standard library.

LIBRARY

LIBRARY: Textually included source code.

```
LIB(RARY)  <file_name>
```

This directive is valid anywhere in the body of the code, but use in the beginning of a program aids readability. The "filename" is the DOS text file which is to be included in the compile at this point in the source. LIBRARY and INCLUDE differ only in the case that a file name is used that has previously been used in the same compile. LIBRARY will ignore a the request if a file_name appears twice. INCLUDE will process the file regardless of whether or not it had appeared in the compile previously.

LOCAL

LOCALs: Local internal RAM storage allocation.

```
LOCAL <size_or_type> <variable_name> ( '[' <count> ']' ) (<initial_value(s)>)
   or
LOCAL <size_or_type> < variable_name> ALIAS <variable_name> (<offset>)
```

This directive is used inside of program blocks prior to the BEGIN statement. This directive allocates LOCAL data from the bottom of memory. A pointer to that memory location is placed on the internal RAM stack which grows down from the top of memory. Because local allocations are returned to the RAM pool after a function finishes executing, local variables are often more memory efficient than global values. Global values execute faster however, and may even be more space efficient if the variable can be safely re-used in multiple functions, possibly using the ALIAS statement. Arrays of LOCAL variables can be defined by using the '[]' characters and an element count. Take care not to exceed the stack space of the device when allocating arrays. LOCAL values exist only while the program is executing, for this reason, current debugger implementations are not able to watch or examine LOCAL values by symbol name.

The ALIAS option can be used to make the defined variable overlay an existing variable. This can prove useful for building up larger types or for creating special combined types.

MEMORY

MEMORY: Specify memory constraints for a host processor

```
MEMORY HIGH <upper RAM limit>
MEMORY LOW <lower RAM limit >
MEMORY EEPROM <sequence breaks in EEprom>
```

This directive usually appears in the token library for a device. It tells the compiler how to assign global memory and when to generate warnings about EEprom sequence breaks. The values of these limits are defined by the version of the TICkit token interpretter and the type of size of each EEprom connected to it.

OPERATION

OPERATIONs: Internal operation code definition.

```
OPER(ATION) <size_or_type> <operation_name>
    parameters....
BEGIN hexadecimal values....
ENDOP(ERATION)
```

This directive can only appear outside of code blocks and informs the compiler of internally implemented functions. These functions are identical to source level functions except that they operate faster and must be implemented in the token interpreter.

PARAMETER

Parameters: Define the parameter list and symbolic argument names for functions or operations.

```
PARAM(ETER) <size_or_type> <argument_name>
```

This directive can only be used inside of procedure blocks between the OPERATION or FUNCTION directive and the BEGIN directive. PARAMETERs inform the compiler how to handle the argument or parameter list for the FUNCTION or operation in which they appear. PARAMETERs also assign a symbolic name to the argument so that they may be used indirectly by the FUNCTION in which they appear. PARAMETERs are temporary names that pointer to the variables used in the function call.

PROTOTYPE

PROTOTYPEs: Declare a function without creating the procedure.

```
FUNC[TION] <return_type> <function_name>
    parameters...
PROTO[TYPE]
```

The PROTOTYPE directive is used to define a function symbol without actually defining the procedure associated with the function. This is useful when doing recursive applications, or any other time that a function will be referenced before it is defined. The FBASIC single pass compiler characteristic requires that programs be written in a "top down" style. The use of PROTOTYPES for all functions in a program frees the programmer from the "top down" requirement.

RECORD

RECORDs: Used for making symbolic maps of external memory allocations

```
REC(ORD) <record_name>
    fields....
ENDREC(ORD)
```

This block can only appear outside of a procedure block. Record blocks are used to declare relative locations of items logically grouped in EEprom memory, or other memory areas. The structures can not be nested, but may lexically recurs. The defined structures can then be declared using the ALLOCATE statement, which actually reserves program space for the records. When a record symbol appears in an expression, it evaluates to a constant of SIZE "word_size". This constant can then be manipulated as a pointer to any sort of memory device.

REPEAT

REPEATs: Marks the beginning of a loop with no looping condition.

```
REPEAT
    procedural statements....
( STOP )
    procedural statements....
( SKIP )
    procedural statements....
UNTIL <logical_expression> or LOOP
```

This directive causes the enclosed block to repeat while the repeat_condition is true or until the exit_condition is true. The STOP directive will exit the body of the loop and the SKIP directive will cause the loop to perform the next iteration without finishing the body of the loop by skipping to the statement at the bottom of the block. Looping directives can be lexically nested.

RETURN

RETURNs: Returns to the line immediately following the last GOSUB.

```
RETURN
```

This statement can only appear within a procedural block. RETURN will cause the execution to resume at a point immediately following the last GOSUB that was executed. Because return addresses are stored on the stackby the GOSUB statement, and removed by the RETURN statement. The number of RETURNs executed in a function must exactly match the number of GOSUBs executed in a function.

SEQUENCE

SEQUENCEs: Establish a sequence for external storage.

```
SEQ(UENCE) <sequence_number> <initial offset>
SEQ(UENCE) <sequence_number> <size, type or record> ~
     ~ <symbol_name> ( '[' <count> ']' )
```

This statement can only appear outside of procedural blocks. SEQUENCE is used to establish either a beginning offset for a given sequence number or to indicate the location of a storage element or array at the current offset for the sequence number. The offset is increased to the first byte past the storage required if a storage element or array is referenced. SEQUENCE is very similar to ALLOCATE except that it does not effect the EEPROM allocation and that there can be more than one sequence for a program. Up to 10 sequences, each uniquely identified by a sequence number, can be used for a program in this version of the FBASIC compiler.

<count> is used to create an array for the <symbol_name> at the current sequence offset.

SIZE

SIZEs: Define a symbol to one of the intrinsic data sizes of the compiler.

```
SIZE <size_symbol> <number_of_type_size>
```

This directive appears in the "fbasic.lib" file. This directive is used to assign a symbol which is easier to remember and more concise than the numbers that the compiler recognizes for size designators. The default sizes are: none, byte, word, and long. None uses no storage, byte uses one 8 bit location, word uses two 8 bit locations, and long uses four 8 bit locations. Only long is arithmetically signed.

TYPE

TYPEs: Define a logical meaning to a data size.

```
TYPE <symbolic_meaning> <size_symbol>
```

This directive, which is similar to the SIZE directive, is used to more loosely assign a meaning to a data item. The symbolic meaning does not override the SIZE of the data item but will prevent another data item which has a different symbolic meaning from being assigned to this data item. By using these restricting measures, the compiler can prevent an accidental misuse of data items of the same physical size, but different logical meanings.

VECTOR

VECTORs: Infrom the Compiler about hardware Interrupt Vectors in TICkit memory. Vectors are attributes of the Microprocessor and the Interpreter firmware.

```
VECTOR <symbolic_name>
```

This directive appears only in the standard token library. Each use of the VECTOR statement defines a function name to be associated with the physical vector of the processor and the processor firmware. The first VECTOR directive assignes the first vector slot. Each subsequent appearance of the VECTOR directive assigns subsequent vector memory locations.

WATCH

WATCHs: Marks data element as a watch point in subsequent debugging sessions

```
WATCH GLOBAL <type> <symbol_name> (<initial_value>)
```

The WATCH directive is used to place a symbol in the watchpoint list of the debugger. Using this directive may save time when debugging more complex programs. Only global values can be watched by the debugger in current versions.

WHILE

WHILEs: Marks the beginning of a loop with a condition to loop.

```
WHILE <logical_expression>
    procedural statements....
( STOP )
    procedural statements....
( SKIP )
    procedural statements....
UNTIL <logical_expression> or LOOP
```

This directive causes the enclosed block to repeat while the repeat_condition is true or until the exit_condition is true. The STOP directive will exit the body of the loop and the SKIP directive will cause the remainder of loop to be omitted by skipping to the statement at the bottom of the loop. Looping directives can be lexically nested.

6 Standard Function Library

6.1 Standard Libraries: "....What do they contain, Books?"

A programming language like FBASIC is very lean. It contains only the basic building blocks of programs, but very few functional parts. The standard library provides most of the functinality of FBSIC. The standard library is a set of functions, operations, definitions, and declarations. The standard library contains things like math functions, input and output functions, bit manipulation, etc.

Physically, the standard library is a collection of program fragments the author of the language assumes are useful to the programmer. The programmer can call functions from the standard library that are required for a larger application. Also, the programmer may decide that certain functions of the standard library are inappropriate or unnecessary for a given application. For this reason, the library is broken down into smaller library files and organized in a sort of hierarchy where library files depend on functions in other files to get the job done. The programmer must determine which library files to reference in his program in order to make an efficient program.

Normally, the programmer will use a define statement and the FBASIC.LIB file to include the appropriate standard library for the processor revision being used. Therefore, the following two lines usually appear as the first directives of a program:

```
DEF tic62_a
LIB fbasic.lib
```

As mentioned above, a programmer may wish to have greater control over which elements of the standard library are included in a program. When this is the case, the programmer must pick and choose elements from the library files. This is easy to do since each library file can be examined and modified with a text editor.

The standard library also includes a reference to extended functions not in the firmware of the processor. To exclude these libraries from your program use the following define before the reference to fbasic.lib

```
DEF tic62_a
DEF operations_only
LIB fbasic.lib
```

6.2 Standard Library Summary

The following sections of the manual divide the functions of the standard library according to function groups. The group headings are:

1. Assignment and size conversion functions
2. Mathematical Functions
3. Bit manipulation functions
4. Logical relational test functions
5. Input and output functions
6. EEprom read and write functions
7. IIC peripheral funtions
8. Parallel Buss (LCD) functions
9. Timing and Counting Functions
10. RS232 functions
11. Console functions
12. System, interrupt and miscellaneous functions
13. Integrated peripheral functions

Library organization is as follow. Notice that including fbasic.lib automatically includes the proper token library and the proper token extension library. You need to explicitly include any other library (like rs_fmt.lib) if you want the functions in it. Simply add LIB rs_fmt.lib into your program before you reference any functions of that library. Also be aware of any DEF statements that the library may expect. All libraries can be viewed with a text editor.

fbasic.lib = FBASIC keyword and size declaration.
- token.lib (tic62c.lib) = token interpreter operation declarations.
  - tokext.lib (ticx62c.lib) = Extension functions. Use DEF opertions_only to exclude.

ee.lib = larger size EEprom functions
rs232.lib = rs232 string and numeric functions
- rsstring.lib = send a null terminated string
- rsbyte.lib = byte to string of numbers
- rsword.lib = word to string of numbers
- rslong.lib = long to string of numbers
- rsfmt.lib = formatted long to string of numbers

con.lib = consolse string and number functions
- constrin.lib = send a null terminated string
- conbyte.lib = byte to string of numbers
- conword.lib = word to string of numbers
- conlong.lib = long to string of numbers
- confmt.lib = formatted long to string of numbers

lcd.lib = LCD buss functions for controlling HD44780 based LCD modules
- lcdinit.lib = initialize the LCD and buss

lcdstrin.lib = send a null terminated string
lcdbyte.lib = byte to string of numbers
lcdword.lib = word to string of numbers
lcdlong.lib = long to string of numbers
lcdfmt.lib = formatted long to string of numbers
lcdchar.lib = character generator programming function
lcdscroll.lib = scrolling routines to make an LCD look like a terminal

6.3 Additional Libraries Summary

When an additional library is included, the minimum size of your program increases because all functions referenced in the extend library are put into your program whether you use them or not. Usually, you use the extend library when you want to develop a program quickly.,.

Various libraries may be placed on release disks. These libraries are often hardware dependent. Other extended libraries can be found on the Protean BBS. Libraries for serial A/D chips and serial clock chips are a few examples. You can use a simple text editor to view these library files. Notes on their use will be contained as comments in the library files. Do not be intimidated by library files, they are simply small functions and provide a nice way to increase the number of tools available to you. Every time you write a function for dealing with a specific type of hardware device or any time you develop a section of code you think you will re-use, put that function into a library file so you can access it easily in future programs.

6.4 Assignment and Size Conversion Functions

Assignment is the most basic of programing functions. The contents of one memory variable or constant is copied into another memory variable. The truncate functions simply drop bytes of higher order than the result requires. This is, in essence, a modulus function of either 256 or 65536. The to_xxx functions append 0 value bytes to the higher order bytes of the result. The return value of to_xxx functions have the same numeric value as the argument only in a larger variable size.

= Assignment

```
none =( byte dest, byte source ) token.lib
none =( word dest, word source ) token.lib
none =( long dest, long source ) token.lib
```

Multi-precision assignment function. The contents of the source value is copied into the destination.

trunc_byte Truncates a larger size to a byte

```
byte trunc_byte( long arg ) token.lib
byte trunc_byte( word arg ) token.lib
```

Truncates the argument to a byte size. Any information in the more significant bytes is discarded.

trunc_word   Truncates a larger size to a word

`word trunc_word( long arg ) token.lib`

Truncates the argument to a word size. Any information in the more significant bytes is discarded.

to_word   Extends a smaller size to a word

`word to_word( byte arg ) token.lib`

Extends the argument to a word size by placing zeros in the more significant bytes.

to_long   Extends an (argument) to long size

```
long to_long( byte arg ) token.lib
long to_long( word arg ) token.lib
```

Extends the argument to a long size by placing zeros in the more significant bytes.

Conversion Function Examples:

```
; determine the most significant Hex digit of a word

FUNCTION none main
    LOCAL word in_val 10000
    LOCAL byte char_val
BEGIN
    =( char_val, trunc_byte ( /( in_val, 4096 )))
    IF >( char_val, 9 )
        =( char_val, +( char_val, '0' ))
    ELSE
        =( char_val, +( char_val, -  ( 'A', 10b )))
    ENDIF

    con_out_char ( char_val )
ENDFUN
```

6.5 Mathematical Functions

The mathematical functions are used to perform arithmetic in FBASIC. The mathematics functions can be viewed as a "prefix" notation for expressions. In expressions where order is significant, like subtraction and division, The first argument is the value that is operated on, while the second argument is the value of the operation. In division then, the first argument is the numerator and the second argument is the denominator and the value returned is the quotient.

+ Arithmetic Sum

```
byte +( byte arg1, byte arg2 ) token.lib
word +( word arg1, word arg2 ) token.lib
word +( byte arg1, word arg2 ) token.lib
word +( word arg1, byte arg2 ) token.lib
long +( long arg1, long arg2 ) token.lib
long +( long arg1, word arg2 ) token.lib
long +( long arg1, byte arg2 ) token.lib
long +( word arg1, long arg2 ) token.lib
long +( byte arg1, long arg2 ) token.lib
```

Multi-precision addition function. Two arguments are added together. The result is returned as the value of the function.

++ Increment by One

```
byte ++( byte arg ) token.lib
word ++( word arg ) token.lib
long ++( long arg ) math32.lib
```

Multi-precision increment single argument. The return value of the function is one plus the argument value.

- Arithmetic Difference

```
byte -( byte arg1, byte arg2 ) token.lib
word -( word arg1, word arg2 ) token.lib
word -( byte arg1, word arg2 ) token.lib
word -( word arg1, byte arg2 ) token.lib
long -( long arg1, long arg2 ) token.lib
long -( long arg1, word arg2 ) token.lib
long -( long arg1, byte arg2 ) token.lib
long -( word arg1, long arg2 ) token.lib
long -( byte arg1, long arg2 ) token.lib
```

Multi-precision subtraction function. The result of arg1 less arg2 is returned as the value of the function.

- Arithmetic Inverse (change sign)

```
long -( long arg ) math32.lib
```

Change sign function. The complement of arg is returned as the value of the function.

-- Decrement by One

```
byte --( byte arg ) token.lib
word --( word arg ) token.lib
long --( long arg ) math32.lib
```

Multi-precision decrement single argument. The return value of the function is the argument value less one.

* Arithmetic Product

```
byte *( byte arg1, byte arg2 ) token.lib
word *( word arg1, word arg2 ) token.lib
word *( byte arg1, word arg2 ) token.lib
word *( word arg1, byte arg2 ) token.lib
long *( long arg1, long arg2 ) token.lib
long *( long arg1, word arg2 ) token.lib
long *( long arg1, byte arg2 ) token.lib
long *( word arg1, long arg2 ) token.lib
long *( byte arg1, long arg2 ) token.lib
```

Multi-precision multiplication function. The result of arg1 multiplied by arg2 is returned as the value of the function.

/ Arithmetic Division

```
byte /( byte arg1, byte arg2 ) token.lib
word /( word arg1, word arg2 ) token.lib
word /( byte arg1, word arg2 ) token.lib
word /( word arg1, byte arg2 ) token.lib
long /( long arg1, long arg2 ) token.lib
long /( long arg1, word arg2 ) token.lib
long /( long arg1, byte arg2 ) token.lib
long /( word arg1, long arg2 ) token.lib
long /( byte arg1, long arg2 ) token.lib
```

Multi-precision division function. The result of arg1 divided by arg2 is returned as the value of the function.

% Arithmetic Modulus (Remainder)

```
byte %( byte arg1, byte arg2 ) token.lib
word %( word arg1, word arg2 ) token.lib
byte %( byte arg1, word arg2 ) token.lib
word %( word arg1, byte arg2 ) token.lib
long %( long arg1, long arg2 ) token.lib
long %( long arg1, word arg2 ) token.lib
long %( long arg1, byte arg2 ) token.lib
word %( word arg1, long arg2 ) token.lib
byte %( byte arg1, long arg2 ) token.lib
```

Multi-precision remainder function. The remainder of arg1 divided by arg2 is returned as the value of the function. For 32 bit functions, the sign follows that of arg1.

array_byte Calculate Address of a byte array element

```
word array_byte( word offset, word index ) token.lib
```

This function returns a word value which is the address of an element of an array which starts at "offset" and which is the "index" numbered element.

array_word Calculate Address of a word array element

`word array_word( word offset, word index ) token.lib`

This function returns a word value which is the address of an element of an array which starts at "offset" and which is the "index" numbered element.

array_long Calculate Address of a long array element

`word array_long( word offset, word index ) token.lib`

This function returns a word value which is the address of an element of an array which starts at "offset" and which is the "index" numbered element.

array_size Calculate Address of an array element

`word array_size( word offset, word size, word index ) token.lib`

This function returns a word value which is the address of an element of an array which starts at "offset" and which is the "index" numbered element. All elements in the array are assumed to be of "size" number of bytes.

Mathematics Function Examples:

```
; program to count the numbers from 100 to 2000 by 10
FUNCTION none main
    LOCAL word cnt_val
BEGIN
    rs_param_set ( debug_pin )  ; setup console to use
                                ; the same connection as
                                ; the debugger
    =( cnt_val, 100 )           ; set count value to 100
    REP
        con_out ( cnt_val )
        =( cnt_val, + ( cnt_val, 10 ))
    UNTIL >( cnt_val, 2000 )
ENDFUN
```

6.6 Bit Manipulation Functions

The bit manipulation functions work on byte values only. Each bit of the arguments have the function performed on them. For example, an "AND" function is really 8 AND functions where the result of each of the AND operations is placed in the 8 bits of the return value of the function. These functions are usually used for masking out specific bits for test or combination from bytes and words. Logical functions are typically used as conjunctions for comparative functions (==,>,<). Logical functions are only available for byte types.

b_and 8 and 16 bit Bitwise logical and function

```
byte b_and( byte arg1, byte arg2 ) token.lib
word b_and( word arg1, word arg2 ) token.lib
```

The result is the bit by bit AND of arguments one and two. The 8 bit version of this function can also be used as a logical AND but the and() function is recommended for logical conjunction.

b_or 8 or 16 bit Bitwise logical OR function

```
byte b_or( byte arg1, byte arg2 ) token.lib
word b_or( word arg1, word arg2 ) token.lib
```

The result is the bit by bit OR of arguments one and two.

b_xor 8 or 16 bit Bitwise logical exclusive or function

```
byte b_xor( byte arg1, byte arg2 ) token.lib
word b_xor( word arg1, word arg2 ) token.lib
```

The result is the bit by bit EXCLUSIVE-OR of arguments one and two.

b_not 8 or 16 bit Bitwise logical complement function

```
byte b_not( byte arg ) token.lib
word b_not( word arg ) token.lib
```

The result if the bit by bit NOT of the argument. Therefor, all bits that are 1 in the argument are returned as 0 and vice versa.

>> 8 and 16 bit arithmetic shift argument to the right

```
byte >>( byte arg ) token.lib
word >>( word arg ) token.lib
```

All bits of the argument are shifted toward bit 0. The least significant bit is discarded as a result and zero is placed in msb.

<< 8 and 16 bit arithmetic shift argument to the left

```
byte <<( byte arg ) token.lib
word <<( word arg ) token.lib
```

All bits of the argument are shifted toward the msb. The most significant bit is therefore, discarded. 0 is placed in the LSB.

b_set Set bits in an 8 or 16 bit field by mask

```
none b_set( byte field, byte mask ) tokext.lib
none b_set( word field, word mask ) tokext.lib
```

Bits are set in the field argument on the basis of which bits are set in the mask. Any bits which are set in the mask will be set in the field argument. Bits in the mask which are zero, will leave the cooresponing bits in the field argument unchanged. These functions are useful for conserving space by using bits as boolean flags.

b_clear Clear bits in an 8 or 16 bit field by mask

```
none b_clear( byte field, byte mask ) tokext.lib
none b_clear( word field, word mask ) tokext.lib
```

Bits are cleared in the field argument on the bsis of which bits are set in the mask. Any bits which are set in the mask will be set in the field argument. Bits in the mask which are zero, will leave the cooresponding bits in the field argument unchanged.

b_test Tests bits in an 8 or 16 bit field by mask

```
byte b_test( byte field, byte mask ) tokext.lib
byte b_test( word field, word mask ) tokext.lib
```

This function tests specific bits in a field. If any of the bits specified by the mask are set in the field 0xffb is returned. If all the specified bits are zero, the function returns 0b. This is a convenient way to test bits used as boolean flags.

Bitwise Function Examples:

```
; Routine to read data from a ADC0831 A/D chip

FUNCTION byte ad_read       ; 'Returns a byte'
    LOCAL byte count 0b     ; a Byte counter
BEGIN
    pin_low(clk)            ; make pin an output,
                            ; needed when sharing buss
    pin_low (cs)             ; enable chip
    pulse_out_high (clk,10w) ; toggle clk to get start bit
    REPEAT
        pulse_out_high(clk,10w)   ; toggle clk to get bits
        =(exit_value,<<(exit_value))    ; shift bits
        =(exit_value, ~
          ~b_or(exit_value , ~
          ~b_and (pin_in (data),1b)))
                                  ; mask bit and add to data
        ++(count)
    UNTIL ==(count,8b)

    pin_low(data)          ; return bus data line to output
    pin_high(cs)           ; disable chip
ENDFUN
```

6.7 Logical And Relational Test Functions

Logical relational functions are used in conditional flow control expressions, like IF or WHILE. Relational functions return 255 (0xff) if the two arguments meet the relational condition, or 0 if they do not. The bitwise combination logic functions, "and", "or", "not", and "xor" can be used with these functions provided all true values in the expression have all bits set (0xff).

== Multi-precision relational test for equal

```
byte ==( byte arg1, byte arg2 ) token.lib
byte ==( byte arg1, word arg2 ) token.lib
byte ==( byte arg1, long arg2 ) token.lib
byte ==( word arg1, byte arg2 ) token.lib
byte ==( word arg1, word arg2 ) token.lib
byte ==( word arg1, long arg2 ) token.lib
byte ==( long arg1, byte arg2 ) token.lib
byte ==( long arg1, word arg2 ) token.lib
byte ==( long arg1, long arg2 ) token.lib
```

If the result of arg1 less arg2 is equal to zero, 0xff is returned. Otherwise, a 0 is returned as the value of the test.

>= Multi-precision rel. test for greater than or equal

```
byte >=( byte arg1, byte arg2 ) token.lib
byte >=( byte arg1, word arg2 ) token.lib
byte >=( byte arg1, long arg2 ) token.lib
byte >=( word arg1, byte arg2 ) token.lib
byte >=( word arg1, word arg2 ) token.lib
byte >=( word arg1, long arg2 ) token.lib
byte >=( long arg1, byte arg2 ) token.lib
byte >=( long arg1, word arg2 ) token.lib
byte >=( long arg1, long arg2 ) token.lib
```

If the result of arg1 less arg2 is greater than or equal to zero, 0xff is returned. Otherwise, a 0 is returned as the value of the test.

<= Multi-precision relational test for less than or equal

```
byte <=( byte arg1, byte arg2 ) token.lib
byte <=( byte arg1, word arg2 ) token.lib
byte <=( byte arg1, long arg2 ) token.lib
byte <=( word arg1, byte arg2 ) token.lib
byte <=( word arg1, word arg2 ) token.lib
byte <=( word arg1, long arg2 ) token.lib
byte <=( long arg1, byte arg2 ) token.lib
byte <=( long arg1, word arg2 ) token.lib
byte <=( long arg1, long arg2 ) token.lib
```

If the result of arg1 less arg2 is less than or equal to zero, 0xff is returned. Otherwise, a 0 is returned as the value of the test.

> Multi-precision relational test for greater than

```
byte >( byte arg1, byte arg2 ) token.lib
byte >( byte arg1, word arg2 ) token.lib
byte >( byte arg1, long arg2 ) token.lib
byte >( word arg1, byte arg2 ) token.lib
byte >( word arg1, word arg2 ) token.lib
byte >( word arg1, long arg2 ) token.lib
byte >( long arg1, byte arg2 ) token.lib
byte >( long arg1, word arg2 ) token.lib
byte >( long arg1, long arg2 ) token.lib
```

If the result of arg1 less arg2 is greater than zero, 0xff is returned. Otherwise, a 0 is returned as the value of the test.

< Multi-precision relational test for less than

```
byte <( byte arg1, byte arg2 ) token.lib
byte <( byte arg1, word arg2 ) token.lib
byte <( byte arg1, long arg2 ) token.lib
byte <( word arg1, byte arg2 ) token.lib
byte <( word arg1, word arg2 ) token.lib
byte <( word arg1, long arg2 ) token.lib
byte <( long arg1, byte arg2 ) token.lib
byte <( long arg1, word arg2 ) token.lib
byte <( long arg1, long arg2 ) token.lib
```

If the result of arg1 less arg2 is less than zero, 0xff is returned. Otherwise, a 0 is returned as the value of the test.

<> Multi-precision relational test for not equal

```
byte <>( byte arg1, byte arg2 ) token.lib
byte <>( byte arg1, word arg2 ) token.lib
byte <>( byte arg1, long arg2 ) token.lib
byte <>( word arg1, byte arg2 ) token.lib
byte <>( word arg1, word arg2 ) token.lib
byte <>( word arg1, long arg2 ) token.lib
byte <>( long arg1, byte arg2 ) token.lib
byte <>( long arg1, word arg2 ) token.lib
byte <>( long arg1, long arg2 ) token.lib
```

If the result of arg1 less arg2 is not equal to zero, 0xff is returned. Otherwise, a 0 is returned as the value of the test.

and Perform logical AND conjunction on two bytes

```
byte and( byte arg1, byte arg2 ) tokext.lib
```

This function returns 0xffb only if both arguments are logically true. In other words, this function returns 0b if either of the arguments is 0b. Use this function when combining relational tests in logical expressions.

or Perform logical OR conjunction on two bytes

`byte or( byte arg1, byte arg2 ) tokext.lib`

This function returns 0xffb if either of the two arguments is logically true. In other words, this function returns 0b only when both of the arguments is 0b. Use this function when combining relational tests in logical expressions.

not Perform logical NOT on a byte

`byte not( byte arg ) tokext.lib`

This function returns 0xffb only if the argument is zero. In other words, this function returns 0b if the argument is any value other than 0b.

Examples:

```
FUNCTION none main
    LOCAL in_temp
BEGIN
    REP
        =( in_temp, ad_read())
        IF or(  <( in_temp,35 ), > ( in_temp, 112 ))
            allarm( in_temp )
        ELSE
            do_other_stuff( in_temp )
        ENDIF
    LOOP
ENDFUN
```

6.8 Input and Output Functions

The input and output functions represent the TICkits interface to the real world. All of these functions are implemented as high speed internal PIC routines. Most of these routines refer to a pin_number argument. The pin number is a byte that ranges between 0 and 15. The pin numbers 0 through 7 correspond to the pins labeled D0 through D7 on the TICkit. The pin numbers 8 through 13 correspond to the pins labeled A0 through A5 on the TICkit. Pin number 14 is labeled R/W and pin number 15 is labeled DL on the TICkit. Just as this implies, the I/O pins on the TICkit can often serve different roles in different programs. Pins may serve as data or address bus pins, general I/O pins, or a serial connections.

pin_high Make pin a high logic output

`none pin_high( byte pin_number ) token.lib`

Make the specified pin an output and set it to a high voltage level. The pins are numbered 0 through 15 where 0 is the data port's pin 0 and 15 is the address port's pin 7.

pin_low Make pin a low logic output

```
none pin_low( byte pin_number ) token.lib
```

Make specified pin an output and set it to a low voltage level. The pins are numbered 0 through 15 where 0 is the data port's pin 0 and 15 is the address port's pin 7.

pin_in Make pin an input and return logic level

```
byte pin_in( byte pin_number ) token.lib
```

Return a logical value representing the logical voltage level of the specified pin. A true value is returned if the pin has a logical high value input to it. The pins are numbered 0 through 15 where 0 is the data port's pin 0 and 15 is the address port's pin 7.

aport_get Get byte representing pin levels of address port

```
byte aport_get() token.lib
```

Read all 8 pins from the address port into a byte.

dport_get Get byte representing pin levels of data port

```
byte dport_get() token.lib
```

Read all 8 pins from the data port into a byte.

aport_set Set pin levels of address port

```
none aport_set( byte pins_values ) token.lib
```

Sets all 8 pins in the address port to the levels specified by pins_values.

dport_set Set pin levels of data port

```
none dport_set( byte pins_values ) token.lib
```

Sets all 8 pins in the data port to the levels specified by the pins_values.

atris_get Get status of address pin tristate levels

```
byte atris_get() token.lib
```

Returns all 8 bits from the address direction register. A zero in a bit indicates that the corresponding pin is an output.

dtris_get Get status of data pin tristate levels

```
byte dtris_get() token.lib
```

Returns all 8 bits from the data direction register. A zero in a bit indicates that the corresponding pin is an output.

atris_set Set tristate levels for address pins

```
none atris_set( byte dir_values ) token.lib
```

Sets all 8 bits of the address direction register according to dir_values. A zero in a bit indicates the corresponding pin is to be an output.

dtris_set Set tristate levels for data pins

```
none dtris_set( byte dir_values ) token.lib
```

Sets all 8 pins of the data direction register according to dir_values. A zero in a bit indicates the corresponding pin is to be an output.

pulse_in_low Measure duration of a low pulse

```
word pulse_in_low( byte pin_number ) token.lib
```

Measures the duration of a low pulse on the specified pin. A zero is returned if either no pulse is detected or if the pulse is greater than .65535 seconds in duration. Each count is 10 microseconds.

pulse_in_high Measure duration of a high pulse

```
word pulse_in_high( byte pin_number ) token.lib
```

Measures the duration of a high pulse on the specified pin. A zero is returned if either no pulse is detected or if the pulse is greater than .65535 seconds in duration. Each count is 10 microseconds.

pulse_out_low Generate a low pulse on a pin

```
none pulse_out_low( byte pin, word dur ) token.lib
```

Generates a low pulse of the specified duration on the specified pin.
Each count produces a 10 microsecond duration.
NOTE: Pin must be made an output before executing this function.

pulse_out_high Genereate a high pulse on a pin

```
none pulse_out_high( byte pin, word dur )token.lib
```

Generates a high pulse of the specified duration on the specified pin.
Each count produces a 10 microsecond duration.
NOTE: Pin must be made an output before executing this function.

cycles Generate square wave cycles on a pin

```
none cycles( byte pin, word cycles,~
   ~word high_time, word cycle_time ) token.lib
```

Generates the specified number of square wave cycles on the specified pin, with the specified high and cycle periods. All times are specified in approx. 3 us intervals. By keeping the high time one half of the cycle time, a 50% duty cycle square wave can be generated. By varying the duty cycle of the wave, the cycles function can be used as analog to digital conversion by connecting a capacitor between the output pin and ground. Up to a 16 bit resolution can be supported using this method. Use a constant as the fixed wave length of the conversion. The voltage out will correspond to the ratio of the high_time divided by the cycle_time multiplied by the high voltage. Frequencies as low as 2.5 cycles per second and as high as 60K cycles per second can be generated using this function.
NOTE: Pin must be made an output before executing this function.

rc_measure Measure the resistance/capacitance at a pin

`word rc_measure( byte pin ) token.lib`

Measures the discharge time of a resistance and capacitance circuit. This function can be used to determine either the resistance or the capacitance in such a circuit. The resistance and capacitance should be wired in parallel between the I/O pin specified and ground. A zero will be returned if either the discharge time is too low, or the charge/discharge time is too high. Appendix A describes this circuit in greater detail.

Examples:

```
; resistance to voltage converter

FUNCTION none main
    LOCAL word res_val
BEGIN
    pin_low( 9b )                      ; discharge cap
    REP
        =( res_val, - ( rc_measure ( 9b ), 1000 )
        ; RC circuit at pin 9
        cycles ( 10b, 100, res_val, 39000 )
        ; D/A circuit at pin 10
        ; assume full range value is 40000 and low value
        ; is 1000.
    LOOP
ENDFUN
```

6.9 Eeprom Routines (Pointer Dereferencing)

The EEprom routines access information contained in the TICkit eeprom by using a 16 bit address. This is the same memory that is used to contain the TICkit program. When an FBASIC program is compiled, the compiler calculates the amount of space required by the procedure and all ALLOCATIONs. The first EEpromlocation that is not used by the program is placed in a special vector at the beginning of the EEprom by the compiler. The two bytes contained at locations 0x0004 and 0x0005 of the EEprom form a 16 bit word which is the address of the first available EEprom space. This address and all addresses higher than it are available for a program to use. Much of this address space may not be usable if no EEprom device has been installed for that area. The standard 2K EEprom TICkits have a total address space from 0x0000 to 0x07ff. 8K EEprom configured TICkits are initially shipped with only an 8K EEprom installed, but an additional 7 devices may be installed which brings the address range up to a full 64K. The programmer will need to code programs with the known upper limit of memory to prevent an unsuccessful read or write to an illegal address.

The ALLOCATE directive bypasses some of the complexity mentioned above. The ALLOCATE statement will reserve EEprom space for data use. The address of any allocation or component field of an allocation is known in an expression simply by referencing the full field and allocation name. The programmer must still exercise caution to ensure that allocations do not exceed the physically implemented limit of the EEprom.

ee_read Read a byte at EEprom address

```
byte ee_read( word address ) token.lib
```

Reads a byte from the EEprom at the specified address. Reads that are out of the valid address space (no eeprom maps to that address) will cause unpredictable results that may result in premature program termination. The programmer must therefore assure that the address is valid. EEprom address 4 and 5 contain the low and high bytes of the address of the first available eeprom byte. All space from this point to the end of the EEprom storage address space is available for program use. The ALLOCATE keyword can be used to allocate EEprom data space in a structured way.

ee_read_word Read a word at EEprom address

```
word ee_read_word( word address ) ee.lib
```

Reads a word from the EEprom at the specified address. Reads that are out of the valid address space (no eeprom maps to that address) will cause unpredictable results that may result in premature program termination.

ee_read_long Read a long at EEprom address

```
long ee_read_long( word address ) ee.lib
```

Reads a long from the EEprom at the specified address. Reads that are out of the valid address space (no eeprom maps to that address) will cause unpredictable results that may result in premature program termination.-

ee_write Write a byte to EEprom address

```
none ee_write( word address, byte data ) token.lib
none ee_write( word address, word data ) ee.lib
none ee_write( word address, long data ) ee.lib
```

Writes the contents of the argument data to the EEprom at the specified address. See ee_read for more details.

EEprom Examples:

```
; use record and allocate to record purchases
LIB ee.lib                     ; library for ee_write_word

RECORD each_buy
    FIELD word cust_no
    FIELD word quantity
    FIELD word prod_no
ENDREC

ALLOC word last_purchs
ALLOC each_buy purchs 100     ; make space for 100 purchases

GLOBAL cur_purchs 0


FUNCTION none main                  ; list purchases
    LOCAL word temp_purchs
    LOCAL word purch_count 0
BEGIN
    ee_read_word( cur_purchs, last_purchs ) ; read last rec
    =( temp_purchs, purchs )    ; points to first record
    WHILE <( temp_purchs, cur_purchs )
        ++( purch_count )
        con_out( purch_count )
        con_out_char( ' ' )
        con_out ( ee_read_word( ~
             ~ +( temp_purchs, cust_no@each_buy )))
          ; display customer number
        con_out_char( ' ' )
        con_out( ee_read_word( ~
          ~ +( temp_purchs, quantity@each_buy )))
             ; display quantity
        con_out_char( ' ' )
        con_out( ee_read_word( ~
          ~ +( temp_purchs, prod_no@each_buy )))
        ; display product number
        con_out_char ( '\r' )
        con_out_char( '\l' )

        =( temp_purchs, +( temp_purchs, each_buy ))
    LOOP
ENDFUN
```

6.10 IIC Peripheral Functions

Starting with version 2.0 of the TICkit interpreter, Generic I2C bus operations are supported for limited peripheral connections using the existing clock and data lines. These lines, which connect to the EEprom and also be used to connect to I2C peripherals with compatible command protocols. Such a device is the Protean X-Tender device. When placing additional peripherals on the I2C bus wires, care must be used to ensure the electrical requirements of the 400k bit per second connection are conformed to. This may require 10k ohm terminations on the physical ends of the lines, special routing of the lines, and special logical address selection of the devices sharing the line. All devices must conform to the three byte or four byte protocol specifications:

1. Address Byte: bit0=R/W, upper seven bits must be a unique device address
2. Command Byte: This byte command the addressed device to do something
3. Data Byte(s): The byte(s) is either read or written on the basis of the Address byte bit0. This is usually a parameter for a command, or the result of the previous command. If the function is a word function, the low byte is sent first.
4. In Read operations, the above protocol is modified. If the R/W bit of the address byte is 0, the address and command bytes will be sent but a re-start will be issued instead of any data transfere following the command byte. If the R/W bit is set, the address and command bytes are skipped and only the following occurs.
5. The address byte is sent with the R/W bit set.
6. Data Byte(s): The single or double byte (TICkit 57 only for double byte) data is received by the TICkit. The TICkit can be paused by the sending device holding the clock for the first data bit of transfer. The sending device must not hold the TICkit for longer than the internal watchdog timer (approx 16ms) or a TICkit reset may occur.

Three functions implement this protocol. The user must ensure that the address bit is set appropriately for reading or writing. Additionaly, notice that the Address/command word used in all of the I2C functions is a passback parameter. If there is an error communicating to an I2C device, the upper byte of the Address/command word is cleared. The Interpreter will attempt to communicate with a device for approximately 16ms (or more if a prescaler is used with the internal watch dog timer) before clearing the address byte and continuing past the I2C function.

i2c_write Write a command and data byte to bus

```
none i2c_write( word addr_comm, byte data ) token.lib
```

This function will write a byte to the addressed device. The address and command bytes are concatenated to form the addr_comm byte. The exact address and command will vary from one peripheral device to another.

The address byte of addr_comm will be cleared if the function fails. The word data version of this function is only available in the TICkit57

i2c_read Read a byte from an addressed device

`byte i2c_read( word addr_comm ) token.lib`

This function will transmit an address and a command, then wait to read back a byte from the addressed device. The exact protocol used in this function depends upon the level of the R/W bit of the device address. If the R/W bit is low (write level) an address byte and command byte will be sent before the data read is performed. The address byte of addr_comm will be cleared if the function fails.

I2C I/O Examples:

```
=( write_addrcomm, 0x80c2w )
i2c_write( write_addrcomm, 0x8b )     ; select A/D channel 0
                                      ; on I2C Xtender periph.
IF <( write_addrcomm, 256w )
    call i2c_error()                  ; handle error with I2C
ELSE
    =( in_voltage, i2c_read( 0x80c2w ))     ; read voltage
ENDIF
```

6.11 Parallel Bus And Lcd Functions

The bus functions implement a limited traditional parallel microprocessor bus. This bus may have either 8 or 4 data lines and may have up to 6 address lines for a total address space of 32 read and 32 write locations. Bus configurations with 4 lines can be made to write 8 bit values by sending two 4 bit values in succession. This works with LCD modules that support 4 bit nibble modes. Before any bus transfer, the bus routines must be set up with a special control byte. The upper two bits of this byte define the mode of the bus. Bit 7 determines if the data bus is 8 lines wide or 4 lines wide. Bit 6 has meaning only for 4 bit buses and determines if 8 bit values are to be sent on the bus by automatically sending two nibbles for every 8 bit value. The remaining bits of the control byte (bit 0 through bit 5) determine which of the address lines to use for bus operations and which lines to leave as general purpose I/O. If the bus is 4 lines wide only pins D4 through D7 are used for bus operations. Any of these bits that are high indicate that the corresponding address pin should be used to bus operations. Between bus operations, all selected address pins are set to a low level, effectively addressing location 0x00.

Data lines may be used for general purpose I/O between bus operations provided that the bus is set up again before the next bus operation.

The lower three address lines (A0 thru A2) maintain their levels longer than the upper address (A3 thru A5) lines. This prevents any race conditions that may exist between device selecting logic and the R/W, data lines, and the device select lines. For this reason, The lower three lines should be used as register select lines while the upper address lines should be used to select between devices on the bus. The meanings of the upper address lines combined with the fact that the address of zero is used as the "deselect" means that locations 0x00 through 0x07 should not be used by any devices on the bus. Map all address decoding to select device by requiring at least one of the upper address lines (A3 thru A5) to be high.

Some common LCD functions are documented here. These functions are contained in the libraries mentioned in their prototype. These functions assume the pressence of three defined symbols. Symbol lcd_bus_mask specifies which of the address lines are to be used by bus functinos. Symbol lcd_data_reg specifies the bus address for the data register. Symbol lcd_cont_reg specifies the bus address for the control register of the LCD module.

buss_setup Setup address and data pins for bus I/O

```
none buss_setup( byte mode_and_mask ) token.lib
```

Sets up the external bus routines. the mode_and_mask specify what the data bus width will be, if 4 bit wide how may nibbles to send, and which lines from the address port to dedicate to use as address lines. Bit 7 specifies the width of the data bus. A high indicates that all eight lines of the data I/O port are dedicated to bus functions. A low indicates that only bits 4 through 7 are dedicated to bus functions. Bit 6 is ignored for 8 bit operations but indicates how many nibbles to send for each bus function if the data bus is 4 bit. A high in bit 6 causes all bus functions to perform two 4 bit nibble transfers for every operation to transfer a complete 8 bit byte. If bit 6 is low, only bits 4 through 7 of any bus read or write operation will be transferred. Bits 0 through 5 of mode_and_mask are used to reserve I/O lines of the address port for bus address lines. If any of these bits are high, the bits will be used to select devices on the bus during read and write operations. Any bits of mode_and_mask that are low are unaffected in future bus read/write operations. All bus read or write operations effect pin 6 of the address port. This line is used as the read/write line for the address bus. This line is normally a high output after bus_setup but is brought low during bus write operations. Items on the address bus are selected whenever their address is placed on the used address port lines. Whenever a bus operation is not taking place, all used address lines are brought low. This effectively selects bus address zero. Therefore, no devices on the address port can be mapped to address zero.

buss_read Read a byte from bus address

```
byte buss_read( byte address ) token.lib
```

Reads a byte from the external bus at the specified address. Read method will conform to the current bus setup. Unused address or data I/O lines will not be affected by this function.

buss_write Write byte to bus address

```
none buss_write( byte addr, byte data ) token.lib
```

Writes a byte to the external bus at the specified address. Writes conform to the current bus setup. No unused address or data I/O lines are affected by this function.

lcd_init4 Initializes an LCD module for 4 bit data bus

```
none lcd_init4( ) lcdinit4.lib
```

This function sets up the TICkit bus and sends the necessary commands to initialize a 44780 based LCD module for 4 bit data transfer.

lcd_init8 Initializes an LCD module for 8 bit data bus

```
none lcd_init8( ) lcdinit8.lib
```

This function sets up the TICkit bus and sends the necessary commands to initialize a 44780 based LCD module for 8 bit data transfer.

lcd_cont_wr Writes a byte to LCD control register

```
none lcd_cont_wr( byte control )  lcdsend.lib
```

Use this function to write to the control register of a 44780 based LCD module. This function automatically ensures previous command is complete.

lcd_data_wr Writes a byte to LCD data register

```
none lcd_data_wr( byte data_val )  lcdsend.lib
```

Use this function to write to the data register of a 44780 based LCD module. This function automatically ensures previous command is complete. The data register is either character generator data or display data depending on the last write to the control registers address control.

lcd_string Writes a string to the LCD

```
none lcd_string( word string_addr )  lcdstrin.lib
```

This function writes a string of bytes to the LCD from a location in EEprom. The string must be null terminated. No control characters are acted upon.

lcd_out Writes a number to the LCD

```
none lcd_out( byte value ) lcdbyte.lib
none lcd_out( word value ) lcdword.lib
none lcd_out( long value ) lcdlong.lib
```

This function writes a number to the LCD screen. Three versions of this routine write either a byte, a word or a long value to the LCD.

lcd_fmt Writes a formatted long to the LCD

```
none lcd_fmt( long value, word form )  lcd_fmt.lib
```

This function writes a formatted number to the LCD screen. The format is determined by a string contained in EEprom (pointed to by argument form). Each character in the format string cooresponds to a digit. The character meanings are as folllows:

$	Print a '$' character in the output
#	Print a number if this or a previous digit was non-zero
0	Print a number even zero, forces following #'s to print
X	Do not print a number digit, but account for its position
.	Print a decimal point

Bus I/O Examples:

```
; Check that LCD is ready to receive data and write
; assume LCD is already initialized

FUNCTION none lcd_write
    PARAMETER byte lcd_out
BEGIN
    WHILE >( bus_read( lcd_cont_reg ), 0x80b )
    LOOP

    bus_write( lcd_data_reg, lcd_out )
ENDFUN
```

6.12 Timing and Counting Functions

The TICkit has no built in time keeping capability except for the microprocessor clock. However, by executing a known number of PIC instructions, a delay of known duration can be caused. This delaying technique is used to produce the time base for the following functions.

In addition to the delaying technique, the TICkit can take advantage of the PIC's internal RTCC(real time clock counter) to count rising or falling edges on the RTCC input, or to count machine clock cycles. The mode of the RTCC is set by using the "rtcc_" functions.

One additional capability of the PIC is used to generate longer delays and reduced power operation. Each PIC has a built in watchdog timer. This timer is a crude internal RC circuit that will reset the PIC if the Capacitor is not recharged before it is fully discharged. The watchdog timer is unavailable as a seperate user controlled resource, but is used by the interpreter for trapping unexplained errors like I2C timeout or for use with the sleep functions. This method of timing is relatively imprecise, but is still useful for creating a low power delay. The sleep function uses this method.

delay Delay processing for milliseconds

`none delay( word millisecond ) token.lib`

Delays program execution for the specified number of milliseconds.

sleep Delay processing and conserve power for a time

```
none sleep( byte sleep_periods ) token.lib
```

Puts the processor to sleep for the specified amount of sleep periods. Each period is nominally 18ms. This function, due to internal PIC organization, will modify the RTCC edge and source settings. NOTE: the time base for this function is an internal RC discharge rate and is affected by temperature and environmental conditions like the characteristics of the IC or supply voltage variations. The base delay of this function is typically 18ms at 25 degrees C, but can vary between 9ms and 30ms. Note: it is possible to assign the RTCC prescaler to the sleep timer using custom OPERATION directives not contained in the standard library. This will dramatically increase the sleep interval. If you really want to do this, examine the token library and observe how the rtcc_int_256 and rtcc_ext_rise operations are created. These operations simply set the PIC OPTION register.

rtcc_get Get the current count of the RTCC register

```
byte rtcc_get() token.lib
```

Reads the 8 bit contents of the RTCC register.

rtcc_set Set the count of the RTCC register

```
none rtcc_set( byte count ) token.lib
```

Sets 8 bit value, "count" into the RTCC register.

rtcc_int RTCC source is internal clock

```
none rtcc_int() token.lib
```

Sets the source for the RTCC to be the internal clock. This is the oscillator frequency divided by 4. This clock is inactive during sleeps.

rtcc_int_16 RTCC source internal and prescaled by 16

```
none rtcc_int_16() token.lib
```

Sets the source for the RTCC to be the internal clock. This is the oscillator frequency divided by 64. This clock is inactive during sleeps.

rtcc_int_256 RTCC source internal and prescaled by 256

```
none rtcc_int_256() token.lib
```

Sets the source for the RTCC to be the internal clock. This is the oscillator frequency divided by 1024. This clock is inactive during sleeps.

rtcc_ext_rise RTCC source is external clock

```
none rtcc_ext_rise() token.lib
```

Sets the source for the RTCC to be the external pin and clocks on the rising edge of any signal on this pin. This pin should be tied high or low if not used.

rtcc_ext_fall   RTCC source is external clock

`none rtcc_int() token.lib`

Sets the source for the RTCC to be the external pin and clocks on the falling edge of any signal on this pin. This pin should be tied high or low if not used.

rtcc_count   Count while delaying for milliseconds

`byte rtcc_count( word milliseconds ) token.lib`

Clears the RTCC register then counts pulses in the RTCC while the TICkit delays for n milliseconds. The 8 bit contents of the RTCC is returned after the delay. This function is useful for determining frequency of an AC signal up to about 50kHz.

rtcc_wait   Wait until RTCC count rolls over to zero

`none rtcc_wait() token.lib`

TICkit execution will pause until the RTCC register rolls over to a count of zero. This function can be used in conjunction with RTCC_SET and RTCC_INT_256 to implement a real time clock. By setting the RTCC count before a section of program is executed and then waiting for the RTCC count to roll over to zero following the program segment, the programmer can ensure the segment will take the same amount of time to execute for each time it is executed. This makes it possible to create real time loops.

Timing Examples:

```
; determine the frequency of an input square wave
; using the RTCC

FUNCTION word freq_get
BEGIN
    rtcc_ext_rise()      ; rising edge-external
    =( exit_value, rtcc_count( 100 ))
          ; count pulses for 100ms
          ; measures between 0 and 2550 Hz signals to
          ; the nearest 10 Hz.
ENDFUN
```

6.13 RS232 and Communications Functions

The PICs used by the TICkit57 and TICkit62 have no serial communications hardware built into them. Therefore, the token interpreter uses special software operations in the TICkit to simulate asynchronous serialcommunications hardware. This software relies on loops of PIC instructions to generate the timebase for the serial timing. This method does not produce the exact timing for the standard baud rates but produces acceptable results for rates from 300 to 9600 baud with a 4mHz PIC clock and 300 to 19200 baud with a 20 mHz PIC.

The routines also support a true and an inverted input/output through a general purpose I/O pin. All these parameters are set with a special parameter byte. Bit 7 of this bit indicates if the communication is inverted (bit7=1) or if the communications are true (bit7=0). Bits 4 through 6 determine the baud rate where 000=300 baud and 110=19200 baud. Bits 0 through 3 determine which general purpose I/O pin to use where 0000 is pin D0, 1000 is pin A0, and 1111 is the DL pin (which is used for debugging purposes also).

One additional parameter can be specified using the rs_stop_chek and rs_stop_ignore functions. If set to ignore, the stop bit of any value received from the serial communication is not to be tested for framing accuracy. This provides one additional bit of time for processing consecutive data received through the serial port, but does not allow the detection of a framing errors or break levels.

Special care must be exercised when using the serial receiving routines. The routines must be executing when data is transmitted from the sending device, otherwise the startbit will not be sensed and either no information or erroneous information will be received. This introduces an unavoidable timing problem for bursts of more than one byte of information to be received through the serial functions. Any processing of the byte just received must take place in one half bit time (or 1.5 bit time if the stop bit is not sensed) to ensure that the next byte will be received intact. This provides very little time for processing at high baud rates.

The TICkit57 has only a single byte receive routine for RS232. The TICkit62 has a multi-byte buffer and can receive up to 128 bytes of serial in a block (128 is a theoretical limit, in actual use, some of the TICkit memory will be used for processing and stack. On the TICkit62, 64 bytes is probably the maximum buffer than can be used for serial blocks).

The TICkit62 assumes that a message format will be used frequently on the TICkit. This is accomidated by features in the recblock function that wait for break levels or a specific address byte before actually capturing serial data. Also, the TICkit 62 can generate handshaking signals for a normal serial stream. Whenever the buffer gets full, the handshake line will change level and signal the transmitting device to pause while the the TICkit digests the buffer information.

rs_param_set   Set RS232 parameters

```
none rs_param_set( byte type_baud_pin ) token.lib
```

Setup baud rate and pin number for serial RS232 communications. This function also sets the pin level meanings. Bit 7 of type_baud_pin determines if the communications signal is inverted and open sourced, or if it is true and totem pole. If bit 7 is high, the signal is inverted and the driver is open sourced when transmitting which means the line must be pulled low. If bit 7 is low, the signal is true and the transmit driver is a totem pole configuration. A totem pole configuration should not be "wire-ored" to prevent stressing the output electrically. Bits 4 through 6 determine the baud rate of the communications routines. A value of 0 in these bits selects a 300 baud rate. A value of 14 selects a 19,200 baud rate. All baud rates in between follow the same doubling pattern. Bits 0 through 3 determine the pin for communications where 0 is data line 0 and 15 is address line 7. The standard console and debug parameter is a setting of 0xDF (Inverted, 9600 baud, pin 15-address line 7).

rs_break Send RS232 break condition

`none rs_break() token.lib`

Sends a break condition for the baud rate and pin specified by the rs_parameter. This function literally sends a space level for 13.5 bit times.

A break condition is technicaly a space level for at least 10 consequtive bit times. Normally, because rs232 consists of a start bit, 9 data bits, and at least on stop bit, no more than 9 consequtive space levels should occur before a mark level. Since an idle rs232 line is at mark level, a break condition will never occur as long as normal communication is taking place. Historically, break conditions were used to communicate a piece of information which is extraordinary to the normal data stream. The name "break" is derived from time share systems in which a break condition was used to interrupt the remote computer's program execution and return to an OS prompt. Breaks are also used to indicate the beginning of data frames etc. in serial packet protocols. Most asynchronys receivers interpret a break as a framing error with a data result of 0. By testing for this condition, breaks can be detected and used to advantage. The TICkit62 can use a break condition as a prefix for an address byte in the rs_recblock function.

rs_param_get Get RS232 parameters

`byte rs_param_get() token.lib`

Reads the current output type, baud rate, and pin number for serial RS232 communications. See the rs_param_set function for more details.

rs_send Send byte out RS232 pin (TICkit57)

`none rs_send( byte data, byte brk ) token.lib`

Send the value "data" out conforming to the RS232 timing standards. Input and output levels, as well as baud rate and pin are determined by the current contents of the rs_param register. A non-zero value for "brk" will cause an incorrect stop bit level to be sent, which is interpreted by most receivers as either a framing error or a break condition.

rs_send Send byte out RS232 pin (TICkit62)

`none rs_send( byte data ) token.lib`

Send the value "data" out conforming to the RS232 timing standards. Input and output levels, as well as baud rate and pin are determined by the current contents of the rs_param register. Break conditions can be generated using pulse functions or pin functions and timing delays.

rs_receive Receive byte in RS232 pin (TICkit57)

```
byte rs_receive( word wait, byte err )   token.lib
```

Receive a byte through a general purpose I/O pin. Input and output levels, as well as baud and pin information are determined by the current contents of the rs_param register. This function will wait approximately (16us * wait) for a start bit before returning an error. A zero value for wait, or a value greater than 65280 will cause an indefinite wait for a start bit. Error codes are: 0 = no error, 1=framing error (break), 2=timeout for start, 4=no initial mark level.

rs_receive Receive byte in RS232 pin (TICkit62)

```
byte rs_receive( byte wait, byte control, byte err )   token.lib
```

Receive a byte through a general purpose I/O pin. Input and output levels, as well as baud and pin information are determined by the current contents of the rs_param register. This function will wait approximately (4096us * wait) for a start bit before returning an error. A zero value for wait produces an indefinite wait for a start bit. The control byte is used to select a general purpose pin for handshake (lower four bits) and to enable handshaking by setting bit 4. Error codes are: 0 = no error, 1=framing error (break), 2=timeout for start, 4=no initial mark level.

rs_recblock Receive array of bytes in RS232 pin

```
byte rs_receive( byte wait, byte control, byte address,~
  ~byte buffer[], byte buf_size )   token.lib
```

Receive a block of bytes through a general purpose I/O pin. Input and output levels, as well as baud and pin information is determined by the current contents of the rs_param register. This function will wait approximately (4096us * wait) for a start bit before returning an error. A zero value for wait will cause an indefinite wait for a start bit and indefinate wait for all characters in a message. The return value indicates if there was an error, and if so how many characters were received. If the return value is 0, no errors occured and the entire block was received. If an error occurs, the return value will be 128 les the number of bytes not received.

The address byte is the byte to match before bytes are captured into the buffer. The buffer is an array of bytes that must be at least as large as buf_size to prevent adjacent memory from being overwritten. The rs_recblock function will continue to capture serial data until either the function times out or the buffer is filled.

The control parameter contains information about handshake, block qualifying and message timing. The lower four bits of the control byte are the handshake pin. If bit 7, the most significant bit is set, the rs_recblock will wait for a break on the line before receiving a block. If bit 6 is set, rs_recblock will wait for a byte that matches the address byte before receiving the block. If bit 5 is set, rs_recblock will wait for 32*buf_size character attempts otherwise rs_recblock will wait for 8*buf_size character attempts. If bit 4 is set, rs_recblock will assert the handshake line, otherwise the handshake pin will remain unchanges by rs_recblock. In order for proper break detection to work, the RS system must be set to check the stop bit using rs_stop_chek function.

rs_string Send a string of bytes out RS232 pin

`none rs_string( word string_addr ) rs_str62.lib, rs_str57.lib`

This function sends a string of characters located in EEprom out of a general purpose pin in RS232 serial format. The pin, baud_rate, and levels are defined by the rs_param_set function. The string must be null-terminated.

rs_delay Delay one and one half RS232 bit times

`none rs_delay() token.lib`

Delay one and one half bit time. Use this function to produce the minimum required delay when sending a serial byte on the same pin just used to receive serial information. This delay is required to prevent a framing error or data overrun. Additional time delay may be required if the sending device will need to do any processing before being ready to receive serial data. The baud rate used to calculate the delay time is contained in the rs_param register.

rs_stop_chek Set RS232 stop bit protocol on

`none rs_stop_chek() token.lib`

This function causes the level of the stop bit to be checked after each RS232 byte is received. Framing errors can only be detected by checking the stop bit. Additional time is required to check the stop bit, however. Some programs may wish to ignore the stop bit to gain more time for handling continuous serial information.

rs_stop_ignore Set RS232 stop bit protocol off

`none rs_stop_ignore() token.lib`

This function causes the level of the stop bit to be ignored after each RS232 byte is received. Additional time for processing continuous serial information is available by ignoring the stop bit. Framing errors can only be detected by checking the stop bit, however.

rs_fmt Sends a formatted long out RS232 pin

`none rs_fmt( long value, word form ) rs_fmt57.lib, rs_fmt62.lib`

This function formats the long argument value into a string of characters on the basis of the string form. This function allows control of leading and trailing zeros, decimal point placement, and dollar sign. The format string is a null terminated string contained in EEprom. The characters that have special meaning are as follows:

$	Print a '$' character in the output
#	Print a number if this or a previous digit was non-zero
0	Print a number even zero, forces following #'s to print
X	Do not print a number digit, but account for its position
.	Print a decimal point

Examples:

```
; listen for a break with this node's address

FUNCTION none wait_addr
    LOCAL byte errorval
    LOCAL byte rs_buffer[32]
BEGIN
    rs_stop_check()              ; test stop bit for valid
    rs_param_set ( rs_invert | rs_2400  | pin_a2 )
                 ; inverted, 2400, pin 10
    REP
        =( errorval, rs_recblock( 100b,~
         ~ rs_cont_brk | rs_cont_addr | rs_cont_wait, ~
         ~ 'A', rs_buffer, 32b ))
        IF errorval
        ELSE
            ; message received OK
             proc_pack()  ; process serial packet
        ENDIF
    LOOP
ENDFUN
```

6.14 Console Functions

The console functions use the internal serial I/O routines of the token interpreter to communicate to a console computer. The console functions use the information contained in the rs_param_byte to determine which pin, baud rate, and line levels to use for the communication. To communicate with the supplied software, the line must be inverted, and the baud rate must be 9600 baud. Furthermore, to use the debugger console, only pin 15 (DL) can be used for the console pin.

The console functions perform special handshaking to ensure that the TICkit is listening while the console sends data to it. Therefore, these routines should not be used to send non-TICkit protocol serial information. Use the serial communications functions listed above for that purpose.

con_test Test for the existance of a console

`byte con_test() token.lib`

Returns zero to indicate a console listening on the console pin. Any other value returned indicates that no console is listening.

con_in_char Get a character from console (TICkit57)

```
byte con_in_char( word wait ) token.lib
```

Get an ASCII character from the console. The "wait" value indicates that the function should wait for only wait*16us interval. This produces a maximum delay of approximately one second. A zero for wait, or a value greater than 65280 will cause the function to wait indefinitely for input. Any characters typed on the console are not echoed locally by the console.

con_in_char Get a character from console (TICkit62)

```
byte con_in_char( byte wait ) token.lib
```

Get an ASCII character from the console. The "wait" value indicates that the function should wait for only wait*4096us interval. This produces a maximum delay of approximately one second. A zero for wait causes the function to wait indefinitely for input. Any characters typed on the console are not echoed locally by the console.

con_in_byte Get a byte from the console (TICkit57)

```
byte con_in_byte( word wait ) token.lib
```

Get a value of size byte from the console. This function waits the same as the con_in_char function. Digits typed on the console while entering the number are echoed locally within console.

con_in_byte Get a byte from the console (TICkit62)

```
byte con_in_byte( byte wait ) token.lib
```

Get a value of size byte from the console. This function waits the same as the con_in_char function. Digits typed on the console while entering the number are echoed locally within console.

con_in_word Get a word from the console (TICkit57)

```
word con_in_word( word wait ) token.lib
```

Get a value of size word from the console. This function waits the same as the con_in_char function. Digits typed on the console are echoed locally.

con_in_word Get a word from the console (TICkit62)

```
word con_in_word( byte wait ) token.lib
```

Get a value of size word from the console. This function waits the same as the con_in_char function. Digits typed on the console are echoed locally.

con_in_long Get a long from the console (TICkit57)

```
long con_in_long( word wait ) token.lib
```

Get a value of size long from the console. This function waits the same as the con_in_char function. Digits typed on the console are echoed locally.

con_in_long Get a long from the console (TICkit62)

```
long con_in_long( byte wait ) token.lib
```

Get a value of size long from the console. This function waits the same as the con_in_char function. Digits typed on the console are echoed locally.

con_out_char Send a byte character to the console

```
none con_out_char( byte data ) token.lib
```

Send an ASCII character to a console. The byte "data" is sent to the console with instructions to the console to display it as an ASCII character.

con_out Sends a numeric value to the console

```
none con_out( byte data ) token.lib
none con_out( word data ) token.lib
none con_out( long data ) token.lib
```

The value "data" is displayed on the console in decimal format. Long values are signed using the two's complement convention.

con_string Send a string of bytes out console pin

```
none rs_string( word string_addr ) cn_str.lib
```

This function sends a string of characters located in EEprom out of a general purpose pin in console serial format. The pin, baud_rate, and levels are defined by the rs_param_set function. The string must be null-terminated. The Debugger can recieve this type of signal.

con_fmt Sends a formatted long to the console

```
none con_fmt( long value, word format ) cn_fmt.lib
```

This function formats the long argument value into a string of characters on the basis of the string format. This function allows control of leading and trailing zeros, decimal point placement, and dollar sign. The format string is a null terminated string contained in EEprom. The characters that have special meaning are as follows:

$	Print a '$' character in the output
#	Print a number if this or a previous digit was non-zero
0	Print a number even zero, forces following #'s to print
X	Do not print a number digit, but account for its position
.	Print a decimal point

Examples:

```
; test for a console and add two signed numbers

FUNCTION none main
    LOCAL long val1
    LOCAL long val2
BEGIN
    rs_param_set( debug_pin )
    IF not( con_test())
        =( val1, con_in_long( 0 ))
        =( val2, con_in_long( 0 ))
        con_out( +( val1, val2 ))
    ENDIF
ENDFUN
```

6.15 System, Interrupt and Miscellaneous Functions

The system functions are used to break or re-establish communication with the debugger, to control the interrupt detection, and to reset the TICkit under software control. Using the debug_on() function while developing a program can be a very useful method of tracing a program. Often, the programmer is only wishing to trace a small section of a program. By placing the debug_on() function at the beginning of the section, the user can allow the program to operate at full speed until it reaches the desired section, then the user can single step or watch variables for just the code in question. When done tracing that section, the user can press 'E' of the debugger and the code will again execute at full speed. This is often much faster than running a program in monitor code while looking for a break point.

The interrupt capability of the TICkit allows a special function in the program to be called at the request of an external device. Provided that the interrupts have been enabled, when the /IRQ input line of the TICkit is brought low, immediately after the current TICkit token finishes executing, the function named IRQ will be executed (vectored at EEprom location 0x0002 and 0x0003). The interrupt is disabled as the IRQ function is called. Therefore, the program must re-enable it to sense any additional interrupts. Usually the interrupts are re-enabled as the last line of the IRQ routine. The TICkit62 has the added interrupt capability to sense multiple events caused by internal hardware stimulus. For example, an interrupt can be generated when the Timer1 16bit counter rolls over. This interrupt is useful to implement a real-time clock in background. For the TICkit62 there are three interrupt vectors:

1. /IRQ pin input - when line is brought low and interrupt occurs (57 and 62)
2. Stack overflow - when RAM memory is exceeded this interrupt occurs (62)
3. Internal peripheral - when a pre-programmed peripheral condition occurs (62)

The names of the functions call by each of these vectors is defined in the token library for each processor. By default they are: "irq", "stack_overflow", and "global_int" respectively.

The internal peripheral interrupt (global_int) can be caused by multiple event sources. To help limit what events can cause this interrupt, and to determine what event caused an interrupt while servicing it, mask registers and flag registers are used. Each bit of the mask and flag registers coorespond to an event source. An event source will cause an interrupt only if its cooresponding mask bit is set, otherwise that source is ignored. Likewise, if multiple event sources are allowed to generate the interrupt, the servicing routine will need to determine which source caused this interrupt. This is done by examining the contents of the flag registers. Only the event which caused the interrupt will have its cooresponding bit set. Also, the flag for the interrupt source will need resetting at the end of the interrupt service routine.

debug_on Turn debug protocol on

`none debug_on() token.lib`

Attempts to establish a connection to a debugger on the console computer. The TICkit will try to establish this connection for approximately 1.5 seconds.

debug_off Turn debug protocol off

`none debug_off() token.lib`

Terminates the debug connection with the console. Forces the program to execute in fast, un-monitored mode.

irq_on Turn interrupt sensing on

`none irq_on() token.lib`

Enables the Interrupt Service Requests. A low level on the /IRQ line will cause execution to resume at the function named "IRQ" immediate following the execution of the current token. Because the interrupt service flag is disabled by each service request, the service request routine will normally re-enable interrupts on exit by executing this function. This function also enables hardware interrupt processing on the TICkit62. Therefore, when using this function, be sure that all internal global interrupts are disabled, or the appropriate "global_int" function exists for handling TICkit62 interrupts.

irq_off Turn interrupt sensing off

`none irq_off() token.lib`

Disables the Interrupt Request line. Use this function to prevent any interrupt from distracting the TICkit from a time sensitive program.

reset Resets the token interpreter

`none reset() token.lib`

Simulates a power on start.

int_cont_set Sets control byte for global_int (TICkit62)

`none int_cont_set( byte control_bits ) token.lib`

This function sets the bits of the TICkit62's global interrupt control register. This register contains the status and mask of several interrupts and also masks the peripheral interrupts. The bit assignments for the interrupt control register are as follows (standard defines for these detailed in the DEFINES section) :

bit 0 = Set if bits 4 thru 7 of the Data Port have changed.
bit 1 = Set if bit 0 of the Data Port has received an edge.
bit 2 = Set if RTCC (tmr0) has overflowed.
bit 3 = Enable Data port bits 4-7 change interrupt
bit 4 = Enable Data port bit 0 edge interrupt
bit 5 = Enable RTCC (tmr0) overflow interrupt
bit 6 = Enable peripheral interrupt sources.
bit 7 = Unused, must be set to zero.

int_cont_get Gets control byte for global_int (TICkit62)

`byte int_cont_get() token.lib`

This function gets the bits of the TICkit62's global interrupt control register. This register contains the status and mask of several interrupts and also masks the peripheral interrupts.

int_flag_set Sets Peripheral Flag byte (TICkit62)

`none int_flag_set( byte flags ) token.lib`

This function sets the bits of the TICkit62's peripheral interrupt flag register. This register contains the status of peripheral interrupts. Usually, this function is simply used to clear a serviced interrupt. Defines for each bit's meaning are listed in the DEFINES section of this chapter. The bits are as follows:

bit 0 = Timer 1 overflow.
bit 1 = Timer 2 overflow.
bit 2 = CCP1 module interrupt (Capture or Compare)
bit 3 = SSP module (I2C port I/O)
bit 4 = not used
bit 5 = not used
bit 6 = not used
bit 7 = not used

int_flag_get Gets Peripheral Flag byte (TICkit62)

`byte int_flag_get() token.lib`

This function gets the bits of the TICkit62's peripheral interrupt flag register. This register contains the status of peripheral interrupts. This function is used to determine which peripheral interrupts are pending and need service.

int_mask_set Sets Peripheral Mask byte (TICkit62)

`none int_mask_set( byte mask ) token.lib`

This function sets the bits of the TICkit62's peripheral interrupt mask register. This register contains the masks of peripheral interrupts. Only the devices which their bits set will generate an interrupt. Bits map the same as the peripheral flag register.

int_mask_get Gets Peripheral Mask byte (TICkit62)

`byte int_mask_get() token.lib`

This function gets the bits of the TICkit62's peripheral interrupt mask register. This register contains the masks of peripheral interrupts.

Examples:

```
; handle an interrupt - also uses debug_on to create a type
; of fast break point. Program executes at full speed until
; debug_on.

FUNCTION none irq      ; this irq handler will display the
                      ; string then connect to a debugger
                      ; if it is present, then, under debug
                      ; control, return to the main process.
BEGIN
    con_string( "responding to interrupt\r\l" );
    debug_on()
    irq_on()
ENDFUN

FUNCTION none main
BEGIN
    rs_param_set( debug_pin )
    irq_on()
    REP
    LOOP
ENDFUN
```

6.16 Peripheral Control Functions

The processors on which the FBASIC interpreters are implemented have special I/O resources for performing more complex tasks. These resources, called peripherals, can operate while the main processing function is doing something else. The TICkit 57 has the RTCC (Tmr0) as its only peripheral device. As an example, the RTCC can count pulses or clock cycles while the program continues to operate. The Functions used to control these devices would not normally be considered part of a standard library. However, because of the inteded use of the processors for control applications, the functions controlling the peripherals are assumed to be central to the task. For this reason, peripheral control functions are included in FBASIC's standard library. Because the availability of peripherals is very processor dependent, be sure to code with only the resources of the processor you will eventually use.

The TICkit 57 has only the RTCC for a peripheral. It's functions are outlined in the timing section of this chapter.

The TICkit 62 has the RTCC timer, but it also has two more timers, a module which can be programed to compare the count in timer 1 with a preset value and interupt the processor on a match, or it can capture the count of timer 1 when the CCP pin is activated, or it can use timer2 to generate a 10bit PWM signal and output that signal on the CCP pin. The TICkit 62 also has an SSP (synchronys serial port) for use as an I2C port. All of these resources are controlled by writing special control and data registers. Once setup, the peripheral devices operate in background while the program proceeds.

tmr1_cont_set Sets TMR1 control register (TICkit62)

`none tmr1_cont_set( byte control_bits ) token.lib`

This function sets the bits of the TICkit62's timer 1 control register. The meanings of the bits of this register are as follows:

bit 0 = Enables timer1 counter
bit 1 = When set clk is A0 pin, otherwise clk is OSC/4
bit 2 = Synchronizes clk with OSC when set
bit 3 = Enables oscillator circuit on A0 and A1
bit 4 = Bits 4 and 5 select prescale value of 8(11), 4(10),
bit 5 = 2(01) or 1(00)
bit 6 = not used
bit 7 = not used

tmr1_cont_get Gets TMR1 control register (TICkit62)

`byte tmr1_cont_get() token.lib`

This function gets the bits of the TICkit62's timer 1 control register.

tmr1_count_set Sets TMR1 count (TICkit62)

`none tmr1_count_set( word count ) token.lib`

This function sets the count of the TICkit62's timer 1.

tmr1_count_get Gets TMR1 count (TICkit62)

`word tmr1_cont_get() token.lib`

This function gets the count of the TICkit62's timer 1.

tmr2_cont_set Sets TMR2 control register (TICkit62)

`none tmr2_cont_set( byte control_bits ) token.lib`

This function sets the bits of the TICkit62's timer 2 control register. The meanings of the bits of this register are as follows:

bit 0 = Bits 0 and 1 select prescale value of 1(00), 4(01)

bit 1 = or 16(1x)
bit 2 = Enables timer 2 counting
bit 3 = Bits 3,4,5, and 6 select the postscale divisor
bit 4 = 0000 is divide by 1
bit 5 = while 1111 is divide by 16
bit 6 = Therefore, divisor = postscale setting + 1
bit 7 = not used

tmr2_cont_get Gets TMR2 control register (TICkit62)

`byte tmr2_cont_get() token.lib`

This function gets the bits of the TICkit62's timer 2 control register.

tmr2_count_set Sets TMR2 count (TICkit62)

`none tmr2_count_set( byte count ) token.lib`

This function sets the count of the TICkit62's timer 2.

tmr2_count_get Gets TMR2 count (TICkit62)

`byte tmr2_count_get() token.lib`

This function gets the count of the TICkit62's timer 2.

tmr2_period_set Gets TMR2 period register (TICkit62)

`none tmr2_period_set( byte period ) token.lib`

This function sets the contents of the TICkit62's timer 2 period register.

tmr2_period_get Gets TMR2 period register (TICkit62)

`byte tmr2_period_get() token.lib`

This function gets the contents of the TICkit62's timer 2 period register.

ccp1_cont_set Sets CCP1 control register (TICkit62)

`none ccp1_cont_set( byte control_bits ) token.lib`

This function sets the bits of the TICkit62's CCP1 control register. The meanings of the bits of this register are as follows:

bit 0 = Bits 0,1,2 and 3 select the mode of the CCP
bit 1 = 0000 = off, 01xx = capture mode
bit 2 = 10xx = compare mode, 11xx = PWM mode
bit 3 = bits 0,1 modify capture and compare modes.
bit 4 = In PWM mode this is the lowest order duty bit
bit 5 = In PWM mode this is the next lowest order bit
bit 6 = not used
bit 7 = not used

ccp1_cont_get Gets CCP1 control register (TICkit62)

`byte ccp1_cont_get() token.lib`

This function gets the bits of the TICkit62's CCP1 control register.

ccp1_reg_set Sets CCP1 register (TICkit62)

`none ccp1_reg_set( word contents ) token.lib`

This function sets the contents of the TICkit62's CCP1 register. Depending on the mode of the CCP this can be a comparison value for timer1, a result of a capture on timer1, or the lower 8 bits of the CCP1 register are the higher 8 bits of the PWM duty cycle.

ccp1_reg_get Gets CCP1 register (TICkit62)

`byte ccp1_reg_get() token.lib`

This function gets the contents of the TICkit62's CCP1 register.

ssp_cont_set Sets SSP control register (TICkit62)

`none ssp_cont_set( byte control_bits ) token.lib`

This function sets the bits of the TICkit62's SSP control register. The meanings of the bits of this register are as follows:

bit 0 = Bits 0,1,2 and 3 select the mode of the SSP
bit 1 = 01xx = slave only modes
bit 2 = 10xx = master support with slave modes
bit 3 = See defines for complete mode list.
bit 4 = Clk enable (allows clk to go high)
bit 5 = Enable the SSP (switches control of A3 and A4)
bit 6 = Receive Overflow flag
bit 7 = Write Collision detected

ssp_cont_get Gets SSP control register (TICkit62)

`byte ssp_cont_get() token.lib`

This function gets the bits of the TICkit62's SSP control register.

ssp_buffer_set Sets SSP Buffer (TICkit62)

`none ssp_buffer_set( byte contents ) token.lib`

This function sets the contents of the TICkit62's SSP buffer. Effectively, this function is used to transmit data on the I2C port.

ssp_buffer_get Gets SSP Buffer (TICkit62)

`byte ssp_buffer_get() token.lib`

This function gets the contents of the TICkit62's SSP buffer. This reads data received from the I2C port.

ssp_addr_set Sets SSP Address (TICkit62)

`none ssp_addr_set( byte contents ) token.lib`

This function sets the contents of the TICkit62's SSP address register. Interupts and data reception/transmission only takes place after a start bit and a match to this address on the I2C port.

ssp_addr_get Gets SSP Address (TICkit62)

`byte ssp_addr_get() token.lib`

This function gets the contents of the TICkit62's SSP address register.

ssp_status_get Gets SSP Status (TICkit62)

`byte ssp_status_get() token.lib`

This function gets the contents of the TICkit62's SSP status register. The meanings of the bits of this register are as follows:

bit 0 = Receive Buffer full (byte in buffer for reading)
bit 1 = Update Address required (place in ssp_address)
bit 2 = Current message is a read message
bit 3 = Start bit was detected last
bit 4 = Stop bit was detected last
bit 5 = Last byte received was a data byte (not an address)
bit 6 = not used
bit 7 = not used

Examples:

```
; Sample program to illustrate using TICkit62 SSP to do
; I2C slave operations. Keep in mind that the master in this
; system must transmit data with spacing between bytes. If
; using a TICkit as the master, use the sim_i2c library to
; generate the signals.

DEF tic62_a
LIB fbasic.lib

GLOBAL byte iic_addr 0b
GLOBAL byte iic_comm 0b

FUNC none irq
BEGIN
    irq_on()
ENDFUN
```

```
FUNC none global_int
    LOCAL byte iic_data
BEGIN
    =( iic_data, ssp_buffer_get())
    IF b_and( ssp_stat_get(), ssp_stat_data )
        IF iic_comm
                con_out_char( '\r' )
                con_out_char( '\l' )
                con_out_char('A')
                con_out( iic_addr )
                con_out_char( ' ' )
                con_out( iic_comm )
                con_out_char( ' ' )
                con_out( iic_data )
        ELSE
            =( iic_comm, iic_data )
        ENDIF
    ELSE
        IF b_and( ssp_stat_get(), ssp_stat_read )
            ssp_buffer_set( 0x18b )
        ELSE
            =( iic_addr, iic_data )
            =( iic_comm, 0b )
        ENDIF
    ENDIF

    ssp_cont_set( ssp_mode_slave7 | ssp_con_clken |~
       ~ssp_con_enable )
    int_flag_set( 0b )
    irq_on()
ENDFUN

FUNC none main
BEGIN
    rs_param_set( debug_pin )
    int_cont_set( int_con_periphe )
    int_mask_set( int_mask_ssp )
    int_flag_set( 0b )

    ssp_addr_set( 0x80b )
    ssp_cont_set( ssp_mode_slave7 | ssp_con_clken |~
       ~ssp_con_enable )
    irq_on()

    REP
    LOOP
ENDFUN
```

Examples:

```
; This program will generate a square wave of varying duty
; cycle on the CCP1 pin. This method is used to perform PWM
; control of motors etc.

DEF tic62_a
LIB fbasic.lib

GLOBAL word duty    ; only the lower 8 bits are used.

FUNC none main
BEGIN
    rs_param_set( debug_pin )
    =( duty, 0 )
    pin_low( pin_a2 )
    tmr2_cont_set( tmr2_con_on )
    tmr2_period_set( 255b )          ; determines frequency
    ccp1_cont_set( ccp_pwm )
    REP
        ccp1_reg_set( duty )
        delay( 10 )
        ++( duty )
    LOOP
ENDFUN
```

6.17 Constant Symbols Defined in Libraries

```
      DEFINE buss_8bit 0y10000000b
      DEFINE buss_4two 0y01000000b
      DEFINE buss_4bit 0y00000000b

      DEFINE debug_pin 0xDFb
      DEFINE rs_invert  0x80b
      DEFINE rs_19200  0x60b
      DEFINE rs_9600   0x50b
      DEFINE rs_4800   0x40b
      DEFINE rs_2400   0x30b
      DEFINE rs_1200   0x20b
      DEFINE rs_600    0x10b
      DEFINE rs_300    0x00b

      DEFINE rs_cont_brk  128b
      DEFINE rs_cont_addr 64b
      DEFINE rs_cont_wait 32b
      DEFINE rs_cont_hand 16b

      DEFINE pin_A7     0x0Fb
      DEFINE pin_A6    0x0Eb
      DEFINE pin_A5    0x0Db
      DEFINE pin_A4    0x0Cb
      DEFINE pin_A3    0x0Bb
      DEFINE pin_A2    0x0Ab
      DEFINE pin_A1    0x09b
      DEFINE pin_A0    0x08b

      DEFINE pin_D7    0x07b
      DEFINE pin_D6    0x06b
      DEFINE pin_D5    0x05b
      DEFINE pin_D4    0x04b
      DEFINE pin_D3    0x03b
      DEFINE pin_D2    0x02b
      DEFINE pin_D1    0x01b
      DEFINE pin_D0    0x00b

      DEFINE false     0x00b
      DEFINE true      0xFFb

      DEF tmr1_con_on   0y00000001b  ; turn on timer1
      DEF tmr1_con_ext  0y00000010b  ; external, rising edge source
      DEF tmr1_con_sync 0y00000100b  ; synchronize to osc clk
      DEF tmr1_con_osc  0y00001000b  ; enable oscillator
                                     ; (inverter and feedback)
      DEF tmr1_con_pre1 0y00000000b  ; prescaler divide by 1
```

```
DEF tmr1_con_pre2 0y00010000b  ; prescaler divide by 2
DEF tmr1_con_pre4 0y00100000b  ; prescaler divide by 4
DEF tmr1_con_pre8 0y00110000b  ; prescaler divide by 8

DEF tmr2_con_on     0y00000100b  ; turn on timer2
DEF tmr2_con_pre1   0y00000000b  ; prescaler divide by 1
DEF tmr2_con_pre4   0y00000001b  ; prescaler divide by 4
DEF tmr2_con_pre16  0y00000010b  ; prescaler divide by 16
DEF tmr2_con_post   0y01111000b  ; mask for postscaler
                                 ; (divide by value)

DEF ssp_mode_slave7  0y00000110b  ; slave only - 7bit address
DEF ssp_mode_slave10 0y00000111b  ; slave only - 10bit
DEF ssp_mode_master  0y00001011b  ; master support
                                  ; - slave disabled
DEF ssp_mode_mast7   0y00001110b  ; master support
                                  ; - slave 7bit address
DEF ssp_mode_mast10  0y00001111b  ; master support
                                  ; - slave 10bit address
DEF ssp_con_clken    0y00010000b  ; clock enable
                                  ; ( not held low )
DEF ssp_con_enable   0y00100000b  ; SSP module enabled
DEF ssp_con_overflow 0y01000000b  ; indicates receiver overflow
DEF ssp_con_collide  0y10000000b  ; collision during
                                  ; write to transmit reg

DEF ssp_stat_full    0y00000001b  ; receive buffer is full
DEF ssp_stat_addr10  0y00000010b  ; 10 bit address to be read
DEF ssp_stat_read    0y00000100b  ; current buss cycle is read
DEF ssp_stat_start   0y00001000b  ; IIC start bit last received
DEF ssp_stat_stop    0y00010000b  ; IIC stop bit last received
DEF ssp_stat_data    0y00100000b  ; data byte in register
                                  ; (not address)

DEF ccp_off          0y00000000b
DEF ccp_capt_fall    0y00000100b
DEF ccp_capt_rise    0y00000101b
DEF ccp_capt_rise4   0y00000110b
DEF ccp_capt_rise16  0y00000111b
DEF ccp_comp_set     0y00001000b
DEF ccp_comp_clear   0y00001001b
DEF ccp_comp_int     0y00001010b
DEF ccp_comp_event   0y00001011b ; reset timer1 for CCP1
                                 ; - start A/D for CCP2
DEF ccp_pwm          0y00001100b
DEF ccp_pwm_bit0     0y00010000b
DEF ccp_pwm_bit1     0y00100000b
```

```
      DEF int_con_periphe 0y01000000b  ; all other peripherals enable
      DEF int_con_tmr0e   0y00100000b  ; timer 0 overflow enable
      DEF int_con_pind0e  0y00010000b  ; pin_d0 interrupt enable
      DEF int_con_portde  0y00001000b  ; data port change enable
      DEF int_con_tmr0f   0y00000100b  ; timer 0 overflow flag
      DEF int_con_pind0f  0y00000010b  ; pin_d0 interrupt flag
      DEF int_con_portdf  0y00000001b  ; data port change flag

      DEF int_flag_ssp   0y00001000b  ; mask for SSP (IIC) port
      DEF int_flag_ccp1  0y00000100b  ; mask for CCP1 sources
                                      ; (compare or capture)
      DEF int_flag_tmr2  0y00000010b  ; mask for timer2 roll-over
      DEF int_flag_tmr1  0y00000001b  ; mask for timer1 roll-over
```

7 The Console Program

7.1 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 "consol**e**xe" 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.

7.2 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.

8 The Debug Program

8.1 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.

8.2 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.

8.3 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 Watch points. 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 Break points. 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 tracethrough 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.

9 The Compiler Program

9.1 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.

9.2 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.

9.3 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.

9.4 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
```

9.5 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.

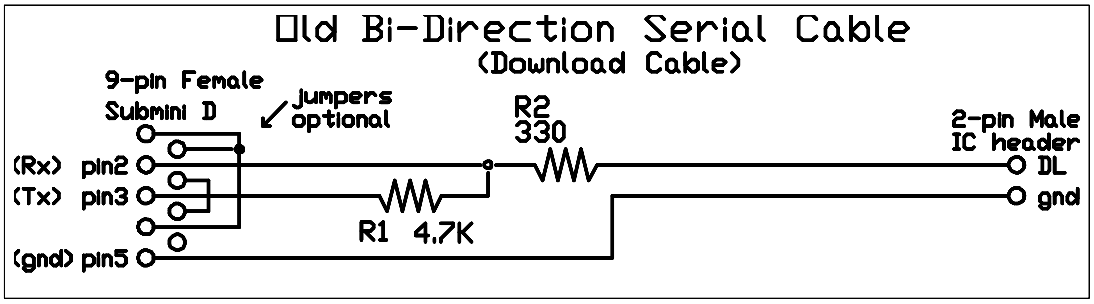
9.6 Compiler Method of Setting Break and Watch Points

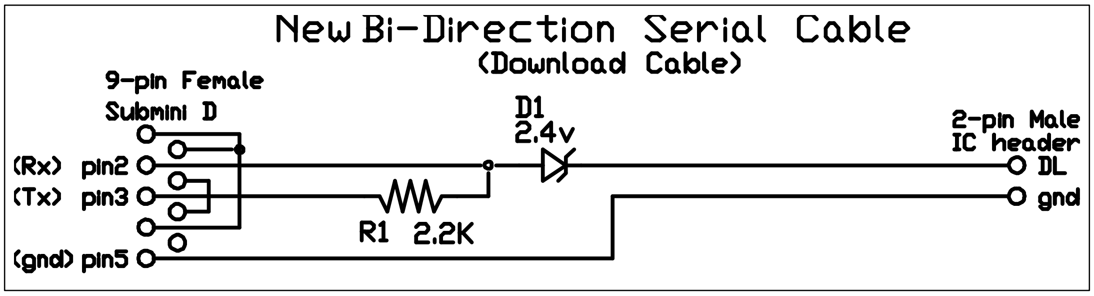
The compiler can also set default Break and Watch points in the symbol file. Use the keyword, "BREAK", at the beginning of any procedural line to associate a break point with that line. This keyword has absolutely no effect on the token file, but places a '+" in the symbol file at that line. The line 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: Circuits

A.1 Download Cable(s)

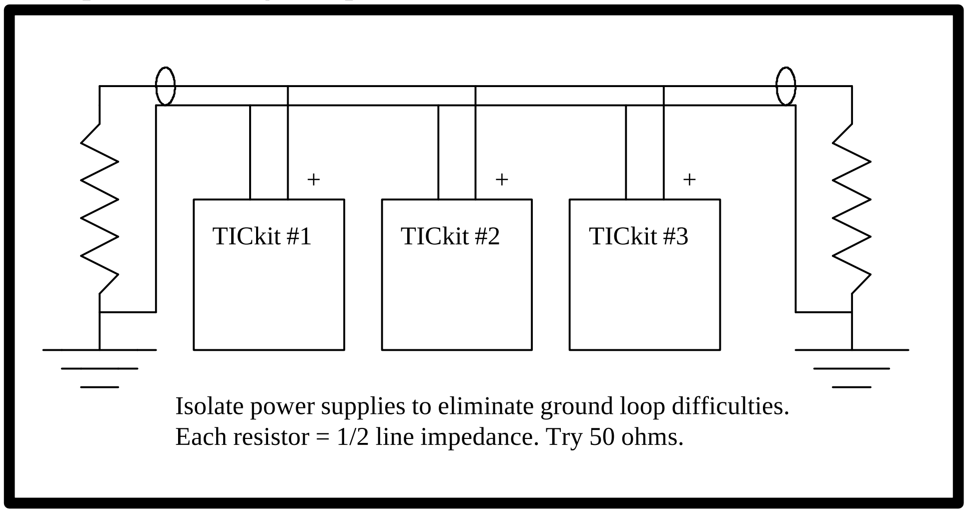




Pin 3 of the Consolecomputer's 9 pin serial port is a transmit pin. When the Console is not transmitting data, this pin will be low (-9 to -12 volts). This is an RS232 idle or stop bit state. The 4.7K ohm resistor acts as a pull down resistor to cause Pin 2 of the Console's port to see an idle state, also. Pin 2 is the receive line for the console. The + pin of the DL port on the TICkit will also see the -9 volt signal, but will shunt it to ground via the 330 ohm current limiting resistor. Either the TICkit or the Console can raise the voltage on the data line by simply transmitting data. When the TICkit transmits data, a voltage divider is formed between the PIC's output and the output of the Console's RS232 output. Because the leg of the divider to the Console's output has a much greater resistance, the PIC's output has priority over the Console's output.

When using this type of bi-directional data cable, The TICkit must be programmed to invert the RS232 signal. The TICkit will use an open source output causing low outputs to be "high impedance", while high outputs will be approximately 5 volts.

A.2 Multi-drop connection of multiple TICkits.


+
+
+
TICkit #1
TICkit #2
TICkit #3
Isolate power supplies to eliminate ground loop difficulties.
Each resistor = 1/2 line impedance. Try 50 ohms.

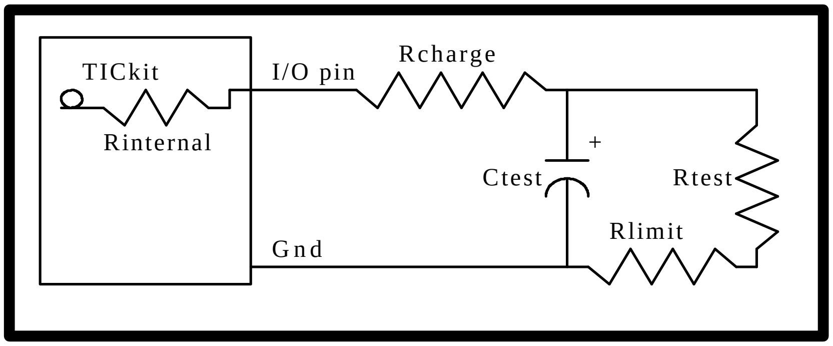
Multiple TICkits can be connected together using a shared wire configuration. By matching the pull down resistance to the characteristic impedance of the transmission line, long lengths can separate TICkits while maintaining good data connection. An example of this type of connection is shown above. At 9600 baud, reliable communication can be expected up to 1000 feet. Longer lengths can be acheived using lower baud rates and/or better terminations.

This type of connection also requires that the RS232 configuration use the inverted option. The user can adopt a protocol that uses framing errors to identify message addresses. By enabling stop bit interrogation, the TICkit RS232 serial library can be made to generate, as well as detect, framing errors. Using this sort of "9 bit" technique allows message headers to contain a special byte with a framing error to distinguish the header from the data stream.

The TICkit 62 has a special function just for doing this type of network communication called rs_recblock. The example below illustrates the program lines necessary for this type of communication.

```
; Sending program for a TICkit 62

=( index, 0b )
rs_break ()              ; send a break level
rs_send( 3b )           ; send the node address. In this case,
                        ; send to node 3 (note node 0 should not
                        ; be used as this may be implemented as
                        ; a broadcast to all nodes address in
                        ; the future.
REP
    rs_send( buffer[ index ] )
                        ; data to be sent is containded in the
                        ; array buffer with 10 bytes
    ++( index )
UNTIL >=( index, 10b )

; receiving program fragment
WHILE rs_recblock ( 0b, rs_cont_brk, 3b, buffer, 10b )
    ; the above will continue to loop until a 10 byte
    ; block is received without errors addressed to
    ; node 3. The resulting data will reside in buffer
    ; for use by the rest of the program.
LOOP
```

A.3 The RC measurement Circuit



This circuit, coupled with the rc_measure function in the TICkit standard library, will measure either a capacitor, a resistor, or both using a timing method. Rcharge and Rlimit resistors may be omitted, but are useful for discussion purposes to preventing boundary problems. As a rule of thumb, the product of Ctest multiplied by the sum of Rtest and Rlimit should equal one where the value of Ctest is in farads, and the values of Rtest and Rlimit are in ohms. Therefore, a value for Ctest of 10uf, a 100k ohm value for Rtest and a value of 0 for Rlimit will produce approximately a 16 bit count range. Accuracy with this measurement method can vary from tenths of percents at high R and C values to 5 percent for low R and C values. For this reason, using an Rlimit resistor of 1k ohms can generate higher accuracy. Any count offset introduced as a result of Rlimit can be compensated for by subtracting a constant from the resulting counts.

The RC measurefunction works by assuming that the capacitor is mostly discharged. This assumption will be true provided that the pin was either held low for a short period, or if an RC measurement was the last I/O function on this. The routine then charges the capacitor by internally connecting the pin to a high logic level. The capacitor will charge rapidly with only the internal resistance and any Rcharge resistance to slow its rate of charge. The routine monitors the voltage on the output pin approximately every 10us. When the routine sees a high level voltage, the pin is held high for an additional 768us. The pin is switched to an input and the time until the voltage on the capacitor falls to a low level is the value returned as the RC measurement. The count is fairly linear with respect to the Rtest and Ctest values, however, there are sources of error.

First, the initial threshold is only accurate to the RC measure routines ability to see the instant the capacitor is charged to a high level. Because the pin is only sampled every 10us, there is a window of error. By increasing the Rcharge resistance or by using a larger capacitor, the effect of this inaccuracy can be minimized. The side effect of increasing these values is that it takes longer for the entire measurement to be performed. Also, if the charging time is longer than .65535 seconds, a 0 will be returned from the function.

Another source of error is caused by the divider formed between the Rtest and the Rinternal. If Rtest is very low, the charging voltage may actually be less than the high threshold voltage. This will prevent the Capacitor from charging to the high threshold. When this happens, the entire measurement takes too long, and 0 is returned. By using an Rlimit of approx. 1K ohms, this possibility is minimized.

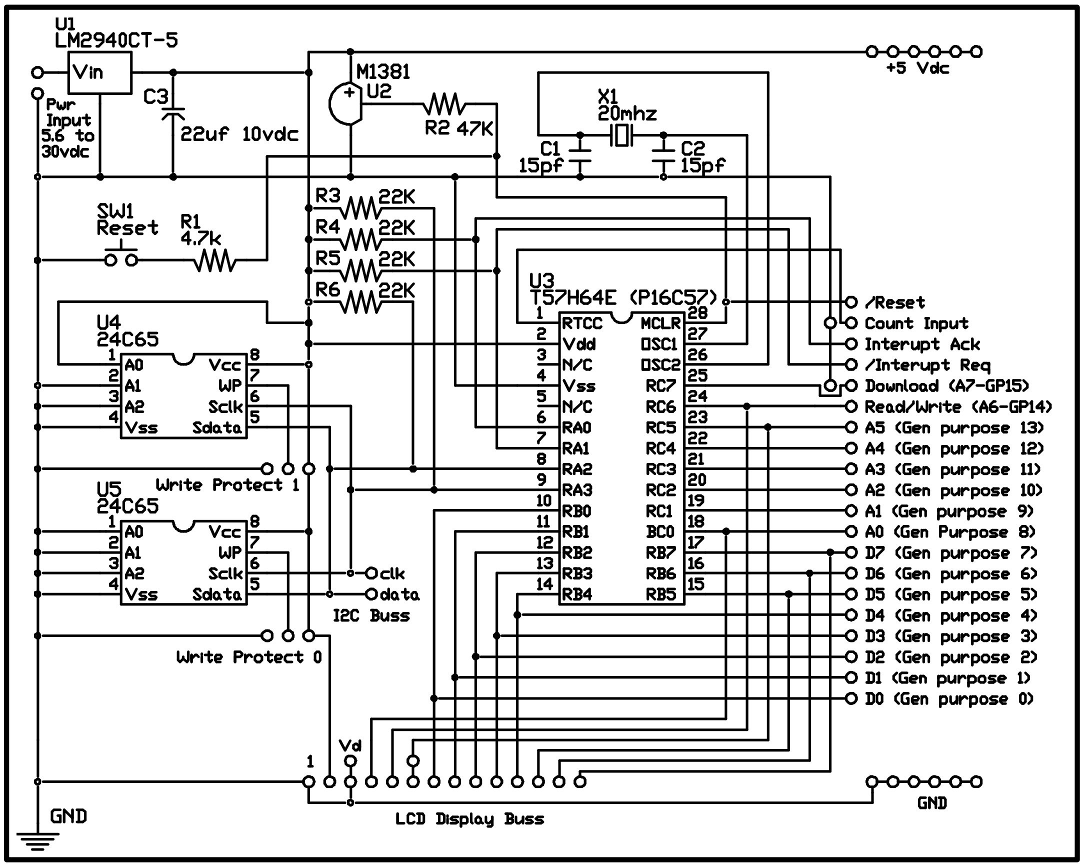
Finally, the Rinternal value and the threshold for a high or low level on the pin is not precise. The PIC was not designed to be a comparitor, so there will be shifts due to environmental conditions. The RC measurement routine is useful for qualitative results, but the user must exercise caution to ensure the required accuracy of data when using this routine.

Examples:

```
; This program repeatedly measures the RC network and displays
; the result on the console. In this example a 10K pot was
; used with a 10uf capacitor.

DEF tic62_a
LIB fbasic.lib

FUNC none main
BEGIN
    rs_param_set( debug_pin )
    REP
        con_out(  rc_measure( pin_a0 ))
        con_out_char( '\r' )
        con_out_char( '\l' )
        delay( 100 )
    LOOP
ENDFUN
```

Appendix B: TICkit57 Hardware

B.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 EEprom configuration, all lines A0, A1, and A2 should be grounded, as shown.

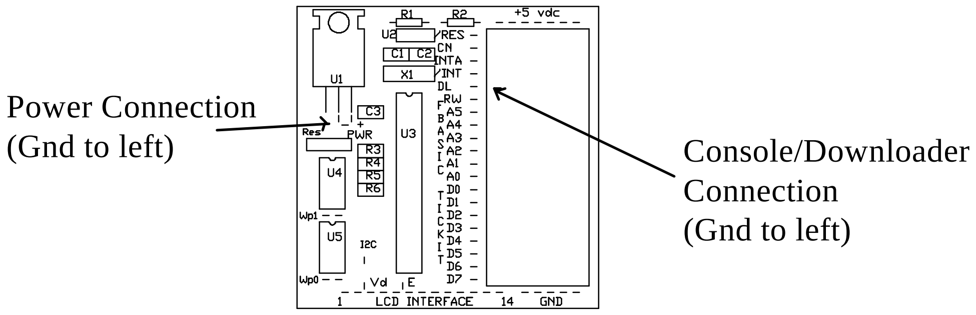
Support for two EEproms addressed in blocks 0 and 1 is provided and each EEprom may be individually write protected.

B.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.

B.3 Component Placement Diagram

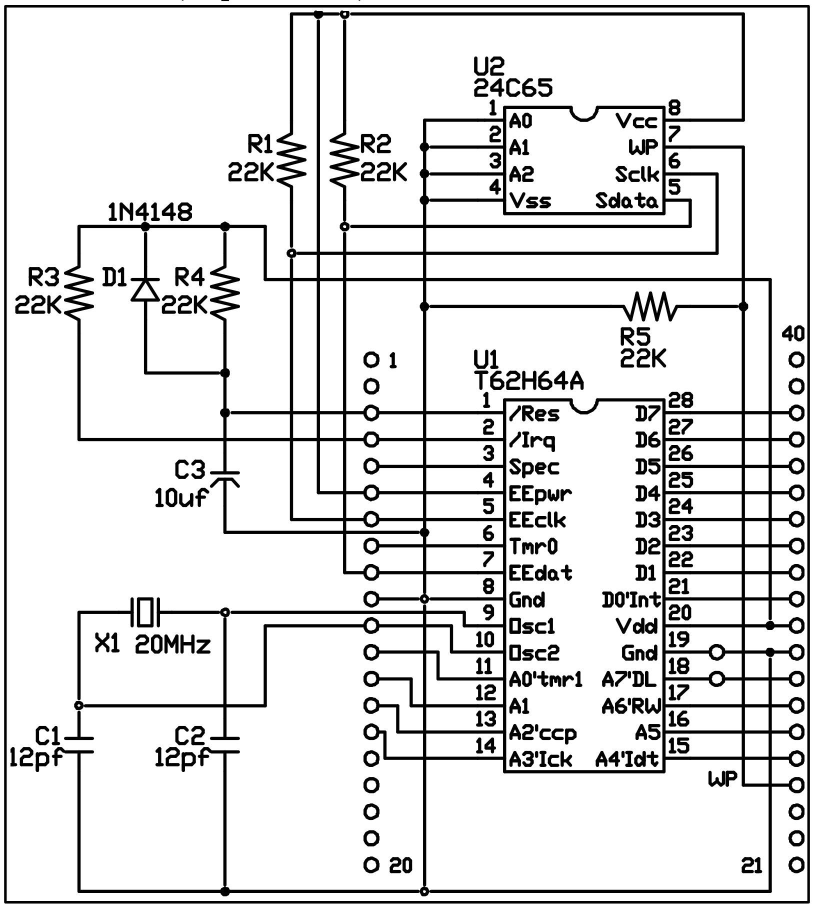


The above diagram shows the locations of components and pins for the TICkit. One important point to notice, is that the data group of outputs is numbered in the opposite order from the address group pins. This is simply a placement issue, but there is a possibility of confusion when wiring components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

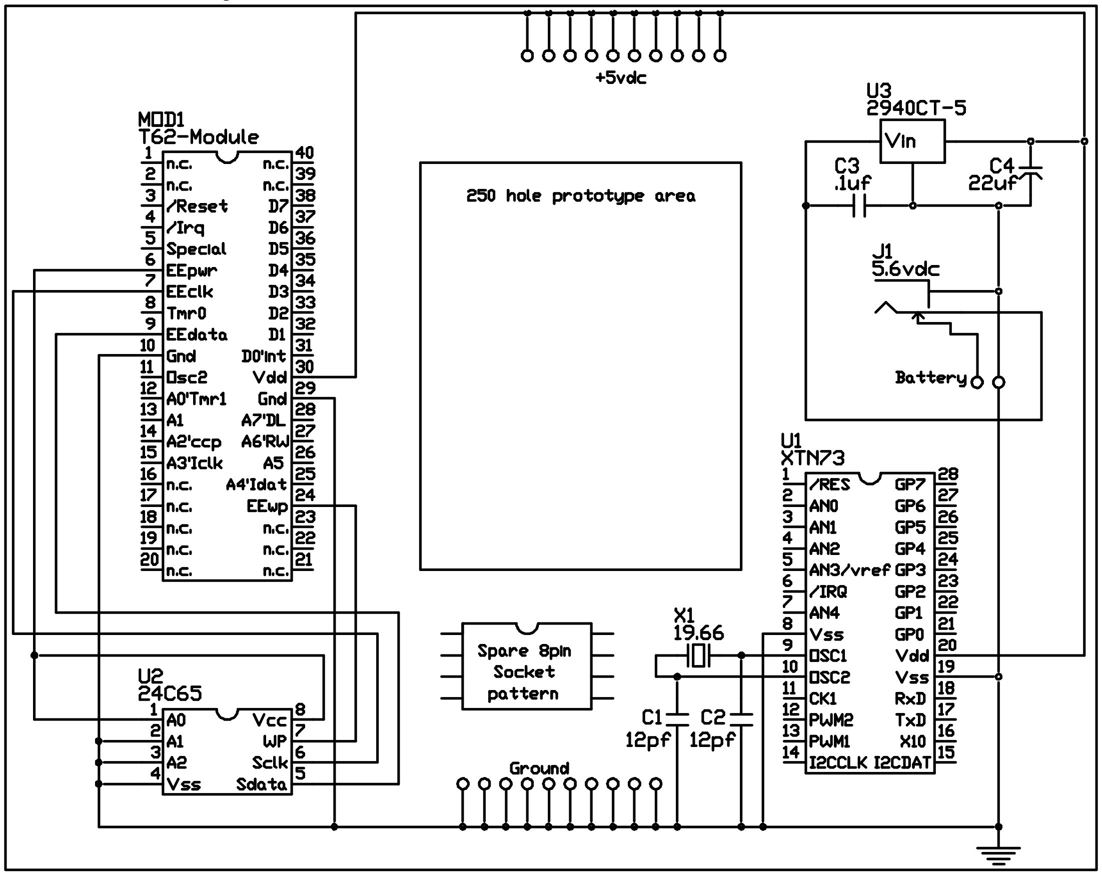
Appendix C: TICkit 62 Hardware

C.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.

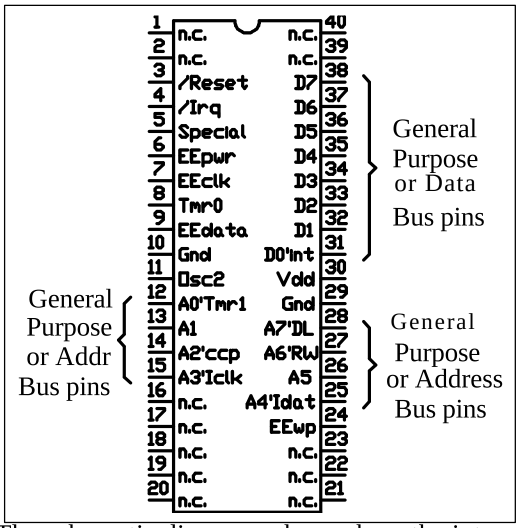
C.2 TICkit 62 Project Board Schematic

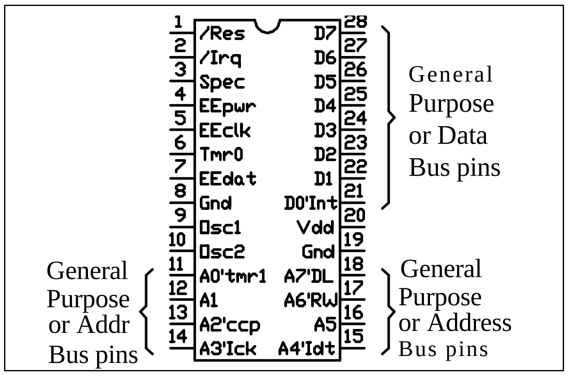


The T62-PROJ project board provides a means for wiring up simple TICkit 62 based projects. A 40 pin IC socket accepts the TICkit 62 module. Additionally, a +5 vdc regulated power supply is implemented on the board. Simply connect any DC source between 5.6 and 18 vdc into the coaxial power connector (center -). 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.

C.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.

C.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.