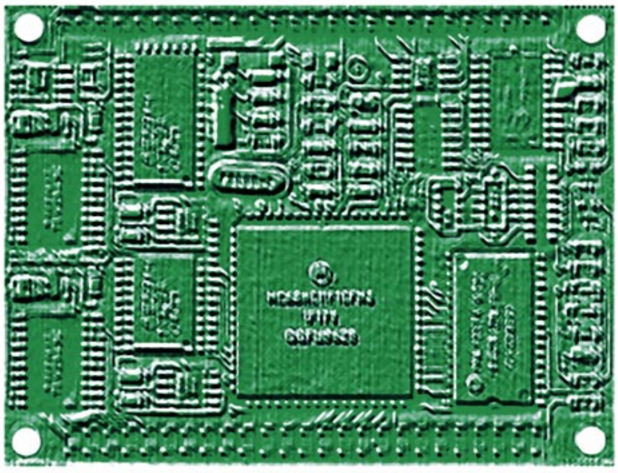
TFX-11

REMOTE DATALOGGER / CONTROLLER ENGINE



USER'S GUIDE

ONSET COMPUTER CORPORATION

TFX-11 HARDWARE
TFBASIC PROGRAMMING
TFTOOLS IDE

Credits

TFX-11 User Guide Project Director: Tom Mignone
Writers: Tom Mignone, Lon Hocker, Jim Dodd
Editors: Tom Mignone, Jim Dodd, Mark McPike
Illustrations: Tom Mignone, Robert Kelley, Lon Hocker
Layout and Design: Tom Mignone
Cover Design and Graphics: Tom Mignone

Special thanks to: Tom Sullivan, Al Bellows, Tom Wilson, Pierre Tillier, John Godley, and Skip Pelletier for their many helpful comments, corrections, and suggestions.

D-2469-A 8/1/97

INTRODUCTION

Onset Computer Corporation

536 MacArthur Boulevard.

P.O. Box 3450

Pocasset, MA 02559-3450

Tel. (508) 563-9000

Fax. (508) 563-9477

tech_support@onsetcomp.com

www.onsetcomp.com

ONSET document D-2469-A

8/1/97

Introducing your TFX

Your new Onset TFX-11 is a state-of-the-art datalogger/controller, the most recent in Onset's line of Tattletale datalogger/controller engines. The TFX combines Onset's years of experience in design and manufacture of high quality, easy to use datalogger/controller engines with the latest in compact, low power, high-performance, components and technology.

The Onset supplied development software features the TFTools Integrated Development Environment (IDE), greatly simplifying program development with Onset's TFBASIC programming language. TFBASIC is a tokenized version of BASIC, with language extensions added by Onset to greatly simplify common datalogging and control functions.

The TFX-11 hardware combines large, non-volatile data storage capacity and microampere range low power modes with small size. The resulting developer's package is well suited to rapid development of intelligent instruments from a simple temperature logger to a complicated, long term, remote logging and control device.

TFX-11 Features

- Dual Processors : Motorola 68HC11F1 and PIC16C62
- Small size 2.4 in. x 3.2 in. x 0.5 in.
- Ultra low power mode - less than100µA HYB mode (50µA typical)
- Wide input voltage range for power supply (5.5-18 VDC) with thermal shutdown protection.
- 472k Non-volatile data and program storage
- 128k battery backed RAM for R/W datafile and program
- High speed parallel data interface off-loads 512k Serial FLASH (SFLASH) in less than 30 seconds.
- Development communication and SFLASH program/data upload and off-load via Standard PC parallel Port
- 1 RS-232 hardware UART serial port
- 11 channel 12-bit A/D converter
- 8 channel 8-bit A/D converter, each channel may be optionally configured as a digital input.
- 16 TTL Digital I/O pins (two eight bit ports)
- 8-bit bus interface with chip selects for expansion

- Programs in TFBASIC and Assembler inside full-featured DOS based Programming IDE. (Contact Onset for stand-alone DOS tools)

TFBASIC LANGUAGE OVERVIEW

TFBASIC is a tokenized dialect of the familiar BASIC (Beginners All-Purpose Symbolic Instruction Code) with language extensions. These extensions, added by Onset, simplify datalogging and control tasks. Although the fundamental structures of the languages are similar, there are many important differences. The following discusses the language, with particular emphasis on this dialects additions, omissions and exceptions.

How to use this manual

For those of you familiar with Onset's TTBASIC or TXBASIC, please refer to the section “What’s New and Different in TFBASIC” at the end of Chapter 6. If this is your first Tattletale, be sure to read through Chapter 1 ”Getting Started” and complete the tutorials.

Specifications

Electrical	· Input	5.5 -18VDC input at 0.5 W max dissipation, 50 mA. max.
	· Power Consumption (typical)	Running 14 mA. , Sleep 4 mA., HYB < 100 μA.
Environmental	· Operating Temperature Range	0 to 70 degrees C.
	· Relative Humidity	0 to 95%, non-condensing
Performance	· Hardware UART Baud Rates	300, 600, 1200, 2400, 4800, 9600, 19200, 38400
	· Max Sampling Rate	800 samples/sec from TFBASIC, 3200+ using assembly language
	· A/D accuracy 12-bit	+/- 1.5 LSB
	· A/D accuracy 8-bit	+/- 1.5 LSB
Dimensional	· Board Dimensions	3.2” L x 2.4” W x 0.5” H
	· Weight	1 ounce

Host Computer Requirements

The TFX-11 was designed to be programmed and off-loaded using an IBM compatible PC. The minimum system requirements are :

- 25 MHz 486SX or greater minimum running MSDOS 6.xx
- 640K RAM with 580K available for program
- Hard Drive with 1MB available for program and related file storage
- (1) 3.5" 1.4MB floppy drive
- (1) parallel port
- (1) available RS-232 serial port
- Microsoft or compatible Mouse

Parallel Port Note: The TFX-11 comes complete and ready to run using only a single serial port on the Host PC. For access to all the features of the TFX-11 the host computer also requires at least one standard parallel printer port connection. The parallel interface on the TFX-11 was designed to be compatible with all PC parallel ports including ECP and EPP, when configured to run in standard mode.

Registering your TFX-11

Thank you for choosing Onset. In order to be eligible for Onset's free technical support and a free subscription to our TattleTips informational newsletter you must mail back your registration card. On it are some questions about you and your opinions about our products. Please be assured this information will be kept confidential. It is for our internal use only and WILL NOT be sold or otherwise distributed, but will be used by us to assist in developing future products, product improvements, all designed to best meet your current and future needs.

Technical Support Policy, Warranty, and Disclaimers

Read This Before You Contact Onset for technical assistance!

Onset provides technical support through a variety of channels. The first and most important is this manual - we have made every effort to provide you with all the information you need in an easy to read format - please check through it thoroughly before calling us. If you cannot find what you need in the manuals please be sure you understand what types of end user problems we do and do not support by reading the following paragraphs.

Software/ Programming problems

Onset provides support for its software only and does not provide instruction in how to program. It is expected that the end user have some familiarity with programming principles and language constructs. We provide an introduction to data logging computing and code examples in the text of this user's manual and on disk to help you make the transition from software only to software dependent hardware control and data logging.

Hardware/circuit design problems

Onset provides support for its hardware only when it does not appear to meet our published specifications or otherwise perform as advertised. Onset does not provide instruction in electronics and electronic instrument design other than what is included in this manual and related application notes. Also, as it is says in the disclaimer, Onset does not guarantee its code to be free from errors. It is again assumed you have some familiarity with electronic principles and practices or have access to someone who does. We try to provide an introduction to data logging fundamental concepts and techniques and simple demonstration code examples in this user's manual, with the source code to most examples reproduced on the disk .

Warranty

Hardware will be repaired or replaced (at Onset's discretion) if found to be defective in materials or workmanship for a period of one (1) year. Software will be replaced only if the disk supplied is found to be defective. All the programs and other files are offered as-is. If you choose to use the software Onset cannot be held liable for any damages or loss of data incurred as a result of its use. While we hope you find this software useful, it is up to you to determine the applicability, fitness, and correctness for your application.

Returns

All returns require that you first request and obtain a Return Materials Authorization (RMA) number from Onset ! The RMA helps us track your return in our computer and speeds processing. To obtain an RMA, call Onset sales support. Have available the Model number, serial number, invoice number, and a brief explanation of the problem. They will assign you a number which should be plainly visible on the outside of the package and referred to in any correspondence. Returns without prior authorization may be delayed. NOTE : some returns may be subject to a restocking fee.

Disclaimers

Onset does not authorize or approve the use of this equipment in life support or related safety equipment, and cannot be held responsible for any injury or death as a result of it being used in any related application.

If you choose to use the supplied software you do so at your own risk. Onset cannot be held liable for any damages incurred as a result of its use. While we hope you find this software useful, it is up to you, the user, to determine the applicability, fitness, and correctness for your application. We encourage your feedback for corrections, amplifications, interesting applications and other comments. Application notes or suggestions supplied to us are not eligible for compensation unless details have been worked out in advance. We will incorporate them, when and where appropriate, for the benefit of all users.

Contacting Onset

Preparing to contact Onset

If you have searched our manuals, help files and other sources of assistance and still have not found an answer, then you need to contact Onset. Before you contact Onset please have the following information available and be well prepared to describe your problems or difficulties by doing the following:

Please have handy the following information:

- Host computer brand name and model
- Number of MB of installed RAM
- Processor type
- System Clock speed
- Operating system name and version number
- TFTools version number
- TFX board serial number
- PIC software version number
- Installed or attached devices (modems, network, printers etc.) and their corresponding ports.

Please be sure to include this information in any correspondence. If you phone, have this information readily available. If possible (and if appropriate to the problem) please try to be seated at your computer when you call.

Syschk diagnostic shareware

Syschk is a shareware program that does an excellent job of cataloging your PC's internal hardware configuration as well as statistics on its performance. It will uncover and list out all the PC specific information requested above, as well as much more. Especially helpful is its identification of COM ports and the devices attached to them. Highly recommended, it is available for download at www.syschk.com

How to contact Onset

You can contact us by phone, fax, email, U.S. mail, and the internet. See the inside front cover of this manual for the current numbers. Our internet World Wide Web site and FTP site contain the latest software and example programs for the TFX.

TABLE OF CONTENTS

CHAPTER 1

Getting Started

Getting Started

Component Checklist

The following is a list of what you should have received in your development kit :

- This Users Guide
- PR-11 Breadboard
- 1.4M 3.5" TFTools floppy Disk
- Parallel cable, DB25 to 9-pin mini-DIN
- Serial cable, DB9 to 3.5mm Stereo Phone jack
- 9 volt battery
- plastic case
- Combo screwdriver
- MC68HC11 Reference Manual
- MC68HC11F1 Technical Data Book
- Assorted jacks for the PR-11 breadboard
- Goodie bag containing :
  - (1) 10k thermistor
  - (1) 10k 0.1% resistor
  - (1) FET

If anything appears to be missing please contact Onset immediately!

What you will need to get started

- A TFX-11 board (purchased separately from the development kit).
- An IBM compatible computer meeting the minimum system requirements as described previously.
- A fine point electronics soldering iron and electronics type solder.

Additionally items you will find extremely useful

- A benchtop power supply, 9V battery, or 9V battery eliminator
- A Voltmeter or Multimeter
- An Oscilloscope
- The reference book by Horowitz and Hill, The Art of Electronics , Cambridge University Press. (ISBN 0-521-37095-7)

Quick Start Tutorial

Let's try it out!

Acknowledging that it is a rare person that wants to read the entire manual before using their Tattletale, this section presents some simple instructions and examples to get you up and running.

Do you understand BASIC?

Do you have some familiarity with electronics? If you are reasonably familiar with BASIC and have a fundamental understanding of analog and digital circuits, continue with this tutorial. If not, you can wing it, but should probably consider reading an introductory text or finding a mentor. For a good all around electronics reference we recommend Horowitz and Hill, The Art of Electronics.

OK, unpack your Tattletale

Remember - it's all CMOS and thus static sensitive! Normal care in handling CMOS devices is all that must be observed to prevent damage to the Tattletale. You must take extra precautions if you are in a very dry climate and get a shock every time you touch something metallic; if you like wearing wool sweaters; if you work on a wool rug. It is always good CMOS practice to touch the negative (-) battery lead first when picking up the Tattletale. Even better is to use a grounded wrist strap and table mat when handling. Be warned - static damage is easily identified and is NOT covered under warranty!

Load the TFTools software

Create a directory on your hard disk where you want to store the files and copy all the files from the distribution disk to this directory. The first time you run TFTools be sure to add the appropriate command line parameters to designate your serial port location (see the TFTools serial communications section of the TFTools IDE chapter for an explanation of the command line parameters). Once this is done the command line parameters will be stored in a configuration file so the next time you will not need to specify them.

Connect the serial cable

The serial cable has a DB9F connector to connect to the Host computer and a mini stereo phone plug to connect to the TFX breadboard. PC serial ports are either DB9M or DB25M. If you only have a DB25M COM port available you will need to get a DB25F - DB9M adapter, available at your local Radio Shack.

TFTools supports COM1 thru COM4 on your Host PC. This connection is used to communicate with the program while it is running. Be sure to remember which COM port you physically connect to, and that no other devices conflict with it! (see note below)

IMPORTANT NOTES regarding PC COM ports

Note 1. PCs share interrupts on the COM ports. Typically COM1 and COM3 share one, and COM2 and COM4 share another. Only one device using interrupts can be active on either pair at one time. Therefore, if your mouse is connected to COM1 DO NOT use COM3, but restrict your choice to COM2 or COM4. If you have a mouse on COM1 and MODEM on COM2 then you may use COM4, as long as you do not use the MODEM or load its drivers while TFTools is active.

Note 2. Some of the earlier SVGA video cards had a conflict with COM4 in IBM 8514 mode. This appears as incorrect colors on the display as well as system lockup. If you are using IBM 8514 display mode you cannot use COM4.

Start TFTools

Assuming you have loaded the software into a directory on your hard disk or have the floppy disk in one of the drives, move into the directory with TFTOOLS.EXE. Then type TFTOOLS at the DOS prompt and hit return.

From the DOS prompt, go to the directory that has the TFTools.exe enter the command **TFTools**. The program will launch and display the main window and within that the terminal window.

A Short TFTools Tutorial

Who should use this tutorial?

This tutorial is recommended for those who have never used a TattleTale along with an Onset integrated development environment (IDE), such as TTools, TxTools, or Crosscut. If you have you can probably skip this section.

TFTools, Step-by-step

The following procedures show you how to start the TFTools program, open a new editing window, and how to enter a short TFBASIC program.

TFX-11 connections for serial communication

With the serial communication cable connected to the Tattletale and to the computer, connect the TFX-11 to a power supply or battery. The Tattletale start-up message will be displayed and the '#' prompt will appear below it

Press the ENTER key. The '#' prompt should be displayed again. This verifies that the serial interface is operating correctly.

Pull down the File menu and select New. This will open a new text editor window labeled "Untitled".

Enter a TFBASIC example program into the text editor

Now you are ready to start entering a TFBASIC program. For this brief tutorial of TFTools you will be entering a simple TFBASIC program and then debugging it to demonstrate to you some of the basic features.

Type the following **exactly** as shown, including all spaces. (Ignore the error in the first line - it is there on purpose):

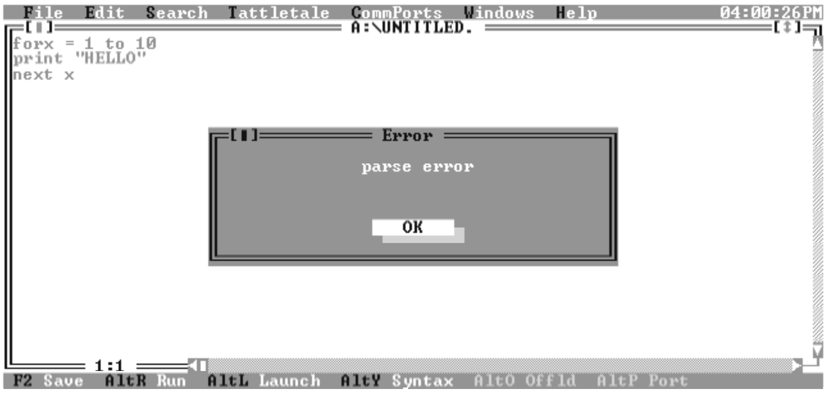
```
forx = 1 to 10
print "Hello"
next x
```

Save your program to a file

Pull down the "File" menu and select "Save". A dialog box will appear with the cursor in the name box at the end of the default name "Untitled". Type the name "TUTORIAL.TFB" and press the ENTER key. The file will be saved under this name in the same directory as the TFTools program.

Perform a syntax check

Pull down the Tattletale menu and select "Syntax Check" (you can also type ALT-Y). If you typed in the program exactly as was shown, a small error dialog box should be displayed in the middle of your screen:


File Edit Search Tattletale CommPorts Windows Help 04:00:26PM
A:\UNTITLED.
Error
OK
1:1
F2 Save AltR Run AltL Launch AltY Syntax AltO Offld AltP Port

```
forx = 1 to 10
print "HELLO"
next x
```

parse error

Parse Error! This error was introduced intentionally to demonstrate how the integrated parser detects syntax errors in your programs. The parser does a syntax check that detects common syntax errors such as misspelled keywords or missing arguments, but it does NOT detect programming logic errors!

To exit the "Parse error" dialog press the ENTER key or click on the OK button. If you get a message "Not enough DOS memory"

You need to set aside more conventional DOS memory for TFTOOLS. There are two ways to do this. The first and simplest is to go under the menu selection Tattletale Options and change the entry Maximum ASM lines to 0. Click on OK and try to do the syntax check. If you still get the "Not enough DOS memory" message then go to the options dialog again and reduce the Maximum symbols entry to 100. If this does not fix the problem you have too little convention DOS memory available and will have to try method two, which is to exit TFTOOLS and remove any TSRs or other memory resident programs until you get at least 580kB of conventional free. If you are running DOS 6.xx you may use the MEM command from the DOS prompt to determine your available memory. You may also use MEMMAKER to help you move some of the TSRs out of conventional memory and into upper memory.

Fix that error! Upon exit from the "Parse error" dialog box the cursor will be placed in the line (but not necessarily the exact spot) with the error. Using the mouse or arrow keys set the cursor under the "x" after the word "for". The "x" will then be underlined, showing you the current position of the editing cursor. Press the SPACE bar to enter a space between the "x" and the word "for".

Again hit an ALT-Y to perform a Syntax Check. If your correction was successful the "No Errors" dialog box should be displayed

The "No Errors" box signal that the parser did not find any errors. The Syntax Check command tokenizes the program in the edit buffer and reports any errors, but does not attempt to load or run the program.

To exit the "No Errors" dialog press the ENTER key or click on the OK button.

Remember, just because the "No Errors" box is displayed does not mean you don't have any logic errors in your program, it only means no syntax errors were detected.

Congratulations!

You now have written and debugged a TFBASIC program. While it may not be the most useful program ever written, it does demonstrate the basic operations of the TFTools Integrated Development environment (IDE).

Back to the TFX-11

This completes the TFTools introductory tutorial. The other TFTools commands and options are explained in detail in Chapters 2 and 3.

Attach a 9-volt battery or power supply

With your computer is running TFTools and the terminal window is displayed , power up the Tattletale by connecting its battery clip to a battery or power supply. If you use a power supply, make sure that the polarity is not reversed when you connect the power to the board, or you may damage the board.

We strongly recommend a current limited power supply for development. A 50 mA current limit will protect your Tattletale from most common disasters. It's not hard to go through a lot of nine-volt batteries when developing a program for a Tattletale. For example, just leaving a Tattletale plugged in running at full speed over a weekend can use up half of an alkaline nine-volt battery. The Archer battery eliminator (Radio Shack Catalog No. 270-1552B) plugs into a wall plug and the Tattletale's battery snap connector! It comes with a long cord and works nicely in all applications we've tested and is available through Radio Shack at a price (at the time of this writing) of $6.95. Its electrical current is very limited so if you add external circuitry make sure it can deliver the current or the Tattletale will not power up properly.

Sign-on message

If you have done everything correctly at power on, you will see the Tattletale sign-on as shown below.

```
Tattletale Model 11.00
TFBASIC Version 1.00
(C) 1996 Onset Computer Corp.
All Rights Reserved
#
```

Don’t worry if your TFBASIC version number is different, as long as it is greater than 1.00! If your Tattletale signed on then you are all set and may skip the next section. Otherwise...

Troubleshooting If the Tattletale didn't sign on

Let's go over the likely causes:

(If the parallel port cable is connected, please disconnect it before proceeding.)

Cable loopback test. First re-check the settings in the Port/Baudrate dialog box found under the CommPorts... Serial Port... menus. The baud rate should be 19200, and the Port radio button selected should match the Port the cable is connected to. With the terminal window open, short the tip of the phone jack to the next ring down (a paper clip works well) and type characters at the keyboard. Characters type should appear in the terminal window. If characters do not appear, check the cable connection and try again. If characters still do not appear, you may have a faulty cable or COM port. If there is another COM port available, try moving the cable to it and repeating the above procedure. Don’t forget to reset the Port setting in the Port/Baudrate Dialog if you change to another COM Port .

If the above test did not work you might try connecting to another computer. If that is not possible, insert a break-out box between the terminal and the Tattletale serial cable and verify that your terminal is driving pin 2.

Check the Battery or Power Supply. If the connections are OK, perhaps the battery you are using to power the Tattletale is bad. Test it under load to see if its voltage is correct for its rating.

If the battery checks out and you are convinced that the Tattletale isn't working, go back to the break-out box and use an oscilloscope to look for the signal transmitted from the Tattletale. This signal should appear on the serial connector at pin 3. Each time you power up the Tattletale, it will send its sign-on message out on this pin and you will see a digital signal on this pin switching between -4.5V and +4.5V. If you don't see a signal on this pin, the Tattletale may indeed have a problem. You can send it back to Onset for a check-out, but I'll wager that you tried the wrong pin or have a dead battery! Double check and check again!

If the Tattletale passed the scope test, loop back to the first step above and again check the connection--the Tattletale is sending, you're not receiving. Press Return. The Tattletale should respond with its prompt :

```
#
```

If this test succeeds it proves that the serial interface can send as well as receive. The number sign, '#' is analogous to the C\>' prompt in DOS and means the Tattletale is idle while waiting for a command.

It just won't work - returning the TFX-11 to ONSET

So nothing you tried worked...

If the above tests failed, or you do not have the tools to properly perform the tests, then you may need to return the TFX to Onset. Please contact us at the numbers in the front of this manual to obtain an RMA and shipping instructions for returns.

PLEASE! - DO NOT RETURN THE TATTLETALE WITHOUT FIRST OBTAINING A RETURN MATERIALS AUTHORIZATION (RMA)

Attempting a return without an RMA will delay processing.

The Parallel Cable

The parallel cable should be connected to the Tattletale only after power has been applied to the TFX-11. If the parallel cable is attached before power is applied the TFX-11 may not power up correctly.

The parallel cable connects the DB25F (printer) connector on the host computer to the 9 pin mini-din connector on the PR-11 board. This cable allows high speed offload of data stored to the flash as well as the ability to record your program in the flash. A program stored in this manner will be copied to RAM and executed at power up of the TFX-11

All is well, let's continue!

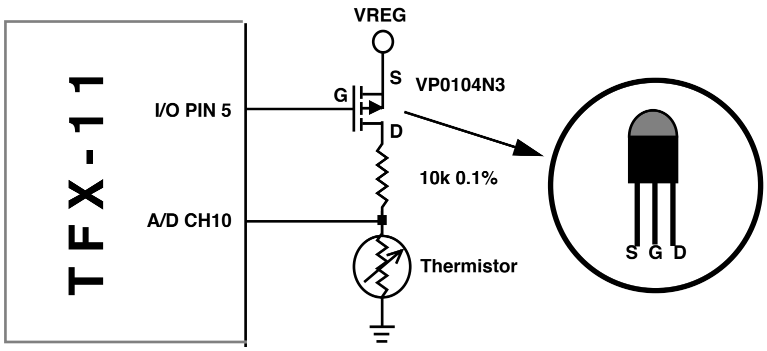
The TFX is up and running, so now you are ready try to a short example program. Now that you know the fundamental techniques needed for running TFBASIC programs, you can use them to run the example programs in the next section.

Build a Datalogger, One Step at a Time

Building a Temperature Monitor

This section illustrates how to build a simple datalogger application, starting with a simple temperature monitor and then extending its utility by adding useful features.

Build this circuit onto your breadboard :



The components to construct this circuit are in the goodie bag supplied with the development kit. The parts include the thermistor, a 10k resistor, and an n-channel FET. The resistor (10k, 0.1%) biases the thermistor and together they form a resistor divider, the center of which is tapped to obtain the temperature signal. The FET acts as an ON/OFF switch that is used to supply power to the thermistor whenever a temperature measurement is made.

Enter this program with the editor

```
       sleep 0
start: sleep 100
       pset 5
       print temp(chan(10))
       pclr 5
       goto   start
       stop
```

The Commands

This example code prints the temperature out on the HOST terminal display. Five commands unique to TFBASIC, SLEEP, TEMP(), PSET, PCLR and CHAN( ), are used. These five commands are fundamental to many datalogging and control applications and are explained below. Be sure you have a good understanding of how they work before proceeding.

SLEEP The SLEEP instruction combines two important features: precise timing and power conservation. Each count of the SLEEP argument represents 10 ms (1/100 of a second), so SLEEP 100 sets the SLEEP countdown timer for one second. When using the SLEEP function it should always be first initialized by a SLEEP 0 instruction. This resets the SLEEP timer.

When a SLEEP command is invoked it checks the already running SLEEP timer from the previous SLEEP command to see if it has expired. If not, it waits in a low power mode until it does expire. When the timer finally expires, the SLEEP timer is updated with the new value from the current command, and the program proceeds to execute the instruction statements that follow until the program reaches the next SLEEP statement. At this point it again checks the sleep timer and again waits until it expires before updating the counter and allowing the program to proceed. NOTE: SLEEP commands, to be effective, must allow sufficient time for the intervening statements to execute. If the command between the two SLEEP instructions take longer than the SLEEP interval, the SLEEP timer will expire BEFORE the next SLEEP command is reached. This generates a non-fatal error, signaled by the program writing an '*' out the serial port, identifying the fact that precise timing has been compromised due to a timer overrun. To reset the SLEEP timer the statement SLEEP 0 should be used, otherwise the first invocation of SLEEP will generate an '*'.

PSET and PCLR PSET sets the corresponding pin(s) to outputs and then sets them to +5V. PCLR also sets the corresponding pin(s) to outputs, but then sets them to 0V.

CHAN() The function CHAN(n) returns a digital value proportional to the analog voltage at pin n.

TEMP() The thermistor is in a divider circuit with a 10K resistor. TFBASIC has a convenient function, TEMP, which converts the output of the converter to a temperature in hundredths of degrees C. The command only works correctly when used with the divider circuit and components illustrated above.

Run it in TFTools

While still in the editor window select Run from under the Tattletale menu to load and run the program. When you run the program you should see the following output - hit <CTRL-C> to halt the program :

```
2340
2340
2340
2340
2340
2340
2340
^C    (a CTRL-C halts the program)
#
```

A slightly more complex program

The following enhanced program demonstrates a few more commands and also adds some descriptive comments which are valuable for code intelligibility and maintenance.

```
        //******* SAMPLE SIMPLE DATA LOGGING PROGRAM ***********
        //**** first set up variable parameters****
Start:
        print
        input "Enter time interval (1/100 sec) : "tInterval
        print
        input "Enter channel to read : (0-10) "Channel
        //****INITIALIZATION****
        onerr exit// quit when memory overflows
        sleep 0// initialize interval
        //**** LOGGING ****
getdata:
        sleep tInterval
        Store #2, chan(Channel)
        goto getdata
        //*** ENDING ***
exit:
        print “Logger full”
        stop
```

Double slashes //

This allows the entering of comment text. The tokenizer ignores any text on the line following the double slash //. Comments enhance readability and maintainability and take up no space in the executing code. Their frequent use is recommended.

Blank line Blank lines are permissible in TFBASIC. Use them to separate logical blocks to make your program more readable.

INPUT This command allows you to assign values to variables as the program is executed. Notice there is no punctuation between the string constant and the variable name.

ONERR When ONERR is used, the program responds to an error instead of printing the error message 'How?'. We know this example will eventually overflow the datafile and cause an error; the ONERR command forces this error to cause a jump to line labeled 'exit' instead.

STORE The Store command sends data to the non-volatile serial Flash (SFLASH) datafile. Once data is written to this datafile it cannot be read or modified by the program. The datafile pointer is available as the read-only variable DFPNT. After each write to the datafile DFPNT is automatically updated to point to the next available location. Data is retrieved after the program is completed. The channel command returns the two bytes from the A/D converter padded out to four bytes for compatibility with the standard TFBASIC integer variable. The #2 after the Store command is a formatting command that says only store the two least significant bytes of the four bytes, which, in this instance, are the only bytes that contain data.

STOP This command ends program execution. It is added here for readability since it really isn't needed here. The last line of the program halts execution anyway.

The above code, employing the ONERR command, is a little fancy; the following code works just as well:

```
for A = 1 to dfmax
 sleep tInterval
 Store #2, chan(Channel)
next A
```

Here 'dfmax' is a predefined variable that returns the size of the datafile.

FOR / NEXT This command, along with NEXT, forms a powerful looping structure in TFBASIC. The loop starts with the assignment of the first specified value (1) to A and executing all the code up to the NEXT command. At the NEXT command, the variable (A) increments and the program goes back to the FOR line. The new value of the variable is then compared with the value after the TO; if it is less, the intervening code between the FOR and the NEXT is executed again, if not execution is passed to the line following the NEXT statement.

Note that the current drain after executing the STOP command is as low as it would be executing SLEEP; the logger is waiting for an incoming character. Data can then be off-loaded.

Retrieving stored data

To retrieve the data from the SFLASH you need to be running TFTools. The TFBASIC program, if running, must be halted and the # prompt must appear in the terminal. Next, the TATTLETALE menu in TFTools is pulled down, where either XMODEM off-load or Parallel off-load will copy the data from the datafile to a file on the host PC. For Parallel off-load to work the parallel cable must be connected.

Making your program boot on power up

The program may be stored in the non-volatile flash memory along with the TFBASIC operating system. When stored the user program will begin execution from its beginning at power on reset or when relaunch is selected from the Tattletale menu. The parallel cable must be connected to allow the program to be written into flash. To write the program to flash select Tattletale Launch (instead of Run) from the menu. This will load the program and start its execution. To erase the program and get the # prompt back select Load OS only from the Tattletale menu.

CHAPTER 2

The TFTools IDE

Understanding the TFTools Integrated Development Environment

Introduction to TFTools

(For latest release information please be sure to read the README files on the distribution disks)

TFTools runs on a host IBM or compatible PC and provides a complete interactive programming environment for coding and debugging in TFBASIC, as well as a complete terminal program for uploading and downloading data and programs between the host PC and the TFX-11.

DOS only

TFTools is a DOS application. There currently is no Windows version, but it can run from Windows as a DOS application.

The user interface used in TFTools is based on Borland International's DOS based Turbo Vision, which looks and acts a little like Microsoft Windows but runs in text mode only. This interface allows use of a Microsoft compatible mouse to select and manipulate items.

The methods of using the mouse, clicking on an object to select it and dragging to move an object, are similar to the techniques used in Microsoft Windows. At this time, only the left mouse button is recognized. Middle and right buttons are ignored. This may change in future versions.

What, you don't have a mouse?

If you don't have a mouse, you can use special keys and key combinations to select and manipulate objects on the screen. Later in this text we describe the keyboard equivalents for mouse actions.

TFBASIC Programming environment

TFTools provides a user–friendly, multi-window programming editor for developing and maintaining your TFBASIC programs. TFTools also includes an integrated tokenizing compiler for generating tokenized TFBASIC code for the TFX-11. There are two main windows, the Terminal Window and the Editor window. Switching between windows is as simple as a mouse click (or a single keystroke).

Integrated Editor

The TFTools integrated text editor can edit a program up to 64K bytes in size. In addition to the normal features of a text editor such as Open File and Save File the editor has specific features to make the debugging and running of your programs that much easier. From within the editor hitting Alt-Y provides instant access to a syntax checker. When the syntax checker finds an error, it identifies the nature of the error with a dialog box and places the cursor on the line where the error was found. This allows rapid corrections of simple mistakes decreasing the time it normally would take to find and correct the error.

When the program compiles successfully, hitting ALT-R will load and run your program, automatically switching you to the Terminal window.

Terminal Window

TFTools has an integrated terminal program with a scrolling history buffer for debugging and interacting with your running Tattletale programs. This terminal program communicates with the TFX-11 through an Onset supplied cable connected to a PC serial (COM) port.

Program upload and data off-load use a standard PC (LPT) parallel port connected by a special cable supplied by Onset.

Summary

TFTools is a necessary part of development on the TFX-11. It was designed to provide a single environment for common operations with the TFX-11. It allows you to write, debug and tokenize TFBASIC programs as well as upload the programs and off-load data.

TFTools Serial Communications

Serial Port configuration file

TFTools attempts to initialize the PC's COM Port using three different methods. If no configuration file is present (TFTools.CFG) and if no command line options are used, by default COM2 is opened with parameters of 19200 baud, 8 data bits, no parity and 1 stop bit. If a configuration file is present (a configuration file is automatically created or updated with the current configuration parameters each time TFTools terminates), the COM Port is set up as specified in the configuration file. Command line options can be used to override the values stored in the configuration file. The command line options are used in this manner (items in square brackets are optional) :

Invoking TFTools from the DOS prompt

```
TFTOOLS [filename] [-p port] [-b baud] [-h]
```

It is permissible to have spaces between the option letters (p, b) and the argument (port, baud). Options can be specified in any order.

Explaining the Command line

TFTOOLS is the name of the executable program and the only non-optional item.

[filename] The name of a DOS text file that will be opened in an edit window.

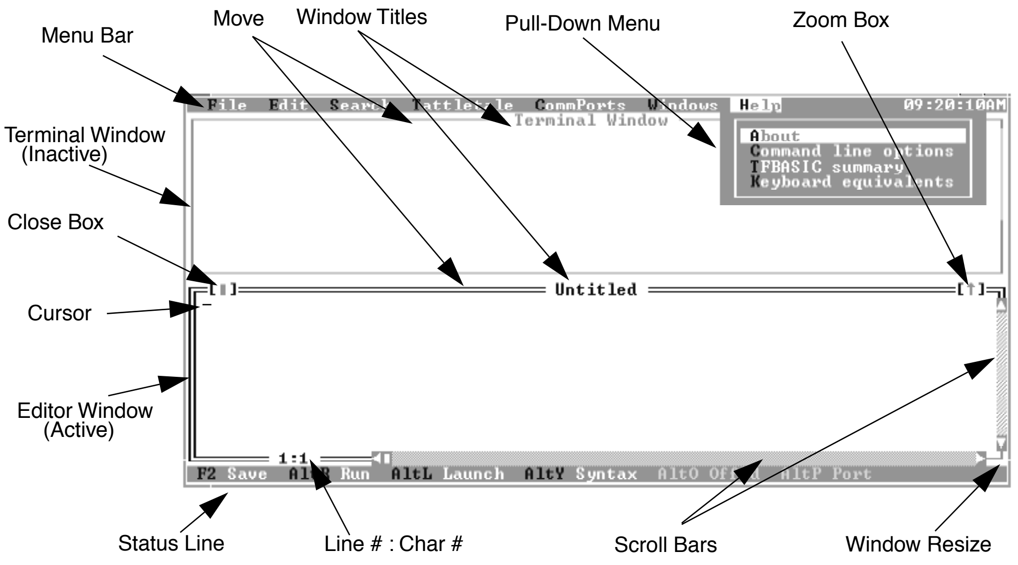
[port] A number from 1 to 4 to specify the Com Port number.

[baud] One of these: 300, 600, 1200, 2400, 4800, 9600, 19200, or 38400

The **'h'** option displays this information in case you forget next time you run the TFTOOLS.

Navigating the TFTools Main Windows and Controls

The TFTools Main Screen



Explanation of TFTools Windows

There are two windows you will be working with in TFTools, the Terminal windows and the Editor window. These windows share many operational features; they may be independently moved, resized, overlapped, zoomed, and scrolled. There may be multiple editor windows open, but only one terminal window. Only one window may be active at a time, and this window will appear on the top of the others. The windows continue to exist and be displayed, (as long as the active window does not cover them) even if they are not the active window. The active window has a double line border, and a window in the background has a single line border with no close box, scroll bars, or zoom box.

Startup screen

When you first start TFTools, you will immediately see one window which fills the screen. This window displays any characters that are received by the serial port (including any characters you have typed and the Tattletale has echoed back).

By clicking on the Zoom box, you can toggle the size of the window between full screen and a smaller size. You can move the window by placing the mouse cursor anywhere along the top of the Title frames and then pressing and holding the left mouse button down while moving the mouse. The window frame will follow the mouse until you release the button.

There are four scroll arrows located at the end of each scroll bar. Clicking on these causes the text in the window to scroll one line in the direction of the arrow, either horizontally or vertically. Click and hold on these arrows to scroll continuously. You can move more quickly through a document by click-hold-dragging on the “Thumb”. The Thumb is the small box that only appears in the scroll bars when the text overflows the window boundaries, either vertically or horizontally. Moving the Thumb toward the top moves closer to the start of the document and moving the Thumb toward the bottom moves closer to the end of the document. The window contents won't scroll until the Thumb is released.

Terminal Window.

When you first start TFTools the Terminal window fills the screen. This window displays characters that are received by the host PC serial port. Keyboard characters go out the host PC serial port to the Tattletale’s main UART. Tattletale replies sent out via the TFX-11 main UART appear in this terminal window. There can be only one terminal window and, while it can be hidden by an editor window, it cannot be closed. The Terminal Window has a 16000 character circular buffer. This holds about 8 pages of packed text. When the buffer fills, the oldest characters are discarded. Characters in this window cannot be saved using cut and paste, but they can be captured to file. Only characters received after Capture to File has been activated will be saved. The number of characters saved is limited only by disk space.

Editor Window

This is the text editor window. The title area contains the full path and the name of the file being edited. When first opened using the menu commands File New the default name is “Untitled”. Notice the Close box in the upper left corner (called Cancel in a dialog window). When clicked on, the Close box attempts to close the DOS file associated with this window. If the window text has been modified since the last save you will be asked if you want to save the changes. You can agree to save changes, throw out all changes since the last save, or cancel the close.

When you open a program file, the text of the file is viewed in a File Edit Window. To open this window with the file you would like to edit, select Open from the File menu and you will be presented with the Open file dialog box.

The Editor

Many edit windows may be open at once, but only one can be active. Each edit window is an independent text editor. Text may be cut from one window and pasted to another. Characters are entered just before the flashing underline “_” cursor. Backspace removes characters before the cursor and Delete removes characters above the cursor. If a block cursor is displayed, the editor is in overwrite mode and each new character typed will overwrite the one under the cursor. The cursor type and editing mode are selected with the INS and DEL keys.

Editor windows continue to exist and be displayed even if they are not currently active. In the previous figure the Terminal Window isunselected, and therefore the border is a single line and the scroll bars (if any) are invisible. In contrast, the Untitled window has the double line border because it is selected.

To select a block of text, click and hold the mouse button over the beginning of the block and drag the mouse to the end of the block. When you release the mouse button, the cursor will appear at the point you release and the block will be selected. You can then cut, copy, paste or clear this block. Moving the mouse pointer will have no effect until you click again, which will deselect the block. Moving the cursor causes the selection to be lost, however.

You can copy and paste between edit windows by selecting the text in one window and copying it to the clipboard, then switching to another window and pasting. You cannot use cut and paste in the Terminal window. The contents of the Terminal Window may not be cut or pasted, but the using the Save to File command will store incoming characters to a file which can later be opened and edited.

Undo

The integrated editor has a limited Undo capability, using a Hot-Key keyboard command or Undo in the Edit Menu. It will only Undo one operation back, and as soon as you move the cursor, the Undo buffer is cleared and the previous operation cannot be undone.

Window titles This is always “Terminal Window” for the terminal window. In an edit window, if the file is opened using New and has not yet been saved, it will be labeled “Untitled”. Otherwise it contains the full path and name of the file being edited.

Zoom Box By clicking on the Zoom box, you can toggle the size of the window between full screen and a smaller size. The smaller size is determined by your last resizing of the window, which can be done manually by “grabbing” the resize corner and setting it to the size you want, or by using one of the Window menu commands, Tile or Cascade.

Move The Move arrow points to the top frame of the window, where the titles are located. You can move the window by clicking and holding anywhere along this top frame and then moving the mouse. The window frame will follow the mouse until you release the button.

Scroll Arrows Scroll arrows cause the text in the window to scroll one line in the direction of the arrow. Click and hold on these arrows to scroll continuously.

Scroll Box “Thumb” You can move more quickly through a document by click-hold-dragging on the scroll box Thumb. Moving the Thumb toward the top moves closer to the start of the document, and moving the Scroll Box toward the bottom moves closer to the end of the document. The window does not scroll until the Scroll Box is released.

Resize The size of the window can be adjusted by click-hold-dragging on the Resize corner of the window frame.

Zoom By clicking on the Zoom box, you can toggle the size of the window between full screen and a smaller size.

Close File The Close box appears only in Editor windows since the Terminal window cannot be closed. The Close box is in the upper left corner of the window. The Close box attempts to close the file displayed in this window. If the text has been modified since the last save you will be prompted with a dialog box asking if you want to save these changes. You can agree to save changes, throw out all changes since the last save, or cancel the close.

The Open file dialog box

This dialog box consists of four parts. The Name box contains a file name or wild card string. The Files box contains a list of files and sub-directories in the current directory. There is a scroll bar at the bottom of this box so you can access all the listings. At the bottom of the dialog box is information on the currently selected file. There are four buttons down the right side of the dialog: Open, Cancel, Parent and All (the label of this button can change, see below).

Choosing a file in the Open File dialog

To choose a file for opening, type the name of the file into the Name field and click Open (or hit the Enter key). A simpler way is to use the Tab key to move focus to the Files list box and then use the arrow keys to highlight the file you want to open. When the file is highlighted, click Open (or hit the Enter key). The simplest way to open a file is is to double click on the name in the Files list box.

Wild Cards

You can enter a wild card string (with optional path) in the Name box and the Files box will attempt to show all files matching this string. Be careful, if you use a name of a file that doesn't exist, and you click OK, a new window will open with that name. Then if you save the window, a file of that name will be created on your DOS disk. To cancel this dialog, you have three choices: click on the Cancel button, click on the Cancel box (on the left side of the top box frame) or hit the Escape key.

Switching directories

Selecting the Parent button moves you up one level in the directory tree and displays the files and sub-directories available there. The Next button cycles among three choices. It begins with the label All to show all files and sub-directories. If you click it once, it will change to TFB meaning that only files ending in the extension ".TFB" will be displayed (and all sub-directories). You can use this extension for all your TFBASIC programs and this button will show only those. If you click the button again it will change to Dir and show only the sub-directories in the current directory. Clicking again returns to All displaying all files and sub-directories.

The bottom button can be customized. You can save up to five extensions of your own (in addition to the All and Dir options which are always available). Hold down the Shift key while you select the button. You will see a dialog box that allows you to select any of five input boxes so you can type up to three characters in any box. Blank entries are ignored.

Navigating the IDE without a Mouse

Equivalent Keyboard commands

If your host computer does not have a mouse, if you only have one serial port and need to give up the mouse to connect the TFX-11, or if you just prefer the keyboard to the mouse, you may use keyboard command equivalents to perform all the functions of the mouse.

Dialog Box Navigation	Keyboard Equivalents
Cancel	Esc
OK	Enter
Move within group	Up/Down Arrows
Toggle check box	Space
Toggle Radio Button	Space
Next Group	Tab
Previous Group	Shift-Tab

Editing Control Functions	Keyboard Equivalents
Move Cursor	Arrow Keys
Cursor Word Left	Ctrl-Left Arrow
Cursor Word Right	Ctrl-Right Arrow
Delete Line	Ctrl-Y
Marking Blocks	Shift-Arrow keys
Select to Beginning of line	Shift-Home
Select to End of line	Shift-End
Select to Top of Page	Shift-PgUp
Select to Bottom of Page	Shift-PgDn
Begin Block	Ctrl-K B
End Block	Ctrl-K K
Move cursor one page	PgUp, PgDn
Move to Beginning of Line	Home
Move to End of Line	End
Delete to End of Word	Ctrl-T
Delete character at cursor	Del or Ctrl-G
Toggle Insert Mode	Ins
Delete to end of line	Ctrl-Q Y
Move one character left	Ctrl-A
Move one word left	Ctrl-S
Move one character Right	Ctrl-D
Move one word Right	Ctrl-F
Delete Character to Left	Backspace or Ctrl-H

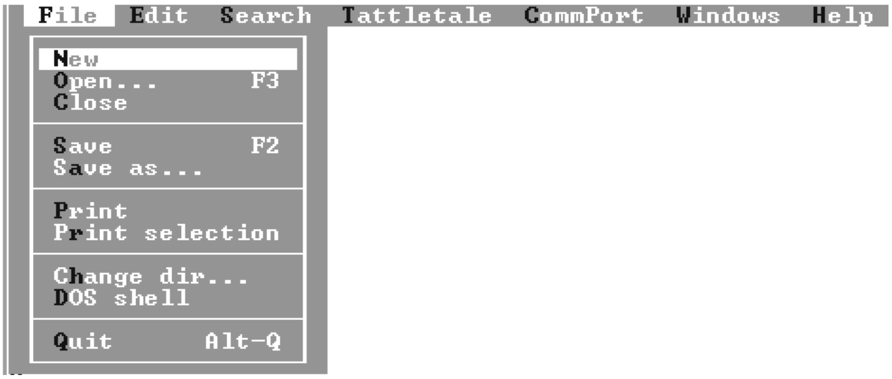
Main/Submenu items		Hot Key	Key sequence	Term	Edit
File		Alt-F			
	New	-	Alt-F N	*	*
	Open...	F3	Alt-F O	*	*
	Close	-	Alt-F C		*
	Save	F2	Alt-F S		*
	S**a**ve as...	-	Alt-F A		*
	Print	-	Alt-F P		*
	P**r**int Selection	-	Alt-F R		*
	C**h**ange Dir...	-	Alt-F H	*	*
	DOS shell	-	Alt-F D	*	*
	Quit	Alt-Q	Alt-F Q	*	*
Edit		Alt-E			
	Undo	Ctrl-U	Alt-E U		*
	Cu**t**	Ctrl-X	Alt-E T		*
	Copy	Ctrl-C	Alt-E C		*
	Paste	Ctrl-V	Alt-E P		*
	C**l**ear	Ctrl-Del	Alt-E L		*
	Paste **D**ate/time	Alt-D	Alt-E D	*	*
	Set PC Time	-	-	*	*
	Show clipboard	-	Alt-E S	*	*
Search		Alt-S			
	Find...	-	Alt-S F		*
	Find **a**gain	Ctrl-L	Alt-S A		*
	Replace	-	Alt-S R		*
TattleTale		Alt-T			
	Run	Alt-R	Alt-T R	*	*
	XMODEM off-load	-	Alt-T X		
	Erase Datafile	-	Alt-T E		
	Launch	Alt-L	Alt-T L		
	Rel**a**unch	-	Alt-T A		*
	Parallel offload...	Alt-0	Alt-T P	*	
	Suspend TFX-11	-	Alt-T S		*
	Load **O**S only	-	Alt-T O		*
	S**y**ntax Check	Alt-Y	Alt-T Y		*
	O**p**tions...	-	Alt-T P		*
CommPort		Alt-C			
	Serial Port...	Alt-P	Alt-C S	*	
	Hex display	Alt-X	Alt-C H	*	
	Capture to file...	Alt-Z	Alt-C C	*	
	Parallel Port...	-	Alt-C P	*	
Windows		Alt-W			
	Tile	-	Alt-W T	*	*
	Cascade	-	Alt-W C	*	*
	Next	F6	Alt-W N	*	*
	Previous	Shft-F6	Alt-W P	*	*
	25 line screen	-	Alt-W 2	*	*
	50 line screen	-	Alt-W 5	*	*
	C**o**lor Screen	-	Alt-W O	*	*
	Blk/wht screen	-	Alt-W B	*	*

Help		Alt-H			
	About	-	Alt-H A	*	*
	Command line options	-	Alt-H C	*	*
	TFBASIC summary	-	Alt-H T	*	*
	Keyboard equivalents	-	Alt-H K	*	*

CHAPTER 3

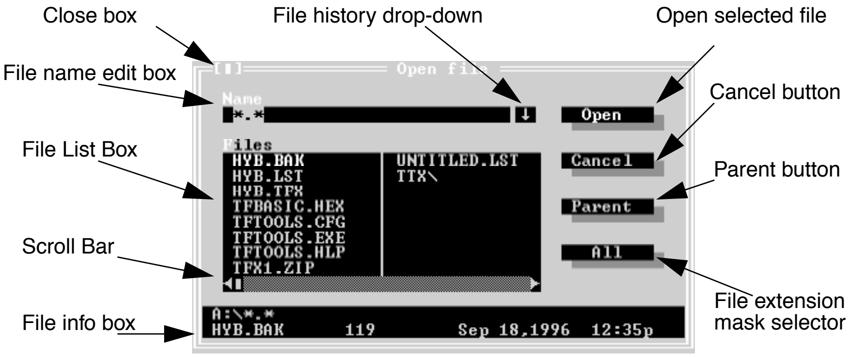
TFTools Pulldown Menu Command Reference

FILE



New Select this item to create a new program file. The default name will be "Untitled". When you go to save it you will be prompted for a new name to save it as.

Open... This command is used when you want to open an existing program file. It reveals the Open File dialog box which is shown below.



```
*.*
HYB.BAK        UNTITLED.LST
HYB.LST        TTX\
HYB.TFX
TFBASIC.HEX
TFTOOLS.CFG
TFTOOLS.EXE
TFTOOLS.HLP
TFX1.ZIP

A:\*.*
HYB.BAK       119       Sep 18,1996   12:35p
```

File name edit box. This is where the name of the file you want to open is entered. By default, the Name box will contain *.* and the list box will display all files and subdirectories. You can enter a wild card string (with optional path) and the Files list box will show all files matching this string. If you enter the name of a file that does not exist and click Open you will open a new editor window with that name. Then when you close the window, a file of that name will be created on your disk. To choose an existing file for opening, click once on the file in the file list box and it will appear in the file edit box. Then click Open (or press the Enter key) and the file listed in the “Name” field will be opened

File List box. A single click on a file name here will select it and copy the name to the file name edit box. A double click on the file name in this box will open it directly.

Open. This button has the same effect as hitting the Enter key, which is to open (or create if it doesn’t exist) the file that is in the Name box.

Cancel button and Close box. There are three ways to close this box without making any changes; press the Escape <Esc> key, click on Cancel, or click on the Close box.

File history drop down. This displays a history of the most recent files opened. It is not saved between sessions.

Parent Button. This moves you up to the next higher directory. There is a “..\” selection in the File list box which appears if you are in any other directory than the root. Clicking on this will have the same effect as clicking the Parent button.

File info box. The first line is the current path and file mask as specified by the file name edit box. The second line displays the name, size in bytes, and creation date and time of the currently highlighted file in the file list box.

File extension mask selector. This button selects up to seven predefined file masks. The active file mask determines which files will be displayed in the list box. All is equivalent to *.*. Each time the button is clicked the next extension in the list is selected. When the end of the list is reached it loops back to the first entry.

The “All” and “Dir” options are always available. In addition, this list can include up to five extensions of your own. To enter your own extensions you must first enter the Extensions dialog box. To do this you must hold down the shift key and then click the File extension mask selector button.

To add your own extensions, enter up to three characters in each box. These extensions are saved in the CFG file when you exit TFTools. The next time you want to use one of the new extensions, just click on the selector button until the desired extension is displayed.

Close Close the currently selected program file. If changes have been made in the file since it was opened, you will be asked if you want to save changes before closing.

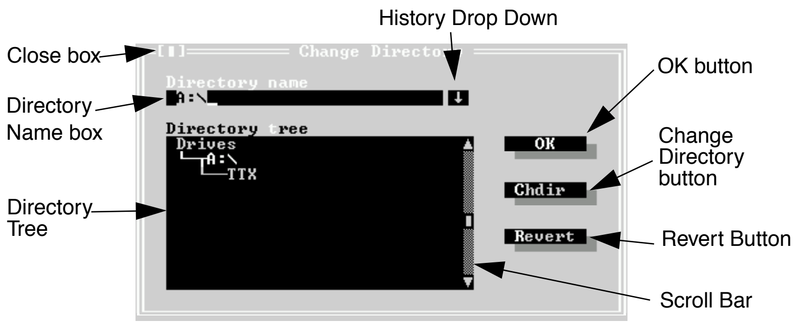
Save Save the program in the currently active edit window to the program file of the same name.

Save As This opens a dialog that is similar to the Open file dialog. It is used to save the program in the currently active edit window to a file of a different name. You will be prompted to enter the new file name. If the file already exists, it will ask to confirm before overwriting the file. If you select a file from the file list it will save to that name, but it will first verify that you do want to overwrite the existing file.

Print Send the program in the edit window to the DOS PRN device for printing. In this version of TFTools, no header or page eject commands are sent to the printer.

Print selection Send the part of the text that is selected (highlighted) to the printer. If no text is currently selected, you will be notified and no action will take place.

Change dir... Opens a dialog that allows you to change the current DOS working directory. You will be in this new directory when you exit TFTools.



Directory name box. This is where the name of the directory you want to move to is entered or displayed. By default, the Name box will contain the current directory. No files are displayed.

Directory tree. This is a graphical display of the directory hierarchy. A single click on a directory or subdirectory name will select it and copy the name to the edit box. A double click the Directory name in this view will move you into that directory automatically.

File history drop down. This displays a history of the most recent directories opened. It is not saved between sessions.

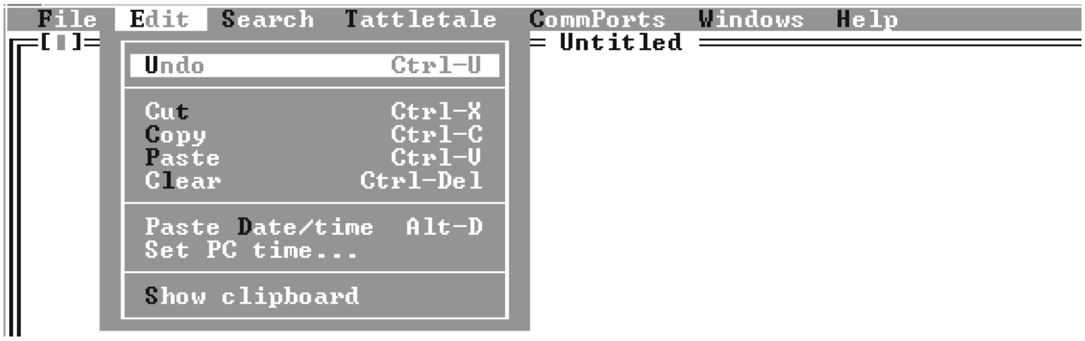
Change Directory button (Chdir). This button moves you to the directory highlighted in the tree window. It is the same as double clicking on the directory in the window. If the name in the Directory name box is different than the directory highlighted, the name box entry will be overwritten.

Revert. This puts you back in the same directory as when you entered the dialog.

DOS shell Suspends TFTools and launches a new copy of the DOS shell. IT DOES NOT REMOVE TFTOOLS FROM MEMORY so there won't be much memory for other programs! This will also close the serial port. You can execute any DOS commands here; even other communications programs. When you're done, use the DOS command EXIT to return to TFTools. The serial port will be opened in the same state it had before, except that any characters appearing at the serial port while running the second DOS will be lost. Don't forget: TFTools is still in memory.

Quit Exit TFTools, free up its memory and return to the DOS command interpreter.

EDIT



Undo Select this item to undo an editing action. Be aware that once you move the cursor, all Undo information is lost.

Cut Remove the currently selected text and save it in the clipboard for later pasting.

Copy Save a copy of the currently selected text in the clipboard for later pasting. Unlike Cut, this does not remove the selected text.

Paste Insert whatever is in the clipboard at the cursor. Use Cut or Copy to get text into the clipboard.

Clear Remove the currently selected text without saving a copy in the clipboard. You can Undo a Clear but not after the cursor has been moved.

Paste Date/time Insert a text string showing the current date and time of your PC's clock into the document. If the Terminal Window is currently selected, a date and time string will be sent out the serial port to the Tattletale. Just selecting this item sends a string of the form:

```
02/13/93 13:53:52
```

where 02 is the month, 13 is the day and 93 is the year. If you hold the Shift key down while this is selected, a longer date/time string of this form is used:

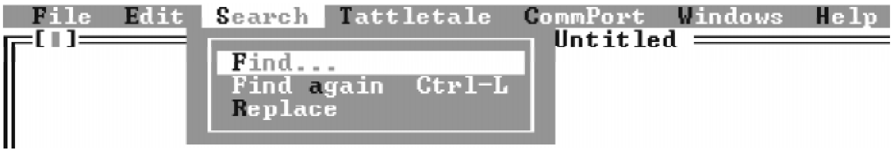
```
Friday, February 12, 1993, 13:53 PM
```

Holding down the Control key while executing this command causes the country-code information in your configuration file to be checked. The date and time will then be pasted in the format normally used in your country.

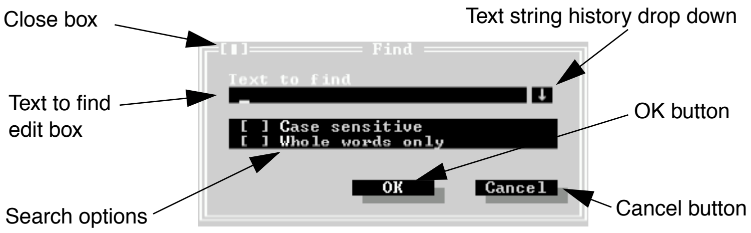
Set PC Time This selection allows the user to set the PC's internal clock without having to shell to DOS. On opening it displays the current PC time. If you modify the time and click OK it will become the new PC time. The PC time is stored in the TFX-11 on launching, so accurate time here will help assure correct time in the TFX-11.

Show clipboard Opens an editor-type window showing the contents of the Clipboard. You can edit the contents of the Clipboard using the normal editing commands. Only portions of the Clipboard that are selected are available for pasting into other edit windows, so be sure to select that portion of the Clipboard text before exiting this window.

SEARCH



Find... A dialog box will appear to allow you to enter a string to look for in the document.



OK. This button has the same effect as hitting Enter key, which is to move you to the directory that is displayed in the directory name box and exit the dialog.

Cancel. Choose Cancel if you want to exit and forget the whole thing.

Close box. There are two ways to close out of this dialog box without any changes; press the Escape <Esc> key, or click on the Close box.

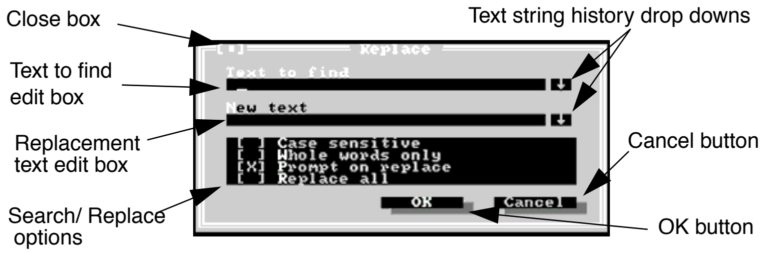
Text to find edit box. Type the string to search for into the box labeled “Text to find”. Qualify the search with the search options check boxes.

Text string history drop down. This displays a history of the most recent directories opened. It is not saved between sessions.

Search Options. You can check either or both of the options, Case sensitive and Whole words only, by clicking the mouse between the brackets to the left of the labels. Alternatively you may select either or both of these options by using the Tab and Arrow keys to work the highlight down to the selection, and then hitting the Space key to toggle the check mark on and off.

Find Again Once a string has been found with Find, you can continue to look through the document for more occurrences of the same string. It is usually easier to use the Ctrl-L keyboard equivalent for this command.

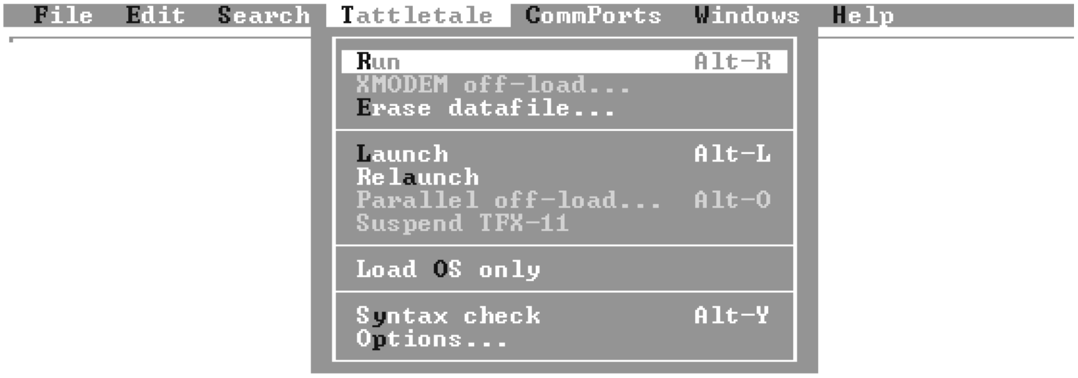
Replace... A dialog box will appear to allow you to enter a text string to search for and a second string to replace it with.



```
[■]            Replace
Text to find
New text
[ ] Case sensitive
[ ] Whole words only
[X] Prompt on replace
[ ] Replace all
```

Use of this dialog is similar to that of the Find dialog except that a second text box is available and there are some more options. Be aware that entering nothing in the New text box means that found text will be erased. Notice that Prompt on replace is normally on (that box is selected with an x). Checking Prompt on replace and Replace all will automatically look for the next occurrence of Text to find after each replace but will ask you first before a replacing text.

TATTLETALE



Run If selected while the Terminal Window is active, a Run command is sent to the Tattletale to execute the program currently in RAM. If selected while an edit window is active, the program in the editor is first checked to see if it has been modified. If it has been modified, TFTOOLS then compiles it and uploads it to the Tattletale. Then the focus switches to the Terminal Window and TFTOOLS sends a RUN command to the Tattletale, starting execution of the loaded program. If power is totally removed from the Tattletale the program will be lost.

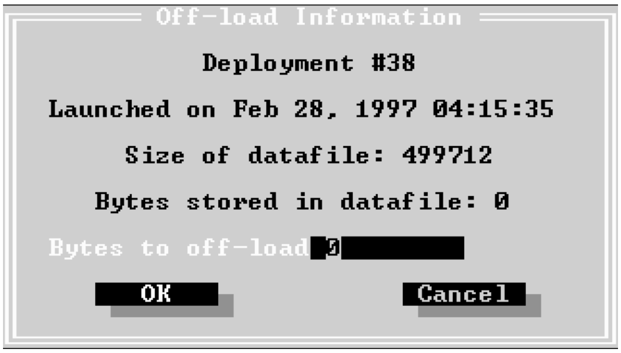
XMODEM Off-load... This initiates an XMODEM off-load of the flash datafile on the TFX-11. Since this is done serially it will be substantially slower than using the parallel version. This option is deigned for small files or remote applications that only have a serial connection. Otherwise the parallel version is recommended.

Erase datafile... This command allows erasure of the flash datafile over the serial port. It does not erase the program. To access this with a serial only link you must be able to break the TFBASIC program and return to the # prompt.

Launch Causes the program in the edit buffer to be tokenized and loaded into the Tattletale's flash storage. The Datafile is erased and the datafile pointer is reset to 0. If there is data in the Tattletale it puts up a dialog box that allows you to save the data before the Launch proceeds.

Relaunch Causes a reset of the Tattletale which will restart the currently burned in program. The datafile is not erased nor is the datafile pointer reset.

Parallel off-load... This command is only enabled in the Terminal Window. It presents you with a dialog box to allow you to choose how to off-load the datafile of the Tattletale. You cannot offload less than the number of bytes given in the “Bytes to off-load” edit box, but you may offload more, up to the number specified in “Size of datafile”. After finishing with this dialog box check OK and a second dialog opens that allows you to select the name of the file to save the data to. You can save this file anywhere in the DOS directory tree.


```
Off-load Information
Deployment #38
Launched on Feb 28, 1997 04:15:35
Size of datafile: 499712
Bytes stored in datafile: 0
Bytes to off-load 0
```

Deployment number. Number of times unit has been launched.

Launched on. This is the PC date and time stored when the program was launched. It is also used to set the TFX-11 internal clock on launch.

Size of datafile. This is the total number of bytes available for storing data in the SFLASH after the program and other information is stored.

Bytes stored in datafile. This is the total amount of data bytes stored in the SFLASH.

Bytes to off-load. Input box to enter the number of the bytes of the datafile to off-load.

OK. Clicking here will start the data off-load

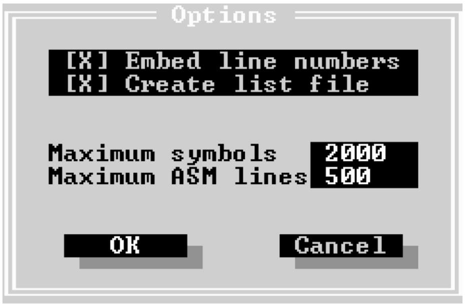
Cancel. Clicking here exits without any action.

Suspend TFX-11 Places the TFX in its lowest power state.

Load OS Only Loads only the TFBASIC operating system and requires the parallel cable be attached to do so. This erases any program loaded in the Tattletale so the next time it boots up it will give a sign-on message followed by the # prompt. This returns the TFX-11 to the way it was when it was shipped.

Syntax check Tokenizes the program in the edit buffer and reports errors but does not attempt to load the program into the Tattletale. This is affected by the Option flags (see below). Upon a successful check, an information box will appear showing the size (in bytes) of the compiled program, the size of the program header (this is information generated by the compiler and will be added to the front of the program) and the size of the variables area used by this program.

Options... Sets startup options for tokenizer and program upload options:



[X] Embed line numbers
[X] Create list file

Maximum symbols 2000
Maximum ASM lines 500

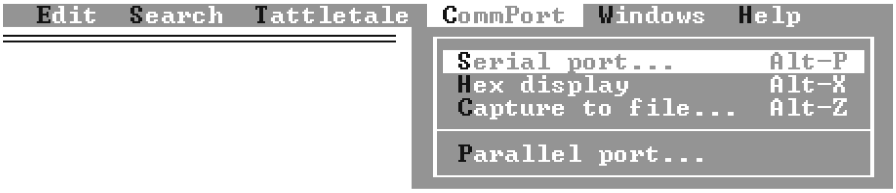
Embed line numbers. Line numbers relating to the text file will be inserted in the listing file. This helps relate tokens to actual code.

Create List File. The next time the tokenizer runs (whether through the Run or Launch command or the Syntax check command), an annotated listing of the program is stored in a DOS text file. The file will have the extension LST.

Maximum symbols. This sets the amount of memory allocated by TFTools for its symbol tables. Reduce this number if you get an “not enough DOS memory” message.

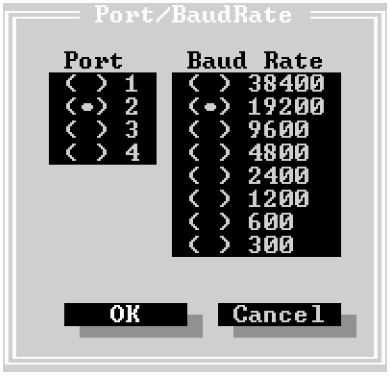
Maximum ASM lines. This setting determines the amount of memory allocated for compiling assembler instructions. If you do not have any assembly code you may reduce this number to allow more memory for symbols. Reduce this number if you get an “not enough DOS memory” message.

COMMPORT



```
Edit  Search  Tattletale  CommPort  Windows  Help
                          Serial port...       Alt-P
                          Hex display          Alt-X
                          Capture to file...   Alt-Z
                          Parallel port...
```

Serial Port... Allows you to set the com port parameters. These values will be stored in a file called TFTOOLS.CFG when you exit TFTools. Notice that these items are groups of radio buttons and only one item of a group can be selected at any one time.



```
      Port/BaudRate
 Port      Baud Rate
 ( ) 1     ( ) 38400
 (•) 2     (•) 19200
 ( ) 3     ( ) 9600
 ( ) 4     ( ) 4800
           ( ) 2400
           ( ) 1200
           ( ) 600
           ( ) 300

   OK        Cancel
```

The Port radio button selects one of the four serial ports available, and the baud rate default is 19200.

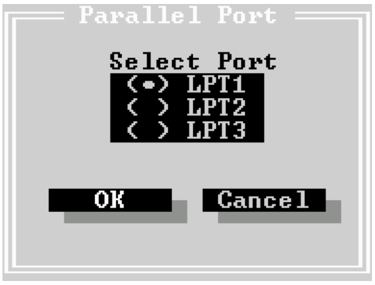
Hex display Toggles the Terminal Window into and out of hexadecimal display mode. In this mode, any incoming characters are displayed in hexadecimal form in rows of 16 characters on the left side of the screen. The ASCII (printable) equivalent is displayed in 16 character rows on the right side of the screen. Toggling this mode always forces output to start on a new line.

Capture to File... This selection permits the collection of all terminal screen data, both input and output, to be captured to a disk file. It is useful for debugging or direct recording and storing of data.

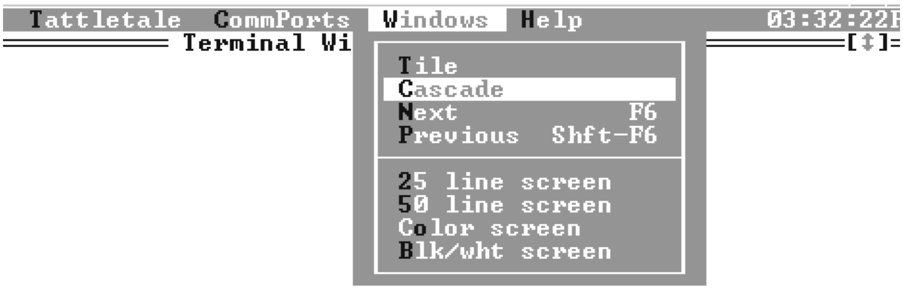
The first time you select this command in a session you will be presented with the open file dialog box, which prompts you for the name of the file to capture to. (For details on the operation of the Open file dialog box, see “Open...” under the Main Menu command “File”) A default name of CAPTURE.TXT is already filled in and can be accepted by simply clicking on Open or hitting the Enter key. After the file is open and active you can use Alt-Z or the menu item to toggle the capture mode on and off. Also, the word "CAPTURE" is displayed in the Status line in the lower-right corner of the display. In color mode, the status line background is changed to green as another reminder. When you toggle capture off, the file is closed and the status line returned to normal.

There are two file capture modes, Append and Overwrite. The default mode is overwrite and that means each time you activate Capture to file with a file already open you will overwrite and therefore loose the contents of the existing file. You can change this to append by holding down the Shift key while selecting Capture mode. You will be asked if you want to change the Capture mode to append. Special note - To capture to a printer you may use the special DOS logical name of PRN as the file name.

Parallel Port... This opens the following dialog which presents you with the selections for the Parallel port to be used to communicate with the Tattletale. Your selection will become the start-up default as it will be automatically saved in the configuration file when you exit the program.

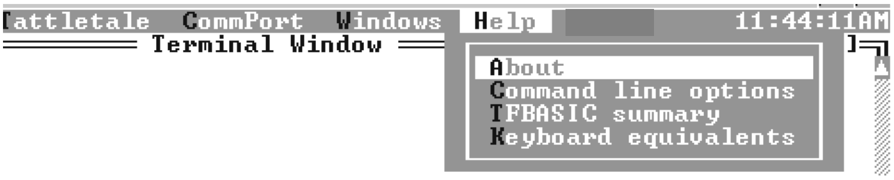


WINDOWS



Tile, Cascade	These allow you to automatically rearrange all the open windows on your screen.
Next, Previous	These will bring different windows into focus in forward or backward order. These are only needed if a particular window is not visible (in which case, the mouse can be used to bring it into focus by clicking on it) or if you have no mouse.
25 line screen	Sets the display mode to the default 80x25 character display.
50 line screen	If you have an EGA or VGA screen, this item toggles the screen to a higher resolution mode. EGA is capable of a 43 line per screen mode and VGA is capable of a 50 line per screen mode. This has no effect on CGA screens.
Color screen	Puts a color-capable screen into color mode. Don't use this if you have a monochrome screen.
Blk/wht screen	Puts the screen in a black and white mode. This is most useful on monochrome LCD screens.

HELP



About Selecting this item displays the About Box dialog showing the version number of TFTools you are using. The information located in here is important if you ever have to contact technical support.

Command line options Shows options you can set when you start TFTools from the DOS command line.

TFBASIC summary A list of TFBASIC commands and keywords in alphabetical order. To see more of the list, use the arrow keys or use the mouse to move the scroller thumb. You can get a brief explanation of a command by double clicking on it with the mouse. Keyboard users can step through the commands with the TAB key and select a command by hitting Enter. If there is a "See also..." highlighted selection, you can change to that topic by selecting it in the same way you selected the original command. When you're done with the help system, either click on the Close box or hit the Escape key.

Keyboard equivalents A list of TFTools menus and editor actions and the key or key combinations that trigger them. It is essentially a copy of the tables at the end of Chapter 2 under the heading “Navigating the IDE without a Mouse”. When you're done with the help system, either click on the Close box or hit the Escape key.

CHAPTER 4

TFBASIC Language Reference

Legend

· str$	string
· x	expression
· v	variable
· m	format
· \x	8-bit character given by x
· $	Inline assembly code
· label	line reference
· [ ]	the commands enclosed within these brackets are optional

Predefined Read-only variables in TFBASIC

These variables allow you to access certain useful internal values in TFBASIC. If you try to assign a value to these variables it will have no effect. In the future it may generate either a syntax or HOW? error.

BAUDGET	Returns the value of the main UART's current baud rate
DFERASED	Returns 1 if the SFLASH datafile is empty
DFMAX	Maximum datafile address (size of datafile - 1).
DFPNT	Points to the datafile location that will receive the next data byte.
MODEL	Contains model number of TFX.
VERS	Version of TFBASIC * 100.
FPERR	Floating point errors. Each of the five errors has a bit assigned to it. FPERR is cleared any time it is accessed so it should always be assigned to a user variable first before checking it.
BBPWR	Returns a value that indicates whether the power source is from the main supply or battery backup. A value of 0 indicates the main power supply is active, and a non-zero value means the power source has switched to the backup battery.
INTSTATUS	This is a read only variable that can be used anywhere in the program to see if the PIC interrupt is enabled. 0 = disabled, 1 = enabled. This variable is set when TFBASIC successfully completes the PICINT command, and is reset when XIRQ completes processing of the related interrupt.

TFBASIC Quick Reference (grouped by function)

Program Commands	· ASM $...end	Assemble HC11 code inline
	· ASM<addr>...end	Assemble HC11 code at <addr>
	· CALL x1,x2[,v]	Call x1 with registers = x2; returning in v
	· CBREAK label	Go to label if CTRL-C character detected
	· CBREAK	Return CTRL-C handling to default
	· COUNT <v>	Start counting transitions on I/O 0 and store in variable
	· COUNT	Stop counting transitions
	· DIM <label> (size)[,(size)]	Dimension variable <label> to 'size'
	· FOR v=x1 to x2 [STEP x3]... NEXT v	Initiate iterative for loop
	· GOSUB label	Execute subroutine at label
	· RETURN	Return from subroutine
	· GOTO label	Go to label
	· IF x ... [ELSE...] ENDIF	Execute 1st command block if x is true, else 2nd block
	· INPUT [s] v[,x][,#x][,\x][;]	Prompt with s, load variable v (see full description for complete arg list)
	· ONERR label [,v]	Go to label if error [, error in v]
	· ONERR	Return error handling to default behavior
	· POKE addr, value	Store byte 'value' at address 'addr'
	· PRINT "s", [#n][,x][,\x][;]	Print string to console
	· REPEAT ... UNTIL x	Repeat command block until x is true
	· STOP	End program execution
	· UGET x1, x2, v, x3	Store data from software UART to string
	· USEND baud, <string>	Send string data out software UART
	· VSTORE addr, value	Store value to UEEPROM address addr
	· WHILE x ... WEND	Execute command block while x is true
	· XMIT+, XMIT–	Enable, disable console output
Datafile Storage Commands	· STORE [#n],x...	Store x to EEPROM, using n bytes
Functions	· ABS(x)	Absolute value of x
	· AINT(x)	round float down to integer
	· ASFLT(x)	Interpret x as float

	· ATN(x)	Arctangent of x
	· BAUDSET (x)	Sets the main UART baud rate
	· COS(x)	Cosine of x
	· COUNT (x)	Return number of cycles at I/O pin 0 in time x
	· CHAN(x)	A-D conversion of channel x
	· EXP(x)	Return e raised to the x power
	· FIX(x)	Integer part of x as integer closer to zero
	· FLOAT(x)	Convert integer x to float
	· FVAL(str$)	Convert string to float
	· INT(x)	Integer part of x as integer more negative
	· INSTR([x,] str1$, str2$)	returns a substrings position in a string
	· IVAL(str$)	String to integer
	· LEN(str$)	Return length of string
	· LOG(x)	Natural log of x
	· LOG10(x)	Common log of x
	· MID(str$,x1,x2)	Return substring of str$
	· PEEK(addr)	Byte at address 'addr'
	· PERIOD (x1, x2)	Time for x1 cycles of signal to pass - x2 is timeout
	· SIN(x)	Sine of x
	· SQR(x)	Square root of x
	· STR (["s"][,#n][,x][,\x])	Create string
	· TAN(x)	Tangent of x
	· TEMP (x)	Convert x to degrees C (times 100) for thermistor input
	· VARPTR (v)	Address of variable v
	· VGET (x)	Return value of UEEPROM at address x
Digital I/O Control	· PCLR x1 [,x2 . . .]	Clear specified I/O pins to low state
	· PIN (x1 [,x2, . . .])	Value formed from specified input pins
	· PSET x1 [,x2 . . .]	Set specified I/O pin to high state
	· PTOG x1 [,x2 . . .]	Toggle state of specified I/O pin
	· PICINT x1 [, x2, x3]	Set up external PIC interrupt
	· SDI x	Shift in x bits
	· SDO <string>	Shift out string expression

	· SDO x1, x2	Shift out x1 value using x2 bits
	· TONE x1, x2	Send x2 cycles of square wave of period x1 - continuous if x2=0
Low Power and Time Commands	· HALT	Stop in lowest power mode
	· HYB (x)	Dormant mode
	· RATE	Change SLEEP interval timing
	· READRTC [v]	Transfer hardware RTC time from PIC to local RTC
	· RTIME [v]	Translate from softtware RTC or variable time-in-seconds to '?' array
	· SLEEP x	Sleep till x*10 ms from last SLEEP
	· SETRTC [v]	Set the hardware RTC time in the PIC
	· STIME [v]	Write '?' array data to software RTC or variable

TFBASIC Quick Reference (alphabetical)

· ABS(x)	Absolute value of x
· AINT(x)	round float down to integer
· ASFLT(x)	Interpret x as float
· ASM $...end	Assemble HC11 code inline
· ASM<addr>...end	Assemble HC11 code at <addr>
· ATN(x)	Arctangent of x
· BAUDSET (x)	Sets the main UART baud rate
· BAUDGET	Gets the main UART baud rate
· CALL x1,x2[,v]	Call x1 with registers = x2; returning in v
· CBREAK label	Go to label if CTRL-C detected
· CBREAK	Return CTRL-C handling to default
· CHAN(x)	A-D conversion of channel x
· COS(x)	Cosine of x
· COUNT <v>	Start counting transitions on I/O 0 and store in variable
· COUNT	Stop counting transitions
· COUNT (x)	Return number of cycles at I/O pin 0 in time x
· DIM <label> (size)[,(size)]	Dimension variable <label> to 'size'
· EXP(x)	Return e raised to the x power
· FIX(x)	Integer part of x as integer closer to zero
· FLOAT(x)	Convert integer x to float
· FOR v=x1 to x2 [STEP x3]... NEXT v	an iterative loop
· FVAL(str$)	Convert string to float
· GOSUB label	Execute subroutine at label
· GOTO label	Go to label
· HALT	Stop in lowest power mode
· HYB (x)	Dormant low power (uA) mode
· IF x ... [ELSE...] ENDIF	Execute 1st command block if x is true [,else execute 2nd block]
· INPUT [s] v[,x][,#x][,\x][;]	Prompt with s, load variable v (see full description for complete arg list)
· INSTR([x,] str1$, str2$)	returns a substrings position in a string
· INT(x)	Integer part of x as integer

· IVAL(str$)	String to integer
· LEN(str$)	Return length of string
· LOG(x)	Natural log of x
· LOG10(x)	Common log of x
· MID(str$,x1,x2)	return substring of str$
· ONERR label [,v]	Go to label if error [, error in v]
· PCLR x1 [,x2 . . .]	Clear specified I/O pins to low state
· PEEK(addr)	Byte at address 'addr'
· PERIOD (x1, x2)	Time for x1 cycles of signal to pass with x2 as timeout
· PICINT x [,y ,z]	Set up external PIC interrupt
· PIN (x1 [,x2, . . .])	Value formed from specified input pins
· POKE addr, value	Store byte 'value' at address 'addr'
· PRINT ["s"][,#n][,x][,\x][;]	{any mix}
· PSET x1 [,x2 . . .]	Set specified I/O pin to high state
· PTOG x1 [,x2 . . .]	Toggle state of specified I/O pin
· RATE	Change SLEEP interval timing
· READRTC [v]	Transfer hardware RTC time from PIC to local RTC
· REPEAT ... UNTIL x	Repeat command block until x is true
· RETURN	Return from subroutine
· RTIME [v]	Translate from softtware RTC or variable time-in-seconds to '?' array
· SETRTC [v]	Set the hardware RTC time in the PIC
· SIN(x)	Sine of x
· SLEEP x	Sleep till x*10 ms from last SLEEP
· SQR(x)	Square root of x
· STIME [v]	Write '?' array data to software RTC or variable
· SDI x	Shift in x bits
· SDO <string>	Shift out string expression
· SDO x1, x2	Shift out x1 value using x2 bits
· STOP	End program execution
· STORE [#n],x.	Store x to EEPROM, using n bytes
· STR (["s"][,#n][,x][,\x])	Create string
· TAN(x)	Tangent of x

· TONE x1, x2	Send x2 cycles of square wave of period x1 - continuous if x2=0
· UGET x1, x2, v, x3	Store data from software UART to string
· USEND baud, <string>	Send string data out software UART
· TEMP (x)	Thermistor conversion, degrees Cx100
· VARPTR (v)	Address of variable v
· VGET (x)	Return value of UEEPROM at address x
· VSTORE x, y	Store y to UEEPROM address x
· WHILE x ... WEND	Execute command block while x is true
· XMIT+, XMIT–	Enable, disable console output

ABS *absolute value*

Syntax: ABS(<x>)

Description: ABS returns the absolute value of the expression in the parenthesis. The function takes either an integer or a floating point argument and returns a corresponding integer or floating point value.

Example:

```
print abs(7)
print abs(-7)
print abs(7.0)
print abs(-7.0)
```

Output:

```
7
7
7.000000E0
7.000000E0
```

Cautions: Integer arguments outside the range of Tattletale integers (-2147483648 to 2147483647) will halt the program with a run time error.

Remarks: None.

See Also:

AINT *round float down to integer*

Syntax: value = AINT(<x>)

Description: AINT returns the next integer value less than the argument. The value is returned as a float. The argument must be a float. If it is an integer, it will be converted to float first.

Example:

```
inData! = 23.7
result! = aint( inData! )
print "aint of ",#.5F, inData!, " = ", result!
result = aint( –inData! )
print "aint of ",#.5F, –inData!, " = ", result!
```

Output:

```
aint of 23.70000 = 23.00000
aint of –23.70000 = –24.00000
```

Cautions: Remember, this function does not simply strip off the fractional part of the argument. Negative numbers return the next lower whole number!

ASFLT *interpret argument as float*

Syntax: ASFLT(<x>)

Description: Integer and float variables both take up four bytes of storage. There is nothing in the storage format that allows the two to be distinguished from each other. Therefore all variables are assumed to be integer unless otherwise designated. ASFLT is used to tell TFBASIC to interpret data retrieved from storage as a floating point value.

Example:

```
FltVal! = 100.0             // create floating point value
vstore 10, FltVal!          // store it in UEEPROM

print vget(10)              // retrieve and print as integer
                            // (default)

print #F, asflt(vget(10))   // retrieve and print as float

stop
```

Output:

```
1120403456
100.00
```

Cautions: This function does not convert the data - it only tells TFBASIC how to interpret it correctly, assuming it was stored as a floating point value.

Remarks: None.

See Also: VGET

ASM *assemble to memory*

Syntax: ASM $ or ASM <address>

<code>

...

...

END

Description: Assembly language can be written directly using ASM. HC11 instructions, with some limitations (see the TFBASIC Assembly Language reference for details), are supported in all addressing modes. Assembly starts when it encounters the ASM command and stops when it encounters the END command. This assembler allows the use of named labels and it can access TFBASIC variables by name.

If the optional <address> is included, the code is assembled starting at the specified address. If <address> is replaced with the $ character, assembly is done in line and automatically called when reached.

Labels: Labels can be used in the assembly code for flow control and to define local variables. Labels MUST start in the first column. Labels can be up to 32 characters long and must begin with a letter or an underscore (_). The only valid characters in a label are upper and lower case characters, the numbers and underscore. The label name can be terminated with a colon (when the label is defined) but this is not necessary in the assembler. These labels will not be accessible to TFBASIC except through the CALL command and TFBASIC labels are not accessible to the assembly code (although TFBASIC variables are).

Opcodes: The TFBASIC assembler recognizes most of the opcodes defined in the Motorola literature. See the section "TFBASIC Assembly Language" for details. Opcodes must have at least one character of whitespace (space character or tab) in front of them on the line OR a label terminated with a colon.

Inline assembly code (ASM $) This version allows you to install assembly language code that will be executed in line with the TFBASIC code and has the form:

```
<TFBASIC code>
ASM $  (nothing else on this line, not even comments!)
<assembly code>
<assembly code>
.
<assembly code>
end
<TFBASIC CODE>
```

In the assembly section, everything after a semicolon and up to the end of the line is considered a comment. The assembler does not recognize TFBASIC comments. A more detailed explanation of this form of ASM is given in the TFBASIC Assembly language section in this manual.

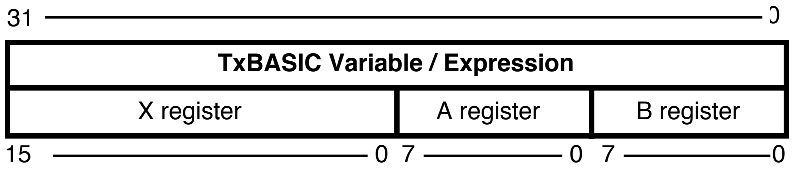
Notice that the first form of the ASM command provides no way to initialize the A, B or X registers before entering the assembly code section. This can be done with the second form of ASM.

Assembly to an address (ASM <address>)

When the interpreter reaches this point in the program, it DOES NOT EXECUTE THE ASSEMBLY CODE. Instead it loads the code to the address specified by the ASM command until it finds the 'end' statement:

```
<TFBASIC code>
ASM <address> (nothing else on this line, not even comments!)
<assembly code>
<assembly code>
<assembly code>
end
<TFBASIC CODE>
```

The A, B and X registers have a total of 32 bits. CALL initializes these on launch using '<input parameters>', and returns their values at exit in the '<optional output variable>' In both cases the registers are packed the same way:



The assembler automatically appends an RTS to the end of your code. If your assembly routine is launched by an interrupt you should end your code with an RTI. The assembler will append an RTS to this but it will not be executed.

A more detailed explanation of this form of ASM is given in the TFBASIC Assembly language section earlier in the manual.

Radix: Assembly code allows more methods of defining the number base of constants. You have these options IN THE ASSEMBLER ONLY in defining a constant 19 decimal as:

- hexadecimal: 13H or &H13 or H'13 or $13 notice &H works as in TFBASIC
- octal: 23O or 23Q or Q'23 or @23
- binary: 10011B or B'10011 or %10011
- decimal: 19 or 19D or D'19 - decimal is the default number base

Remarks: none

See Also: CALL and the "TFBASIC Assembly Language" manual section.

ATN *arctangent*

Syntax: value = ATN(<x>)

Description: ATN returns the angle (in degrees!) of the expression in the parenthesis. The function takes a floating point argument and returns a floating point value. An integer argument will be converted to float first.

Example:

```
tangnt! = 1.0
degrees! = 0.0

degrees! = atn(tangnt!)
print “The arctangent of “, #7.1F, tangnt!,“ is “,#6.3F,degrees
```

Output:

```
The arctangent of   1.0 is 45.000
```

Remarks: If the argument is +inf the result is 90.0

If the argument is -inf the result is -90.0

Cautions: **DON’T FORGET!** The result is in degrees, not radians.

See Also: COS, SIN, TAN

BAUDSET *Set the baud rate of the main UART*

Syntax: Baudset (x)

Description: Sets the send and receive baud rate of the main UART to the selected baud where x = baud. Baud is one of these: 300, 600, 1200, 2400, 4800, 9600, 19200, or 38400

Remarks: The default baud rate on reset is always initialized to19200.

Cautions: **DON'T FORGET!** Changing the baud rate on the TFX-11 will NOT automatically change the baud rate of TFTools. If the baud is set incorrectly CTRL-C will no longer work to break the program.

See Also: BAUDGET

BAUDGET *Get the baud rate of the main UART*

Syntax: value = BAUDGET

Description: Baudget returns the value of the current baud rate of the main UART. The function returns an integer. Value is one of the following bauds: 300, 600, 1200, 2400, 4800, 9600, 19200, or 38400

Remarks: The default baud rate on reset is always initialized to 19200.

Cautions: None

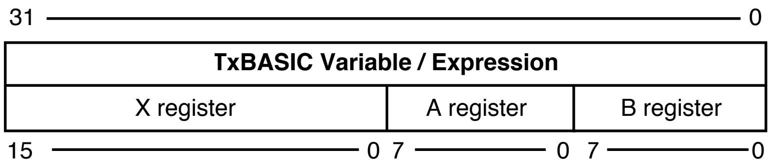
See Also: BAUDSET

CALL *Call an assembly language subroutine*

Syntax: CALL addr or <asm label>, regs [, var]

CALL <x1>,<x2> [,<v>]

Description: CALL executes a user loaded assembly language subroutine. The A, B, and X registers are loaded with the value in the "regs" expression on entry to the subroutine, and on exit, the optionally specified variable returns with the register contents. The packed format for register passing is shown below:



Cautions: When you call an assembly language program, you are leaving the warmth and safety of the TFBASIC programming environment. Obviously, the power to access all of the registers, ports, and memory also affords a path to catastrophic program crashes which may be very difficult to diagnose.

Remarks: None.

See Also: ASM command and the "TFBASIC Assembly Language" section of the manual.

CBREAK *go to label on CTRL-C*

Syntax: CBREAK [label]

Description: This command allows you to redirect the flow of your program when it receives a Ctrl-C character. Normally, the program terminates and returns to the monitor. If you use the CBREAK command (followed by a label name) at the beginning of your program, it will vector to the label when it receives a Ctrl-C character. To return to the normal Ctrl-C action, use CBREAK with no argument.

Example: (indentation for clarity only)

```
       CBREAK getdata         // goto 'getdata' when Ctrl-C hit
       sleep 0
savedata:
       for icount = 1 to 2000  // collect 2000 data pts
        store #2,chan(1)
        sleep 5
       next icount
       goto savedata           // reset datafile pointer to start
getdata:
       print "ready to off-load"
       stop
```

This code fragment shows one use for CBREAK. When a Ctrl-C character is received, the program will vector to the code at label 'getdata' and be at the # prompt awaiting the command for an XMODEM off-load.

Remarks: The Ctrl-C handler may be changed any number of times by executing CBREAK with different arguments.

As with ONERR , CBREAK must be executed to be effective. For this reason, you should put CBREAK near the beginning of the program.

Disable CTRL-C breaks by writing a zero byte to address 9C hex (POKE H'9C,0). Re-enable breaks by writing a non-zero byte (POKE H'9C,1). A count of CTRL-C characters will continue to be updated at address 9B hex. To clear this before you re-enable break-outs use (POKE H'9B,0).

Cautions: Do not vector CBREAK to a label inside a subroutine. When a CTRL-C is detected the program will act as if you did a GOTO into the subroutine. It will not have a proper return address on the stack and will probably crash.

CHAN *get result of A-D conversion*

Syntax: value = CHAN(<x>)

Description: The CHAN command returns a digital value corresponding to ratio of the voltage at the input channel (specified by <x>) to the converter's reference voltage input. The TFX has eleven 12-bit channels, channels 0 thru 10, and eight 8-bit channels, channels 11 thru 18, for a total of nineteen analog input channels.

The analog inputs are designed to handle signals that range from 0 to the converter's Vcc , typically +5V. The result of the conversion is left justified to produce a 16-bit result, regardless of the number of bits in the converter. The result of the 12-bit converter is shifted left four bits (multiplied by 16), and that of the 8-bit converter is shifted left eight bits (x256). To to restore to there unjustified form divide by their multipliers.

Example:

```
// **** CHAN EXAMPLE *****
for counter = 1 to 10
value = chan(0)
print #016B, value,'B  ',#04H, value,'H  ',#1D,value
next counter
```

Input mapping:

CHAN	PIN	CHAN	PIN	CHAN	PIN	CHAN	PIN
0	A36	6	A42	11	A26	17	A31
1	A37	7	A43	12	A28	18	A33
2	A38	8	A44	13	A30		
3	A39	9	A45	14	A32		
4	A40	10	A46	15	A27		
5	A41			16	A29		

Remarks: All 19 channels are set to make ratiometric readings. channels 0-11 may be modified to make absolute measurements by the addition of an external precision reference. See Using the onboard A/D converters

A/D Channels 11-18 may be reconfigured to be digital inputs.

Cautions: Specifying channels less than 0 or greater than 18 will generate a run-time error.

See Also: Section Using the Onboard A/D converters.

COS *cosine*

Syntax: value = COS(<x>)

Description: COS returns the cosine of the expression in the parenthesis. The argument must be in degrees. The function takes a floating point argument and returns a floating point value. An integer argument will be converted to float first.

Example:

```
degrees! = 0.0  // init arg (notice '!' means it's a float)
result! = 0.0   // just to force 'result' to be a float
for i = 1 to 6
   result = cos(degrees)
   print "The cosine of ",#5.1F,degrees," is ",#6.3F,result
   degrees = degrees + 72.0
next i
```

Output:

```
The cosine of   0.0 is  1.000
The cosine of  72.0 is  0.309
The cosine of 144.0 is –0.809
The cosine of 216.0 is –0.809
The cosine of 288.0 is  0.309
The cosine of 360.0 is  1.000
```

Remarks:

Cautions: **DON’T FORGET!** The argument is in degrees, not radians.

See Also: SIN, TAN, ATN

COUNT *count positive edges at I/O line 0*

Syntax: cycles = COUNT (<x>) count function

or

COUNT <v> count command

Description: There are two versions of COUNT. The count function counts the number of square wave cycles (positive edge is counted) appearing at I/O line 0 during a specified time. Specify the duration <x> in hundredths of seconds between 1 and 65535.

The count command works in the background incrementing variable <v> at every positive edge on I/O line 0. COUNT <v> starts the background counter and COUNT with no arguments disables the count interrupt. The COUNT and PERIOD functions may not be used while the COUNT command is running.

Examples: Write this program in TFTools:

```
for N = 2000 to 6000 step 2000
print "Set up for ", N, " Hz, hit <cr>...";
input "" A;  // input to dummy var to wait
print "  reads ", count(100), " Hz"
next N
```

Output:

```
Set up for 2000 Hz, hit <cr>...  reads 1998 Hz
Set up for 4000 Hz, hit <cr>...  reads 4000 Hz
Set up for 6000 Hz, hit <cr>...  reads 6003 Hz
```

Write this program in TFTools:

```
count backCount       // start background count, store in backCount
SLEEP 0
SLEEP 2
A=backCount           // save current count, counting continues
SLEEP 100
B=backCount
PRINT B—A             // get second count, show difference
count                 // stop the background count
PRINT COUNT (100)     // now count with function
```

Output:

```
1998
1997
```

Remarks: Triggering off the positive edge. In the same amount of time, the PERIOD function will return a more accurate timing measurement. COUNT has two advantages:

- It can be simpler to use.
- The number of transitions in 'duration' can be unknown.

Cautions: The maximum input frequency is about 30KHz. Rates higher than that may return erroneous results. Durations greater than 65535 or less than 0 will generate run-time errors.

See Also: PERIOD.

DIM *dimension array*

Syntax: DIM <label> (size) [,(size)]

Description: This command is used to define an integer or floating point array in TFBASIC. Any legal variable name can be used as an array name as long as its size is defined first by using this command. All array members take 4-bytes - just like any TFBASIC variable - integer or floating point.

DIM allows you to 'dimension' an array. It must appear in the program before any reference to members of the array. Attempts to access members outside the array's boundaries (as in Array1(21) or Array1(-1) in the example below) result in a 'HOW?' error.

Examples:

```
// make Array1 have 20 elements numbered 0 to 19
DIM Array1(20)

// 2 floating point elements numbered 0 and 1
DIM Array2!(2)

// make Array3 have 10 elements numbered 0 to 9,
// each representing 5 elements numbered 0 to 4
DIM Array3(10,5)
```

Remarks: String arrays are not allowed in TFBASIC

All arrays in TFBASIC are 0 based. References to the members of the Array1 defined above will look like Array1(0), Array1(1), Array1(2) ... Array1(19). Notice that the index starts at 0 and ends at 19, giving the same number of elements as the number in parentheses.

Cautions: The @() array and the ?() are predefined and therefore automatically available. DO NOT attempt to DIM these again! The @ array dimension is 15232.

Once a name has been defined as an array, it cannot be used for a variable name, and vice versa.

See Also: RTIME, STIME, READRTC, SETRTC; also Data storage options (@ array) and TFX Timekeeping (? array)

EXP *raise e to a power*

Syntax: value = EXP(<x>)

Description: EXP returns the base of the natural logarithms (e= 2.71828...) raised to the power of the expression in the parentheses. The function takes a floating point argument and returns a floating point value. An integer argument will be converted to float first.

Example: build this program with your editor:

```
arg! = 0.125
result! = exp(arg!)
print “e raised to “, #5.3F, arg!, “ is “, #6.3F, result!
```

Output:

```
e raised to 0.125 is 1.133
```

Remarks: To raise the value X to the power Y use the equation:

```
        value = exp(Y*log(X))
```

For this to work correctly X must be greater than 0 and Y*log(X) must fall into the valid range for exp arguments (see Cautions below).

Beware that these numbers lose accuracy as X or Y approach their limits.

Cautions: The range of input arguments is -87.33654 to 88.72283. Arguments greater than 88.72283 will result in a floating point Overflow error, (FPERR=2) with the result equal to +INF. Arguments less than -87.33654 will result in a floating point Underflow error, (FPERR=1) with the result equal to 0.0. In both cases execution is not stopped.

See Also: LOG

FIX *convert a float to an integer*

Syntax: value = FIX(<x>)

Description: FIX returns the next integer value closer to zero than the argument. The function takes a floating point argument and returns an integer value. An integer argument will be converted to float first and then converted back to integer.

Example: build this program with your editor:

```
finp! = 5.329
result = fix(finp!)
print  “fix of “, #8.3f, inp, “ = “, #D, result

result = fix(-finp!)
print  “fix of “, #8.3f, inp, “ = “, #D, result
```

Output:

```
fix of  5.329 = 5
fix of -5.329 = -5
```

Remarks: None

Cautions: Arguments outside the range of Tattletale integers will generate a runtime error, halting the program.

See Also: FIX, INT

FLOAT *convert integer to float*

Syntax: value = FLOAT(<x>)

Description: FLOAT returns the floating point representation of the argument. The argument must be an integer.

Example: build this program with your editor:

```
inp = 123
result! = float(inp)
print "float of ", inp, " = ", #5.1F, result
inp = -77
result = float(inp)
print "float of ", inp, " = ", #5.1F, result
Output:
float of 123 = 123.0
float of -77 = -77.0
```

Cautions: Single precision floating point has a precision of 24 bits while integers have a precision of 32 bits. Arguments outside the range of -16777215 to 16777215 will lose precision when converted to floating point. This Loss of Precision error (FPERR = 8) does not stop program execution.

A non-integer argument will cause an error in the tokenizer.

See Also: FIX

FOR *for - next loop*

Syntax: FOR var = initial TO final [STEP inc] [statements] NEXT var

FOR <v> = <x1> TO <x2> [STEP <x3>] [statements] NEXT <v>

Description: FOR loops provide one of four methods of looping available in TFBASIC. Here "var" can be any TFBASIC integer variable, and "initial", "final", and "inc" are integer expressions. The variable "var" will first be initialized to the value of the expression "initial", and then the section of code between the 'FOR' statement and the 'NEXT' statement will be repeated until "var" is greater than the value of the expression "final". After each pass, "var" will be increased by the value of the expression "inc". If STEP and "inc" are omitted, a step value of one is assumed. The limit ("final") and step ("inc") are evaluated and stored each time the loop is tested for continuation.

Examples:

Build this program with your editor:

```
//prints out the sequence 7,14,21,28,35,42,49
X = 7
FOR A = X TO X*X STEP X
PRINT #4, A;
NEXT A
```

Output when run in TFTools:

```
7  14  21  28  35  42  49
```

Now build this program with your editor:

```
//This example demonstrates the use of "for - next"
//loops for formatted print-outs
FOR A = 1 TO 3
FOR B = 1 TO 5
PRINT #5, B;
NEXT B
PRINT
NEXT A
```

Output when run in TFTools:

```
1    2    3    4    5
1    2    3    4    5
1    2    3    4    5
```

Remarks The test for continuation occurs at the beginning of the loop and the STEP and LIMIT expressions are evaluated each time this test is done. This allows you to use a GOTO to exit the loop. Also, because this structure stores nothing on the stack, you can nest these loops as deeply as you like and a GOTO can be used to exit any number of FOR loops. Use of GOTO to exit a loop, while useful in certain circumstances, is NOT generally considered good programming practice! If you feel the need to use a GOTO to exit, chances are the whole construct could be more effectively coded using a WHILE or REPEAT.

Cautions: STEP may be positive or negative, but do not use STEP 0!

At this time, only integer variables and expressions can be used in the FOR loop specification.

See Also: WHILE, REPEAT, GOTO

FVAL *convert string to floating point value*

Syntax: FVAL(str$)

Description: FVAL takes a string representing a floating point value and converts it to its numeric floating point variable equivalent.

Examples:

```
astr$  = “1.234e1”
aflt!  = FVAL(astr$)
print #5.2f, aflt!
```

Output:

```
12.34
```

Remarks: Valid characters are 0-9, +, -, ., E, and e. If the number contains other characters it will terminate the conversion at the unrecognized character. In the case of “12.3x5” the number returned will be 1.23

If the number evaluated is out of the floating point range it will generate the appropriate FPERR but will not halt the program.

See Also: IVAL

GOSUB *go to subroutine, saving return address*

Syntax: GOSUB label

...

...

RETURN

Description: The GOSUB and RETURN commands allow you to use subroutines in TFBASIC. The label specifier can be a line number or label. The RETURN statement signals the end of the subroutine. GOSUBs can be nested at least 20 deep.

Examples:

Build this program with your editor: (indentation for clarity only)

```
'***** GOSUB EXAMPLE 1 *****
       GOSUB SUB1
       GOSUB SUB3
       STOP

SUB1:
       PRINT "1st Subroutine"
       GOSUB SUB2
       RETURN
SUB2:
       PRINT "2nd Subroutine"
       RETURN
SUB3:
       PRINT "too much excitement - PLEASE STOP!"
       RETURN
```

Program Output:

```
1st Subroutine
2nd Subroutine
too much excitement - PLEASE STOP!
```

See Also: GOTO

GOTO *go to label*

Syntax: GOTO label

Description: GOTO causes an unconditional transfer of the program to the specified label.

Examples:

```
'***** GOTO EXAMPLE 1 *****
REM  simple unconditional jump

       GOTO SKIP
       PRINT "you won't see this"

SKIP: PRINT "you will see this"
       STOP
```

Remarks: GOTO can be used to exit any number of nested FOR, WHILE and REPEAT loops. Good programming practice avoids using GOTO.

See Also: WHILE, REPEAT, FOR

HALT *stop in lowest power mode*

Syntax: HALT

Description: This command puts the Tattletale into its lowest power mode where it has a typical power drain of <100 µA. It is up to the user to configure all I/O and external devices into their lowest power states. The only exits from this mode are disconnecting and re-connecting the main battery, or interrupting the process by commanding a LAUNCH (restart) from the HOST via the parallel port.

Example:
```
HALT
```

Remarks: None

Cautions: Before going into the dormant mode using 'HALT', all of the unused I/O lines need to be converted to inputs and asserted according to their pull-down or pull-up resistor configuration. Any hardware attached should drop into its minimum power drain state when the lines are so asserted. In order to make use of the low power HALT state you must provide termination for all of the unused input lines or set them to outputs.

See Also: HYB, SLEEP, STOP

HYB *very low power mode with wakeup*

Syntax: HYB x

Description: This command puts the Tattletale into its lowest power mode where it has a typical power drain of <100 μA. The parameter is the number of seconds to remain in this state before being awakened by the supervisory alarm in the PIC. The only other exits from this mode are disconnecting and re-connecting the main battery, commanding a restart from the HOST via the parallel port, a PICINT interrupt, or a high to low transition on the IRQ line.

Example:

```
HYB 0
for i=1 to 12
HYB 5
print “Awake from HYB with ? = “,?
next i
```

Remarks: In order to make use of the low power HYB state you must provide termination for all of the unused lines.To make the most of the low power capabilities of the HYB command all I/O and peripherals should be configured to their lowest power states. Unused I/O lines may be set to outputs, or set to inputs if pull-ups or pull-downs are attached. Any power switched device should be powered off.

Timekeeping. When the HC11 is in HYB its clock is shut down - when it awakens from a HYB it automatically reads the PIC time value into the ? variable.

Cautions: Care must be taken when setting a driven input to an output. If you are sure the state will not change (for example if it is connected to a pullup or pulldown resistor) during the HYB duration this is what may be done:

- Read the state of the input using PIN
- Depending on the state of the pin read use PSET or PCLR to set to output to the correct state.

See Also: HALT, SLEEP, STOP, PICINT, INTSTATE read-only variable.

IF *branch on result of comparison*

Syntax:

IF expression

...command...

[...more commands...]

[ELSE

...command...

...block 2...]

ENDIF

Description: IF allows you conditional control of your program. If the 'expression' is true (does not evaluate to 0), command block 1 is executed otherwise, if the ELSE block exists, command block 2 is executed. The ELSE block is optional but the ENDIF is not. If one of the operands of the comparison is a floating point value, the integer is treated like a float and the operands are compared as floating point values.

Examples: (indentation for clarity only)

```
input Value
if  Value > 100    // execute block 1 if Value > 100
    print "The value “,Value,” is > 100 - not allowed!"
    Value = 100    //end of command block 1
else               // otherwise, execute command block 2
    print "The value is NOT greater than 100 - OK"
endif              //end of command block 2
print “Value = “,Value
```

Output, trial 1:

```
27
The value 27 is NOT greater than 100 - OK"
Value = 27
```

Output, trial 2:

```
127
The value 127 is > 100 - not allowed!"
Value = 100
```

Remarks: The result of a comparison using one of the relational operators is: 0 for false, and 1 for true. Do not write programs depending on this, however.

See Also: Relational Operators

INPUT *get value from console (buffered)*

Syntax: INPUT ["prompt"] var[;]

INPUT ["<s1>"] <v1> [;] [,\x1] [,#x2] [,x3]

Description: INPUT allows you to assign a value to a variable entered through the main UART from a terminal. You can use a string (in double quotes) as a prompt. A default prompt of the variable name is used if you don't include one. If you don't want a prompting string, use a zero-length string (quotes with no intervening characters). Input will accept all variable types; integer, floating point, and string. Floating point input can be in either fixed point or scientific notation. The default terminator is a <CR>, and the default time-out is none. By default up to 255 characters can be input. As explained below, these defaults may be overridden by the optional parameters each of which must be INTEGER expressions:

Additional Switches:

\x1 **Set terminating character.** This byte value represents the terminating character for the input stream. The special case \~ means no terminator, useful with the [#x2] parameter (below) for binary transfers of fixed length with all values possible.

#x2 **Set byte count.** This expression fixes the count of characters to read as input before termination. If a specific termination character is defined it takes priority over the count. If the terminating character is present in the input stream, it will terminate input before the total count is reached.

x3 **Set time-out.** If present, sets a non-zero time-out in increments of 0.01 seconds. If the time between incoming characters exceeds this value then input is terminated and the program proceeds, whether or not there are terminating characters or a terminating character count set. **IMPORTANT NOTE:** If the value ‘0’ (zero) is specified for the time-out then INPUT checks the buffer to see if a character is available; if there is a character it returns with it, otherwise it returns immediately. Similar to the INKEY$ function in other BASICs.

; No <CR> <LF> on termination of input. By default INPUT sends out a <CR> <LF> to the terminal when it exits, even if terminating on a byte count or timeout.

Example:

```
// check if a key was hit without waiting for one
input “hit a key to stop” key$,#1,0;
if key$<> “”
 stop
endif
```

Example:

```
// enter a floating point number
input floatVar!
input intVar
input "Type a floating point value-> "test!
input "Type an integer value-> "number
print
print #10.3F,floatVar,#10D,intVar,#12.6S,test,#10D,number
```

Program Output: (typed responses in bold face)

```
floatVar: 9.351e3
intVar: 54321
Type a floating point value-> 134.55
Type an integer value-> 9999999999 <too large an integer!>

? 999999999

9351.000     54321  1.345500E2 999999999
```

Remarks: Entering a carriage return alone in response to an input command assigns zero to the variable.

A trailing semicolon after the variable specifier causes the input command to inhibit echoing the terminating carriage return.

Entering a Control-C during an input leaves the variable unchanged.

Cautions: The expression entered from the terminal is evaluated after each prompt. If an integer value outside the range -2147483647 to 2147483647 is input, a '?' is displayed to request corrected input.

A floating point input between -1.175494E-38 and +1.175494E-38 assigns zero to the variable and, if the input is not exactly zero, sets the

FPERR variable to indicate an underflow error. A value of +Infinity is assigned to the variable if the input is greater than 3.402823E+38. A value of -Infinity is assigned to the variable if the input is less than -3.402823E+38. The value of FPERR will be updated to show an overflow occurred for either infinite result. If a floating point input cannot be evaluated, a '?' is displayed to request corrected input.

See Also: PRINT

INSTR *returns a substrings position in a string*

Syntax: INSTR([x,] str1$, str2$)

Description: Return the position in str1$ at which the substring str2$ is first found. Optionally start the search at position x in str1$. If the substring is not found then the function returns 0.

Examples:

```
astring$ = "This is a needle in a haystack"
search$ = "needle"
Offset = instr (astring, search)
if offset <> 0 print mid(astring, offset, len(search))
if offset = 0 print "String not found"
```

Remarks: This function does not return a string. It returns an integer offset to the first character of the substring that can be used on string STR1$.

See Also: MID , LEN

INT *convert float to integer*

Syntax: value = INT(<x>)

Description: INT returns the next integer value less than the argument. The value is returned as an integer. The argument must be a float. If it is an integer, it will be converted to float first.

Example: Build this program with your editor

```
inp! = 5.32987
print "int of ", #8.5F, inp, " = ", #D, int(inp)
print "int of ", #8.5F, -inp, " = ", #D, int(-inp)
```

Output:

```
int of 5.32987 = 5
int of -5.32987 = -6
```

Cautions: Arguments outside the range of Tattletale integers (-2147483648 to 2147483647) will result in a run time error.

See Also: FIX, FLOAT.

IVAL *convert numeric string to integer value*

Syntax: IVAL(str$)

Description: IVAL takes a string representing an integer value and converts it to its numeric integer variable equivalent.

Examples:

```
astr$  = "1234"
anint= IVAL(astr$)
print #4, anint
```

Output:

```
1234
```

Remarks: Valid characters are 0-9, +, -. If the number contains other characters it will terminate the conversion at the unrecognized character. In the case of "123x567" the number returned will be 123.

See Also: FVAL

LEN *return length of string variable*

Syntax: LEN(str$)

Description: LEN takes the string argument and returns an integer value representing the number of characters including terminating characters. The Maximum length is 255 characters.

Examples:

```
TestString = “This is string one”
print “Length of string one is “, len(TestString$”)
print “Length of string 2 is “, len(“this is string 2”)
```

Output:

```
Length of string one is 18
Length of string 2 is 16
```

Remarks: none.

Cautions: none.

See Also: MID, INSTR

LOG *natural logarithm*

Syntax: value = LOG(<x>)

Description: LOG returns the natural logarithm of the expression in the parenthesis. The function takes a floating point argument and returns a floating point value. An integer argument will be converted to float first.

Example: build this program with your editor:

```
arg! = 0.125
result! = log(arg!)
print “Natural Log of “, #%.3F, arg!, “ is “, #6.3F, result!
```

output:

```
Natural Log of 0.125 is -2.079
```

Remarks: Arguments less than or equal to zero will generate a Not-a-Number (NaN) floating point error (FPERR=4), but execution is not stopped.

See Also: EXP, LOG10

LOG10 *common logarithm*

Syntax: value = LOG10(<x>)

Description: LOG10 returns the common logarithm of the expression in the parenthesis. The function takes a floating point argument and returns a floating point value. An integer argument will be converted to float first.

Example: build this program with your editor:

```
arg! = 0.125
result! = log10(arg!)
print “Common Log of “, #%.3F, arg!, “ is “, #6.3F, result!
```

output:

```
Common Log of 0.125 is -0.903
```

Remarks: Arguments less than or equal to zero will generate a Not-a-Number (NaN) floating point error (FPERR=4), but execution is not stopped.

See Also: LOG, EXP

MID *Return a substring of a string*

Syntax: substr$ =MID(str$,x1,x2)

Description: MID returns a substring of the string argument passed to it, where x1 is the offset to the start character and x2 is the number of chracters to read within the string argument.

Example 1:

```
SubStr$ = ""
MainStr$ = "this is a test "
print MainStr$
SubStr$ = mid(MainStr$,3,6)
print SubStr$
print SubStr$ + "really"
stop
```

output:

```
this is a test
is is
is is really
```

Example 2:

```
MyStr$ = "12345"
for index = 1 to len(MyStr$)-1
 print(mid(MyStr$,index,3)
next index
stop
```

output:

```
123
234
345
45
5
```

Remarks: The first character in a string is referenced as 1, not 0. If you use) it will be converted to one. Negative numbers for x1 cause MID to terminate. If the nu8mber of charaters requested goes past the end of the string, then the string returned will automatically terminate at the end.

Cautions: None:

See Also: LEN, INSTR

ONERR *go to label on error*

Syntax: ONERR [label [,var]]

Description: ONERR directs the Tattletale to jump to the specified label if a run-time error occurs instead of printing an error message. Errors are normally flagged as they occur with a 'HOW' comment. If an ONERR line is encountered during execution, the error printout will be skipped and execution will continue at the 'label'. This allows emergency shutdown or recovery from a program error encountered in the field. To return to the normal error action, use ONERR with no argument.

When the ONERR branch is made, the program loses all information about previous GOSUBs.

Example: build this program with your editor (indentation for clarity only)

```
               ONERR MID
               X=1
LOOP1:         X=X*2
               GOTO LOOP1   // find something too big
MID:           A=X
               ONERR LOOP2
LOOP2:         A=A/2
               IF A=0 PRINT "MAX INTEGER = ",X
               STOP
               X=X+A
               GOTO LOOP2
```

Program's Output:

```
MAX INTEGER = 2147483647
```

What & Where?: In addition to the form shown above, TFBASIC allows an optional variable to be specified that will receive the error code number and the address of the token that failed. This value can be examined in the error handling routine to decide what action to take. Be aware that all information on previous GOSUB return addresses is lost. In addition, TFBASIC resets the token parameter stack.

The token address for the error is stored in the least significant two bytes of the variable, and the 'HOW' error number in the most significant two bytes. Use the divide and mod operators to separate out these parts. The token address can be used to look into the *.LST file to get an idea of where the error occurred.

Example: build this program with your editor (indentation for clarity only)

```
ONERR TROUBLE,E //goto TROUBLE if error, error # in E
  A = 2          // initialize variable
  FOR I = 0 TO 99 //execute loop up to 100 times
  A = A * 2     // make A larger, possible error source
  NEXT I           // loop back

TRYTHIS:
  B = TEMP(1000000)// another possible error source
  STOP        // won't get here, second err causes exit

TROUBLE:
   PRINT "Error #", E/65536," found";
   PRINT " at token address ", #H, E % 65536, "H"
   IF E/65536=7
      PRINT "Multiply out of range"
      GOTO TRYTHIS
      ENDIF
   IF E/65536=14 PRINT "TEMP argument out of range"
   STOP
```

Program's Output:

```
Error #7 found at token address 28H
Multiply out of range
Error #14 found at token address 35H
TEMP argument out of range
```

Remarks: As with CBREAK , ONERR must be executed to be effective. For this reason, you should put ONERR at or near the beginning of the program. The error handler may be changed any number of times by executing ONERR with different arguments.

Cautions: Do not vector ONERR to a label inside a subroutine. When a error is detected the program will act as if you did a GOTO into the subroutine. It will not have a proper return address on the stack and will probably crash.

See Also:

PCLR *set I/O pin low*

Syntax: PCLR pin [,pin...]

PCLR <x1> [,<x2>.]

Description: PCLR first converts the specified pins to outputs, and then clears these pins to a logic low (0 volts). The following table maps the argument value to the I/O pin.

PIN	I/O PIN	PIN	I/O PIN
0	A18	16	B20
1	A19	17	B19
2	A20	18	B18
3	A21	19	B17
4	A22	20	B16
5	A23	21	B15
6	A24	22	B14
7	A25	23	B13

Examples:

```
// use two ways to clear I/O lines

FOR A=0 TO 5
PCLR A
NEXT A

PCLR 0,1,2,3,4,5
```

Remarks: In TFBASIC, when multiple pins are specified in a single command, the changes may not take place simultaneously if the pins selected are from different ports.

Cautions: **Pins 8-15 are INPUTS ONLY!** PTOG, PSET and PCLR will generate a HOW? error when used with these pins.

See Also: PSET, PTOG, PIN.

PEEK *read memory byte*

Syntax: PEEK (<addr>)

PEEK (<expr>)

Description: This function returns the value of the byte located at the address <a> in parentheses. The <a> can be any expression that evaluates to a valid address in BANK 0.

Example:
(indentation for clarity only)

```
   print peek ($H74C0)
   A = peek (&H112)

startprog:
   print "square of values from 0 to 9"
   for a = 0 to 9
      b = a * a
      print #5, a, b
   next a
   print "last b value bytes:  ";
   print #02H, peek(varptr(b)),' ';
   print #02H, peek(varptr(b)+1),' ';
   print #02H, peek(varptr(b)+2),' ';
   print #02H, peek(varptr(b)+3)
   print "program starts at ";
   print #04H, labptr(start_prog)
   print "program ends at ", #04H, labptr(end_prog)
endprog:
```

Remarks: This function can return the value of any address in the first 64K address space including the processor registers.

See Also: POKE

PERIOD *measure period of signal*

Syntax: PERIOD (count, timeout)

PERIOD (<x1>,<x2>)

Description: PERIOD measures the amount of time it takes for 'count' cycles of a signal to pass. The input signal must be connected to I/O line 0 and is measured in units of 1/2.4576μSec (about 0.40690μSec). If 'timeout' * 0.01 seconds passes before the prescribed number of cycles transpires, the returned value will be zero. This keeps the Tattletale from locking up forever if no signal is at the input. Period may return incorrect values for input frequencies higher than 30KHz.

Example: (indentation for clarity only)

```
start:         print "COUNT gives ",count(100)," Hz"
               X = PERIOD (100,100)
               if X=0 print "PERIOD gives 0 Hz"
                 goto finish
               endif
               print "PERIOD gives ",122880000/X," Hz"
finish:        stop
```

Output

```
COUNT gives 4997 Hz
PERIOD gives 4996 Hz
```

Cautions: Be careful when dividing anything by PERIOD, since PERIOD can return a zero which would cause a 'HOW?' error.

The maximum value for the count and timeout arguments is 65535.

Remarks: Argument maximum is 65535.

See Also: COUNT

PICINT *external interrupt for wakeup*

Syntax: PICINT x [,y ,z]

Description: This instruction sets up I/O line 16 as an edge-sensitive interrupt. This interrupt originates in the PIC. This instruction sets I/O 16 to input, sets the edge, clears any pending interrupts and then sets the PIC's INTE bit. When an interrupt is received it will awaken the PIC (if asleep) which in turn will signal the HC11 via the XIRQ line. The PIC will clear the interrupt flag and the interrupt enable flag, so the interrupt will remain disabled until the PIC receives another enable instruction. From TFBASIC this may be used as an asynchronous awakening from HYB. If connected to the UDI pin it will awaken the TFX-11 from sleep on receipt of a serial character.

arguments:

- x If 0, interrupt will be disabled. If 1, interrupt is enabled.
- y If 0, interrupt is on falling edge (default). If 1, on rising edge.
- z Address of assembly routine called when interrupt is detected.

Examples:

```
intcount = 0
savecount = 0
cbreak clean
print "Count interrupts on I/O Pin 16"
print "Hit Ctrl-C to exit"

asm &hb000
ldd intcount+2// increment the counter
addd #1
std intcount+2
rts           // notice RTS, code internal to TFBASIC does RTI
end

picint 1,0,&hb000// enable PIC interrupts on negative-going edge

sleep 0
loop:
if intcount <> savecount
       print "intcount = ",intcount
       savecount = intcount
       picint 1,0,&hb000// re-enable PIC interrupt
endif
```

```
sleep 10
goto loop
stop

clean:
picint 0// disable PIC interrupt
print "PIC interrupts disabled"
stop
```

See Also: INTSTATE read only variable, HYB

PIN *read state of I/O pin*

Syntax: value = PIN (pin [,pin.])

value = PIN (<x1> [,<x2>.])

Description: For each pin that is specified by the PIN instruction, the data direction control bit for that pin is set to input. The value that is formed from the states of the specified pins is then returned . If the voltage at a particular pin is above 2.0 volts, the PIN instruction interprets the input as a 1; if it is below 0.7 volts, it is interpreted as a 0. Intermediate values will return unpredictable (indeterminate) results.

This command returns a value of all listed pins in a set order, not depending on the order they are listed in the command arguments.

I/O	PIN	WEIGHT	I/O	PIN	WEIGHT	I/O	PIN	WEIGHT
0	A18	1	8	A26	256	16	B20	65536
1	A19	2	9	A28	512	17	B19	131072
2	A20	4	10	A30	1024	18	B18	262144
3	A21	8	11	A32	2048	19	B17	524288
4	A22	16	12	A27	4096	20	B16	1048576
5	A23	32	13	A29	8192	21	B15	2097152
6	A24	64	14	A31	16384	22	B14	4194304
7	A25	128	15	A33	32768	23	B13	8388608

Comments: If a pin is not listed in the command's argument list its corresponding value is always returned as 0, whether or not it is set.

Remarks: In TFBASIC, when multiple pins are specified in a single command, the pins are handled in sequence as three blocks. First the pins in the block 0 - 7, then the pins in the block 8 - 15, and finally the pins 16 - 23.

Cautions: I/O pins 8-15 do not correspond sequentially to their proto board pin numbers.

See Also: PSET, PCLR, PTOG.

POKE *place byte into RAM*

Syntax: POKE <addr>,<value>

POKE <expr1>,<expr2>

Description: This command stores the least significant byte of the value of the expression <expr2> at the address that results from the evaluation of the expression <expr1>.

Examples:

```
poke &H74C0,123

b=123456789             // write 123456789 to locations

for a=&H74C3 to &H74C0 step -1  // 112H -115H, msb first
  poke a,b%256
  b=b/256
next a
```

Remarks: This modifies memory directly and should be used with care. Its use is restricted to the 64K bank of program memory.

See Also: PEEK, VARPTR

PRINT *print to console*

Syntax: PRINT ["string"][,#format][,value][,\ ascii val][;]

PRINT ["<s>"][,#n][,<x>][,\ <x>][;] {any order or mix}

Description: PRINT can be used to write values, strings, individual characters, and blocks of the datafile to the hardware UART. The formats, strings, values, characters, and datafile blocks must be separated with commas. A trailing semicolon will suppress the trailing carriage return line-feed that is normally sent at the end of a PRINT.

Strings: A string is a set of characters bracketed by either single or double quotes. Strings can have any length, including zero.

String examples

```
print "HELLO"
print "This is a test"
```

Values: Values are expressions that are evaluated at the time of the execution of the PRINT statement. The default format for expressions is one character. All the digits of a number will be printed even if the format specifies too small a space.

Value examples

```
print "A=", A
print "A+5=", A+5
```

Using PRINT with Formatting: Formats are a '#' followed by a numeric value, optionally followed by a type specifier ('D', 'H', 'Q', 'B', 'F', 'S'). The numeric value specifies the minimum number of spaces a value is allowed to take when printed. Values that take less than the specified number of spaces will be filled out (to the left of the value) with spaces unless the first character following the '#' is a '0', in which case the fill character is a zero. All digits of the value will be printed, regardless of the format. Decimal is assumed unless one of the type suffixes ('D' = decimal [the default for integers]), 'H' = hexadecimal, 'Q' = octal, 'B' = binary, 'F' = fixed point float, 'S' = scientific float [the default for floating point]) is specified.

Formatted examples:

```
print #10H,-1,#14Q,-1,#35B,-1
  FFFFFFFF    37777777777    11111111111111111111111111111111

print #010H,-1,#014Q,-1,#035B,-1
00FFFFFFFF 00037777777777 00011111111111111111111111111111111

print #10.2F, 12.345
      12.34

print #10F, 12.345
 12.340001

print #7.2S, 12.345
 1.23E1

print #.5F, 12.345
12.34500

print #0S, 12.345
1.234500E1
```

For floating point numbers, two numbers can be used to separately specify the minimum width and the number of decimal places of precision. Both are optional with the default digits of precision being 6.

Character codes: Individual characters can be sent by preceding their ASCII value with a backslash. You can also specify an expression after the backslash. The least significant byte stored in the variable will be sent.

Character code example

```
// This example sends out a 'bell' control character
print "Strike the bell, second mate!",\7, "Let us go below!"
// Next example prints letters 'a' through 'z' then CR/LF
for i = 97 to 122
 print \i;
 next i
print
```

Trailing semicolon: A trailing semicolon causes TFBASIC to omit the CR, LF that is normally sent at the end of a print statement.

Trailing semicolon example:

```
print “this is “;           // these two lines produce
print “a test”              // identical output

print “this is a test”      // to this one line
```

Remarks: To send a CR without a LF at the end of a line use \13 to send the carriage return, and a semicolon to suppress the normal CR LF. This is useful for updating a value on the display without generating a new line. It will continuously write over the value on the same line.

Cautions: The PRINT command's output is buffered. All print format declarations require a numeric value between the # sign and the type specifiers.

See Also: STORE

PSET *set I/O line high*

Syntax: PSET pin [,pin, pin,...]

PSET <x1> [,<x2>.]

Description: This command sets the data direction register for the specified pins to outputs, and then sets the pins to a logic high (+5 volts).

Example Code:

```
// example 1
FOR A=0 TO 7
PSET A
NEXT A

// example 2
PSET 0,1,2,3,4,5,6,7
```

Either one of these will set all of the output lines high. This is useful in an application where the I/O lines that are not used do not have pull-ups or pull-downs. Setting them to outputs keeps them from being floating inputs that can draw extra current needlessly.

Remarks: See PCLR for I/O pin mapping.

In TFBASIC, when multiple pins are specified in a single command, the pins are handled individually and in sequence as three blocks. First the pins in the block 0 - 7, then the pins in the block 8 - 15, and finally the pins in the block 16 and above.

Cautions: **Pins 8-15 are INPUTS ONLY!** PTOG, PSET and PCLR will generate a HOW? error when used with these pins.

See Also: PCLR, PTOG, PIN

PTOG *toggle I/O line to opposite state*

Syntax: PTOG pin [,pin, pin,...]

PTOG <x1> [,<x2>.]

Description: The PTOG command sets the data direction register for the specified pins to outputs, and then changes the pins to the opposite state they held before this command was executed.

Example code:

```
FOR A=0 TO 13
PTOG 1
NEXT A
```

This example will cause pin D1 to change state 14 times ending up in its original state.

Remarks: In TFBASIC, when multiple pins are specified in a single command, the pins are handled individually and in sequence as three blocks. First the pins in the block 0 - 7, then the pins in the block 8 - 15, and finally the pins in the block 16 and above.

Cautions: **Pins 8-15 are INPUTS ONLY!** PTOG, PSET and PCLR will generate a HOW? error when used with these pins.

See Also: PCLR, PSET, PIN.

RATE *Change sleep interval*

Syntax: RATE <x>

Description: This command assigns a new value for the sleep interval duration. By default a SLEEP 1 equals 10ms., so a SLEEP 100 gives a 1 second interval. This is equivalent to RATE 1. Other acceptable values for RATE are:

	Sleep	interval
RATE	ticks/sec	(ms.)
1	100	10.0
2	200	5.0
3	300	3.33...
4	400	2.5
6	600	1.66...
8	800	1.25
12	1200	.833...
16	1600	.625
24	2400	.4166...
32	3200	.3125
48	4800	.20833...
64	6400	.15625
96	9600	.104166...
128	12800	.078125
192	19200	.0520833...

Example code:

```
for index = 1 to 5
  print ?              // print seconds count
  sleep  100           // sleep for one second
next index
RATE 2                 // twice as many interrupts per second
for index = 1 to 10
  print ?              // print seconds count
  sleep  100           // sleep for one-half second
next index
RATE 1                 // return to default RATE
```

LATCH

Remarks: ? variable timekeeping update rate is NOT affected by different RATES, but is always updated 100 times a second. Actual update time may be delayed slightly when using higher RATE values.

Cautions: Using values for RATE not in the table will cause a run time error. Using a high RATE may cause sleep overruns where there were none before. Use this command with care.

See Also: SLEEP and STPWCH example program in Chapter 5 section assembly language subroutines.

READRTC *load PIC RTC time to local variable*

Syntax: READRTC

or

READRTC <v>

Description: READRTC takes the 4 bytes from the PIC RTC in seconds and copies it directly into the '?' variable, or optionally, into the variable provided as an argument. This time is counted in seconds starting at New Years Day 1980.

Example code:

```
       input "Set time (Y/N)?"answer$,#1

       if answer <> "Y" & answer <> "y"
          goto start
       endif

       input "year:    "?(5)
       input "month:   "?(4)
       input "day:     "?(3)
       input "hour:    "?(2)
       input "minute:  "?(1)
       input "second:  "?(0)

       print          // print out time
       print #02,?(5),"/",?(4),"/",?(3)," ";
       print #02,?(2),":",?(1),":",?(0)
       print

       stime          // convert ?(array) to ? variable
       SetRTC         // also copy this time to PIC
start: ReadRTC        // get back time stored in the PIC
       sleep 0

loop:
       rtime          // convert ? variable to real time
       print #02,?(5),"/",?(4),"/",?(3)," ";
       print #02,?(2),":",?(1),":",?(0),\13;
       sleep 100
       goto loop
```

Output:

```
Set time (Y/N)?y
year:   99
month:  12
day:    31
hour:   23
minute: 59
second: 45

99/12/31 23:59:45

1999/12/31 23:59:45
1999/12/31 23:59:46
1999/12/31 23:59:47
1999/12/31 23:59:48
1999/12/31 23:59:49
1999/12/31 23:59:50
1999/12/31 23:59:51
1999/12/31 23:59:52
1999/12/31 23:59:53
1999/12/31 23:59:54
1999/12/31 23:59:55
1999/12/31 23:59:56
1999/12/31 23:59:57
1999/12/31 23:59:58
1999/12/31 23:59:59
2000/01/01 00:00:00
2000/01/01 00:00:01
2000/01/01 00:00:02
2000/01/01 00:00:03
2000/01/01 00:00:04

^C

#
```

Remarks:

Cautions:

See Also: RTIME, STIME, SETRTC, and section “TFX Timekeeping”

REPEAT *execute loop until expression true*

Syntax: REPEAT

...commands...

...to be executed...

UNTIL expression

Description: REPEAT loops provide one of four methods of looping available in TFBASIC. The code between the REPEAT and UNTIL commands will be executed until 'expression' becomes true. Unlike the FOR and WHILE loops, the testing of 'expression' takes place after the loop has executed so a REPEAT loop will always run at least once. Because this structure stores nothing on the stack, these loops can nest as deeply as you like. GOTO will exit any number of nested REPEAT loops and will not cause stack problems.

Examples:
(indentation for clarity only)

```
// example 1 - force input to be 0 or 1 and count mistakes
begin:
tries = 0
print "Input 0 to exit or 1 to continue"

repeat
  input "Continue? "goAgain
  tries = tries + 1
until goAgain = 0

// goAgain = 1 // remove comment slashes to modify behavior
print "that took you ",tries," tries"

if goAgain = 0
  print "Program terminating"
  gosub CleanUp
  stop
else
  goto begin
endif
```

```
// example 2 - force input between 0 and 100, quit on 99
//             count number out of range and quit after 10

        repeat
          tries = 0
          repeat
            input newNumber
            tries = tries + 1
            if tries >= 10 goto give_up
          until newNumber < 100 & newNumber > 0
        until newNumber = 99

        print "99 was input. Time to stop."
        stop

give_up:
        print "Ten numbers out of range - you are finished!"
        stop
```

Remarks: None

See Also: FOR, GOTO, WHILE

RETURN *return from subroutine*

Syntax: RETURN

Description: RETURN is used with GOSUB to signal that it is time to go back to the calling routine.

Cautions: GOSUB stores the return address on the stack and RETURN uses it to know where to jump back to. If no GOSUB has placed a valid return address on the stack then executing a RETURN without a corresponding GOSUB will take whatever is on the stack and use it as the return address, most likely sending the program off into space. BE SURE that there is no path to a RETURN that is not deliberate. This type of error is usually associated with indiscriminate use of GOTOs.

See Also: GOSUB

RTIME *Read local software real-time-clock*

Syntax: RTIME

or

RTIME <v>

Description: RTIME translates the '?' variable (seconds) to the six elements of the '?()' array. Time is counted in 1/100 ths of a second starting at New Years Day 1980. This command, with STIME, allows translation between the all seconds format and the year, month, day, hour, minute, second format. RTIME <v> translates the variable v, which represents seconds only (no subseconds), to the members of the ? array.

?(0) gets the second (0 to 59)

?(1) gets the minute (0 to 59)

?(2) gets the hour (0 to 23)

?(3) gets the day (1 to 31)

?(4) gets the month (1 to 12)

?(5) gets the year (1980 to 2047).

?(6) gets the number of 0.01 sec ticks (0-99).

Example code: Build this program with your editor (indentation for clarity only)

```
       sleep 0
loop:  rtime
       print 'THE TIME IS  ',?(2),':',?(1),':',?(0);
       print ' on ',?(4),'/',?(3),'/',?(5)
       sleep 100
       goto   loop
```

Output:

```
THE TIME IS  16:24:18 on 5/8/1996
THE TIME IS  16:24:19 on 5/8/1996
THE TIME IS  16:24:20 on 5/8/1996
```

Remarks: Dates beyond 2047 are out of range. Leap years are handled properly.

Cautions: The ? array is predefined, do not initialize with DIM!

See Also: STIME, SETRTC, READRTC, and section “TFX Timekeeping”

SDI *shift register input*

Syntax: value = SDI (bits)

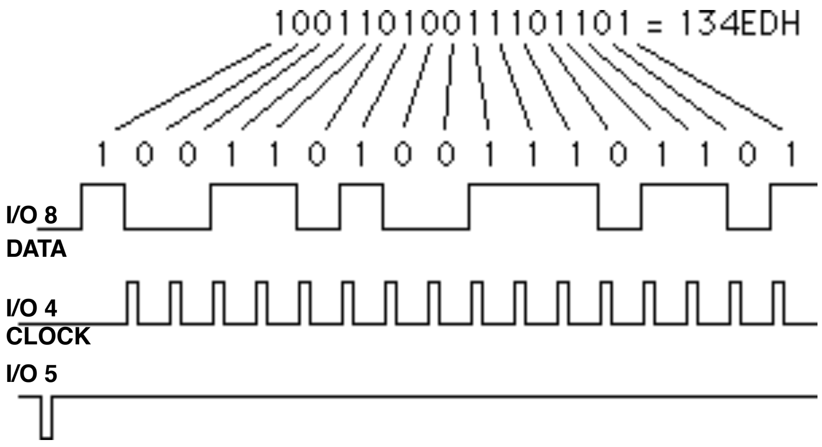
or

value = SDI (<x>)

Description: SDI is designed to bring in a serial data stream from a shift register and return with a value formed from this data stream. SDI first applies a negative going pulse to I/O line 5. This pulse is used to latch data into the shift registers. SDI then shifts in the number of bits of data specified by the value in parentheses, using I/O line 8 as the data input line and I/O line 4 as a clock. The returned value is made from the binary data received (msb first). Clocking occurs on the positive edge. This works nicely with a 74HC165 or 74HC166 shift register.

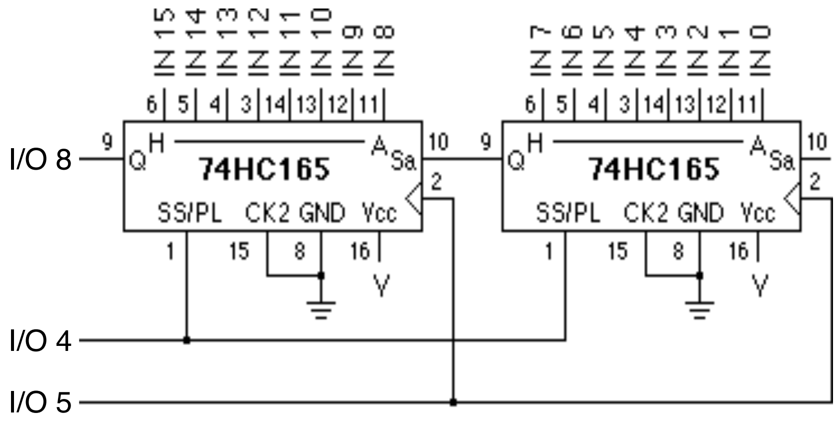
The command A = SDI (N) will cause N bits of data to be clocked in to form an N-bit two's complement number. If less than 32 bits are shifted in, the unspecified MSB's are zeros. The last bit shifted in has a weight of 1, with the preceding bits given weights of 2, 4, 8, 16, etc. If 32 bits are shifted in, the first bit will be the sign bit of the resulting two's complement number. The figure below shows 17 bits shifted in to form the number 134EDH, which is 79085 decimal.

Example of 17 Bits being Shifted



This sequence works well with both 74HC165 and 74HC166 shift registers. The figure below uses two 74HC165s to shift in 16 inputs. By cascading two more shift registers, 32 inputs can be read at once.

74HC165 and 74HC166 Shift Registers



Remarks: I/O lines 4, 5 and 8 will normally be connected to a shift register such as a 74HC165 or 74HC166 as shown above.

Since the most significant bit of the shift register attached to I/O line 8 is always available, an initial 'clock' is not needed. Thus, if you request N bits, there will be N-1 'clock' pulses on I/O line 4.

Cautions: Run time errors will occur if the value in theparentheses is not in the range 1 to 32.

See Also: SDO

SDO *shift register output*

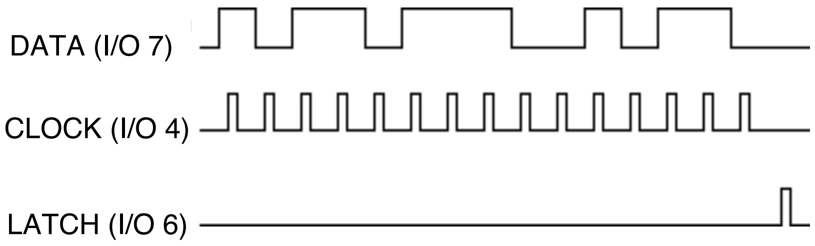
Syntax: SDO value, bits

SDO <x1>,<x2>

or

SDO <string>

Description: **Form 1:**SDO is designed to send a serial data stream made up of the least significant bits of the value. The bits are sent out I/O line 7, using the positive edge of I/O line 4 as a clock. After the last shift pulse, I/O line 6 is used as a positive-going latch pulse. This works nicely with a 74HC595 shift register.

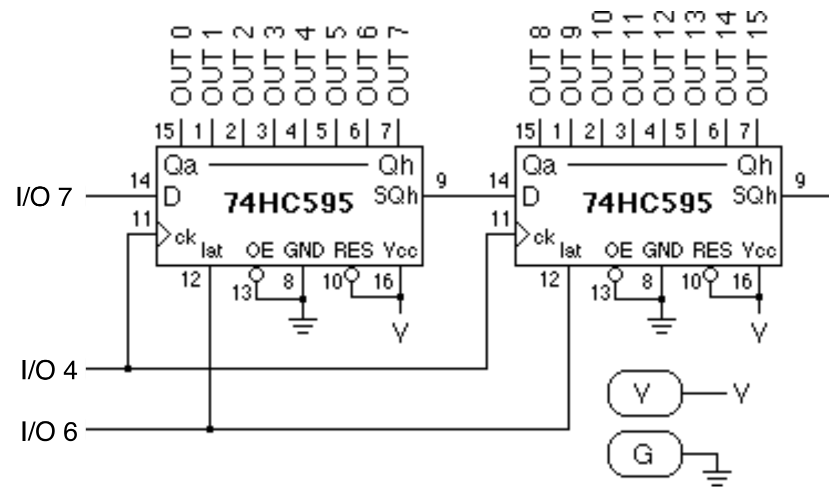


Form 2: The second form of SDO sends characters out the serial line (eight bits and then latch). Timing Lines used by the SDO Command

The figure above shows the timing of the three lines used in the SDO command. The circuit below shows one use of SDO. Here two 74HC595 shift registers are cascaded to form 16 outputs. By adding two more 74HC595's, a total of 32 output lines can be changed with a single SDO command.

NOTE: These format specifiers, #D, #H, #B, #F, #S or #Q are considered ambiguous. Do D or H signify the radix (decimal or hexidecimal) or a variable field width? You can use variables for field width but not with those 12 names (upper or lower case). To get the radix form, use #1D, #1H etc.

SDO Command Timing Lines



Remarks: I/O lines 4, 6 and 7 will normally be connected to a shift register such as a 74HC595 as shown above.

Since the bit on I/O line 7 is always available to be stored in the least significant bit of the shift register, a final 'clock' is not needed. Thus, if you request N bits, there will be N-1 'clock' pulses on I/O line 4.

Cautions: In form 1, the bit's value must be in the range 1 to 32.

See also: SDI and STR

SETRTC *transfer local time in seconds to PIC RTC*

Syntax: SETRTC

or

SETRTC <v>

Description: SETRTC takes the four byte '?' variable time in seconds and uses it to set the time in the PIC, or optionally takes the time in seconds from the variable v if present. Time is counted in seconds starting at New Years Day 1980.

Example code: See READRTC

See Also: RTIME, STIME, READRTC, and section "TFX Timekeeping"

SIN *Sine*

Syntax: value = SIN(<x>)

Description: SIN returns the sine of the expression in the parenthesis. The argument must be in degrees. The function takes a floating point argument and returns a floating point value. An integer argument will be converted to float first degrees! = 0.0 // init arg (notice '!' means it's a float)

Example:

```
result! = 0.0   // just to force 'result' to be a float
for i = 1 to 6
   result = sin(degrees)
   print "The sine of ",#5.1F,degrees," is ",#6.3F,result
   degrees = degrees + 72.0
next i
```

Output:

```
The sine of   0.0 is  0.000
The sine of  72.0 is  0.951
The sine of 144.0 is  0.588
The sine of 216.0 is –0.588
The sine of 288.0 is –0.951
The sine of 360.0 is  0.000
```

Cautions: **DON’T FORGET!** The argument is in degrees, not radians.

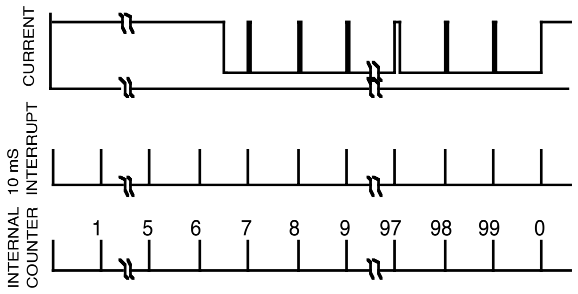
See Also: COS, TAN, ATN

SLEEP *low power wait over a precise time interval*

Syntax: SLEEP tics

SLEEP <x>

Description: SLEEP places the Tattletale in a semi-dormant mode until the number of 1/100-second intervals specified by the tics argument has expired. It does this by comparing the specified 15-bit tics value with the 16-bit value in a free-running counter, called the interval counter, that is incremented every 1/100 of a second. When a match is found, it clears the interval counter and completes the instruction. In this manner, the SLEEP command actually sets intervals between SLEEP commands, and accordingly, is independent of other timing delays. If the comparison shows that the interval counter has a larger value than the tics value specified by the command, the warning '*' will be printed to the primary UART. (At the same time the lsb of the OVRSLP system variable is set. You may check this flag by using the command PEEK(&H9E) and using POKE &H9E,0 will clear it). If 'tics' = 0, no check is made, but the interval counter is reset to zero.



The figure above shows the timing for a SLEEP 100 command that was separated from the previous SLEEP command by commands that took about 65 mS to execute. If the intervening commands had taken more than a second, the current drain would not have dropped, the interval counter would have been reset to zero and a '*' would have been sent out the hardware UART.

See Also: HYB, RATE

SQR *square root*

Syntax: value = SQR(<x>)

Description: SQR returns the square root of the expression in parentheses. This function takes a floating point argument and returns a floating point result. If the argument is an integer, it will be converted to float first.

Examples: build this program with your editor (indentation for clarity only)

```
start:
   input "SQR("argument!;
   print ") = ",#F,sqr(argument)
   goto start
```

run it in TFTools

```
SQR(1) = 1.000000
SQR(2) = 1.414214
SQR(4) = 2.000000
SQR(100) = 10.000000
SQR(1000) = 31.622778
SQR(1000000) = 1000.000000
SQR(123.4) = 11.108556
SQR(-1) = NaN  ("Not-a-Number" because sqr(-1) is imaginary)
SQR(16) = 4.000000 (Note the NaN error didn't stop program)
```

Remarks: Taking the square root of a negative number returns a value of Not-a-Number (specified by the IEEE 754 floating point specification). This will not stop program execution. It sets the Not-a-Number bit in the floating point error variable FPERR.

See Also: EXP

STIME *set local software real-time-clock*

Syntax: STIME

or

STIME <v>

Description: STIME translates the 6 element '?()' array to the '?' variable. Time is counted in 1/100ths of a second starting at New Years Day 1980. This command, with RTIME, allows setting and reading the Tattletale's real-time clock.

Examples: build this program with your editor:

```
input 'THE YEAR IS (1980 TO 2040) ' ?(5)
input 'THE MONTH IS (1 TO 12) ' ?(4)
input 'THE DAY IS (1 TO 31) ' ?(3)
input 'THE HOUR IS (0 TO 23) '?(2)
input 'THE MINUTE IS (0 TO 59) '?(1)
input 'THE SECOND IS (0 TO 59) '?(0)
STOP
```

Output:

```
THE YEAR IS (0 TO 99) 1996
THE MONTH IS (1 TO 12) 5
THE DAY IS (1 TO 31) 7
THE HOUR IS (0 TO 23) 15
THE MINUTE IS (0 TO 59) 23
THE SECOND IS (0 TO 59) 45
```

Remarks: STIME <v> translates the '?' array to the variable <v>. Time is counted in seconds starting at New Years Day 1980. In both cases dates beyond 2047 are out of range. Leap years are handled properly.

Cautions: The ? array is predefined. Do not initialize it with DIM!

See Also: RTIME, SETRTC, READRTC, and the section “TFX Timekeeping”

STOP *stop program execution*

Syntax: STOP

Description: This command stops TFBASIC execution. A STOP command is not needed at the end of the program.

Examples:

Build this program with your editor (indentation for clarity only)

```
      gosub psub
      stop

psub: print 'A very dull sample program'
      return
```

Output A very dull sample program

Remarks: A program can have any number of STOP lines.

See Also: HALT

STORE *store to datafile*

Syntax: STORE [[#size] [,expr] . . .]

STORE [[#n] [,<x>] . . .]

Description: STORE places bytes, words, double words or strings into the next free location of the SFLASH datafile. Formats #1, #2, and #4 are used to specify whether the data is to be stored as one, two or four bytes. #1 specifies that the least significant 8 bits be stored, #2 specifies that the least significant 16 bits be stored, and #4 specifies that all 32 bits be stored. The pointer variable is updated after the data has been stored - it is accessible from TFBASIC as the read-only variable DFPNT. The default format of store is in bytes so that the #1 can be omitted in byte storage commands.

Example - build this program

```
Channl = 10          //assign your particular channel
FOR A=1 TO 5         //store five temp measurements
  T=TEMP(CHAN(Channl))
  PRINT #4,T/100,".",#02,T%100; // print them too
  STORE #2,T
NEXT A
```

Program's Output

```
  21.14  21.14  21.14  21.14  21.14
```

Remarks: The #size specifier is ignored for floating point and string variables. Floats always store as four bytes. Strings store length byte first followed by that number of characters that make up the string.

Cautions: Data in the datafile is inaccesible from the program., and can only be retrieved on offload. If you will need the stored data later for calculations etc. then put a copy in the @ array read/write storage.

See Also: PRINT

STR *assign ASCII formatted output to string*

Syntax: STR ([“s”] [,#n] [,x1] [,\x2])

Description: Create a string by concatenating quoted strings, converted numerical values (optionally formatted), string variables and character constants in any order.

“s”	represents a quoted string
#n	optional numeric formatting
x1	integer, floating point, or string expressions
\x2	character constants

Examples:

```
tempc! = float(temp(chan(0)))/100.0
rh!    = float(chan(2))
BigStr$ = str(“degrees c =”, #6.2F, fempC!,” : RH = “, rh!)
print BigStr$
store BigStr$
```

Output:

```
degrees c = 50.5 : RH = 64.00
```

Remarks: This function formats output exactly like the print statement, but allows assignment of the results to a string variable.

Cautions: After the output string reaches 255 characters in length all other characters are discarded.

See Also: PRINT

TAN *Tangent*

Syntax: value = TAN(<x>)

Description: TAN returns the tangent of the expression in the parenthesis. The argument must be in degrees. The function takes a floating point argument and returns a floating point value. An integer argument will be converted to float first.

Example:

```
degrees! = 0.0  // init arg (notice '!' means it's a float)
result! = 0.0  // just to force 'result' to be a float
for i = 0 to 6
   result = tan(degrees)
   print "The tangent of ", #5.1F, degrees, " is ", #6.3F, result
   degrees = degrees + 72.0
next i
```

Output:

```
The tangent of   0.0 is  0.000
The tangent of  72.0 is  3.078
The tangent of 144.0 is –0.726
The tangent of 216.0 is  0.726
The tangent of 288.0 is –3.078
The tangent of 360.0 is  0.000
```

Cautions: **DON’T FORGET!** The argument is in degrees, not radians.

See Also: SIN, COS, ATN

TEMP *convert number to temperature*

Syntax: value = TEMP(<x>)

Description: TEMP converts the 0-65535 value 'x' to a temperature, assuming that x results from a measurement of one of the A/D channels that is connected to a thermistor voltage divider circuit. The temperature conversion is given in hundredths of degrees C.

Examples: Build this program with your editor

```
for A = 0 to 65520 step 256
   tempVal = temp(A)
   print #4,tempVal/100,".",#02,abs(tempVal) % 100;
   if ((A/256) % 10) = 9 print
next A
```

Program's Output

```
166.00 166.00 166.00 166.00 165.88 158.48 151.08 143.68 136.33 132.00
127.68 123.35 119.05 116.12 113.20 110.27 107.36 105.16 102.96 100.76
98.57  96.82  95.07  93.32  91.57  90.10  88.62  87.15  85.68  84.40
83.13  81.85  80.58  79.48  78.38  77.28  76.18  75.18  74.18  73.18
72.18  71.31  70.43  69.56  68.68  67.86  67.03  66.21  65.38  64.61
63.83  63.06  62.28  61.58  60.88  60.18  59.48  58.83  58.18  57.53
56.89  56.26  55.64  55.01  54.39  53.81  53.24  52.66  52.09  51.51
50.94  50.36  49.79  49.26  48.74  48.21  47.69  47.16  46.64  46.11
45.59  45.09  44.59  44.09  43.59  43.11  42.64  42.16  41.69  41.24
40.79  40.34  39.89  39.44  38.99  38.54  38.09  37.64  37.19  36.74
36.29  35.86  35.44  35.01  34.59  34.16  33.74  33.31  32.89  32.49
32.09  31.69  31.29  30.89  30.49  30.09  29.69  29.29  28.89  28.49
28.09  27.69  27.29  26.89  26.49  26.11  25.74  25.36  24.99  24.59
24.19  23.79  23.39  23.01  22.64  22.26  21.89  21.49  21.09  20.69
20.29  19.91  19.54  19.16  18.79  18.41  18.04  17.66  17.29  16.91
16.54  16.16  15.79  15.39  14.99  14.59  14.19  13.81  13.44  13.06
12.69  12.29  11.89  11.49  11.09  10.71  10.34   9.96   9.59   9.19
8.79   8.39   7.99   7.59   7.19   6.79   6.39   5.96   5.54   5.11
4.69   4.26   3.84   3.41   2.99   2.56   2.14   1.71   1.29   0.86
0.44   0.01   0.41   0.89  -1.36  -1.84  -2.31  -2.79  -3.26  -3.74
-4.21  -4.69  -5.16  -5.64  -6.11  -6.64  -7.16  -7.69  -8.21  -8.76
-9.31  -9.86 -10.41 -10.99 -11.56 -12.14 -12.71 -13.34 -13.96 -14.59
-15.22 -15.87 -16.52 -17.17 -17.82 -18.57 -19.32 -20.07 -20.82 -21.62
-22.42 -23.22 -24.02 -24.97 -25.92 -26.87 -27.82 -28.90 -29.97 -31.05
-32.13 -33.48 -34.83 -36.18 -37.53 -39.36 -41.18 -43.01 -44.85 -47.65
-50.45 -53.25 -56.00 -56.00 -56.00 -56.00
```

Cautions: No attempt has been made to resolve temperatures greater than 166 C or less than -56C. **Using the12 bit converter** gives about 0.03C resolution for temperatures between about 0C and 35C.

Remarks: The Getting Started section shows the schematic for the circuit and describes how to connect the thermistor to take temperature readings

See Also:

TONE *send square wave out*

Syntax: TONE period, count

TONE <x1>,<x2>

Description: TONE allows you to produce a square wave of predetermined period and duration output on I/O line 3 The expression 'period' gives the square wave period, which is measured in multiples of 0.8138µSec. The range of period is 70 (about 17554Hz) to 65535 (about19Hz). The expression 'count' gives the number of cycles to be output, which can be as few as one, or as many as 32767 cycles. This output can be used to drive a speaker and produce notes covering a large portion of the audio spectrum. TONE sets I/O line 3 to an output and leaves I/O line 3 low at the end of a command.

A special form of TONE is available to produce a continuous square wave. Use a value of 0 for the count argument and the square wave will continue (at the period you select) until you execute a TONE with a zero value for both the period and count arguments. You can change the period of this continuous signal “on the fly “ by using TONE with the new period argument and a zero count argument. The square wave period will change on the next transition of the signal.

Example 1:

```
X=1000000/4/44/16// Two octaves above 440 'A'
for A=24 to 0 step –1
@(A)=X
X=X*106/100
next A // Have half tone scale from @(0) to @(23)
Y=500000
tone @(21),Y/@(21)
tone @(23),Y/@(23)
tone @(19),Y/@(19)
tone @(5),Y/@(5)
tone @(14),Y/@(14)// Recognize that?
```

Example 2:

```
sleep 0
for i = 10 to 1 step –1
tone i*60,0     // update continuous signal period
sleep 10        // produce this frequency for 100 mSec
next i
tone 0, 0        // stop the tone
```

Remarks: In the example above, the number of cycles is normalized by the interval to make the length of the note constant. The above poorly-tempered scale is actually remarkably close to a well-tempered scale. PRINT 10000*@(24)/@(0) yields the value 2515, about 0.7% away from the right number (2500). The 1.06 ratio is good to about 1/10 of a half tone per octave.

Cautions: Driving circuitry should expect the quiescent state to be low. If the TONE command is not used for a while after power-up, set that line low with a PCLR 3 command to ensure that the speaker driver is in its low power state!

UGET *bring character in software UART*

Syntax: UGET baud, count, strvar, timeout

UGET <x1>, <x2>, <v>, <x3>

Description: The UGET command treats I/O line 1 as a UART input line to receive serial data. The input expects CMOS levels and has the marking state high, inverted from the normal RS-232 sense. The baud rate is specified by the expression 'baud' and can be any value between 100 baud and 19200. The expression 'count' specifies the number of bytes to be received. 'Strvar' specifies the string variable to be used to store the data as it is received. The data is stored directly into memory and is not echoed to the screen. The expression 'timeout' specifies the time-out (in 1/100ths of a second); this time-out aborts the receiving routine in case of an external system failure and starts at the beginning of the execution of the command. time-out can be set to any value between 0.01 and 655.35 seconds.

Example: (indentation for clarity only)

```
        X$ =""
        UGET 4800,10,X$,800
        IF LEN(X$)<10
         GOTO LT10
        ENDIF
        PRINT "RECEIVED ALL":STOP
LT10:   PRINT "RECEIVED ONLY ",LEN(X$)," CHARACTERS"
```

Output:

```
RECEIVED ONLY 0 CHARACTERS
```

This program will try to receive 10 characters at 4800 baud, storing them int the variable X$, and timing out if all ten are not received in eight seconds. Provisions are also made to detect and deal with external system failure.

Remarks: UGET uses I/O line 1 as a UART input line. Like USEND, this line uses CMOS levels and has a marking high state, inverted from the normal RS-232 sense. The transfer format for the received characters is eight data bits, no parity, with one stop bit (at least). The 'count' parameter has a maximum of 255.

See Also: USEND

USEND *send characters out software UART*

Syntax: USEND baud, <string>

Description: USEND sends serial data out I/O line 2. The argument can be any string expression; constant, variable,or function. The range for 'baud' is 100 to 19200 or a runtime error will occur.

Example 1: (indentation for clarity only)

```
start: input "enter text : "istring$
       usend 300,istring$
       usend 300, "\x0D\x0A"   // send <cr> <lf>
       stop
```

input this string at the prompt followed by <ENTER>

```
THIS IS A TEST
```

Output: :

```
THIS IS A TEST
```

Example 2

```
       USEND 9600,"            seconds from power-up";
loop:  X=?
       USEND 9600 str(\13,#8,X/100,".",#02,X%100)
       GOTO loop
```

This simple program continuously displays the time on a terminal connected to I/O line 2 through a level shifter.

Remarks: USEND uses I/O line 2 as a UART output line. Like UGET, this line uses CMOS levels and has a marking high state, inverted from the normal RS-232 sense. The transfer format for the transmitted characters is eight data bits, no parity, with one stop bit (at least).

Cautions: There is no breakout from USEND, and sending 220K bytes at 110 baud will take about 6 hours!

See Also: UGET, STR

VARPTR *get address of named variable*

Syntax: VARPTR (<variable>)

Description: This function returns the address of the variable named in parentheses. Since variables are defined when they are first used in a program, the variable can be used here before being defined elsewhere. Also, the variable can be an array but it cannot include the index value. Be aware that VARPTR returns the address of the most significant byte of the variable. The least significant byte = VARPTR (data) + 3.

Examples: (indentation for clarity only)

```
print varptr(data)   //address of beginning of 'data' array (MSB)
print varptr(data)+3 //address of beginning of 'data' array (LSB)

start_prog:
   print "square of values from 0 to 9"
   for a = 0 to 9
      b = a * a
      print #5, a, b
   next a
   print "last b value bytes :  ";
   print #02H, peek(varptr(b)), ' ', peek(varptr(b)+1), ' ';
   print #02H, peek(varptr(b)+2), ' ', peek(varptr(b)+3)
end_prog:
```

See Also: PEEK, POKE

VGET *get variable from user EEPROM*

Syntax: value = VGET(<expr>)

Description: VGET returns the corresponding variable value from the User EEPROM (UEEPROM). Although there are 128 bytes available in the UEEPROM, they can only be assigned to in 4-byte blocks. These correspond to locations 0-31.

Example:

```
// Store and retrieve two bytes and a word in a single location

CalValA = 124
CalValB = 37
CalValC = 1027
VSTORE 0, (CalValA + CalValB*256 + CalValB*65536)
...
CalValA = (VGET(0) & &H000000FF)
CalValB = ((VGET(0)/256) & &H000000FF)
CalValC = ((VGET(0)/65536) & &H0000FFFF)
```

Remarks: VGET can be used to retrieve unique user calibration parameters stored in the UEEPROM by VSTORE. This permits instruments to have identical TFBASIC programs while allowing for variations in sensors calibrations.

All values stored in the UEEPROM are assumed to be integer. ASFLT can be used to recover a floating point value stored in the UEEPROM.

```
FLPTV! = 80.86
VSTORE  20, FLPTV!
FLPTV! = ASFLT(VGET(20))
```

Cautions: The UEEPROM is only readable and writable from TFBASIC. Any values stored will not be off-loaded with the main flash off-load unless deliberately stored into the main flash by your TFBASIC program.

See Also: VSTORE, ASFLT

VSTORE *store variable to user EEPROM*

Syntax: VSTORE <address>, <expr>

Description: VSTORE stores the 32-bit value of expr at the UEEPROM address. There are 128 bytes available in the UEEPROM, but they can only be assigned to in 4-byte blocks. Valid addresses are 0-31.

Example:

```
TempCalVal = 5
PressCalVal! = 1.05
HumidCalVal! = 0.24

VSTORE 0, TempCalVal         // store an integer value
VSTORE 1, PressCalVal!       // store a floating point value
VSTORE 2, HumidCalVal!

CurrTemp = CHAN(6)* VGET(0)
CurrPress! = float(CHAN(1))* ASFLT(VGET(1))
CurrHumid! = float(CHAN(3))* ASFLT(VGET(2))
```

Remarks: Both integer and float variables take four bytes, and thus either will neatly fit into one location. A simple VGET will return the proper integer value, but ASFLT() is necessary to recover a floating point value stored in the UEEPROM. No special commands are necessary to store a floating point value.

Cautions: The UEEPROM is only readable and writable from TFBASIC. Any values stored will not be off-loaded with the main flash off-load unless deliberately stored into the main flash by your TFBASIC program.

The UEEPROM has a finite number of write cycles, so avoid frequent rewriting of locations such as might happen if the VSTORE command appears in a loop.

See Also: VGET, ASFLT

WHILE *loop while expression true*

Syntax:

WHILE expression

...commands...

...to be executed...

WEND

Description: WHILE loops provide one of four methods of looping available in TFBASIC. The code between the WHILE and WEND commands will be executed as long as 'expression' is true. Like the FOR loop, and unlike the REPEAT loop, the testing of 'expression' takes place before the loop is executed. Because this structure stores nothing on the stack, you can nest these loops as deeply as you like. A GOTO can be used to exit any number of nested WHILE loops.

Example: (indentation for clarity only)

```
// collect data as long as I/O pin 0 is low

    onerr HandleError, errVar
    dfPointer = 0
    sleep 0

    while pin(0) = 0
      store dfPointer, #2, chan(0), chan(2)
      sleep 100
      wend

    print "Finished logging, ready to off-load"
    stop

HandleError:
    if errVar/65536 = 4
       print "Ran out of datafile. Stopped logging"
    else
print "Logging stopped by error #",errVar/65536
    stop
```

Remarks: none

See Also: FOR, GOTO, REPEAT

XMIT+, XMIT– *enable, disable console output*

Syntax: XMIT–

or

XMIT+

Description: On receiving the XMIT– command, the Tattletale will stop echoing characters received by the main UART. XMIT– remains in effect until the Tattletale receives XMIT+ or a power-on reset. Under the influence of XMIT–, characters sent to the hardware UART are simply thrown away.

Example:

```
xmit–
print "You won't see this"
xmit+
print "You WILL see this"
print "Input a number: ";
xmit–
input "this prompt will be lost"aNumber
xmit+
print "the number is ", aNumber
```

Output:

```
you WILL see this
Input a number: <type 123 here> the number is 123
```

Remarks: XMIT– is most useful when you don't what characters entered into the Tattletale to be echoed.

Cautions: Remember, everything is suppressed by XMIT–, even error messages!

See Also: ITEXT for another method of not echoing input characters.

TFBASIC Error Messages

There are two different kinds of errors recognized by TFBASIC. The first type is caught in the tokenizer. These are usually editing or syntax errors. The second type of error is caught at run-time. These are usually mathematical bounds errors. When an error is encountered in an executing program, execution is halted, and the word "HOW?" and the error number are displayed followed by the address of offending token. The token address can be found in the Token List file created by the Alt-T command in TFTools. The error numbers with causes are listed below:

HOW? errors

- 1 : not used in TFBASIC
- 2 : array variable index out of range
- 3 : not used in TFBASIC
- 4 : STORE out of range of the datafile
- 5 : not used in TFBASIC
- 6 : integer divide by zero
- 7 : integer multiply overflow
- 8 : not used in TFBASIC
- 9 : integer add or subtract overflow
- 10 : ABS argument = -2147483648
- 11 : A-D channel not supported
- 12 : I/O pin not supported
- 13 : attempt to set input only pin to output
- 14 : input to TEMP out of range
- 15 : not used in TFBASIC
- 16 :SDI requested <1 or >32 bits
- 17 : COUNT timeout > 65535
- 18 : PERIOD argument out of range
- 19 : not used in TFBASIC
- 20 : not used in TFBASIC
- 21 : not used in TFBASIC
- 22 : not used in TFBASIC
- 23 : not used in TFBASIC
- 24 : SLEEP interval > 32767
- 25 : UGET/USEND baud rate out of range
- 26 : not used in TFBASIC

- 27 : TONE parameter out of range
- 28 : not used in TFBASIC
- 29 : date/time input to STIME out of range
- 30 : integer input to RTIME out of range
- 31 : HYB interval overflow
- 32 : SDO requested <1 or >32 bits
- 33 : CALL address > 65535
- 34 : not used in TFBASIC
- 35 : not used in TFBASIC
- 36 : VSTORE/VGET index out of range (>31)
- 37 : UGET timeout > 65535
- 38 : STIME out of range
- 39 : not used in TFBASIC
- 40 : not used in TFBASIC
- 41 : PRINT field width > 255
- 42 : FIX/INT result overflow
- 43 : OFFLD x,y; where x > y
- 44 : Bad argument to BAUDSET
- 45 : Flash EEPROM command failed
- 46 : not used in TFBASIC
- 47 : array index out of bounds
- 48 : not used in TFBASIC
- 49 : Illegal RATE argument
- 50 : not used in TFBASIC
- 51 : stack running low
- 52 : PIC command failure
- 53 : not used in TFBASIC

- For applications where these responses are undesirable, they can be replaced by a 'goto on error' response using the ONERR command.

CHAPTER 5

TFBASIC Assembly Language Reference

TFBASIC Assembly Language Reference

TFBASIC Assembly Language

The TFBASIC tokenizer (running on the host computer) has a built-in assembler. The tokenizer switches from generating tokens to assembling when it encounters the ASM command and switches back to tokenizing when it encounters the END command. This assembler allows the use of named labels and it can access TFBASIC variables by name.

Labels, assembler

Labels can be used in the assembly code for flow control and to define local variables. Labels must start in the first column. Labels can be up to 32 characters long and must begin with a letter or an underscore (_). The only valid characters in a label are upper and lower case characters, the numbers and underscore. The label name can be terminated with a colon (when the label is defined) but this is not necessary in the assembler. These labels are accessible to the CALL command as long as they are defined before CALL is invoked. TFBASIC labels are not accessible to the assembly code (although TFBASIC variables are).

Assembler Opcodes

The TFBASIC assembler recognizes all of the opcodes defined in the Motorola manuals with some exceptions including all those using the Y register. These include ABY, CPY, DEY, INY, LDY, PSHY, PULY, STY, TSY, TYS, XGDY, CPD, STOP, BRCLR, BRSET, IDIV, and FDIV.

Opcodes must have at least one character of whitespace (space character or tab) in front of them on the line OR a label terminated with a colon.

Referencing HC11 control registers

The Motorola reference manual places the control registers starting at location 1000H. On startup the TFX-11 relocates these registers to start at location 0. When referencing the control registers in your assembly code don’t forget to subtract the 1000H from the addresses given in the Motorola manual to get the correct address.

Two forms of ASM

Assembly routines can be either executed in-line with TFBASIC commands or accessed as subroutines from TFBASIC using the CALL command depending on the argument that follows ASM.

In-line assembly (ASM $)

The assembler (built in to the tokenizer) assembles from the line after the ASM $ until it detects the END command. The interpreter will switch from interpreting tokens to executing the assembly code when it reaches the

ASM$ statement, and go back to interpreting tokens when it reaches the END statement. No RTS, RTI or other special ending command is needed before the END statement.

Example This example shifts each bit of a TFBASIC variable one bit to the left with the most significant bit rotated around to the least significant bit position. Note how the TFBASIC variables are handled. The name ToBeRotated points to the most significant byte of the variable.To access the other three bytes of the variable:

To get this byte of variable:	Use this name:
most significant byte	ToBeRotated
next most significant byte	ToBeRotated+1
third most significant byte	ToBeRotated+2
least significant byte	ToBeRotated+3

Example 1:

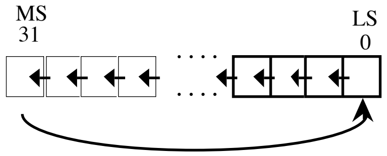
```
       input “Value to rotate: “ ToBeRotated
       input “Number of bits to rotate: “NumberShifts
       print “value before rotation “,ToBeRotated
asm $
       ;nothing else on above line - not even comments!
       ldab   NumberShifts+3        ;get number of shifts in
                                    ; B register
loop   beq    leave                 ;if number of shifts is
                                    ; zero, exit
       rol    ToBeRotated+3         ;ls byte, ms bit to carry,
                                    ; garbage into ls bit
       rol    ToBeRotated+2         ;carry into ls bit, ms bit
                                    ; into carry
       rol    ToBeRotated+1
       rol    ToBeRotated           ;done, except ls bit of ls
                                    ; byte is garbage

       bcc    no_carry              ;branch if carry = 0
       ldaa   ToBeRotated+3         ;get here if carry bit set
       oraa   #1                    ; so set the ls bit
       bra    endloop

no_carry
       ldaa   ToBeRotated+3         ;carry bit (from ms bit)
                                    ; is 0 so
       anda   #&HFE                 ;clear the ls bit

endloop
       staa   ToBeRotated+3         ; restore the ls byte
       decb                        ; count shift completed
       bra    loop
leave  end
       //on the above line, leave is an assembler label and end
       //end is a TFBASIC command

       print "value after rotation ",ToBeRotated
       stop
```



The figure above shows the operation of the assembly routine for one pass around the loop. Each TFBASIC variable is 32 bits wide. The figure shows the four most significant and four least significant bits of variable ToBeRotated. The value in variable NumberShifts tells how many times to do this operation. In the assembly section, everything after a semicolon and up to the end of the line, is considered a comment. The assembler does not recognize TFBASIC comments.

This first form of the ASM command provides no way to initialize the A, B or X registers before entering the assembly code. The second form of ASM, described below, does.

Assembly to an address (ASM <address>)

This form of embedding assembly code differs from the above in that when the token interpreter reaches this point in the program, it does not execute the assembly code!. Instead it copies the assembly code to the address specified in the argument following the ASM command. After placing the code at the specified address the token interpreter continues execution with the tokens following the assembly code. To actually execute the assembly routine, you must use the **CALL** command.

Also with this form, an RTS instruction is automatically appended to any code you write. This is fine if you're writing normal assembly subroutines but you cannot use this method to write data to a RAM address. Use the POKE command instead.

The TFBASIC 'CALL' command syntax

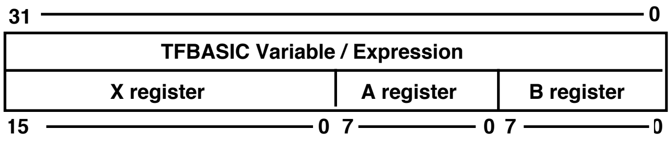
CALL <x1>, <x2>, [<v>]

x1 the address of the assembly routine - this is the memory address or a predefined label following the ASM command.

x2 the state of the X, A and B registers to be upon entering the subroutine

v an optional variable name to receive the state of the X, A and B registers when the subroutine returns to TFBASIC.

The translation of the **x2** and **v** arguments into the microprocessors registers:



In the example below, the B register is filled with the ASCII value of a character to be sent out the UART. The X register is filled with the number of times to send this character (beware, zero in the X register will send it 65536 times!).

Example 2:

```
// do not use location 400H if using CSIO1 and CSIO2 chip selects
print “Start out in TFBASIC”
asm &H0400
             ;assembly routine will load at address74C0H >
test   ldaa   H’2E          ;test if transmit register empty
       anda   H’80          ;test bit
       beq    test          ;branch back if not empty

       stab   H’2F          ;send character in B reg by storing
                            ; in register 13H
       dex                  ;Count down one character
       bne    test          ; if not zero, branch back for more
       end
       print “ASM code has been loaded at 0400H”

       sleep 0
       sleep 50      // must wait for PRINT output to finish

       call &H0400,&H10041
       // X register= 1, A register= 0, B register= 41H =‘A’
       call &H0400, &H20042
       // X register= 2, A register= 0, B register= 42H =’B’
       call &H0400, &H30043
       // X register= 3, A register= 0, B register= 43H =’C’
       call &H0400, &H1000D
       // X register= 1, A register= 0, B register= 0DH = CR
       call &H0400, &H1000A
       // X register= 1, A register= 0, B register= 0AH = LF
       print “Finished”
```

Notice the lines sleep 0 followed by sleep 50. The PRINT command in TFBASIC sends all its output to a buffer. The characters in this buffer are sent out the UART in the background as fast as the baud rate will allow, but this is slow compared to the speed of the CPU. When the PRINT command is completed, TFBASIC continues on with the next command while characters may still be waiting in the buffer to be sent out the UART. The assembly routine we wrote bypasses the UART buffer and will interfere with characters being automatically output from the buffer. So, we pause to let the buffer clear before using our routine.

Radix Notice the new methods of defining the number base of constants. You have these options IN THE ASSEMBLER ONLY. For example, the constant 19 decimal can be defined as:

```
Decimal is the default base
&H works the same as in TFBASIC

decimal:       19D or D'19 or 19
hex:           13H or &H13 or H'13 or $13
octal:         23O or 23Q or Q'23 or @23
binary:        10011B or B'10011 or %10011
```

ASM mnemonics and addressing modes

The TFBASIC assembler uses the same opcode mnemonics that Motorola uses in its HC11 literature. All assembler mnemonics must have at least one character of whitespace (space character or tab) in front of them on the line or a label terminated with a colon.

Single Byte (Inherent) The single byte instructions, like NOP and RTS, can be preceded and/or followed by an arbitrary number of spaces and tabs.

Indirect (Indexed) These instructions use the 'X' register as a pointer. They have the form:

```
MNEMONIC <expr>,X
```

where <expr> is a positive offset and must evaluate to a number in the range of 0 to 255, and the 'X' (upper or lower case) must follow the comma without any intervening space. Examples:

```
LDAA 0,X      ; load A reg with byte pointed to by X
STAA 7*8-4,X  ; store A reg at address contained in X + 52
```

Extended Extended instructions may address within the full 64K memory space. They have the form

```
MNEMONIC <expr>
```

If <expr> evaluates to a number outside the range of 0 to 65535, the assembler (in the tokenizer) will print an error message. Examples:

```
LSR H'74C0   ; logical shift right of the byte at 74C0H
ROL 118H     ; rotate byte at 118H left through carry
```

Extended/Direct Many instructions have shortened versions (called 'direct' by Motorola) for addressing page zero memory (address < 256), as well as full length versions for addressing all of memory. The 'direct' version of the instructions takes two bytes instead of three. The assembler first evaluates the <expr>. If the result is in the range 0 to 255, the assembler uses the direct form; otherwise, the assembler uses the extended form. Examples:

```
SUBB 10     ;direct, subtract byte at addr 10 from B reg
SUBB 256    ;extended,subtract byte at address 256 from B
CMPA 255    ;direct, compare A reg with byte at addr 255
LDAA 7*128  ;extended, load A reg with byte at address 896
```

Relative The branch instructions form a displacement from the address of the byte immediately following the branch instruction to the address that is to be branched to. In the form used by the assembler, the address is specified, not the displacement. The assembler will then calculate the displacement. You can use either a label or an absolute number to specify the address. Examples:

```
BNE _NoMore   ; branch if not equal to label '_NoMore'
BRA 74C0H     ; branch always to address 74C0H
```

Relative branches must be within +127 and -128 of the start of the next instruction. If the displacement does not fall within this range, a tokenizer error will result.

Immediate An immediate instruction deals with a value instead of an address. The form of the immediate instruction is shown below:

```
MNEMONIC #<expr>
```

There must be white space (tabs or spaces) between the MNEMONIC and the '#'. There can be white space between the '#' and the <expr>. The assembler checks <expr> and makes sure it is in range (0 to 255 for some commands, 0 to 65535 for others). Examples:

```
LDAA #&H23          ; load A reg with 35 (23H)
LDD  #&H1234        ; load A/B registers with 4660 (1234H)
SUBD #7*16+&H4000   ; subtract 16496 (4070H) from A/B reg
```

Data Test and Bit Manipulation These instructions read the addressed memory, the first operand, and modify this value using the second operand. The modified value is then written back to the target address.

```
BSET   Flags,H’80    ;Set MSB in location Flags
BCLR   Flags,H’80    ;Clear MSB in location Flags-
                     ; note the bit cleared is set
                     ; in the operand mask.
```

These instructions perform a read-modify-write on their target locations. Caution must be used when read-modify-write instructions such as BSET and BCLR are used with I/O and control registers because the physical location read is not always the same as the location written. See the Motorola documentation for more information.

Summary of TFBASIC Assembler Directives

The assembler makes some directives and pseudo-ops available. Here is a brief listing of the directives in the TFBASIC assembler. The second section goes into more detail and provides examples of how the directives are used.

The TFBASIC assembler directives are grouped into four categories: source control, data declaration, symbol declaration and location control. The source control directives tell the assembler where its source code starts and ends. The data declarations allow you to initialize an area of memory from an expression or string. The symbol declaration directive allows you to assign a numeric value to a symbol name that can be used anywhere a constant is expected. The location controls allow you to set or modify the location counter (the address to which code is assembled). Notice that the TFBASIC command ASM is in two categories.

Source Control	`ASM`	`Start assembly source (See TFBASIC Command ref.)`
	`END`	`End assembly and return to TFBASIC`
Data Declarations	`DATA`	`Declare initialized data (IEEE)`
	`DB`	`"Define Byte" (Intel)`
	`FCB`	`"Form Constant Byte" (Motorola)`
	`DW`	`"Define Word" (Intel)`
	`FDB`	`"Form Double Byte" (Motorola)`
	`FCC`	`"Form Constant Characters" (Motorola)`
Symbol Declarations	`EQU`	`Assign value to symbol`
Location Control	`ALIGN`	`Force location counter alignment`
	`RES`	`Reserve uninitialized data`
	`DS`	`"Define Storage" (Intel)`
	`RMB`	`"Reserve Memory Block" (Motorola)`

ALIGN

Syntax: `[label]align  expr   [; comments]`

Description: The align directive forces the location counter to align to a boundary that is a multiple of the value specified by the expression in the operand field.

Examples:

```
_align:        align  2      ; align on word boundary
               data   "word"
               align  4      ; align on long boundary
               data   "long"
               align  256    ; align on page boundary
               data   "page"
               align  32768  ; align on enormous boundary
               data   "huge"
               align  1      ; back to byte boundary
               data   "byte"
```

Remarks: Values in the align expression must not contain any forward references.

DATA, DB, FCB, DW, FDB, FCC

Syntax:

```
[label]data[.size] expr|"string" [,expr|"string"...][; comments]
[label]db     expr|"string" [,expr|"string"...] [; comments]
[label]fcb    expr|"string" [,expr|"string"...] [; comments]
[label]dw     expr|"string" [,expr|"string"...] [; comments]
[label]fdb    expr|"string" [,expr|"string"...] [; comments]
[label]fcc    <delim>"string" <delim> [; comments]
```

Size Specifiers (for 'data' directive only)

.b	.B	Byte
.s	.S	Short(2 Bytes)
.w	.W	Word (2 Bytes)
.l	.L	Long (4 Bytes)
.q	.Q	Quad (8 Bytes)

Description: The various data declaration directives instruct the assembler to generate initialized data from expressions and strings. The size of the data generated for each expression is determined either explicitly with "dot-size" suffixing as in the data directive, or implicitly by the directive name. Expression values are truncated to fit into objects of the specified size. The data directive and the "dot-size" suffixes shown in the table conform to the guidelines of the IEEE-694 standard for microprocessor assembly language. The **fcb** (Form Constant Byte), **fdb** (Form Double Byte), and **fcc** (Form Constant Characters) are provided as a convenience to programmers more comfortable with Motorola pseudo-ops, while the **db** (Define Byte), and **dw** (Define Word) are available for programmers preferring Intel mnemonics. With the exception of the **fcc** directive, all of the data directives accept any combination of strings and expressions. Strings used with these directives must be enclosed inside double-quotes. Use two consecutive double-quotes to generate one double-quote character in the output data.

The **fcc** directive generates data from string expressions bracketed by the first character encountered in the operand field. Use two consecutive occurrences of the delimiting character to generate one instance of the character in the output data.

Examples:

```
_data: data   "sample string"
       data   "several", "different", "strings"
       data   "several", 0, "terminated", 0, "strings", 0
       data   "sample string with quote ("") character"
       data   "string with byte expression", 10
```

```
;
       data   0,1,2,3       ; yields: 00 01 02 03
       data.b 0,1,2,3       ; yields: 00 01 02 03
       data.w 0,1,2,3       ; yields: 00 00 00 01 00 02 00 03
       data.l 0,1           ; yields: 00 00 00 00 00 00 00 01
       data.q 0             ; yields: 00 00 00 00 00 00 00 00
;
_db:   db     "string with ending bit 7 hig", 'h' + h'80
_fcb:  fcb    "string with embedded quote ("")"
       db     'ab'          ; should just be h'61
       fcb    ''''          ; the single-quote character
;
_dw:   dw     0,1,2,3       ; yields: 00 00 00 01 00 02 00 03
       dw     'ab'          ; should be h'6162
_fdb:  fdb    ''''''        ; two single-quote characters
;
_fcc:  fcc    :string with colon (::) terminators:
       fcc    ~string with tilde (~~) terminators~
       fcc    /string with forward slash (//) terminators/
```

Limits: Maximum of 255 bytes generated from a single data statement.

END

Syntax: `[label]end    [; comments]`

Description: The end directive instructs the assembler to stop reading the current source file and return control to the TFBASIC tokenizer.

Example:

```
asm $
       ldaa   #12
       ldx    #H'86
       staa   0,x
       end           ; Configuration Byte set
```

EQU

Syntax:

```
label equ expr [; comments]
```

Description

The equ directive permanently assigns a numeric value to a symbol which may be used anywhere a constant is expected. Both the label and operand fields are required.

Examples:

```
FALSE          equ     0
TRUE           equ     ~FALSE
DEBUG          equ     FALSE
DEL            equ     H'7F    ; ASCII delete
CR             equ     H'0D    ; carriage return
LF             equ     H'0A    ; line feed
TAB            equ     H'09    ; horizontal tab
SP             equ     H'20    ; 'space' character
```

Remarks:

Values used in the assignment expression must not contain any forward references. The TFBASIC assembler does not adapt EQU to perform text substitution on arguments which do not evaluate to a number.

RES, DS, RMB

Syntax:

```
[label]res    expr   [; comments]
[label]ds     expr   [; comments]
[label]rmb    expr   [; comments]
```

Description: The res, ds, and rmb directives define a block of uninitialized data equal in length to the value of the expression in the operand field multiplied by the optional size specifier. Blocks declared using the reserve directives affect only the location counter and do not generate any output data in the object file.

Examples:

```
asm    h'4000
_res:  res    h'100          ; should be 4000 - 40FF
       res.b  h'100          ; should be 4100 - 41FF
       res.s  h'100          ; should be 4200 - 43FF
       res.w  h'100          ; should be 4400 - 45FF
       res.l  h'100          ; should be 4600 - 49FF
       res.q  h'100          ; should be 4A00 - 51FF
```

Remarks: Values used in reserve statements must not contain any forward references.

Assembly Language Subroutines

The following locations are the entry points into fixed assembly language routines.

ATOD12	**FD9A.** Get reading from 12-bit A-D converter. Enter with one of the channel numbers (0 - 13) in B register. Channels 0 - 10 specify one of the external A-D input channels. Channels 11, 12 or 13 specify one of the reference channels. Return with MSB value in A register, LSB value in B register with value shifted left four bits.
PDAD12	**FD9D.** Power-down 12-bit A-D converter.
ATOD8	**FDA0.** Get reading from 8-bit A-D converter internal to 68HC11. Enter with one of the channel numbers (0 - 7) in B register. Return with value in A reg, zero in B register.
GETBSR	**FDA3.** Read byte from Bank-switched RAM. Offset into bank in X register, values from 0 to 60927 valid. Byte returned in A register
STRBSR	**FDA6.** Store byte in Bank-switched RAM. Byte to store in A register. Offset into bank in X register, values from 0 to 60927 valid.
STRMEM	**FDA9.** Store byte in A register to next location in datafile.
CHIBUF	**FDAC.** Return next byte in UART input buffer without removing it. Byte returned in A register.
FLUSHI	**FDAF.** Flush UART input buffer.
FLUSHO	**FDB2.** Flush UART output buffer.
OUTTST	**FDB5.** Check for characters in UART output buffer. Number of characters remaining is returned in the A register.
URTGET	**FDB8.** Receive next byte from UART buffer and return in A register. Returns the number of characters remaining in the buffer in the B register. Waits for a byte before returning.
URTSND	**FDBB.** Send byte in A register out UART (via buffer).

URTTST

FDBE. Return number of characters available in UART input buffer in A register.

STPWCH

FD97. Stopwatch functions. Values returned are four-bytes, with MSBs in register X and LSBs in register D. Units are in ticks and will depend on the RATE command. Call with a value of 0-4 in the B register for one of five functions:.

B reg	Function
0	Initialize. Chains the stopwatch interrupt to the system clock and zeroes the clock. Must be called before other functions are used.
1	Remove stopwatch clock. Calling this saves the extra time stopwatch adds to the system clock handler.
2	Clear clock. Clears the stopwatch to zero. Stopwatch keeps running.
3	Get clock. Returns current stopwatch value then clears stopwatch to zero. Stopwatch keeps running.
4	Split clock. Returns current stopwatch value but DOES NOT clear stopwatch value. Stopwatch keeps running

Example showing how RATE interacts with Stopwatch function

```
start: input "RATE value "r
       if r > 32
       goto start           // limit RATE to 32 for now
       endif

       rate r               // set to change timer resolution
       secPerTick! = 1.0 / (100. * r)
       print "Timer resolution = ",#.4f,secPerTick," seconds"

       input "Hit key to start"dummy,#1

       call &hfd97,0        // initialize and zero stop watch
       input "Hit key to stop"dummy,#1

       call &hfd97,3,ticks  // get current stop watch time
       call &hfd97,1  // remove stop watch from system clock
       print #.4f,ticks*secPerTick," seconds"
       goto   start
```

Important Addresses in TFBASIC

These system variables may be accessed in your assembly routines or with PEEK and POKE. Be warned - be sure you know what you are doing or you may have unintended side effects.

time variables	addresses	notes
? variable	60H - 63H	seconds since 1/1/80
? tick	64H	0.01 seconds
? array - ?(0)	65H - 68H	second
? array - ?(1)	69H - 6CH	minute
? array - ?(2)	6DH - 70H	hour
? array - ?(3)	71H - 74H	day
? array - ?(4)	75H - 78H	month
? array - ?(5)	79H - 7CH	year
? array - ?(6)	7DH - 80H	0.01 ticks

other variables

Each variable must be accessed by the address given, not by its name.

· CCOUNT	9BH	Count of CTRL-Cs pending (See CBREAK)
· CENAB	9CH	CTRL-C enable (zero = ignore CTRL-C)
· TRACE	9DH	When set to non-zero to trace execution
· OVRSLP	9EH	Flag that signals SLEEP detected oversleep
· BRKCNT	9FH	Count of RS232 break conditions since last call to FLUSHI or power on reset.
· LNCHSTAT	A0H	Bits are set to indicate certain errors when program is launched accoring to the following:

Bit 0	Set if there was a checksum error of code stored in SFLASH.
Bit 1	Set if there was an error getting clock time from PIC. In this case, the clock will be set to zero (midnight January 1, 1980)
Bit 2	Set if there was an error getting the datafile pointer (DF pointer) from the PIC. In this case, the datafile pointer will be set to zero.
Bit 3	Set if there was an error accessing the SFLASH

Interrupt Vector Table

Overview To allow the interrupt vectors to be user modifiable the interrupt vectors in EEPROM all point to jump vectors that reside in the HC11 RAM. Each RAM vector is made up of a JMP instruction followed by a two-byte address. To change the vector you must replace the RAM address (in most cases it points to an RTI) with the address of your own handler.

Addr	Mnemonic	Explanation	Used by TFBASIC
FD C1	SCI	Serial Communications Interface	UART routines
FDC4	SPI	Serial Peripheral Interface	
FDC7	PAIE	Pulse Accumulator Input edge	
FDCA	PAO	Pulse Accumulator Overflow	
FDCD	TO	Timer Overflow	PERIOD command
FDD0	IC5/OC4	Timer IC4 OC5	TONE command
FDD3	TOC4	Timer Output Compare 4	system clock
FDD6	TOC3	Timer Output Compare 3	UGET command
FDD9	TOC2	Timer Output Compare 2	
FDDC	TOC1	Timer Output Compare 1	
FDDF	TIC3	Timer Input Compare 3	COUNT/PERIOD commands
FDE2	TIC2	Timer Input Compare 2	
FDE5	TIC1	Timer Input Compare 1	
FDE8	RTI	Real-time Interrupt	(alt system clock)
FDEB	IRQ	External Interrupt Request	
FDEE	XIRQ	Extra External Interrupt Request	HYB routine
FDF1	SWI	Software Interrupt	
FDF4	IOT	Illegal Opcode Trap	
FDF7	COPF	“COP” Failure	
FDFA	CMF	Clock Monitor Failure	
FDFD	RESET	RESET used by Bootloader	

CHAPTER 6

TFBASIC Internals

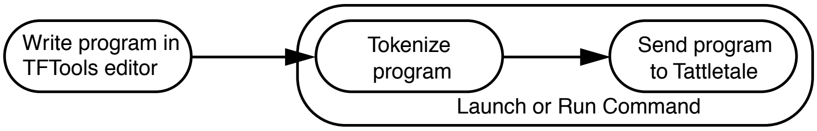
TFBASIC Structure

Overview

TFBASIC adds a tokenizing pass between editing the program and sending it to the Tattletale. This pass is transparent in that it is part of the routine that sends the program to the Tattletale. It's fast too, especially when using the parallel port cable.

The Host program : Editor / Tokenizer / program loader

Before a program is sent to the Tattletale it is tokenized by the host computer, an IBM PC (or compatible), running TFTools. The tokenizer in TFTools reads each command line and splits it into discrete, basic operations, and converts all expressions to their Reverse Polish equivalents. Each operation is defined by a token (or label) and a set of parameters. When tokenized, the program is sent to the Tattletale where it can be run.



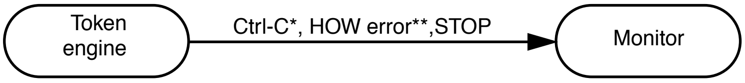
If a syntax error is found during tokenizing the program is not loaded and the offending line is flagged in the editor so that the error can be corrected.

Details of the Tattletale Program

The TFBASIC program runs on the Tattletale and has two distinct parts: the program's tokens, which are generated from your source code, and the token interpreter engine, which executes the tokens. The Tattletale's interpreter engine includes a monitor that is active when the user's program is not running and is identified by the # prompt. The interpreter and monitor are loaded along with the program tokens and makes up the program code that is downloaded to the TFX 's Serial FLASH (SFLASH).

Token Engine

The Token Engine is an interpreter that executes the TFBASIC program tokens, jumping to the Monitor on an error (unless redirected with the ONERR command), receiving a CTRL-C (unless redirected by the CBREAK command), or after executing a 'STOP' or 'HALT'.



* only if not redirected by CBREAK
** only if not redirected by ONERR

Monitor

The monitor has a number of functions that respond to control characters:

CTRL-R. Start the interpreter on the program already in RAM

CTRL-L. Load a tokenized program into RAM

CTRL-E. Erase the datafile

CTRL-O. Start XMODEM off-load of the datafile

CTRL-H. Load and execute a HEX file.

These characters may be entered when the terminal window is active.

On Launch or Power-up RESET

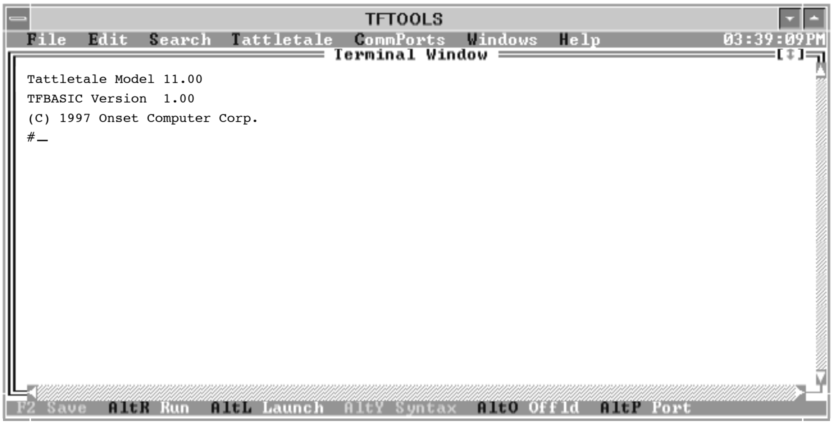
On launch the Tattletale always copies the operating system and program (if present) from SFLASH to RAM. It then checks to see if there is a valid program in RAM by verifying the checksum and performing other checks as described below. Bits are set in location A0H to report the following errors:

- Bit 0 will be set if there was a checksum error (of code stored in the SFLASH).
- Bit 1 will be set if there was an error getting clock (from PIC). In this case, the clock will be set to zero (January 1, 1980 at midnight).
- Bit 2 will be set if there was an error getting DF pointer (from PIC). In this case, the datafile pointer will be set to zero.
- Bit 3 will be set if there was an error accessing the SFLASH.

NOTE: If all is OK the value in this location will be 0.

TFTools : TFBASIC development on a PC

To develop TFBASIC applications in a PC environment you will need the TFTools Integrated Development Environment (IDE). The IDE provides the tools to edit, load, and run programs as well as off-load the stored data and view activity on the serial port.


TFTOOLS
File Edit Search Tattletale CommPorts Windows Help 03:39:09PM
Terminal Window
F2 Save AltR Run AltL Launch AltY Syntax AltO Offld AltP Port

```
Tattletale Model 11.00
TFBASIC Version  1.00
(C) 1997 Onset Computer Corp.
#_
```

TFBASIC Power-up Program Launch

When you first fired up your Tattletale, it printed a sign-on message :

```
Tattletale Model 11.00
TFBASIC Version 1.00
(C) 1997 Onset Computer Corp.
#
```

The Tattletale is running a program written in TFBASIC that was stored in SFLASH at the time of manufacture. This program ends with a STOP command so that the prompt will appear. This section shows you how to make your program launch on power-up instead of the one that prints the message shown above.

CTRL-C

TFBASIC programs, no matter how launched, will break with a CTRL-C unless a CBREAK command has been used to specify a line number to restart to when a CTRL-C is encountered. You can disable CTRL-C breaks by writing a zero byte to address 9C hex. A count of CTRL-C characters will continue to be updated at address 9B hex. Clear this before you re-enable break-outs. See CBREAK.

Power-up for TFBASIC

On power-up TFBASIC copies the program and operating system to RAM and launches that program. Remember that the program will always break out upon finding a CTRL-C unless a CBREAK command has been used to specify a label to vector to when a CTRL-C is encountered.

Making your program permanent

Using the LAUNCH command will erase the contents of the datafile, reset the datafile pointer to 0. It then replaces previously stored program (if any) with a fresh copy of the TFBASIC token interpreter engine and your compiled program tokens.

TFBASIC Integers

Internal representation

Integers are represented as a signed sequence of 32 bits, arranged as four bytes. The format is as follows :

MSB	byte 2	byte 3	LSB
SBBBBBBB	BBBBBBBB	BBBBBBBB	BBBBBBBB

where S is the sign bit and B represents the remaining 31 individual bits in descending order.

The range of Tattletale integers is 2^{32} which, with the sign bit, gives a range of –2147483648 to 2147483647. Any attempt to create a number outside this range will result in a run time error.

Assigning A/D conversion values to a TFBASIC variable.

The A/D converters return two bytes, left justified so that all conversions, no matter the resolution, span a range of 0-65520 for the 12-bit A/D and 0-65280 for the eight bit. The TFX supports only unipolar operation of the A/D so the assignment of the type:

```
A_D_Value = CHAN(12)
```

will always produce the correct value and sign.

Big-endian vs. Little-endian

In the early days of microprocessors Intel and Motorola independently devised equal but opposite storage sequences for the order of significance in multi-byte data items. In Intel’s little-endian world the Least Significant Byte (LSB) is stored first in the lowest address, and the Most Significant Byte (MSB) in the highest address. Motorola’s order, big-endian, is just the reverse. The Motorola processor in the TFX-11 uses the Motorola sequence, but when the data is off-loaded directly from the datafile to an Intel based PC the order for multiple-byte data items is now reversed. If you try to access a four byte integer stored as binary from a BASIC or C program this byte reversal will cause incorrect results. This reversal has been a cause of much confusion, partly because it always works correctly for single bytes.

TABLE 1.

memory address	big-endian TFX	little-endian PC
A+0	MSB	LSB
A+1	...	...
A+2	...	...
A+3	LSB	MSB

NOTE: The terms little-endian and big-endian come from Gulliver's Travels., where Jonathan Swift imagined a never-ending war between two Kingdoms; one, the Big-Endians, who crack open their hard boiled-eggs on the big end, and the other, the Little-Endians, who crack open their hard boiled-eggs on the opposite end.

Byte sequence and VARPTR(x)

Since integers in TFBASIC are stored MSB first the address returned by VARPTR(x) returns the MSB of the integer. The LSB of the integer is at the address VARPTR(x)+3. In the following example it is assumed you want to only return the LSB of the variable, and the other three bytes can be safely ignored.

TABLE 2.

memory address		
A+0	MSB	(VARPTR(x) points here)
A+1	...	
A+2	...	
A+3	LSB	

Example:

```
X= 100        // put a number (<255) in the LSB of X
y = varptr(X)
print "Value of X = ", peek(y)  // pointing to MSB
```

Output:

```
Value of X = 0
```

This is obviously not the correct answer. The +3 added to y in the peek function of the following code corrects the problem:

Example:

```
X= 100
y = varptr(X)
print “Value of X = “, peek(y+3)  // now pointing to LSB
```

Output:

```
Value of X = 100
```

TFBASIC Floating Point

The floating point format used is based on the IEEE 754 single precision floating point standard. The range of representable values is ±1.175494E-38 to ±3.402823E+38 and zero. Also, ±infinity and not-a-number (NaN) are represented.

Internal representation

Single precision floating point numbers are represented in four bytes. The format is as follows :

MSB	byte 2	byte 3	LSB
SEEEEEEE	MMMMMMMM	MMMMMMMM	MMMMMMMM

S = sign bit; 0 = positive, 1 = negative

E = binary exponent with bias of 128

M = mantissa with 23 explicit bits and an implied 1 bit as the most significant bit

In the Tattletales, the most significant byte is at the lowest address of the four bytes. This is important if binary data is transferred directly to a computer that interprets numbers in the opposite order (as all computers with Intel microprocessors do). SEE INTEGERS ABOVE for further explanation.

Variables

When using a floating point variable for the first time, append an exclamation point (!) after the last character of the name. It allows the tokenizer to check that the correct types are being used for arguments. It also allows the tokenizer to automatically convert data to the correct representation.

Example :

```
sineValue! = sin( 90.0 )
sineSquared! = sineValue! * sineValue!
dim sineList!(100)
```

Although the ! type identifier only needs to be used the first time a variable is declared, for clarity it is recommended that it also be used with every instance of the same variable. Notice how the variable was declared as a floating point array. This MUST be done in the DIM statement because this is the first place an array is used.

If an integer value is assigned to a floating point variable, the value will automatically be converted to float before the assignment.

Constants Floating point numbers can be entered by following this rule. The entered number must contain at least one digit with : 1) a decimal point and/or 2) the power-of-ten operator E followed by the power of ten value.

Examples :

```
        250., 2.5E2, 2500.0E-1 and 25E1
```

are all valid representations of 250.0

```
        250, 2.5E2.0, E5 and .E5
```

are all INVALID floating point constants

If a floating point constant is assigned to an integer variable, the constant will first be converted to integer (using the FIX function) and then assigned to the variable.

Arithmetic Operators All but one of the normal TFBASIC arithmetic operators are available for floating point math. The exception is the modulo operator '%'. The tokenizer checks the two operands of each arithmetic operation. If one is a float and the other an integer, the integer is first converted to float and then the operation is performed. The result is considered as type float in the rest of the expression. For example, in this equation :

$$\text{floatValue!} = \underbrace{\overbrace{5 * 2}^{\text{Integer}} + \overbrace{3 * 7.9}^{\text{Float}}}_{\text{Float}}$$

5 and 2 are multiplied as integers (and saved temporarily). Then 3 is converted to float and then multiplied (as floating point) by 7.9. Then the result of 5 * 2 is converted to float and this is added to the (already float) result of 3 * 7.9 and stored as a float in floatValue. If instead we had assigned the result to an integer variable, the only change would be that the final result would be converted to integer just before the assignment; the math operations would not have changed.

Relational operators All TFBASIC relational operators are available for floating point comparisons. The precedence is exactly the same, too. As with the arithmetic operators, the tokenizer checks the two operands of each compare operation. If one is a float and the other an integer, the integer is first converted to float and then the comparison is performed. Unlike math operations, relational operations do not carry the result (or type) of comparisons along in complex expressions. The result of each comparison can only be TRUE or FALSE. For example, in this equation :

$$\text{if } \underbrace{\overbrace{2.3 > 2}^{\text{Float}} \mathbin{\&} \overbrace{18 < 20}^{\text{Integer}}}_{\text{Bit-wise AND}} \text{ print "TRUE"}$$

2 is converted to a float and then compared with 2.3 and the result, TRUE, is saved. Then 18 is compared with 20 (as integers) and this result is TRUE also. Finally, the two TRUE results are ANDed together (the & operator). This is TRUE and the 'print' will be executed. Beware of checking for equality in an IF statement using floating point operands. It's possible that the numbers may appear to be equal (using print commands) but may not truly be equal. You should always test using greater than or equal or less than or equal.

Conversions

There are two kinds of conversions : implicit and explicit. We have mentioned implicit conversions in the previous discussions. If two operands are of different types and one is converted to the other's type, this is an implicit conversion. TFBASIC only makes implicit conversions from integer to float. Explicit conversions are accomplished with the FLOAT, INT and FIX functions. You must watch out for overflow and loss of precision when making explicit conversions. Overflow occurs when a floating point number with a large exponent is converted to integer. For instance, 1.0E15 cannot be converted to integer because the maximum integer is approximately 2.1E9. This is considered an error by TFBASIC. Loss of precision occurs when an integer with too many digits is converted to floating point. Integers in TFBASIC have 31 bits of precision (plus a sign bit) while floating point numbers only have a precision of 24 bits (one bit is an implied 1). TFBASIC does not consider this an error. It just rounds the floating point value to the nearest representation of the integer that it can.

There is an ambiguous situation. This is when floating point values are stored in binary in the SFLASH datafile or in the UEEPROM. Anything stored there is assumed to be integer. In this case you must use the ASFLT function to tell TFBASIC to interpret the data as floating point. See ASFLT in the "TFBASIC Command Details" section.

Floating Point Functions

Here is a list of the functions available for floating point operations. The functions that take a floating point argument will automatically convert an integer to float before executing the function. Functions that take an integer argument will signal an error if passed a floating point argument (except for ABS which can take either type of argument)

TABLE 3.

Name	Description	Argument type	Return type
abs	absolute value	float or int	float or int
asflt	interpret value as float (no conversion made)	integer	float
atn	arctangent (in degrees)	float	float
cos	cosine of angle in degrees	float	float
exp	raise e to power	float	float
fix	float to integer closer to zero	float	integer
float	integer to float	integer	float
int	float to integer less than argument	float	integer
log	natural logarithm	float	float
log10	common logarithm	float	float
sin	sine of angle in degrees	float	float
sqr	square root	float	float
tan	tangent of angle in degrees	float	float

Print / Store formats

There are two floating point formats for printing or storing data as characters. Fixed point notation uses only digits and the decimal point. Scientific notation uses a mantissa (which is really fixed point) followed by an exponent (in powers of ten). The format statement in PRINT and STORE commands for fixed point is:

```
#w.dF
```

where w is the minimum field width of the number (including the sign, integer part, decimal point and fractional part) and d is the number of decimal places in the fractional part of the number. For instance, the number -15.302 would be specified as #7.3F. The format for scientific notation is very similar to fixed point:

```
#w.dS
```

but here, the minimum field width, w, must also include the exponent specifier. For instance, 2.1345E-3 would be specified as #9.4S. The w specifier is optional (it defaults to a minimum field width of zero) and the d specifier is optional (it defaults to six decimals places) but if d is specified, it MUST be preceded by the decimal point. Here are some examples of valid formats and what they produce :

Format for v	v = 12.345	v = -0.987654
#10.2F	12.34	-0.99
#7.2S	1.23E1	-9.88E-1
#6.3F	12.345	-0.988
#.5F	12.34500	-0.98765
#0F	12.345000	-0.987654
#0S	1.234500E1	-9.876540E-1

Floating Point Errors

Errors in floating point operations do not stop program execution. Instead, a bit is set in the read-only variable FPERR. These bits remain set until you access FPERR by copying it to a variable, printing it or storing it. Then FPERR is cleared to zero. If you perform a number of floating point operations that produce different errors, FPERR will have more than one bit set when you finally check its value. The drawing to the right shows which bit is set for each of the four possible floating point errors.

	Cannot compare	Loss of precision	Not-a-Number	Overflow error	Underflow error
Bits 31 - 5 not used	4	3	2	1	0

FPERR variable

When the value of FPERR is printed, each bit takes on the following values : bit 0 = 1, bit 1 = 2, bit 2 = 4 and bit 3 = 8. The values are then added up to produce the final value (just as in any binary number). For instance, if Underflow, Not-a-Number and Loss of Precision errors occur before FPERR is printed, the value 13 (1 + 4 + 8) will be printed.

Here is the explanation for each error :

Underflow. - Bit 0 (weight = 1) is set if a number between +1.175494E-38 and zero results or a number between -1.175494E-38 and zero results. This is not representable in single precision. The result is set to zero.

Example :

```
floatValue! = 3.0E-28 * 3.1E-15          produce the error
print "result = ", #F, floatValue        print the result
print "error = ", FPERR                  print FPERR value
```

Printed :

```
result = 0.000000                value of floatValue
error = 1                        value of FPERR
```

Overflow. - Bit 1 (weight = 2) is set if a number greater than +3.402823E+38 or less than -3.402823E+38 resulted. This is not representable in single precision. The result is set to +Infinity or -Infinity.

Example :

```
floatValue! = -2.0E30 * 1.0E20           produce the error
print "result = ", #F, floatValue        print the result
print "error = ", FPERR                  print FPERR value
```

Printed :

```
result = -INF                    value of floatValue
error = 2                        value of FPERR
```

Not-a-Number. - Bit 2 (weight = 4) is set when the floating point routine has no idea of how to represent the result. One way to get this is to take the square root of a negative number which is an imaginary number. The result is flagged as a 'not-a-number'.

Example :

```
floatValue! = sqr(-4.0)                  produce the error
print "result = ", #F, floatValue        print the result
print "error = ", FPERR                  value of FPERR
```

Printed :

```
result = NaN                     value of floatValue
```

```
error = 4                                   value of FPERR
```

Loss of Precision. - Bit 3 (weight = 8) is set when an integer greater than 16777215 or less than -16777215 is converted to floating point. This is because it takes more than 24 bits to represent such a number and single precision floating point has only 24 bits to represent the precision of a number. TFBASIC will convert the value to the nearest floating point equivalent.

Example :

```
floatValue! = float(17789321)               produce the error
print "result = ", #F, floatValue           print the result
print "error = ", FPERR                     print out FPERR
```

Printed :

```
result = 17789319.000000            value of floatValue
error = 8                           value of FPERR
```

Could not Compare. This error occurs when you attempt to compare any floating point value with NaN or if you try to compare +INF with itself or -INF with itself. NOTE: You can compare +INF with -INF: + Infinity is greater than - Infinity

Characters and Strings in TFBASIC

Overview

String variables are denoted by adding a $ suffix to the name of the variable. When a string variable is declared, 256 bytes are reserved for it. The first byte is the length of the string so a string cannot contain more than 255 characters. 256 bytes are always reserved for the string whether used or not. Like the integer and floating point variables, TFBASIC does not initialize its string variables. If the programmer does not initialize each string variable, it may contain from 0 to 255 characters of any type when it is accessed in the program.

String constants must be enclosed in double quotes. This is different from earlier versions where single or double quotes could be used. String constants can be up to 255 characters in length (although the editor only handles lines 256 characters long). String constants can be used as arguments to commands and functions and can be assigned to string variables.

String Functions in TFBASIC

String functions can be used just as any other function. All other functions can be used as a variable or argument in another expression. Of course, functions returning a string can only be used where a string is expected. Functions returning an integer can only be used when an integer is expected.

Character Constants

You can now use the character 'A' as a synonym for &H41. There can be up to four characters enclosed in single quotes. The type of the character constant is a four-byte integer. Characters are shifted into the least significant byte of the integer as they are read with unused bytes set to zero. For instance, 'A' is a character constant equivalent to value 41 H. 'AB' is equivalent to 4142 H. 'ABC' is equivalent to 414243 H. 'ABCD' is equivalent to 41424344 H. Character constants can be used anywhere an integer value is used. These are especially useful when checking if a character from the UART is equal to one or more characters. For instance, if you got a character from the UART in variable ch:

```
if ch = 'Y' | ch = 'y'
  print "Yes"
else
 if ch = 'N' | ch = 'n'
  print "No"
 endif
endif
```

Special characters can be embedded in a character constant by preceding a pattern with the escape character '\'. See the full explanation under "Other String Operations" that follows.

Other String Operations

String Input. You may input a string to a variable using the INPUT command as in Input "Enter: " astring$.

String output/storing to datafile. PRINT and STORE. Special instructions for STORE: the string is stored with the length byte first followed by the string.

String assignment . String assignment from other string variables and from string constants is performed like any other assignment as in astring$ = "Test Line".

String concatenation . String concatenation is performed using the '+' operator.

Example:

```
astring$ = “abc” + “defg” + “hij”
print  astring$
```

output:

```
“abcdefghij”
```

Embedding Special characters

Special characters can be embedded within a string constant or a character constant by including an escape sequence. Each escape sequence determines the code value for a single character. Escape sequences are used to represent characters that cannot be otherwise represented.

There are two forms of escape sequences, numeric and mnemonic. Both forms start with a backslash. Numeric escape sequences allow you to represent a single character using a numeric code. Mnemonic escape sequences represent the character code with a single character related to its function, such as a \t to represent the TAB character code .

Mnemonic and numeric forms may be used interchangeably. The mnemonic form for a the tab character is "\t" and the numeric equivalent is "\x09". Both these sequences generate a single byte code of 09h .The surrounding double quotes are necessary as all escape sequences are strings.

Example To embed a double quote:

TFBASIC code:

```
print “He said,\”I’m not sure\”.”
```

Output:

```
He said,”I’m not sure”.
```

The following code is equivalent to the example code above:

```
print “He said,\x22I’m not sure\x22.”
```

and will give the same output, although the code it is far less obvious in conveying what the programmer was up to. Note also there is no space after the digits in the second example. The escape sequence terminates with the last character in the sequence. A space included after the sequence will show up as a space in the output.

mnemonic escape sequences list Here is a list of mnemonic escape sequences, the ASCII characters they represent, and their hex character code:

```
\a     BEL    alarm, the bell character          07H
\b     BS     backspace character                08H
\f     FF     formfeed character                 0CH
\n     NL     newline, the line feed character   0AH
\r     CR     carriage return character          0DH
\t     HT     horizontal tab character           09H
\v     VT     vertical tab character             0BH
\”            the double quote character         22H
\’            the single quote character         27H
\\            the backslash character            5CH
```

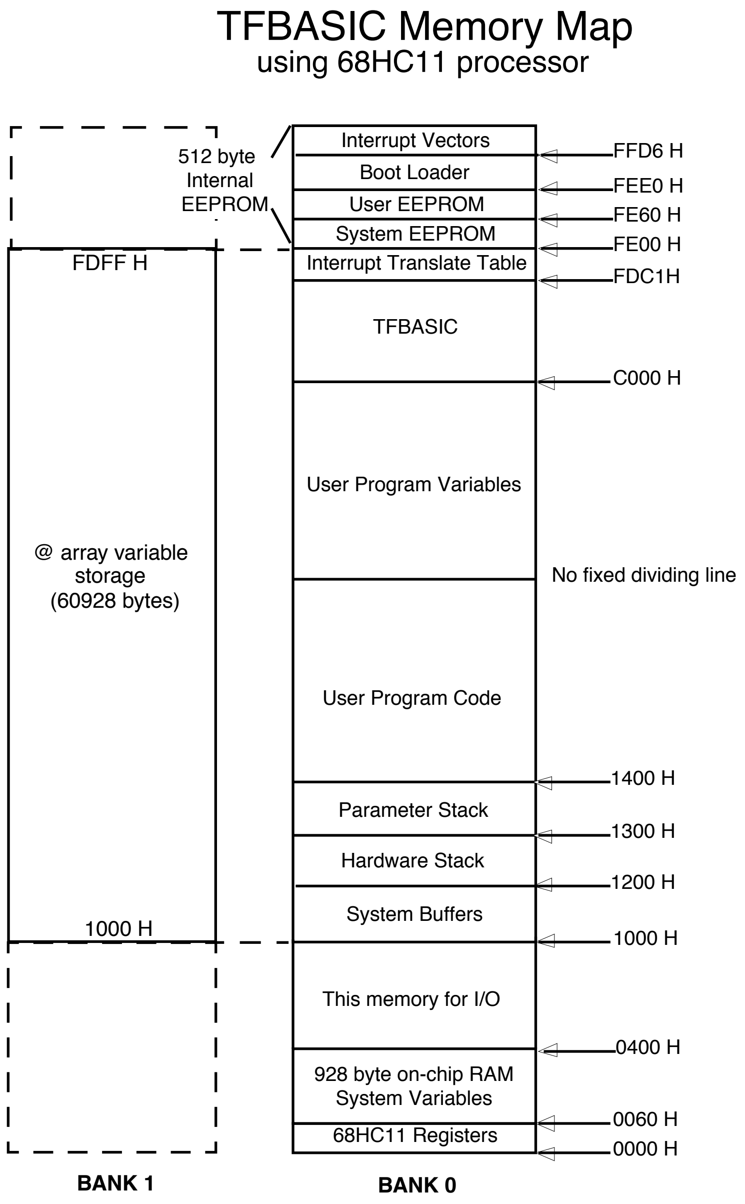
numeric escape sequences list

```
\xhh   where hh is the hex value of the character to
       embed  - BOTH CHARACTERS MUST BE PRESENT!

NOTE:  Octal is not supported.
```

TFBASIC Memory Map

Mapping of RAM memory for the 64K program space (BANK 0) and variable storage space (BANK 1) in the 128K RAM



RAM Memory Overview

The TFX-11 has 128K of RAM which is split into two 64K banks. Bank 0 contains the executing code as well as the HC11 internal memory, registers and EEPROM, which includes the UEEPROM. Bank 1 is the upper 64K of RAM and is devoted to data storage only. Access to this storage requires using the @ array. Because of hardware addressing schemes in the HC11 not all of the BANK 1 64K bytes of RAM is available. The area that is available starts at 1000h and ends at FDFFh, or 60928 bytes. Access to this area requires bank switching which slows operation slightly. The user need not be concerned with bank switching or addressing since TFBASIC takes care of all that when the user makes an assignment statement. The data in memory is referenced by the array's index value, which is zero based. Each @ array variable is four bytes, the typical TFBASIC variable size. There are 15321 4-byte variables, thus the indices range from 0-15320.

Memory map details by address

0000h-0060h

HC11 Hardware Registers. These are the registers of the processor. These may be read and written directly from an assembly routine embedded in your TFBASIC program. The register list is available in TFTools help. For detail information on these registers and what they control consult the Motorola MC68HC11F1 Technical Data Book and the Motorola MC68HC11 Reference Manual.

0060h-03FFh

System Variables. This is the HC11's on board RAM. It is used by TFBASIC to keep track of its operations. Your code should neither read nor write within this area. Reading, while generally a benign activity, will not give you any useful information and future revisions of the software may modify this area so any code written to depend on values read from this area is risky at best. Writing to this area is a sure recipe for disaster.

0400h-0FFFh

I/O memory. Can be mapped for CSIO1 and CSIO2 chip select addressing. Refer to the Motorola MC68HC11F1 technical manual for details. If not used with chip selects it is available as a general purpose RAM.

1000h-11FFh

System Buffers. This area contains the UART buffers and other similar TFBASIC structures. Keep out.

1200h-12FFh	**Hardware Stack.** Do not touch.
1300h-13FFh	**Parameter Stack.** Keep out.
1400h - BFFFh	**User Program code and Variable area.** The lowest part of this area contains the TFBASIC tokens and other code to be executed by the TFBASIC token engine. It grows up from the low address. The User variables grow downward from the location BFFFh.
C000h-FDC0h	**TFBASIC.** This is dedicated to the TFBASIC token engine.
FDC1h-FCFFh	**Interrupt translate table.** The interrupts vectors in the HC11's EEPROM point here. Since this area is located in RAM, these vectors may be modified for those who might like to write there own ISRs. Not for the beginner or the faint of heart.
FE00h-FE5Fh	**EEPROM System Info Area.** This area contains configuration information stored at time of manufacture for use by ONSET and TFBASIC.
FE60h-FEDFh	**EEPROM User Area.** (SEEPROM) This area contains128 bytes accessed as an array by the VSTORE and VGET commands. They have been set aside for user parameter storage. The are read and written as four byte variables indexed from 0 -31.
FEE0h-FFD5h	**Bootloader.** This is the code that loads the program from flash to RAM and starts it executing.
FFD7h-FFFh	**Interrupt Vectors.** These are the primary interrupt vectors which point to the interrupt table in RAM.

What's New and Different in TFBASIC

? variable The ? variable now keeps time in seconds. You can use this value with no worry of wraparound after 200 days. If you need to time events to resolutions of 0.01 seconds, use the new stop-watch functions listed in the Assembly Language Routines section of the manual.

Storing to the datafile The STORE command no longer uses a variable to point into the datafile. All datafile storage is sequential with the datafile pointer being kept internally. The datafile pointer is zeroed at launch and can only increment. Data written to the datafile cannot be read back by the program.

GETBYT The assembly language routine GETBYT that gets a byte from the UART input buffer now does not block and also returns count (in B reg) of characters in the buffer.

IF and IFF IF and IFF have changed. The old IF syntax no longer works. We now use the IFF syntax exclusively - BUT - we use the spelling of IF.

Multiple commands The ':' operator which allowed you to put multiple commands on a single line is no longer available. You can now have only one command on a line.

Removed commands The following commands have been removed from TFBASIC:

```
ADLOOP  BURST   DTOA    GET     GETS
ITEXT   LEFT    OFFLD   OTEXT   PEEKW   PEEKL   POKEW
POKEL   REM     RUN     RIGHT   XSHAKE
```

String functions The STR command is much more flexible under TFBASIC. You can create a string with all the same options available to the PRINT command. But now you have a string variable that can be printed, stored, split into sub-strings, appended to other strings or compared with other strings.

When strings are stored, they are stored with the length of the string first followed by the characters.

PRINT {x,y} removed The PRINT command no longer supports the {x,y} format because TFBASIC does not allow reading from the datafile within a program.

VAL function expanded

The VAL function has been split into IVAL (for integers) and FVAL (for floating point numbers). This allows these functions to be used anywhere. VAL could only be used where it was being asigned to a variable so the compiler could decide if an integer or floating point value was to be output.

No early wakeup from SLEEP

The form of SLEEP with early wake-up (if D1 goes high) has been removed.

ITEXT removed

The INPUT command covers most of the functionality of ITEXT but is more flexible because it stores to a variable.

Datafile pointer variable replaced

with Notice the addition of the DFPNT read-only variable. This was necessary because the programmer doesn’t have control of the datafile pointer as before. The datafile pointer is automatically set to zero when a new program is loaded and is only changed by the STORE command. This makes accidental overwriting of data impossible - unless the user loads a new program without off-loading the data first! TFTools gives a warning if you attempt to do this.

Setting the BAUD rate

There are two new commands to read (GETBAUD) and change (SETBAUD) the baud rate of the main UART. The following baud rates are available: 300, 600, 1200, 2400, 4800, 9600, 19200 and 38400.

Entering comments in the code

The only method of entering comments in a TFBASIC program is with the // operator. You can no longer use the ' at the beginning of a line to make the rest of the line a comment. Neither can you use the REM command to enter comments. The semicolon, ;, is still the only way to enter comments in assembly language code in TFBASIC.

PIN command

The PIN command returns weighted values for the various digital I/O pins when they are high.

PCLR, PSET and PTOG limitations

The PCLR, PSET and PTOG commands only work with digital I/O pins 0-7 and digital I/O pins 16-23. Digital I/O pins 8-15 are input-only pins and work only with the PIN command.

HYB command acts differently

The HYB command does not awake once every 10 seconds to check for an RS-232 BREAK condition at the input to the main UART. You can use the PIC Interrupt or the IRQ line to awake from HYB early.

Timekeeping RTIME and STIME do not access the hardware real-time clock. They make conversions between the TFBASIC software clock clock (the ? variable) and the ? array. To access the RTC in the PIC use SETRTC and READRTC.

UGET/USEND UGET and USEND no longer use the datafile for storage. Instead thay use a string variable. In addition, USEND can now use a string constant or a string variable. The UGET and USEND baud rate limit has been increased to 19200.

CHAPTER 7

TFX-11 Interfacing

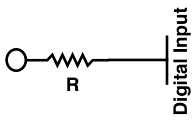
Interfacing to Real World Signals

The Tattletale as delivered is extremely versatile and has many capabilities, but it is far from a complete instrument. This is where you come in. In order to make the Tattletale into the device you envision, you will most likely need to connect external sensors and devices to its different I/O ports. Understanding of the capabilities and limitations of the different I/O options available is critical to completing an instrument or device that works reliably and repeatably.

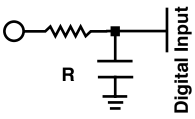
The following sections review general issues associated with interfacing your Tattletale to the outside world. We hope this will give you some background and ideas that will help you make the best choices for your design. Admittedly this sections is brief - for more detail we recommend you read some of the books listed as references, especially the Motorola HC11 manuals and The Art of Electronics by Horowitz and Hill.

Digital Input Protection

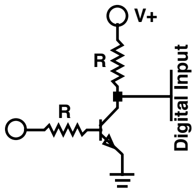
Digital inputs must be protected from signals that exceed the Tattletale's internal bus supply levels. The diagrams below show five techniques that protect the inputs from large current spikes which may cause latch-up.



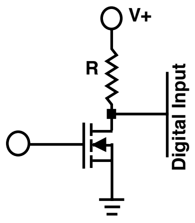
Resistor. The resistor protects the input from surges by limiting the amount of current that can be injected. A typical value might be 100K.



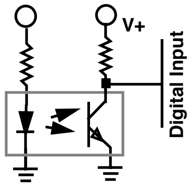
R - C. Adding a capacitor helps protect inputs from high voltage spikes caused by electrostatic discharge. The combination of resistor and capacitor here forms a low-pass filter which changes the response to high speed signals, which include the voltage spikes.



Transistor. This solution isolates the Tattletale's inputs completely from the source, placing the transistor at hazard (the transistor is a lot easier and cheaper to replace than the processor that the inputs are connected to). This design can also perform some level shifting.



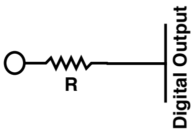
VFET. Same as the transistor, but takes much less current. VFETS are generally harder to obtain, cost a little more, and are more easily damaged by static, but are the best solution for low power drain and minimal loading of the signal source.



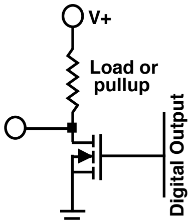
Opto-isolation. This solution has the advantage of total isolation of both the supply and the ground of the source from that of the microprocessor. However, it comes at the expense of substantial current drain, size, and expense.

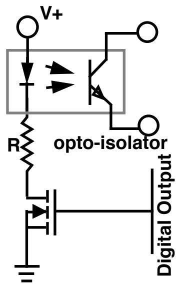
Digital Output Protection

Digital outputs are just as vulnerable as inputs, and they cannot be driven, even transiently, with a signal larger than the 'V' supply or lower than ground. They also have relatively low drive capability and should be buffered if they are expected to drive substantial loads. Some examples of output buffering are given in the figure below.

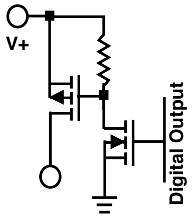


100K resistor. The resistor in this first example will provide protection but very little drive current. It is suitable for connecting to CMOS inputs driven from a separate supply (where there is a possibility that the supplies won't track).





VFET driver. The VN0104 is an N-channel DMOS transistor with an on-resistance of about 4 ohms when the gate is +5 volts and lots of megohms when the gate is grounded. In the configuration shown, it is suitable for driving small relays or opto-isolators. The VN0104 has a Vds of 40 volts, so the solenoid can be powered directly from Vbat or any other convenient source.



Power switch. This last example shows how to configure two VFETs (one P-channel and one N-channel) to, for example, turn on and off a separate voltage to switch Vbat to a separate regulator.

Using the Onboard A/D Converters

Signal Conditioning of Analog Inputs

The analog inputs are designed to handle signals that range from 0 to the converter's Vcc , typically +5V. This full range ratiometric conversion is ideal for potentiometer inputs with the wiper attached to the input and the ends tied to Vsw and ground. Other sensors, such as strain gauges, may need amplification before they can be attached to the converter input.

Ratiometric A/Ds

The TFX-11 has its negative reference tied to ground and its positive reference input tied to the converter's +5V positive supply. This means that if you are measuring your sensor as a fraction of the reference input (a potentiometer, or a bridge), your conversion will give solid, repeatable results. Otherwise you may require an external reference.

External reference

The 12-bit A-D converters will work well with an externally applied precision reference that is 2.5 volts or greater up to Vdd. The converter's accuracy depends on the reference voltage and begins to deteriorate with reference inputs less than 2.5 volts.

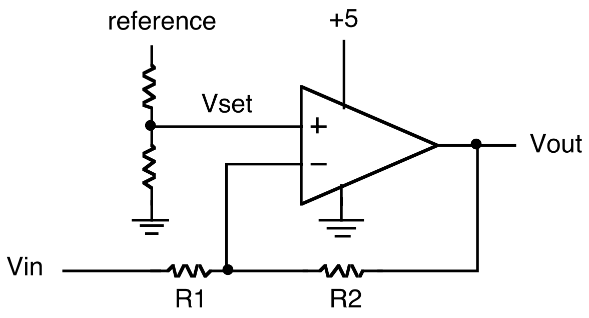
Convert a Bipolar Signal Input to Unipolar

Signals that go both Positive and Negative

We have been asked how to interface a bipolar signal to a Tattletale. Some Tattletales can be made to run in bipolar mode (make sure to add a Schottky diode between V– and ground), others won't. This note shows a simple way to convert a bipolar input to a positive only signal.

The simple schematic below shows an operational amplifier connected in an inverting configuration. Note that this design assumes a relatively low impedance signal is driving Vin.

Circuit for Converting Bipolar to Unipolar



The only tricky thing we have done is to give a positive bias to the non-inverting input to the amplifier. Remembering that op-amps adjust Vout so that the inverting and non-inverting inputs have the same voltage. The equation for this configuration is:

```
Vset = Vin + (Vout – Vin) *R1 / (R1+R2)
```

or, rearranged a little:

```
Vout = ((R1+R2) * (Vset – Vin) / R1) + R1
```

You can change this into two equations, one for gain, another for offset:

```
Gain = ΔVout/ΔVin = - R2/R1,
```

and offset (Vout for Vin = 0):

```
Offset = Vset * (R1 + R2) / R1
```

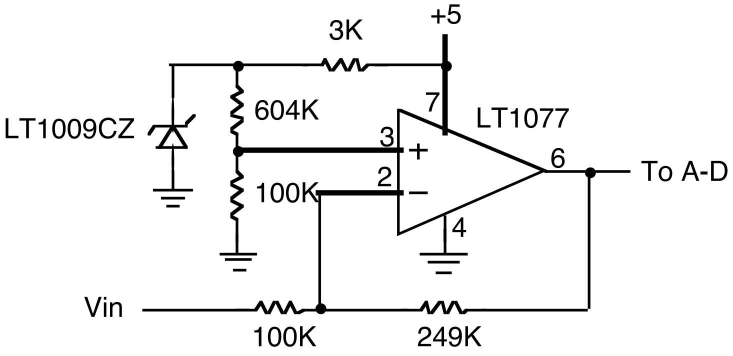
Alternatively:

```
R2/R1 = – Gain
```

```
Vset = Offset * R1 / (R1+R2)
```

Example Application: Suppose we had a signal that ranged from –0.5V to +0.5V, and wanted to translate that to a voltage we could read with our positive signal only A-D. We first add a 2.5V reference to the A-D and translate our input to a signal ranging from 0 to 2.5V. For this we would need a gain of 2.5, and an offset of 1.25V. Our circuit actually gives a negative gain so that –0.5V would translate to 2.5V, and +0.5V will translate to 0V. From the gain and offset equations we can find the ratio R2/R1 (2.5) and Vset (0.357V). If we use standard 1% resistors and the op-amp from Linear Technology we get the schematic shown below.

Example Circuit for Converting Bipolar to Unipolar



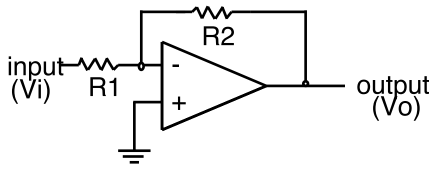
We show a 2.5V reference, also from Linear Tech, fed by a 3K resistor and divided down to reach the 0.357V. The resistors are not the exact values that we need to give the gain and offset we wished for, but come very close. The LT1077's input offset current is 0.25nA max., giving an offset of about 25μV at the input. This, combined with the input offset voltage of 40μV max. is still less than 1 LSB of a 12-bit converter.

Operational Amplifiers

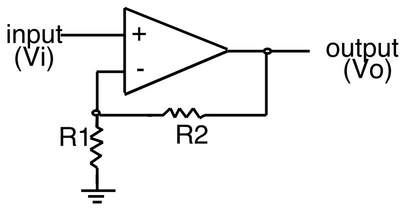
For those of you that are not familiar with the terms “operational amplifier” and “instrumentation amplifier”, this very brief explanation should be enough to get you started.

Operational Amplifiers

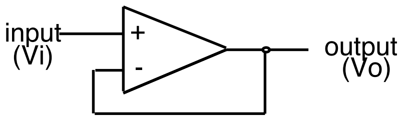
The op amp (as they are commonly called) is an amplifier with both an inverting and non-inverting input, and a single output. For our purposes we will treat it as having an infinite gain, and no current flows through either input. Three commonly used circuits for op amps are shown in the figure below, with the '–' designating the inverting and the '+' designating the non-inverting inputs.



INVERTING



NON-INVERTING



UNITY GAIN BUFFER

The Inverting op amp Configuration

The inverting circuit has a gain of –R2/R1. This makes sense since the op amp will do what is needed to keep the + and – inputs at the same voltage, and any current flowing into the input resistor (R1) will also flow through the 'feedback resistor' (R2). An input voltage Vi will cause a current Vi/R1, and cause an output voltage Vo of –R2(Vi/R1). Note the minus sign: the current flows through R2 away from ground if the current through R1 is flowing toward ground. An op amp is not magic and it cannot give an output that is larger than its positive supply or lower than its negative supply. The Tattletale has a regulated 5V, but its negative supply is ground, so most applications will supply the op amp with ground and +5V. This makes it impossible to use the inverting circuit as drawn, unless the input is a negative voltage.

Advantages: R1 protects input from external voltage

Disadvantages: To get a large gain, R1 must be small, loading the input.

Requires the input signal be negative for the positive signal needed by the Tattletale.

The Non-inverting op amp Configuration

Following the same analysis we did for the inverting configuration, we find that the gain of the circuit is (R1+R2)/R1 for the non-inverting circuit. In this case we are limited to positive inputs, since the Tattletale has no negative supply.

Advantages: High input impedance (will not load down your signal source).

Disadvantages: Takes a positive input only. To handle a negative input in a Tattletale application would require adding a negative supply to the Tattletale

Unity gain buffer op amp Configuration

The analysis for this configuration is simple. Since one of the fundamental operational rules of op-amps states that the output will do all it can to make the + and - inputs equal to each other, we can see that the input will follow the output. Since the signal is directly into the + input, the impedance is essentially infinite and the output impedance is zero. (theoretically of course!). Again we are limited to positive inputs, since the Tattletale has no negative supply. Real op amps will have real values for input and output impedance, as well as potential limitations when attempting to drive a signal near V- supply or the V+ supply.

Advantages: High input impedance (will not load down your signal source). Makes an excellent signal buffer. Low output impedance is appropriate for driving A/D inputs.

Disadvantages: Takes a positive input only. To handle a negative input in a Tattletale application would require adding a negative supply to the Tattletale. Signal may be distorted near the rails.

The Real World

As magical as they appear to be, op amps are not perfect. Some limitations are described below.

Op amp limitations

Current drain. In your Tattletale application you won't want to use any more power than needed. Check the current drain specs on the part you choose.

Supply voltage. Not all op amps work with supply voltages of +5 and ground. In some applications you may even want your circuit to work while the Tattletale is running from 3 volts.

Max output swing. Most op amps can only drive to within about a volt of their positive supply, others to only about a volt of their negative supply.

Input voltage range. Not all op amps are pleased with inputs close to the supply lines; some will work with input voltages <u>below</u> ground. Don't be confused by the inverting configuration described above. In that circuit the input voltage is always at ground.

Input offset voltage. The input offset voltage can be understood as the voltage difference between the positive and negative inputs needed to make the output voltage zero. Ideally this should be zero; in reality many op amps have input offsets of 10mV or more. This offset will appear as an error at the output equal to the input offset times the gain of the circuit.

Input offset current. Many op amps have FET inputs that take almost no current at the input, but some can have sizable currents flowing (nanoamps) into or out of the inputs. This can lead to sizable errors if your input resistor is a large value.

That's not all, but should be enough to give you some idea of what to look for in op amp data sheets.

Op Amps we have used:

We will show some of our biases by giving you short descriptions of three op amps we like. Note that each has its own strengths.

LT1077, 1078, 1079

Single, dual and quad amplifier from Linear Technology (408) 432-1900. This part has lovely current drain, input voltage range and input offset voltage specs.

· Current drain:	40μA typical for each amplifier (μA = microamp, a millionth of an amp)
· Supply voltage:	3 volts minimum
· Max output swing:	ground +6mV to positive supply –1 Volt (mV= millivolt, 1/1000V)
· Input voltage range:	ground –5V to positive supply
· Input offset voltage:	30μV typical (μV = microvolt, a millionth of a volt)
· Input offset current:	0.25nA max. (nA = nanoamp = a milli-micro amp)

TLC1078

Dual amplifier from Texas Instruments (800) 232-3200. Note the superb input offset current specifications.

· Current drain:	15μA typical for each amplifier
· Supply voltage:	1.4V minimum
· Max output swing:	ground to positive supply –1 Volt
· Input voltage range:	ground –0.2V to positive supply –1 Volt
· Input offset voltage:	160μV typical
· Input offset current:	0.1pA max. (pA = picoamp = a micro micro amp)

ALD1704, 1701, 2701, 4701

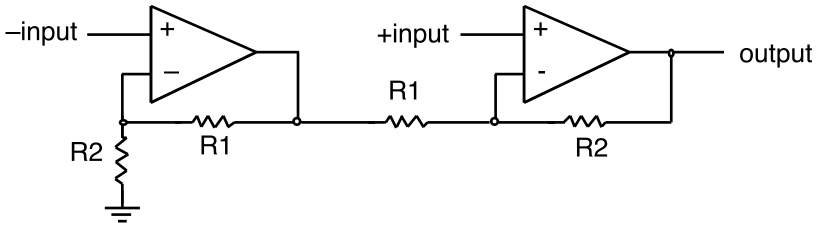
Single, dual and quad amplifier from Advanced Linear Devices (408) 720-8737. This part drives line to line.

· Current drain:	120μA typical for each amplifier
· Supply voltage:	2.0V minimum
· Max output swing:	ground to positive supply –1 Volt
· Input voltage range:	ground –0.2V to positive supply –1 Volt
· Input offset voltage:	2mV <u>for the premium version</u>
· Input offset current:	25pA max. (pA = picoamp = a micro micro amp)

NOTE. If you need the specifications for gain-bandwidth product, noise characteristics, stability, offset drift, temperature coefficients or output drive characteristics, refer to the data sheets from the manufacturer. We do not supply them in this manual. These and other parameters are important in some applications, but we don't have the space to cover everything.

Instrumentation Amplifiers

What if your sensor provides two outputs (like a bridge circuit)? An operational amplifier has both an inverting and a non-inverting input, but one is needed to set the gain of the circuit. You can build an instrumentation amplifier from two operational amplifiers and a small pile of precision resistors using the circuit below:



This circuit has a gain G = (R1+R2)/R1, and works over a range of input voltages that go from ground to 'Max positive output' * R2/(R1+R2).

LT1101 Instrumentation amplifier: Linear Technology's LT1101 packages this all in one 8-pin part and can be set to a gain of 10 or 100 with no external parts.

· Current drain:	75µA typical
· Supply voltage:	2V minimum
· Max output swing:	ground +4mV to positive supply –1V
· Input voltage range:	ground +70mV to positive supply –2V
· Input offset voltage:	50µV typical
· Input offset current:	8nA max.

CHAPTER 8

TFX-11 Hardware Reference

TFX-11 Hardware Reference

Overview of I/O

The TFX-11 provides up to 19 A/D converter input and up to 24 I/O pins. Eight I/O pins are shared between digital input functions and 8-bit A/D functions. Therefore, if all the 8-bit A/D pins are used for digital input, then only the 11 12-bit A/D inputs are left. There a 16 pins available exclusively for digital I/O. Eight are located on the HC11, and eight are located on the PIC. Because of their placement each set has slightly different operational characteristics. Since the HC11 is the primary controller for the TFX-11, TFBASIC has direct access to the HC11 port I/O pins. In contrast the HC11 uses synchronous serial communications with hardware handshaking to operate on the PIC I/O pins.

Selecting Digital I/O pins

Therefore it is recommended that all high speed bit twiddling be performed using the eight bit port on the HC11, while the PIC I/O be used for slower speed operations.

A/D input choices

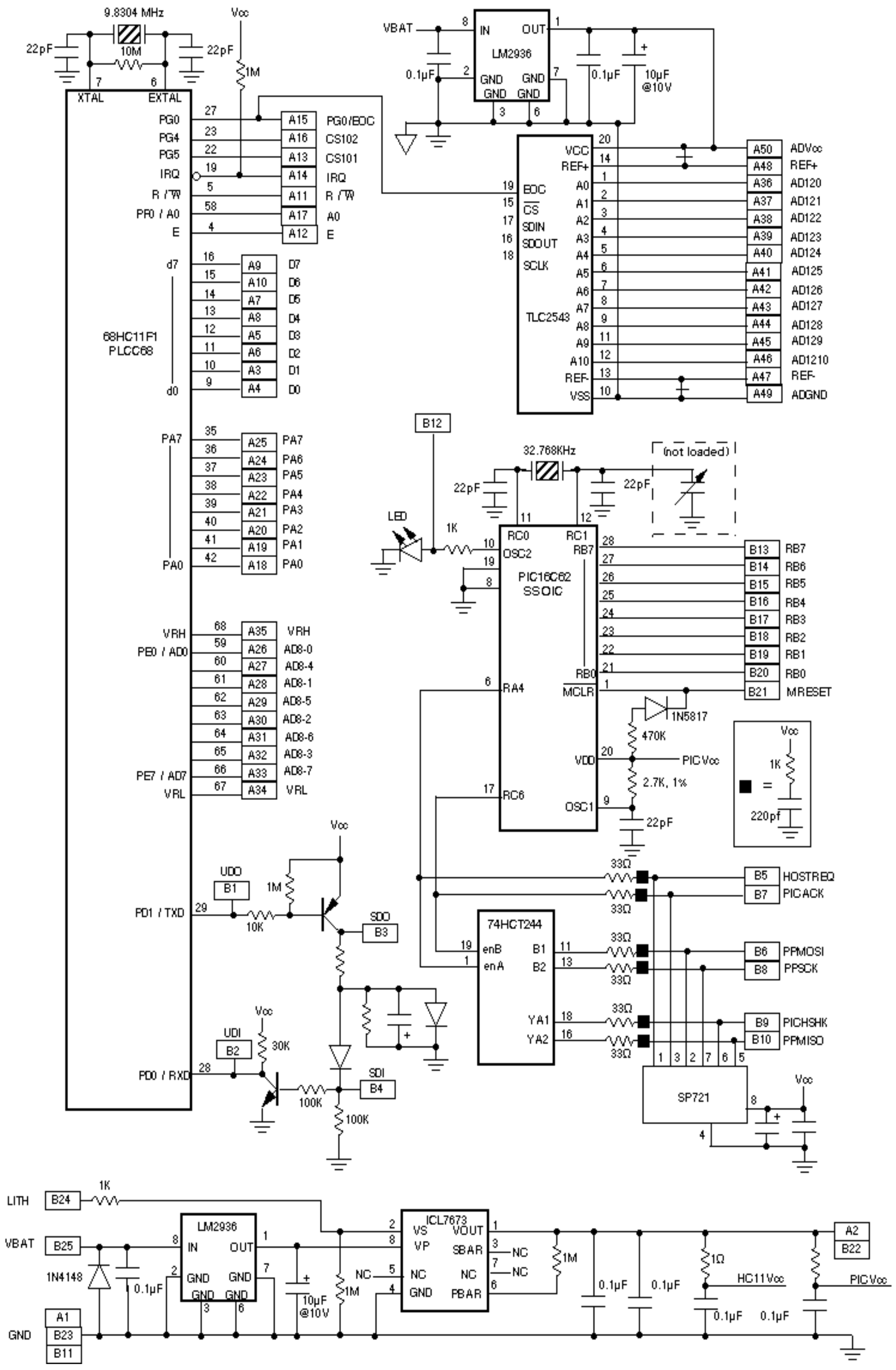
The 8 bit A/D converter is internal to the HC11. These A/D input pins can also double as digital inputs. The 12-bit converter is a separate device and communicates with the HC11 via the SPI bus. Here the choice of which inputs to use is mainly based on resolution required. The 8 bit converter does allow you to sample more quickly due to its lower resolution and its registers being internal to the HC11.

Attaching devices to the bus

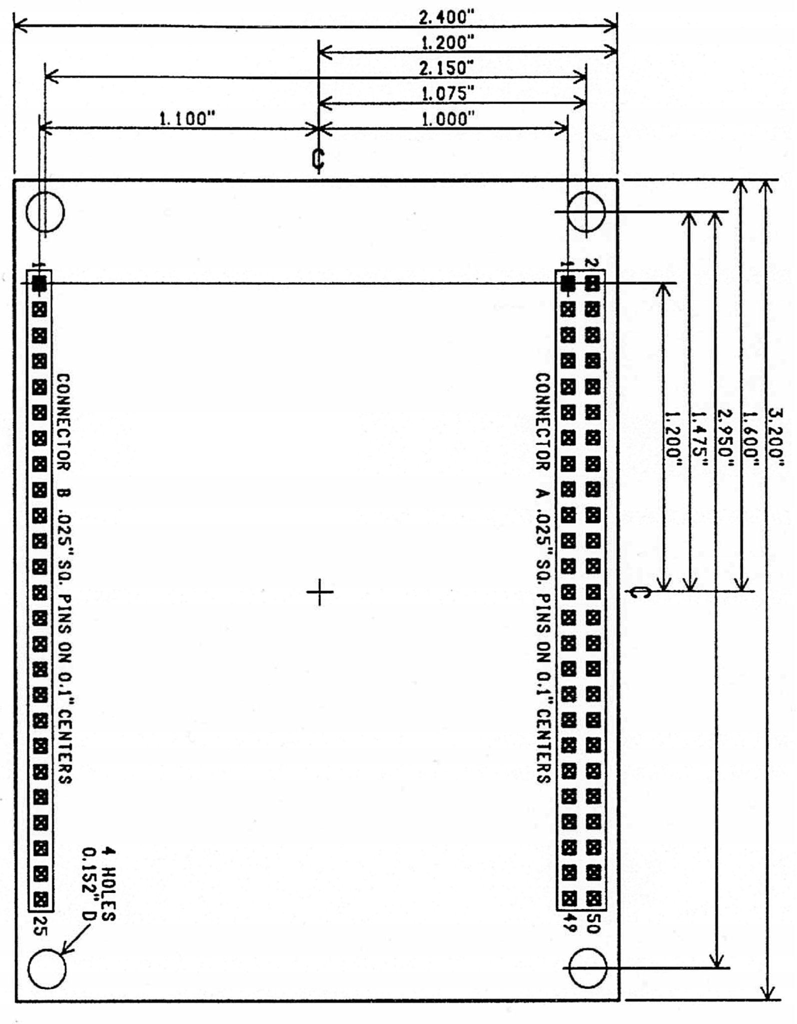
Also available are some of the HC11 bus signals - the data lines, chip selects and other qualifying signals for attaching external memory-mapped items to the HC11. The two chip selects, CSIO1 and CSIO2, may be programmed to map into specific addresses, simplifying the addition of memory mapped I/O devices. (see the TFBASIC memory map in this manual and the HC11F1 Technical Data Book for details).

TFX-11 Reference Schematic

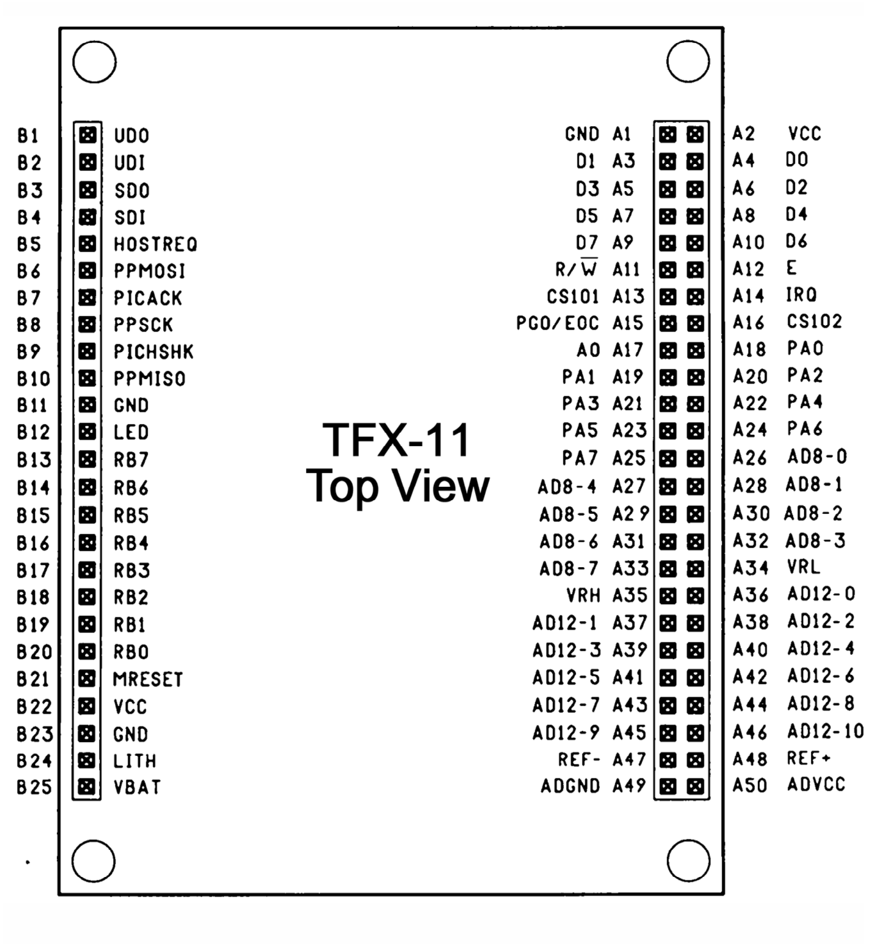
The simplified schematic on the following page identifies only the major components and external connections needed to properly interface additional hardware to the TFX-11



TFX-11 Board Dimensions

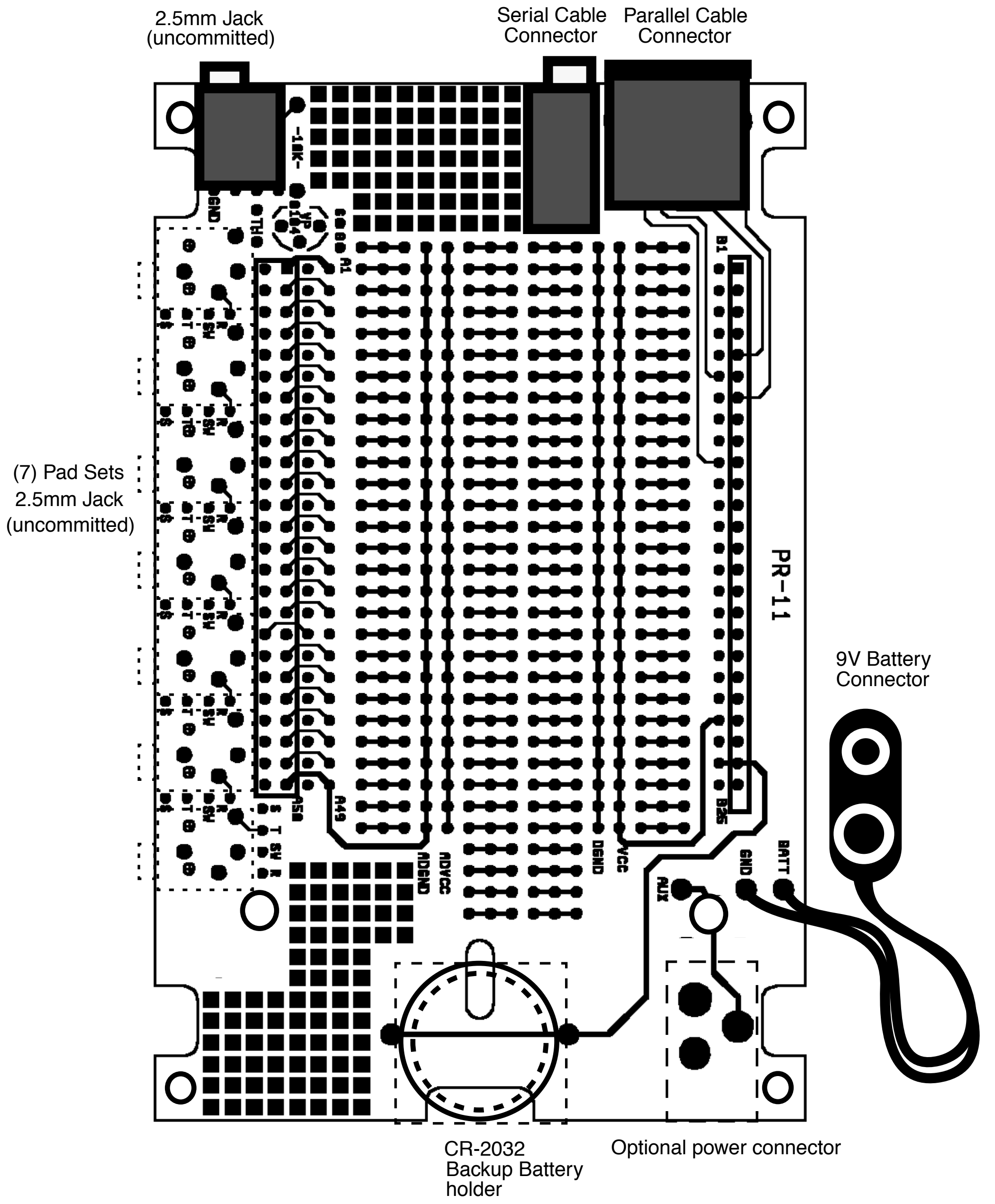


Pin outs as viewed from the top (pins pointing toward you) of the TFX-11



The TFX-11 pins come out the top of the board. When the TFX is connected to the PR-11 prototype board the components on this top surface are inaccessible. Note that the two boards mate face to face, therefore the pinouts of the TFX-11 and the PR-11 are mirror images when viewed from each board's top.

The PR-11 Prototype Board



Overview

The PR-11 Prototype board was designed to support many different configurations and applications. It carries Power, Serial and Parallel jacks as well as pads laid out to take common connectors which you may purchase separately and install yourself as necessary.

The PR-11 board outline is designed to fit inside a standard SERPAC A-279V plastic enclosure, included in the deluxe development kit. The case comes with the parts needed to handle the basic PR-11 board configuration as delivered. Depending on what optional connectors you install you may have to modify the case to access them.

The standard configuration includes a 9V battery connector for power, a 9-pin mini-din connector to mate to the Parallel cable, and a 3.5mm stereo phone jack to mate to the serial cable. The CR2032 backup battery holder is mounted on the underside of the PR-11 board. When the TFX-11 is plugged into the PR-11 board , the PR-11 board provides all the necessary connections to fully operate the TFX-11.

Power Connections

Main power is connected through a 9V battery connector. The fully assembled case has a 9V battery compartment to house the battery if so desired.

Optional Power jack

Another set of pads are on the board to allow mounting of a jack (KYCON KLD-0202-A) compatible with standard wall mount DC power supply modules. The thru holes for this connector are used to attach the tie-wrap used as a strain relief for the 9V battery connector. This tie-wrap must be removed to install the jack. When this connector is used the 9V battery compartment will no longer fit inside the case. The mating connector of the DC power supply must have a 2.1mm inside post and where the inside post must be ground. BE SURE THE POLARITY IS CORRECT BEFORE CONNECTING!

The Backup Battery

The Backup battery is not a requirement for operation of the TFX-11, but can be useful in some applications. As long as the main power is attached the backup battery is out of the circuit. (NOTE: The backup battery has a 1M resistor across it. This is to allow proper operation with the battery removed.) The backup battery allows retention of the clock time and data in the 128K ram and internal to the HC11 and PIC when main power is removed. Data stored in the SFLASH is not affected, but the pointer to the end of data in the SFLASH is lost.

Communications Connectors.

A 9 pin mini-din (LZR Electronics MDJ9PS) and an 3.5mm stereo phone jack (Shogyo International SJ-0375-3RT) are mounted for connecting to the host PC using the Onset cables.

Optional 2.5mm stereo phone jacks

Along one side of the PR-11 board are seven sets of pads designed to accept either 2.5mm stereo (Shogyo SJ-0252-3RT) or mono (LZR Electronics Inc. LZR-RL254) phone jacks. All seven are uncommitted; that is, the connections to the jack's internal contacts are brought out and terminate next to the connector. It is up to the user to determine what jumpers get installed based on the application and the choice of connector used.

Tutorial Area

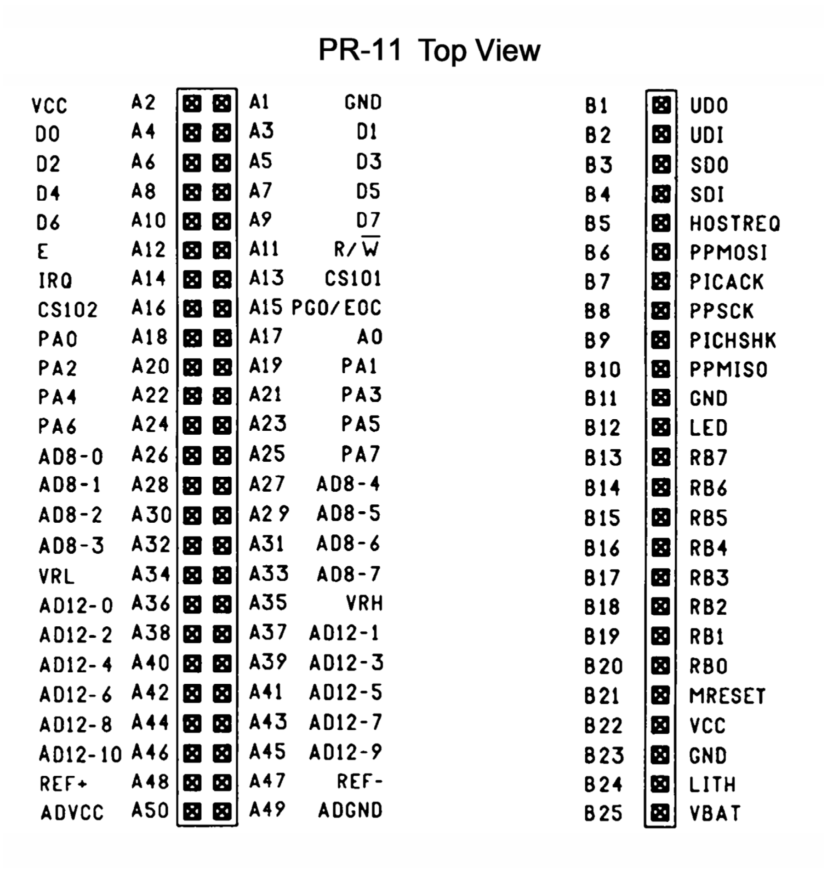
Included on the PR-11 breadboard are pads already laid out and ready to accept a thermistor, resistor and FET that form a simple but accurate temperature measurement circuit.

Designing a custom interface board for the TFX-11

Some of you may want to design your own mating circuit board in place of the PR-11. The required terminations for the 8 bit A/D subsystem as well as specific connector pin assignments are summarized in the table below.

TFX-11 Pin	PR-11 Termination	Comments
B5	MINI DIN pin E2	Parallel Port connector
B6	MINI DIN pin E9	Parallel Port connector
B7	MINI DIN pin E1	Parallel Port connector
B8	MINI DIN pin E7	Parallel Port connector
B9	MINI DIN pin E3	Parallel Port connector
B10	MINI DIN pin E8	Parallel Port connector
B11	MINI DIN pin E4	Parallel Port connector
B3	3.5mm jack	RS232 output
B4	3.5mm jack	RS232 input
A35	+5V	8 bit A/D reference connection
A34	GND	8 bit A/D reference connection
B24	Backup Battery +	Ltihium battery holder
B25	9V battery +	Battery power input

Pinouts as viewed from the top of the PR-11 (socket strips point toward you).



Explanation of Connector Pin Functions

Overview The following paragraphs explain in detail the functions associated with the pins brought out to the PR-11 prototype board.

Important Note for those who want to design there own interface board There are connections made on the protoboard that are required for proper operation of the TFX-11's eight bit A/D converter. If you design your own board you should include these connections or a suitable alternative to assure proper operation even if you don't use the 8 bit A/D. If these pins are left unconnected the internal 8-bit A/D will definitely not operate properly, and may cause other anomalies.

Onset will supply drawings to assist you in designing a mating PC board. Please contact Onset technical support for details.

UDO **(B1).** The output of the hardware UART at TTL levels. The signal is inverted compared to the RS-232 (SDO) output.

UDI **(B2).** The input to the hardware UART at TTL levels. The signal is inverted compared to the RS-232 (SDI) input.

SDO **(B3).** The output of the UART at RS-232 levels. The signal is inverted compared to the TTL (UDO) output.

SDI **(B4).** The input to the UART at RS-232 levels. The signal is inverted compared to the TTL (UDI) input.

HOSTREQ **(B5).** Driven LOW by the Host PC when requesting attention. Bringing this line low will halt a running TFBASIC program. Not meant as a user function.

PPMOSI **(B6).** Communications line from Host PC to TFX-11. Not meant as a user function.

PICACK **(B7).** TFX-11 signal that it has recognized the HOSTREQ. Not meant as a user function.

PPSCLK **(B8).** Host to TFX-11 communication clock. Not meant as a user function.

PICHSHK	**(B9).** TFX-11 signal to host that it is ready for new command. Not meant as a user function.
PPMISO	**(B10).** Communications line to Host PC from TFX-11. Not meant as a user function.
GND	**(B11).** System digital ground.
LED	**(B12).** Output from external activity LED. When oscillating the PIC is active, when not the PIC is sleeping.
RB7-RB0	**(B13-B20).** Highest numbered block of the 24 digital I/O pins. RB0=I/O16, RB7= I/O23. These are on the PIC processor and since they require HC11 to PIC communications rather than direct HC11 port access they are not suitable for high speed toggling such as might be used in tone generation. Use PA0-PA7 instead.
MRESET	**(B21).** Taking this LOW will perform a power-on reset of the TFX-11.
VCC	**(B22).** 5VDC coming from the main LM2936 linear regulator. A maximum of 50mA is available from this regulator, but the amount available for your project will vary depending on the activity of the HC11 and its peripherals. The TFX-11 should draw typically 3.5mA at idle to a peak of about 25mA when making conversions, allowing about 25 mA for your projects. This regulator has short circuit protection, reverse battery protection, and thermal shutdown. Take care not to exceed the power dissipation specification (which changes with ambient temperature and battery voltage) or the TFX-11 will go into thermal shutdown, temporarily halting operations until it recovers. For more detailed information refer to the LM2936 data sheet, supplied in Adobe Acrobat PDF format, on the TFTools/TFBASIC software distribution disk.
GND	**(B23).** System digital ground.
LITH	**(B24).** V+ side of CR2032 Lithium backup battery.
VBAT	**(B25).** Main power input to the TFX-11. This input can be in the range 5.5V to 18V, but care must be taken so as not to exceed the maximum

power dissipation that would result in thermal shutdown of the regulator. See the LM2936 data sheet for more information.

GND	**(A1).** System digital ground.
VCC	**(A2).** See VCC (B22) above.
D0-D7	**(A3-A10).** (See PR-11 drawing for exact pin to signal correspondence). When TFBASIC is running the HC11 is in expanded operating mode so these lines default function is as the external data bus. These lines may be used with signals CSIO1,CSIO2, A0, R/W, and E to add memory mapped external devices.
R/W	**(A11).** Indicates direction of transfers on external data bus.
E	**(A12).** This is the E clock output. It is crystal frequency divided by 4. When the E-clock is low the HC11 is processing. When high there is a data access taking place.
CSIO1	**(A13).** First user available chip select that is mapped to range 400h-7FFh when enabled. Default for the TFX-11 is disabled. Requires understanding of the HC11 and assembly language programming to access. See the MC68HC11F1 technical reference for more information. **NOTE:** Be aware this line may be used in the future by ONSET to attach memory expansion devices to the TFX-11.
IRQ	**(A14).** Falling edge sensitive, maskable interrupt to the HC11 CPU. Set to edge sensitive at reset. May be used for early wakeup from HYB. Sensitivity may not be changed by user. Not used by the TFX-11 operating system. Requires understanding of the HC11 and some assembly language programming to access. See the MC68HC11F1 technical reference for complete information.
PGO/EOC	**(A15).** Used to read the 12-bit A/D converter end of conversion signal.
CSIO2	**(A16).** Second user available chip select that is mapped to range 800h-FFFh when enabled. Default for the TFX-11 is disabled. Requires understanding of the HC11 and assembly language programming to access. See the MC68HC11F1 technical reference for more information.

	NOTE: Be aware this line may be used in the future by ONSET to attach memory expansion devices to the TFX-11.
A0	**(A17).** Lowest external address line. Used to qualify reads and writes for some peripherals attached to the data bus.
PA0-PA7	**(A18-A25).** Digital I/O pins corresponding to pins I/O0-I/O7
AD8-0, AD8-7	**(A26-A33).** 8 bit A/D converter inputs. These pins also act as digital inputs when used with the PIN command. The corresponding PIN parameters are I/O 8 to I/O 15, and the corresponding CHAN parameters are 11-18. Note that AD8-1 thru AD8-6 do not necessarily match connector pins A27-A32 in sequence. Refer to the connector reference for the PR-11 or TFX-11 boards for the correct sequence.
VRL	**(A34).** Provides the LO reference voltage for the HCll on-board 8-bit A/D converter. This pin is NOT connected unless plugged into the PR-11 breadboard. The PR-11 has a jumper that normally connects this signal to ground. The jumper may be cut if you want to connect it to another reference. Bypass capacitors may be required to minimize noise that will affect A/D accuracy.
VRH	**(A35).** Provides the HI reference voltage for the HCll on-board 8-bit A/D converter. This pin is NOT connected unless plugged into the PR-11 breadboard. The PR-11 has a jumper that normally connects this signal to VCC. The jumper may be cut if you want to connect it to another reference. Bypass capacitors may be required to minimize noise that will affect A/D accuracy.
AD12-0, AD12-10	**(A36-A46).** 12 bit A/D converter inputs. The corresponding CHAN parameters are 0-10
REF-	**(A47).** Provides the LO reference voltage for the 12-bit A/D converter. This pin is connected to ADGND via a jumper on the TFX-11 main board. The jumper may be cut if you want to connect another reference.
REF+	**(A48).** Provides the HI reference voltage for the 12-bit A/D converter. This pin is connected to ADVCC via a jumper on the TFX-11 main board.

ADVCC is the regulator that is the dedicated supply for the 12-bit A/D converter. The jumper may be cut if you want to connect an absolute reference.

ADGND **(A49).** Analog ground connection for the 12-bit A/D converter input signals. To avoid introducing noise in the A/D conversions DO NOT attach any other grounds to this point

ADVCC **(A50).** Analog reference for the 12-bit A/D converter input signals. To avoid introducing noise in the A/D conversions DO NOT attach any unrelated device grounds to this point.

TFX Timekeeping

RTC Storage Structures

TFXBASIC has two structures for holding time related information - the ? variable and the ?() array.

The ? variable. The ? variable consists of 4 bytes and increments every second.

The ?( ) array . The ?() array is a separate structure of 7 four-byte integer variables with ?(0) holding seconds (0-59), ?(1) holding minutes (0-59), ?(2) holding hours (0-23), ?(3) holding days (0-31), ?(4) holding months (1-12), ?(5) holding years (1980-2040). The ?() array is only updated by invoking the RTIME command or by directly modifying it with an assignment to one or more of the elements. ?(6) contains the current ticks count (0-99). Ticks occur a the rate of 100 per second and cannot be changed.

Using these predefined structures simplifies date and timekeeping and time-based math calculations. Note that the ?(5), the year array element, contains all four significant digits of the year, not just the last two, to allow simpler and error free date calculations.

Two separate Crystals

Because the PIC and HC11 do not share clock crystals it is possible that the time kept in each processor will drift from the other, therefore a clear understanding of how the two interact is necessary to insure critical timing sequences are not compromised.

Clock Resolution

The PIC keeps time in one second increments. TFBASIC on the HC11 keeps time to 1/100 second. Because of the HYB overhead it is recommended that all intervals below 5 seconds only be handled by SLEEP, and intervals longer than 5 seconds be handled by HYB. Where interval timing must be precise within a sub-second then SLEEP must be used.

PIC as primary RTC

Since the PIC clock crystal is always running, even in HYB mode, the PIC processor is considered the principal RTC on the TFX-11. At Program Launch the Host PC copies its current system clock time, in seconds, to the PIC. When the PIC starts the HC11 by removing it from RESET the PIC time is copied to the TFBASIC ? variable by the TFBASIC startup code, with tenths of seconds being initialized to 0. After this initialization there is nothing explicitly built into the TX to keep the two clocks synchronized, but since the commands available between the PIC and TFBASIC use calculations based on relative time, there is no problem.

When an RTIME or STIME command is executed in TFBASIC it only acts locally, that is it does not affect and is not affected by the time in the PIC. After TFBASIC initialization all timekeeping data is kept locally on the HC11 and it is updated locally, unless otherwise commanded from the TFBASIC program.

Timing dependencies

SLEEP commands. All SLEEP clock timing is relative to the TFBASIC periodic timer interrupt on the HC11. TFBASIC keeps a separate SLEEP timer which, while clocked from the same HC11 periodic interrupt, does not share count values with the ? variable, and therefore is not affected by the RTIME or STIME commands. It can only be modified by a SLEEP x command.

HYB commands. Since the HC11's clock is stopped during HYB, absolute HYB timing is dependent on the PIC's clock crystal. Wakeup from HYB timing is calculated as relative time; as seconds from the current value in the HC11 clock to the requested wakeup time. This value is transferred to the PIC added to the current PIC time to calculate the alarm wakeup time. The HC11 stores locally this offset count as well as the time it entered HYB. When the PIC wakes it up TFBASIC gets the current time. from the PIC Assuming no drift between the two crystals, this technique will wake it up at the exact time requested, with the ? variable containing the correct time for TFBASIC, even if the times on the PIC and HC11 differ. Since the HC11 has been stopped during HYB the only valid time available is the PIC time in seconds.

Clock Drift

Any drift error during a HYB period depends on the PIC's crystal oscillator. Any drift error during TFBASIC operation is dependent on the HC11 crystal.

TFX-11 Data Storage Options

The TFX-11 offers some different data storage options, each designed to suit a particular purpose best. Different methods of write and read access define appropriateness for using each of these different storage areas. The major on-board memory areas are listed below:

- Program and Variables memory, bank 0 of the 128K static RAM
- @ array memory, bank 1 of the 128K static RAM
- UEEPROM, 128 bytes User EEPROM inside the HC11
- 472K bytes Serial flash EEPROM (SFLASH)

Refer to the TFBASIC memory map in Chapter 6, TFBASIC Internals, for a graphical representation of the descriptions that follow.

Program and Variables memory, bank 0 of the 128K static RAM. This is the default RAM bank that contains the running TFBASIC program, token interpreter, and variable storage space. It is TFBASIC's prime operating area.   User storage in this area is accessed directly through TFBASIC variables. The total area available for storage is reduced by the overhead used by the token interpreter and the storage used by the user's program tokens.

This memory is battery-backed and retains its information as long as the backup battery is not removed. This memory is not accessible directly via the parallel port. Any data stored here must be transferred to the SFLASH before off-load or transferred directly out the UART via a running TFBASIC program.

@ array memory, bank 1 of the 128K static RAM. This area of RAM is set aside for random access data storage. It is predefined as a single array containing 15232 elements, called the @ array. Each element of this array is a standard four byte variable. This section of RAM is easily accessible; any assignment to or from an @ array variable transfers the data, automatically performing the bank switching. This area is NOT automatically initialized on startup. As it is battery-backed, it can retain its contents from one reset to another, as long as the backup battery is not removed.

This memory is not accessible directly via the parallel port. Any data stored here must be transferred to the SFLASH before off-load or transferred directly out the UART via a running TFBASIC program.

@ARRAY limitations. The @ARRAY is mainly meant for use as a datafile. Unlike the SFLASH datafile, which is write only, the @ARRAY may be written and read. It is battery-backed and can be considered non-volatile as long as the backup battery is viable. @ARRAY variables CANNOT be used as the following:

- FOR loop variable
- RTIME argument
- STIME argument
- READRTC argument
- SETRTC argument
- CALL return variable
- INPUT variable
- ONERR return variable

This should present no problem as there is plenty of RAM available for regular variables.

UEEPROM, 128 bytes User EEPROM (inside the HC11). This is located in a section of the internal memory map of the HC11. It is read, erased and written in units of four bytes, the standard TFBASIC variable. As with all EEPROMs any one location can only be written a limited number of times (approx 10,000) before it fails. This is in contrast to reads, which are unlimited.

The most appropriate use for this limited memory is to store configuration information or sensor calibration parameters.These values would be written once when the instrument is calibrated or setup and then could be read any number of times as necessary. This technique allows a single program to compensate for sensor or external hardware variations across otherwise identical instruments.

These 128 locations are divided into 32 groups of four making the integer (or float) the default storage unit. They are accessed only by the TFBASIC commands VGET and VSTORE, with valid locations being 0-31. These locations are totally non-volatile - they will remain even if all power is removed.

472K bytes Serial flash EEPROM (SFLASH). This storage is in an external serial linked flash EEPROM, or SFLASH. This SFLASH can be read,written or erased directly by the Host computer via the parallel port, but it is write only to the TFBASIC program. The SFLASH is written to using the STORE command. After each write to the SFLASH, TFBASIC updates an internal pointer variable to point to the next location data can be written to. The only way to erase this memory is to perform a successful OFFLOAD from the Host computer, after which you will be prompted “Do you really want to erase this data?” This is to help protect the data from accidently being corrupted or erased by a programming logic, accidental restart, or other error.

Minimum SFLASH datafile storage

The SFLASH holds both the program and the data. Therefore the amount of data storage is impacted slightly by the size of the program. Given the largest program there will be a minimum of 405k for data. Use the DFMAX read-only variable from inside your program to report the exact amount available.

This storage is best suited to holding data that do not need to be referenced once written.

This storage is non-volatile. The information written into it will remain intact even if all power, including the battery backup power, is removed. Even if the other components are not functioning, it may still be possible to read out the data from the SFLASH.

TFX-11 Hardware Description

Functional overview of the TFX-11

The TFX-11 contains two processors, 128K static RAM, 472k flash EEPROM (SFLASH), and a parallel port and an RS232 serial port for development/communications interface. This section will attempt to provide a general overview of the relationships, responsibilities, and interactions of the major subsystems.

The architecture of the TFX-11 includes an HC11 with 128K of RAM, 64K of which is used for the TFBASIC token engine, the users program, and user variable storage. The TFBASIC program is stored complete in flash as an executable image. This image is created using TFTools on the host computer. It is up to the TFTools host software to put together the image which will be loaded by the HC11 from the flash into its RAM for execution. The initial step is for the host computer to load this image directly into the SFLASH via the parallel port connection. The SFLASH provides permanent (until deliberately erased and rewritten) storage of the program and prepares the way for the next step, which is having the HC11 copy the image into RAM. Once loaded into the RAM the PIC controls start of the program because it controls the HC11's RESET line. The PIC's timing functions can include a delayed start or periodic wakeup and shutdown of the HC11, providing the very lowest power modes when the HC11 is shutdown .

Fundamental to the design of the TFX is it's clean slate (64K RAM) at RESET. Resident on the host computer, TFTOOLS formats (compiles/ assembles) both the user's code and a copy of the token engine into an image which is uploaded to the TFX for execution. The PIC controls RESET and at power on or Launch instructs the HC11 to load this program in the flash and then RESETs the HC11 to start to execute this code.

TFX-11 in Operation

Both the HC11 and the Host PC may become the master of operations, but not simultaneously. The following discusses each as master separately, as well as the supervisory role of the PIC.

Host computer in charge. Under the control of the host computer the devices active are the PIC and the SFLASH. Through the signal HOSTREQ the PIC will detect that the TFX is connected to the host interface and wants service. At this point the PIC commands an orderly shutdown of the HC11. When this process has been completed the PIC is slave to the host computer along with the SFLASH and the A/D. When the PIC has completed configuring itself as slave, it asserts the SFLASH CS and responds to HOSTREQ with a PICACK. The PIC has now given control to the HOST and set up the bus so that the default communication from the HOST is with the SFLASH. The host is not required to communicate to the PIC for any SFLASH related activities, READ,WRITE, or ERASE. If the HOST wants to talk directly to the PIC or the A/D it must grab the attention of the PIC by toggling the HOSTREQ line. The PIC continually polls the state of the HOSTREQ line, which acts a as a PIC CS. If it changes state it can mean only one of two things - either the HOST is no longer requesting control or the HOST is requesting communication with the PIC. If the HOSTREQ remains de-asserted for more than 2 seconds it is understood to be a disconnect - otherwise the PIC disables all CS on the bus and waits for a command from the HOST.

Communications to the PIC from the host include functions such as reading and setting the time, setting an alarm start time or periodic sleep/ wake cycle time, getting status information such as serial number and Model number of the board and version information for the PIC firmware. The primary use of this interface the off-loading the SFLASH data and uploading new program images to the SFLASH using TFTOOLS. The HC11 is completely shutdown while the HOST is active.

PIC processor in charge.

When the HOST parallel interface is disconnected the PIC assumes the position of supervisory controller and holds the HC11 in RESET until the PIC finishes setting up the HC11's environment. In this state the PIC can only communicate with the HC11 via the SPI, and the HC11 must be running for those communications to take place. On power up the PIC's tasks include; detecting if the HOST parallel interface is connected and if not, controlling the orderly power up of the HC11. When the PIC determines the HOST interface is not connected releases the reset it holds on the HC11so it can start running in expanded mode. At this point the PIC becomes a slave to the HC11.

HC11 in charge

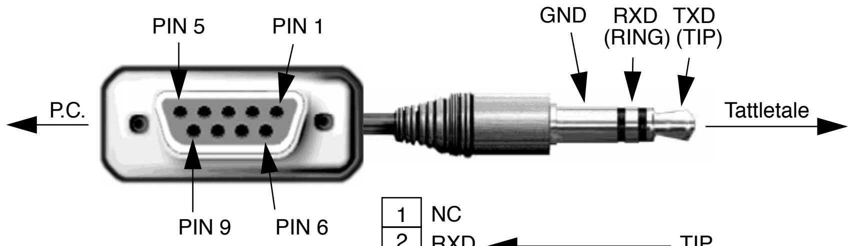
When the HC11 is up and running it is the main processor, with all its program information in RAM. In this mode it is running almost exactly like any of the previous Tattletales, with the PIC acting primarily as the RTC and wakeup alarm.

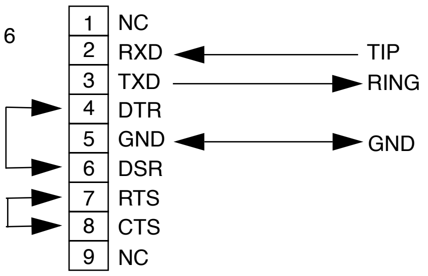
The PIC as RTC and Low Power controller.

With the HC11 in charge the PIC still has the power to take back control in response to external events (i.e. the interface attached) or as necessary to control low power mode requests from the HC11. The HC11 can command the PIC (via the SPI bus) to set an alarm wakeup time, and then the HC11 can put itself into STOP mode. When the PIC alarm times out it uses the XIRQ line to interrupt the HC11 to wake it up, and, after a brief handshake with the PIC over the SPI, the HC11 resumes processing from where it left off. If the handshake does not take place the PIC will assume an operational problem with the HC11 and place it in reset.

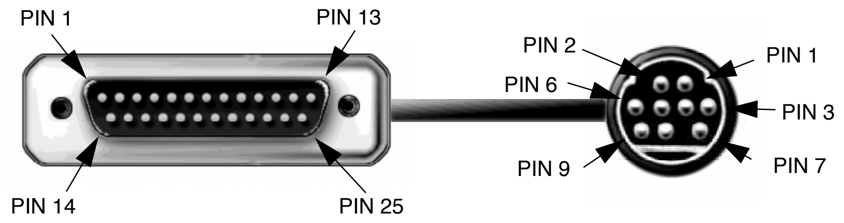
Low Power operation : This can be initiated by the TFBASIC program requesting a HYB interval. The PIC sets its alarm timer to the wakeup time (one second resolution) and when the PIC timer times out the PIC will use XIRQ to awaken the HC11. While in HYB only the PIC is active. HALT will put the TFX-11 in this same low power state, but it does not provide an automatic wakeup.

Serial Cable Pinout





Parallel cable Pinout



2	→	MOSI	4
9	→	SCLK	1
10	←	UNUSED IN	9
11	←	MISO	6
12	←	PICACK	7
15	←	PICHNDSHK	8
16	→	UNUSED OUT	3
17	→	HOSTREQ	5
18-25	↔	GND	2

CHAPTER 9

Glossary of Terms

Glossary of TFBASIC Terms and Definitions

Abbreviations There are no abbreviations allowed for any TFBASIC commands.

Arrays The number of arrays in TFBASIC is limited only by the size of the variable storage area.

Arithmetic Operators The five arithmetic operators have the highest priority of all of the TFBASIC operators. Note that TFBASIC does not have a separate high priority unary minus operator, but instead treats the negation of a constant or variable as zero minus the value of the variable or constant.

These are the TFBASIC arithmetic operators in order of precedence (all operators on the same level are evaluated left to right) :

highest	* / %	Multiplication, Division, Modulo
	+ -	Addition, Subtraction
	> <= > >= <> =	Relational operators
	&	Logical bitwise AND
lowest	|	Logical bitwise OR

See the “TFBASIC Floating Point” section for details about arithmetic operations involving floats.

Assembly Language The TFBASIC tokenizer has a built-in assembler. Code can be assembled in line with the program or into a separate area as desired. See “TFBASIC Assembly Code Programming” section for details.

Break A CTRL-C sent via the primary serial port can break a running TFBASIC program. A special command 'CBREAK' followed by a label can be used to specify the address to restart to when a CTRL-C is received. You can disable CTRL-C breaks by writing a zero byte to address 9C hex. A count of CTRL-C characters will continue to be updated at address 9B hex. Clear this before re-enabling break-outs. See CBREAK.

Case Labels and variable names are case sensitive, commands and keywords are case insensitive.

Comments TFBASIC provides two ways to include comments in your code; one for TFBASIC code, the other for assembly code. In TFBASIC a pair of forward slashes (//) can appear anywhere on a line and cause the rest of the line to be ignored. In assembly code a semicolon (;) is used to separate comments from code. Comments are stripped before the tokenizing pass and therefore do not affect execution code size.

Constants TFBASIC supports string, integer, and floating point constants. Integer constants are signed decimal numbers in the range -2147483648 to +2147483647 or unsigned hexadecimal numbers in the range &H0 to &HFFFFFFFF. Floating point constants must include a decimal point and/or the power-of-ten specifier 'E'. All floating point constants are single precision with a range of ±1.175494E-38 to ±3.402823E+38 and 0.0. See the "TFBASIC Floating Point" section for details. There are no short integer constants, and no octal constants.

Character constants of up to four characters in single quotes are allowed. The character on the right is placed as the least significant byte in the four byte constant.

String constants must be bracketed by double quote characters. String constants can be used with the PRINT, INPUT, and STORE commands.

Datafile : Storage and Retrieval The Tattletales have a special non-volatile data storage area called the datafile, which can only be written from the TFBASIC program. Datafile commands are STORE for writing, and OFFLD for reading. An OFFLD command is not available from the TFBASIC program, but is a menu command in TFTools. The datafile contains a read only pointer that points to the next empty datafile byte, and this pointer automatically increments after each STORE command. NOTE: Once the data is stored in the SFLASH it cannot be read back or written over by the program!

Data Types TFBASIC supports long integers, string constants, and strings as well as IEEE 754 single-precision floating point. There are both implicit and explicit conversion operators to convert numbers between integer and floating point formats and back again. See the "TFBASIC Floating Point" section for details.

Decimal See **Radix**

Division by Zero Integer division by zero will cause the program to stop executing and display the "HOW" message. Refer to the "Errors" heading for more information on error handling. Floating point division by zero DOES NOT stop program execution. As with other floating point errors, it sets a bit in

	the FPERR error variable to indicate an error. Division by zero returns a result of infinity. See "TFBASIC Floating Point".
Editing	Editing is done on the host computer in the TFTools IDE. The TFX-11 board is not running an interpreter and therefore has no facilities for editing.
Errors	A list of error statements and their causes can be found on the "TFBASIC Error Messages" summary page. An error will stop program execution and display a message unless an ONERR command has been executed by the program. The ONERR command is detailed in the "TFBASIC Language Reference" section.
Floating point	Single precision floating point math (IEEE 754) is available in TFBASIC along with a number of trigonometric functions. See "TFBASIC Floating Point" for more information.
Hexadecimals	See **Radix**
IDE	Acronym for Integrated Development Environment
I/O: Analog / Digital	Standard dialects of BASIC use the keyboard and disk storage for data input. In the Tattletale these inputs are augmented by the logger's analog and digital inputs. A number of new commands and functions have been designed to deal with these inputs and outputs simply and efficiently. The analog command to get a value from an A/D channel is CHAN(X), where X is the channel to be read. The digital I/O input read command is PIN(X), and the digital output commands are PSET, PCLR, and PTOG. See the language reference for more information.
Labels, assembler	Labels can be used in the assembly code for flow control and to define local variables. Assembler labels can be up to 32 characters long, must begin with a letter or an underscore (_), and may end with a colon. The only valid characters in a label are upper and lower case characters, the numbers and underscore. The label name must start in the first column of the line. Assembler label names are not required to be terminated with a colon when defined.
Labels, TFBASIC	TFBASIC labels can be up to 32 characters long, must begin with a letter, an underscore (_) or the @ symbol and end with a colon. The only valid

characters in a label are upper and lower case characters, the numbers, underscore and @. TFBASIC labels DO NOT have to begin in the first column of the line. **Line numbers are not acceptable as labels!**

Line Forms There are almost no immediate commands and line numbers are not allowed. Blank lines are permissible and are encouraged for readability.

Line Numbers Line numbers are not allowed in TFBASIC.

Logical Operators The Tattletale supports the two logical operators, AND and OR, which are used for both bit-wise operations and logical connectives. Unlike most other BASIC dialects, the operators are not spelled out, but instead are represented by the symbols "&" for AND, and "|" for OR.

Multiple Statements TFBASIC does not support the use of colons to allow multiple statements on a single program line.

Octal See **Radix**

Overflow Overflow errors are detected during the evaluation of an expression when the intermediate value becomes greater than the maximum long integer (four byte integer) value of 2,147,483,647, or less than the minimum long integer value of -2,147,483,648. Overflow or underflow can occur in floating point numbers if the intermediate value is outside the range ±1.175494E-38 to ±3.402823E+38. See the "TFBASIC Floating Point" section for details. Integer overflow errors cause the program to stop executing and display the "HOW" message. Refer to the "Errors" heading in this section for more information on error handling.

Quotation Marks Single quotes are used to enclose character constants, double quotes enclose character strings. Using single quotes for character strings may generate an error or cause unexpected behavior.

Radix The radix is the number base used to interpret numeric constants, typically either decimal, octal, binary, or hexadecimal. Decimal is the default Radix, and there is no way to globally change this. Any number with a radix other than decimal must be specifically identified using the following techniques: (NOTE : The identifying characters are not case sensitive.)

(Base 16) **Hexadecimal.** Hexadecimal (Base 16) numbers can be entered in TFBASIC by preceding the number with '&H'. The entered number must be unsigned and may include up to eight hexadecimal characters. It will be treated as a signed 32-bit two's-complement number internally. TFBASIC assembly code accepts H', $, 0xxh, in addition to the above. The H' prefix is preferred for designating hex numbers in assembler.

(Base 8) **Octal.** Octal (Base 8) numbers can be entered in TFBASIC by preceding the number with &O where O is the character o, not the digit zero. The entered number must be unsigned and may include up to eleven octal characters. It will be treated as a signed 32-bit two's-complement number internally. TFBASIC assembly code accepts Q' xxQ, and xxO in addition to the above. The Q' prefix is the preferred method for designating octal numbers in assembler.

(Base 2) **Binary.** Binary (Base 2) numbers can be entered in TFBASIC by preceding the number with '&B'. The entered number must be unsigned and may include up to 32 binary characters. It will be treated as a signed 32-bit two's-complement number internally. TFBASIC assembly code accepts B', %, and xxB in addition to the above. The B' prefix is the preferred method for designating binary numbers in assembler.

(Base 10) **Decimal.** Decimal (Base 10) is the default radix in TFBASIC, and therefore requires no identifying code. The entered number must be in the range -2,147,483,647 to 2,147,483,647. It will be treated as a signed 32-bit two's-complement number internally. TFBASIC assembly code has decimal as its default radix, but also accepts D' and xxD as radix identifiers.

Relational Operators TFBASIC supports seven relational operators which are used for comparing two values. Relational operations return 1 if the result of the comparison is true, and 0 if the result is false. See the "TFBASIC Floating Point" section for details about comparisons involving floats. The relational operators are :

<	less than	A<B
<=	less than or equal to	A<=B
>	greater than	A>B
>=	greater than or equal to	A>=B
<>	not equal to	A<>B
><	not equal to	A><B
=	equal to	A=B

SFLASH	This is the main non-volatile storage area. It is a Serial Electrically Erasable Programmable Read Only Memory. To help protect the user from data loss due to program errors the SFLASH interface is designed to only be read out and erased when the TFX-11 is connected to the parallel or serial port. The SFLASH will retain the data for up to ten years with no power applied.
Strings	See **Variables**
String Operators	TFBASIC supports string operations, and includes the functions MID$( ) and LEN$( ). See language reference for details on string functions.
Tabs	Tabs are just fine in TFBASIC, and are used to increase readability.
Timing	In standard BASIC dialects, there is little need to pace a program (the sooner it's over, the happier you are!). In a logging / control application, however, program timing is critical. Timing functions in the Tattletale are handled by the SLEEP and HYB commands. SLEEP puts the logger in a low-power mode for an integral number of 10 ms steps from the wake-up of the previous SLEEP command. This not only provides the necessary timing, but also ensures that the logger is in a low-power mode during the interval. HYB puts the Tattletale in an ultra low power (dormant) mode with wakeup at a fixed time in the future or at regular intervals. HYB is especially appropriate where the application has long periods of quiescence with periodic short bursts of logging and control activity.
UEEPROM	This is a small block of 128 bytes of non-volatile data storage located inside the HC-11 processor. It is accessible only from TFBASIC and can be read and written at any time using VSTORE and VGET. It is specifically meant for calibration or configuration parameters that are unique to the particular instrument. This allows a single program to run on different TFX-11s without having to have different versions of the program that depends on sensor calibration or environmental factors. This data will be retained with no power applied.
Variables	**Integers.** Variable names up to 32-characters long are allowed in TFBASIC. All variables not specifically typed are 4 byte integers **IEEE 754 Floating Point.** Variables intended to be floating point variables must have a **!** suffix.

Strings. Variables intended to hold strings must have a **$** suffix.

Arrays. Arrays must be declared using the DIM statement. An array may be either floating point or integer. Up to two dimensions are allowed and total array size is limited to the size of available memory. String arrays are not allowed in TFBASIC.

White Space

White space is stripped out (except in strings!) before it is sent to the Tattletale. White space is necessary around command, variable and array names.

INDEX