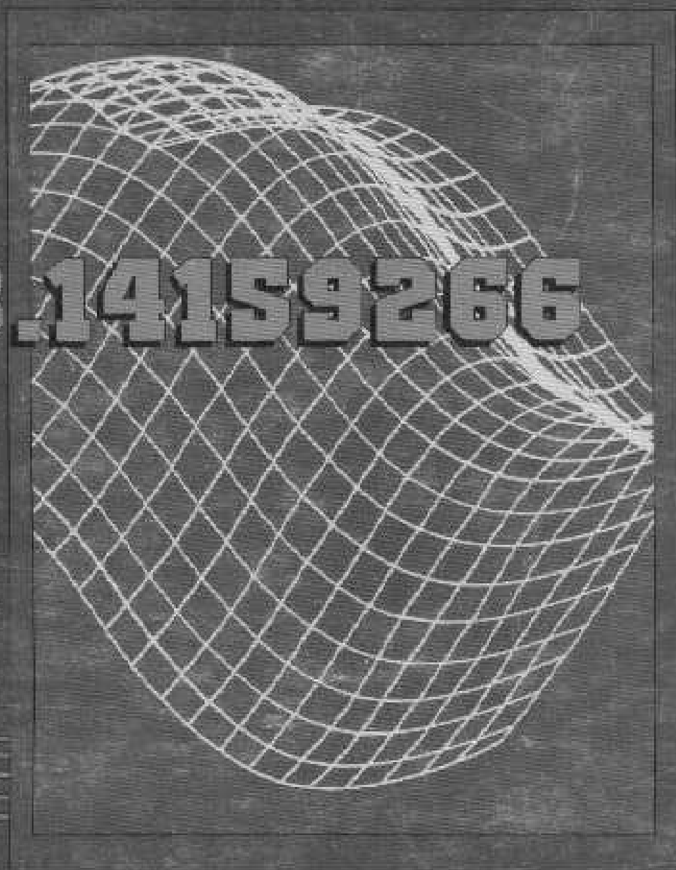
insoft®

TransFORTH

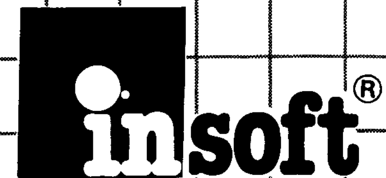
Language Reference Manual

COS
ATN
EXP
LOG
ENG
SCI
READLN
ARRAY
DO
LOOP
BEGIN
WHILE
REPEAT
IF
ELSE
THEN



TransFORTH][B™

by Paul Lutus



For Apple II, Apple II Plus, Apple IIe, Apple III Emulation Mode

TransFORTH II B™
LANGUAGE MANUAL

Disclaimer of all Warranties And Liabilities

Insoft Inc. and Paul Lutus make no warranties, either expressed or implied, with respect to the software described in this manual, its quality, performance, merchantability or fitness for any particular purpose. This software is licensed "as is". The entire risk as to the quality and performance of the software is with the buyer. Should the software prove defective following its purchase, the buyer (and not INSOFT INC., or Paul Lutus, their retailers or distributors) assumes the entire cost of all necessary servicing, repair or correction and any incidental or consequential damages. In no event will INSOFT INC. or Paul Lutus be liable for direct, indirect, incidental or consequential damages resulting from any defect in the software even if they have been advised of the possibility of such damages. Some states do not allow the exclusion or limitation of implied warranties or liabilities for incidental or consequential damages, so the above limitation or exclusion may not apply to you.

The word Apple and the Apple logo are registered trademarks of Apple Computer Inc.

Apple Computer Inc. makes no warranties, either expressed or implied, regarding the enclosed computer software package, its merchantability or its fitness for any particular purpose.

DOS 3.3 Copyright 1979-1981 Apple Computer, Inc.

This manual is copyrighted. All rights are reserved. This document may not, in whole or part, be copied, photocopied, reproduced, translated or reduced to any electronic medium or machine readable form without prior consent, in writing, from Insoft Inc.

Notice

Insoft Inc. and Paul Lutus reserve the right to make improvements in the product described in this manual at any time and without notice.

© 1982 by Insoft Inc.
7933 S.W. Cirrus Dr.
Beaverton OR 97005
(503) 641-5223

Manual written by Phil Thompson

TABLE OF CONTENTS

Disclaimer and Warranty

Table of Contents — Page

CHAPTER ONE: INTRODUCTION

Introduction to TransFORTH — *1-1*

System Requirements — 1-7
What You'll Need to Know — 1-8
About This Manual — 1-9
Starting Up — 1-12

CHAPTER TWO: STARTING TransFORTH

Starting Up — *2-1*

The Data Stack — 2-4
Numbers — 2-8
Manipulating Numbers on the Stack — 2-9
More Words — 2-11
Mathematical Operations — 2-13
Overflow/Underflow — 2-14
Printing Text — 2-15
Summary — 2-16
Problems — 2-17

CHAPTER THREE: DEFINING NEW WORDS

Immediate Mode Execution — *3-1*

Defining New Words — 3-1
Forgetting Words — 3-9
Miscellaneous Thought on Word Definitions — 3-10
Summary — 3-14
Problems — 3-14

CHAPTER FOUR: LOOPS AND TESTS

DO–LOOP 4-1

The Return Stack 4-4
Comparing Numbers 4-5
Decision and Branching Words 4-8
Summary 4-20
Problems 4-21

CHAPTER FIVE: THE TEXT EDITOR

Cursor Movement 5-1

Introduction: Using the Text Editor 5-1
The Text Editor 5-2
Program Compilation 5-10
Comments 5-11
Examples 5-12
Memory Considerations 5-16
Summary 5-18

CHAPTER SIX: DATA STRUCTURES

Variables 6-1

Number Format and Storage 6-5
Arrays 6-7
Strings 6-17
Combining Text and Numerical Data in an Array 6-29
Summary 6-31
Problems 6-32

CHAPTER SEVEN: MISCELLANEOUS WORDS AND FUNCTIONS

Screen Display Words 7-1

Number Formatting 7-2
Program Control Words 7-4
Saving the TransFORTH System 7-6
Miscellaneous Words 7-8
Scientific Functions 7-14
Summary 7-18
Problems 7-19

CHAPTER EIGHT: INPUT AND OUTPUT

The I/O System 8-1

Apple DOS Disk Access 8-11
Using Textfiles for Data Storage 8-12
Saving the Contents of Arrays 8-13
Overlays 8-14
Summary 8-17
Problems 8-18

CHAPTER NINE: GRAPHICS

High Resolution Graphics 9-1

Turtlegraphics 9-10
Larger Graphics Programs 9-15
Screen "Dumps" and Saves 9-16
Low Resolution Graphics 9-17
Summary 9-21
Problems 9-22

CHAPTER TEN: APPLE //e AUXILIARY MEMORY

Installing the Auxiliary Memory Features 10-1

Understanding Auxiliary Memory 10-1
Using Auxiliary Memory 10-2
Saving High-Resolution Pictures in Auxiliary Memory 10-8
Summary 10-9

APPENDIX A: WORD LIBRARY LISTING

APPENDIX B: SYSTEM MEMORY MAP

APPENDIX C: "TECHNICALITIES"

APPENDIX D: TransFORTH FILES

APPENDIX E: DIFFERENCES BETWEEN TransFORTH][AND TransFORTH][B

APPENDIX F: ASCII CHARACTERS

APPENDIX G: INDEX

CHAPTER ONE: INTRODUCTION

CHAPTER TABLE OF CONTENTS: Page

Introduction to TransFORTH ***1-1***

Why TransFORTH 1-1
A Family of Languages 1-3
Comparison with TransFORTH] [1-3
Comparison with GraFORTH 1-3
Comparison with Standard Forth 1-4
Speed 1-5
Error Handling 1-6

System Requirements ***1-7***

Hardware 1-7
Recommended Peripheral Options 1-7
(No) Firmware Requirements 1-7

What You'll Need to Know ***1-8***

Minor Modifications in DOS 3.3 1-9

About This Manual ***1-9***

The Chapters 1-9
The Appendices 1-10
In Reading This Manual 1-11
Request for Feedback 1-11

Starting Up ***1-12***

Registration Card and Product Replacement 1-12
Making Back-ups 1-12
Running the Demonstration Program 1-13
Learning and Using TransFORTH 1-13

Introduction to TransFORTH

The Apple computer has come a long way since it was first introduced in 1977. It was originally designed with an Integer version of the Basic language, some graphics and sound features, a simple access to machine language, and 4 to 16K of RAM memory. Since that time, an incredible number of new features have been added or made available, including more memory, Applesoft Basic, disk drives, 8Ø-column video cards, CP/M, keyboard enhancers, speech synthesizers, even more memory.... The list goes on and on.

One aspect of this expansion is in programming languages being created for the Apple. New versions of Basic, as well as other languages with names like Pascal, C, Lisp, and Forth, seem to be springing up every day. Why bother with another programming language?

Every language has its strengths and weaknesses. Integer Basic is easy to learn, but it cannot handle floating-point numbers, and its reliance on line numbers and GOTOs can sometimes make programs difficult to follow. Applesoft Basic adds floating-point capabilities and a few other features, but shares Integer Basic's lack of structure. Pascal provides a clear structure, using a variety of looping and conditional tests. Pascal also includes a wider variety of data types, and allows the user to give programs and subroutines English names. Unfortunately, Pascal is large and sometimes cumbersome, and it is not interactive like Basic, i.e. you cannot enter and execute Pascal commands directly from the keyboard.

Why TransFORTH?

TransFORTH was developed as a general purpose scientific and business language for the Apple. It is based loosely on the Forth language, but with changes and enhancements designed to make it much more familiar to the average Apple owner. The resulting system then contains the best of several worlds. To be specific:

* TransFORTH is modular: Programs are broken into manageable segments, called "words", which can be given easy-to-remember English names.

* TransFORTH is structured, with a variety of looping and branching constructs that help make programs more readable.

* TransFORTH uses floating-point numbers with 9 decimal digits of accuracy, and includes a number of scientific and mathematical functions.

* TransFORTH is fully compiled, producing machine language code, increasing speed and efficiency.

* TransFORTH is interactive: Commands can be entered and run directly from the keyboard, unlike most compiled languages.

* TransFORTH can be customized: You can create your own special-purpose commands, and add these permanently to the language.

* TransFORTH uses Reverse Polish Notation, a method of entering commands that gives you greater control over the tasks the system performs, and allows you to evaluate complicated expressions without using parentheses.

* TransFORTH is compact, about one-eighth the size of the Pascal system. On 64K systems, you have nearly as much free RAM for programs and data as with Basic.

* TransFORTH includes a sophisticated Input/Output system, for routing data between disks, memory, and printers and other peripherals.

* TransFORTH is DOS 3.3 compatible: Programs and data can be saved as standard DOS binary files or textfiles. This means TransFORTH can access data from other DOS programs.

* TransFORTH programs can be saved as stand-alone systems, which are executable DOS binary files. These files can be run by themselves, and do not require the original TransFORTH system.

* TransFORTH includes a variety of graphics features, including high-resolution and low-resolution graphics, Turtlegraphics, and text printing on the graphics screen.

* TransFORTH can make use of the auxiliary memory found in Apple //e computers with an extended 8Ø-column text card.

A Family of Languages

TransFORTH is part of a family of languages developed by Insoft. TransFORTH][was the first of the family to be introduced. The next one was GraFORTH. GraFORTH is similar to TransFORTH in many ways, but is designed for different needs. The product which this manual describes, TransFORTH][B, is Insoft's new updated version of TransFORTH][. All of these languages are related to each other and to the Forth language in general.

Comparison with TransFORTH][

TransFORTH][B is a direct outgrowth from TransFORTH][, with several specific enhancements and changes. TransFORTH][B works with both the Apple][and //e. It includes a modified disk operating system (DOS) that will automatically load itself into high memory on an Apple //e or an Apple][with a language card or RAM card, freeing up more room for programs and data. TransFORTH][B also uses a somewhat different method of variable storage, which helps make programs shorter, faster, and easier to read. All high-resolution graphics routines are now built into the language. This means TransFORTH][B is completely independent of which Basic resides in the computer. Peripheral card input and output using TransFORTH's I/O system has been made more general purpose and will work with any standard peripheral cards. A number of minor changes have also been made.

Comparison with GraFORTH

While the main emphasis of TransFORTH is in business and scientific programming, GraFORTH is intended for the graphics, educational, and game markets. GraFORTH is a very fast integer language, similar in basic structure to TransFORTH, but with a large number of graphics commands built into the system. These include standard line and point graphics, turtlegraphics, fast character font and block graphics, and 3-D graphics. GraFORTH lacks TransFORTH's floating-point, scientific, array, and sophisticated I/O capabilities. On the other hand, TransFORTH does not have as many graphics features as GraFORTH has.

Comparison with Standard Forth

TransFORTH is a fully compiled version of the Forth language, i.e. TransFORTH programs are converted directly to machine language instructions for speed. Most other versions of Forth (as well as most Apple languages) use a run-time interpreter, slowing programs down.

Two "standards" have been developed for the Forth language: Forth-79 and Fig-Forth. The two standards are very similar, and represent a common ground for writing programs in Forth. Anyone familiar with the Forth standard can write a program on any computer if the version of Forth used adheres to the standard. Unfortunately, the standard then cannot make use of the special features of any one computer.

TransFORTH is instead designed specifically for the computer you own, the Apple. While the general structure of Forth is used, no attempt was made to closely follow the Forth-79 standard. This was based on the assumption that most Apple owners have no great need for writing programs compatible with other computers, and prefer a language which is well suited to the Apple. Perhaps TransFORTH can then be thought of as a language similar to Forth, rather than as a version of Forth.

TransFORTH includes most of the main Forth concepts: Reverse Polish Notation, data and return stacks, word library, and the looping and testing constructs (DO - LOOP, IF - ELSE - THEN, BEGIN - UNTIL, etc.). Several standard TransFORTH words use names borrowed from Apple Basic, rather than Forth, to make them more familiar to Apple users. The contents of variables are stored and retrieved using a different, more readable, method. The looping and branching commands can be run directly from the keyboard (outside of a colon definition), unlike most Forths.

TransFORTH uses standard Apple DOS files rather than the Forth "disk block" structure. DOS compatibility means that data can be shared with other DOS-based programs, and reduces the time it takes to learn TransFORTH.

While most Forths handle only integers, TransFORTH is a fully floating-point language. All numbers on the stack are stored internally in a five-byte floating-point format. Numbers can be saved in memory, however, in one or two byte integer format as well as in floating-point. TransFORTH does not have the Forth <BUILDS - DOES> construct, but instead includes a versatile

built-in array declaration. The array type can be easily used for nearly all common data applications.

Speed

These days, everybody seems to be very concerned with speed. A common practice is to ask for "benchmark" results, which are the times it takes a language or program to perform usually very simple tasks. Unfortunately, some benchmark-type tests are often terribly misleading.

For example, if a simple do-nothing loop in TransFORTH is compared to a similar loop in Applesoft:

```
TransFORTH:  30000 0 DO LOOP
Applesoft:   FOR N=1 TO 30000 : NEXT
```

the two will take about the same amount of time to run. However, when longer (and more realistic) programs are compared, TransFORTH's speed advantage becomes both noticeable and significant. This is because Applesoft gets bogged down hunting for variables to read and line numbers to jump to. Since TransFORTH is fully compiled, it makes direct machine language calls and jumps.

A more realistic benchmark test, "Eratosthenes Sieve" prime number program, is described in the article "A High-Level Language Benchmark" in the September 1981 issue of Byte magazine. You may wish to find the issue and read this informative article. A version of this program written in TransFORTH can be found on the TransFORTH system disk in the textfile "PRIMES". Here is a table containing the times (in seconds) of TransFORTH and other Apple languages running the benchmark:

TransFORTH:	94
Applesoft Basic:	280.6
Integer Basic:	232
GraFORTH:	17.9
Fig-Forth:	20.0
Apple Pascal:	51.6

Notice that TransFORTH is much faster than either version of Basic.

The PRIMES program does not require floating-point capabilities. TransFORTH is slower than some integer languages because it handles five-byte floating-point numbers with every stack

manipulation, rather than two-byte integers. (Apple Pascal loses much of its speed advantage if the programs being compared rely heavily on floating-point and scientific calculations.)

Error Handling

TransFORTH is like other versions of Forth, but unlike languages such as Pascal, in that it does not include comprehensive error checking. TransFORTH does check for most errors, such as division by zero, missing labels, etc. However, for functions that expect numbers within a given range, TransFORTH does not check that the numbers are in that range. For example, if you declare an array with 100 elements, then ask for the value of the 101st element, TransFORTH will not print an error message. It will instead return a nonsensical value. Other functions that expect certain numbers may have unpredictable results if given a "wrong" number.

Range checking was left out of TransFORTH for a good reason: It uses time and memory space, and has no value to a program if the program already works. This of course means that you must be careful to always provide the correct values. If you need error checking, you can write routines to do this. If you don't need error checking, it is not included.

In addition, TransFORTH provides you with the flexibility to easily read from or write to any location in Apple memory. This includes the ability to intentionally or accidentally overwrite a portion of the TransFORTH language itself. If this ever happens, the system may "hang", and you will have to reboot. If the language ever does hang up and can't be recovered with Reset, check your program. You'll probably find that a value was written to an "illegal" area of memory. The memory map in Appendix B shows how TransFORTH uses Apple memory.

System Requirements

Hardware

TransFORTH requires that you have the following miminum hardware components:

An Apple][or Apple][Plus computer with 48K RAM
  or an Apple //e
  or an Apple /// with an Apple][emulation disk
One DOS 3.3 disk drive with controller
A video monitor, and/or
A TV with an RF modulator

Recommended Peripheral Options

In addition to the above, you may also want to use:

A 16K RAM card or language card (for an Apple][), to provide extra memory
A color monitor or TV, for color graphics displays
An 8Ø-column video display card
A second disk drive

(No) Firmware Requirements

TransFORTH was written in 65Ø2 machine language using the ALD System Assembler which was written by Paul Lutus and is also available from Insoft. All floating-point and graphics routines are internal and are therefore completely independent of either Apple Basic (Integer or Applesoft). TransFORTH boots directly from the 'monitor', without a Basic 'HELLO' program, as you will notice by the asterisk prompt (rather than the Basic prompt) during bootup. This makes the boot program independent of any resident language in ROM, avoiding the differences between the various Apple II computers which are sometimes troublesome to software.

What You'll Need to Know

This book is intended to be a complete tutorial and reference manual for TransFORTH][B. It does make a few assumptions, however, that you already know some things about your Apple or can learn about them from the Apple manuals.

You should be generally familiar with the Apple computer itself: the Return key, arrow keys, Reset, the video display.... It's also helpful to have a little programming experience with either Pascal, or Applesoft or Integer Basic. Some of the examples i this manual compare TransFORTH and Applesoft program lines. Even if you know only Pascal, the meaning of the Basic lines should be obvious.

This manual assumes you know what things like "subroutines", "variables", and "arrays" are. How they apply to TransFORTH is described completely, but the general underlying concepts are only outlined here.

TransFORTH uses the standard Apple Disk Operating System Version 3.3, known affectionately as DOS 3.3. If you are at all unfamiliar with the disk operating system and the DOS commands, we suggest you take the time to study the DOS manual which came with your disk drive(s). How well you need to understand DOS depends on what kind of programs you will be writing with TransFORTH.

The manual also does not explain all of the details of Apple's low and high resolution graphics modes. An overview provides most of the basic information. If you've ever used graphics from either Basic or Pascal, you know everything you need to know. If not, a few minutes with the Apple manuals will be time well spent.

Throughout this manual, we'll be referring to 48K Apples and 64K Apples. If you have an Apple][or Apple][Plus with a 16K RAM card or language card, or if you have an Apple //e, then you have a 64K Apple. If you have an Apple][or][Plus without the memory card, then you have a 48K Apple. TransFORTH automatically makes use of the extra 16K of RAM on 64K Apples.

TransFORTH will also run on an Apple /// in emulation mode. The emulation mode acts like a 48K Apple][Plus.

Minor Modifications in DOS 3.3

As mentioned above, minor modifications have been made to the disk operating system that is provided on the TransFORTH system disk. A normal DOS master disk, when booted, will first determine how much memory is available, and load itself into the highest memory area under 48K. It then enters Basic, and loads and runs a Basic "greeting" program.

The TransFORTH disk, however, also checks for RAM higher than 48K, found on 64K Apples as discussed above. If it finds this RAM, it loads DOS here. It then enters through the Apple system monitor (as shown by the asterisk "monitor" prompt, rather than the Basic prompt), and BRUNs the binary file OBJ.FORTH. This file contains the TransFORTH language system. Running OBJ.FORTH actually starts the language.

About This Manual

This book is designed to serve as both a tutorial and a reference manual. The main body of the manual is written in a tutorial format, with many examples for you to try as you read. (If you bought this book alone, the text will explain what the system does at any particular moment.) At the end of each chapter is a concise summary of the main points in the chapter. In most chapters, a set of example problems (with solutions) follows.

A number of appendices are included, providing more technical information along with a quick-reference guide to many of TransFORTH's features.

The Chapters

Chapter Two introduces the fundamentals of TransFORTH, describing the word library and showing you how to manipulate numbers on the stack, do arithmetic, and print numbers and text.

Chapter Three shows how to define new words, either to customize your system or as part of a larger program.

Chapter Four discusses the various looping, testing, and branching words in TransFORTH.

Chapter Five describes how the text of word definitions can be saved using either the TransFORTH text editor or any DOS textfile editor, then compiled onto the word library.

Chapter Six explains the data structures in TransFORTH (variables, arrays, and strings) and how best to use them.

Chapter Seven describes a variety of TransFORTH words, for printing and number formatting, controlling programs and saving a compiled system to disk, and other miscellaneous functions.

Chapter Eight shows you how to use the I/O system to move data, execute DOS commands directly from TransFORTH, manipulate textfiles, and implement overlays for very large programs.

Chapter Nine discusses the graphics features of TransFORTH, including high-resolution and low-resolution graphics, turtlegraphics, and the provided graphics module.

Chapter Ten is for users that have an Apple //e with the extended 80-column text card. It describes how you can use up to 46K of auxiliary memory on the card for storing data.

The Appendices

Appendix A is a quick-reference guide to every word in the TransFORTH language. If you need to know what a given word does, look here first. A brief description of the word is provided, along with the page number where the word is discussed more fully in the manual.

Appendix B is a memory map of the system, showing how TransFORTH uses memory in both 48K and 64K Apples, and includes a table of useful system locations.

Appendix C provides a discussion of some of the more complicated technical aspects of TransFORTH, including word library structure and memory usage, errors, floating point format, and recursion.

Appendix D describes many of the demonstration and utility files on the TransFORTH system disk, and how to use them for various programming applications.

Appendix E lists the differences between TransFORTH][and TransFORTH][B, to assist users who are updating from the old version to the new.

Appendix F is a table of ASCII characters and values.

Appendix G is the index for this manual. The index includes the main references to all TransFORTH words, chapter and section headings, and general topics.

In Reading This Manual

The tutorial part of this manual is sequential in nature. Each chapter builds on the previous chapter. The easiest way to learn the TransFORTH language is to simply read through the chapters in order. Many people find they learn best by reading the manual twice, first away from the computer to gain a better perspective on the system, then again at the computer while following the examples. If you decide to skip around the manual to learn about your favorite features first, we suggest you at least the read chapter summaries, to see what you might have "missed".

Many of the problems at the end of each chapter provide excellent examples in using the features of TransFORTH in actual programs. Whether or not you "work through" the problems is up to you. Either way, studying and testing the solutions will help you become more comfortable with writing your own programs.

After you've become reasonably accustomed to working with TransFORTH, you will probably want to refer to Appendix D. This section includes a number of miscellaneous handy routines that you can use in your programs.

Request for Feedback

We tried to design this manual to be as complete as possible, but we know that something somewhere may be confusing. If there is something about this manual (or the TransFORTH system) that you don't like, and you think it can be improved, let us know about it. And if there is something that worked especially well for you, let us know about that too, so that we can continue to produce high-quality products.

Starting Up

Registration Card and Product Replacement

You should fill out and return the enclosed product registration card as soon as possible. It registers you as one of our customers, which can improve service if you ever have a problem with the TransFORTH disk. It also keeps you up to date. If we decide to release an update to TransFORTH, then we can let you know about it.

If the TransFORTH disk ever fails to boot for any reason, return the disk to Insoft. If the disk itself is in good condition, we will send you a replacement (or recopy the same disk) at no charge. If the disk is damaged, there is a five dollar charge for replacement.

Making Backups

The TransFORTH disk is not copy-protected. You can make back-up copies for your own use with any standard disk copy program, such as COPYA. The easiest way to make customized system disks is to copy the original, then delete any files you don't need from the copy.

You can also copy the TransFORTH system file by file onto an initialized disk. If you do, however, the DOS on your initialized disk will not make use of the top 16K of memory. (This is no problem if you have an Apple][without a RAM card.) In addition, the TransFORTH language will not boot automatically. You can instead write a Basic greeting program that BRUNs the OBJ.FORTH file to start TransFORTH.

We encourage you to make back-ups immediately, before using TransFORTH. Then put the original in your lead-lined vault. Whenever this manual refers to "your TransFORTH disk", use your working copy of the original. That way, if anything goes wrong (your dog mistakes the disk for a frisbee), another copy of the original can be made.

Running the Demonstration Program

To start TransFORTH, insert the system disk in your disk drive and boot it. (Apple][users turn on the machine and type 6 CTRL-P;][Plus and //e users simply turn the computer on.) In a few seconds, the following will appear on the screen:

```
TransFORTH Demonstration? (Y/N) :
```

Type 'Y' to run the TransFORTH demonstration program. The disk will whir for a few more seconds, then the demonstration will begin. This program was written in TransFORTH, to give you a basic "feel" for what the system can do. For now, rebooting is the simplest way to return to TransFORTH.

Learning and Using TransFORTH

There are only two things you need to do in order to begir learning and using TransFORTH:

1. Be adventurous, inquisitive, and have fun!

2. Turn the page to Chapter Two....

CHAPTER TWO: STARTING TransFORTH

CHAPTER TABLE OF CONTENTS: Page

Starting Up	**2-1**
Background	2-2
Words	2-2
Program Execution	2-3
The Data Stack	**2-4**
Numbers	**2-8**
Manipulating Numbers on the Stack	**2-9**
More Words	**2-11**
Mathematical Operations	**2-13**
Overflow/Underflow	**2-14**
Printing Text	**2-15**
Summary	**2-16**
Problems	**2-17**
Solutions to Problems	2-18

TransFORTH is a powerful language in two ways. It has a powerful structure, giving you great flexibility and control over your Apple. It also has powerful features, enabling you to perform sophisticated operations more easily.

In the next three chapters, we'll introduce the basic structure of TransFORTH, giving you the groundwork for learning the language. This chapter will discuss using the stack, printing text, and performing operations using Postfix notation. Chapter Three will describe how to write programs by adding new words to the word library, and Chapter Four will introduce the basic decision and branching words. Individual features will be more fully described in later chapters.

This tutorial contains numerous command and program examples. If you have purchased the TransFORTH system software along with this manual, we strongly encourage you to try these examples. They provide the difference between "book" learning and "experience" learning. As you enter examples, you may mistype something and find yourself in a situation you don't quite yet know how to get out of. If you can't recover things properly by pressing Reset, don't worry: The power switch was put on the Apple for a good reason! Just turn the power off and reboot again (Apple //e users can reboot by pressing CTRL-open apple-Reset), then try to figure out what went wrong. We'll help you along the way.

Starting Up

Insert the TransFORTH system disk in the drive and boot the disk. In a few moments, you should see something like:

```
TransFORTH Demonstration? (Y/N) :
```

We assume that since reading the first chapter, you've already run the demonstration program. Answer "N" to the prompt. The screen will clear and the TransFORTH header will be reprinted along with a "Ready" prompt:

```
TransFORTH ][ B (C) P. Lutus 1982

Ready
```

If you have an 80-column display card (in either slot 3 or the //e auxiliary slot), the 80-column display will be automatically turned on. If you don't have a display card, the 40-column Apple screen is used. On an Apple][, the above displays will appear

as upper-case only characters.

TransFORTH is designed to print both upper and lower case to any external device or the Apple //e video screen, and upper case only to the Apple][display. The program examples in this manual will use both upper and lower case.

The "Ready" prompt is displayed whenever TransFORTH is ready for you to enter commands. If at any time you don't see the "Ready" prompt when you think you should, it may be time to start wondering....

Before continuing, let's back up for a moment to get a better look at what TransFORTH is.

Background

As mentioned in Chapter One, the TransFORTH system is actually a machine language program stored on disk with the name OBJ.FORTH. Booting the disk runs this program.

The language can be divided into two main parts. The first part consists of the compiler and low-level system routines. These routines keep track of internal housekeeping and do the things that need to be done without a lot of fanfare. The second part is the word library. This is the visible, active part of TransFORTH.

Words

The word library is made up of a large number of TransFORTH "words". You can see this list of words by typing the word "LIST". LIST is a TransFORTH word that lists all of the TransFORTH words. (The listing stops every 16 words. You can press CTRL-C to stop the listing, or any other key to continue.)

```
Ready LIST

NOTE
NEGATE
ABS
SIGN
CALL
PREG
YREG
XREG
AREG
POKEW
POKE
.
.
.
```

Everything in TransFORTH is either a "word" or a number. Words can be variables, arrays, commands, subroutines, or programs. Each TransFORTH word accomplishes a particular task. For example, the word "NOTE" plays a note, the word "+" adds two numbers together, and the word "PRINT" prints text. Programs are written, not by entering program "lines" (as in Basic), but by stringing TransFORTH words together.

As you write programs, you will be defining new words with their own unique names. A TransFORTH word name can be any string of ASCII characters, except that it cannot begin with a control character, and there cannot be any spaces or carriage returns in the name. Spaces are used to separate words, and a carriage return is used to end a line, telling TransFORTH to compile the line into memory, then either execute it or save it as part of a new word. Since spaces are used to determine the end of one word and the beginning of another, the spaces in the program examples are very important.

Word names can be in either upper or lower case (or both). However, they must be typed in the same way each time. For example, the name "TEST" is not the same as "test" or "Test".

Program Execution

Here is a program line written in Basic:

```
PRINT 5.6 + 12.8
```


Consider how the computer reads the line and how it acts upon it. First the word PRINT is read, but the computer doesn't yet know what to print. The "5.6" follows, but still nothing can be done because there are more characters to read. With the "+" sign, the computer knows that something is to be added to 5.6. The "12.8" is read, and the end-of-line is finally reached. Only now does the computer have enough information to act upon. It can now add the 5.6 and 12.8 together, then print the result. Note that PRINT was the first item on the line, but the last item to be executed.

Unlike Basic, in most cases TransFORTH executes words in the same order they are read. When it reads a word from an input line, it usually acts on that word immediately. For example, when TransFORTH reads the word "+", it wants to add two numbers together right then and there. This means that the numbers to be added must be waiting somewhere for it. Where do they wait? On the "data stack".

The Data Stack

The data stack is simply a stack of numbers, arranged much like a stack of dinner plates or a deck of cards. Numbers (like plates) can be placed on the top of the stack, or removed from the top of the stack. Most TransFORTH words use the data stack in some way. Some words place numbers on the stack, some words remove numbers, and some do both. For example, the word "+" removes two numbers from the top of the data stack, adds them together, then places the result back onto the top of the stack.

Simply entering numbers places them on the data stack. Type:

```
Ready -4  5.6
```

The numbers -4 and 5.6 have been placed on the data stack. To verify this, type "STACK". STACK is a TransFORTH word that causes the contents of the stack to be displayed after every command or program is executed:

```
Ready STACK

[ -4 ]
[ 5.6 ]
Ready
```

Now place the number 12.8 on the data stack:

Ready 12.8

[-4]
[5.6]
[12.8]
Ready

Note that the stack display is "upside-down". The 12.8 was the last number placed on the stack, so it should be on the top of the stack, but it is displayed below the other numbers. Here's why: Stacks and "top-of-stack" are standard computerese conventions, and we didn't want to break tradition by calling it the "bottom-of-stack". However, the TransFORTH stack can hold up to 55 numbers at a time, while the Apple can only display 24 lines. With an upside-down stack display, even if the stack contains too many numbers and some scroll off the screen, the top stack number (the most accessible value) will still be on the screen.

If you type the word STACK again, the stack display feature will turn off. Hence, STACK "toggles" the stack display on and off. You may want to type STACK a couple of times to verify this, but before continuing with these examples, have the stack display turned on.

With numbers now on the data stack, you can add two of them together. Type:

Ready +

[-4]
[18.4]
Ready

The word "+" removed the 5.6 and the 12.8 from the stack, added them together, then placed the result, 18.4, back onto the stack. Typing "+" again will add the -4 and 18.4 together:

Ready +

[14.4]
Ready

The result has been placed on the stack, but nothing has been done with it yet. If the stack display were off, you would have to print the number in order to see it. The TransFORTH word "."

(a period) removes the top number from the stack and prints it:

Ready .
14.4

Putting numbers on the stack, adding them, and printing the sum can be combined on one line:

Ready 5.6 12.8 + .
18.4

Notice how this compares with the Basic example shown above. The Basic format is <number> <add> <number>, while the TransFORTH format is <number> <number> <add>. This notation, where the numbers (or "operands") precede the plus sign (the "operator"), is called Postfix or Reverse Polish Notation (RPN), and is used in all versions of Forth as well as in many Hewlett-Packard calculators.

This notation might seem "backwards" to people who may be more familiar with Basic or Pascal. Perhaps a better adjective would be "sideways", considering that some computer languages, such as Lisp, place the operator before the operands: <add> <number> <number>. There are two significant advantages to RPN: 1) You have greater control over the machine because you specify exactly what order in which you want the machine to perform tasks. 2) You can evaluate complicated mathematical expressions without using any parentheses.

For example, suppose you want to add 3 to 5, then add 6 to 7, then multiply the two sums together. In Basic, you would enter a line like the following:

PRINT (3 + 5) * (6 + 7)

The parentheses were needed to prevent Basic from performing the multiplication first. With TransFORTH, no parentheses are needed:

Ready 3 5
[3]
[5]

Ready +
[8]

```
Ready 6 7
[ 8 ]
[ 6 ]
[ 7 ]

Ready +
[ 8 ]
[ 13 ]

Ready *     ( The word "*", an asterisk, multiplies the top two
              stack values. )
[ 104 ]

Ready .
104
Ready
```

This example was "unfolded" so that you could see the individual operations. For most applications, you would enter the example on one line:

```
Ready 3 5 + 6 7 + * .
104
Ready
```

You can perform most operations using as few or as many lines as you like. There are a few exceptions, and these will be noted.

You now know how to place numbers on the stack, print numbers, and add or multiply numbers. Here are a few more examples to familiarize you with stack operations:

```
Ready 1 2 3        ( Place the numbers 1, 2, and 3 on the stack. )
[ 1 ]
[ 2 ]
[ 3 ]

Ready 57 43        ( Place 57 and 43 on the stack. )
[ 1 ]
[ 2 ]
[ 3 ]
[ 57 ]
[ 43 ]
```



```
Ready .            ( Print the 43. )
43
[ 1 ]
[ 2 ]
[ 3 ]
[ 57 ]

Ready . .          ( Print both the 57 and the 3. )
573
[ 1 ]
[ 2 ]
```

(Note that the 57 and the 3 were printed without any separating spaces. We'll show how to insert spaces shortly.)

```
Ready 99           ( Place 99 on the stack. )
[ 1 ]
[ 2 ]
[ 99 ]

Ready *            ( Multiply the 2 and the 99 together. )
[ 1 ]
[ 198 ]

Ready 2.2 +        ( Add 2.2 to the 198 on the stack.)
[ 1 ]
[ 200.2 ]

Ready .            ( Print the result. )
200.2
[ 1 ]
```

Numbers

TransFORTH is a full floating-point language. It keeps nine decimal digits of accuracy, and can handle positive and negative numbers as large as 1E38 (1 times 10 to the 38th power) and as small as 1E-38. All values on the stack are stored as floating-point values.

Numbers accepted by TransFORTH use the following general guidelines:

1. At least one numeric digit must appear somewhere in the number, along with any other appropriate symbols.
2. Commas, dollar signs, and plus signs can appear anywhere in the number, and are ignored.
3. A minus sign appearing anywhere before the optional exponent causes the number to be negative.
4. A decimal point is not needed if there are no digits to the right of the decimal.
5. The letter E marks the beginning of the optional exponent.
6. A minus sign after the E causes the exponent to be negative.

Here are some examples using the proper numeric format:

```
0
827
-25
14E4
90.1E6
$600,000
-1.23456789E12
17.9E-8
```

Notice that, while still following the above guidelines, a few unusual formats are accepted by TransFORTH. For example:

```
2+3       Without spaces to separate the characters, this is
read as 23.
2-3       The minus sign causes this to read as -23.
$4$5$6$   This reads as 456.
```

Manipulating Numbers on the Stack

Following the earlier examples, the number 1 was left on the stack. Suppose you want to remove the 1 from the stack without printing it. There are a number of TransFORTH words designed for manipulating stack values, including duplicating, swapping, and dropping values from the stack. While these operations may not sound terribly exciting, you will find that they are very useful for arranging numbers so that they can be correctly used by other TransFORTH words.

The word DROP simply removes the top number from the stack. The

number is lost.

The word DUP duplicates the top stack value, placing the duplicate on the top of the stack.

The word SWAP trades the positions of the top two numbers on the stack.

The word OVER copies the second number on the stack to the top.

These examples continue from the previous ones:

```
[ 1 ]

Ready 5 6      ( Place 5 and 6 on the stack. )
[ 1 ]
[ 5 ]
[ 6 ]

Ready DUP      ( DUPlicate the top stack value. )
[ 1 ]
[ 5 ]
[ 6 ]
[ 6 ]

Ready DROP     ( DROP, or remove, the top number from the stack. )
[ 1 ]
[ 5 ]
[ 6 ]

Ready SWAP     ( SWAP the top two stack values. )
[ 1 ]
[ 6 ]
[ 5 ]

Ready OVER     ( Copy the second stack value OVER the first to
top-of-stack. )
[ 1 ]
[ 6 ]
[ 5 ]
[ 6 ]

Ready DROP . DROP DROP   ( DROP the top stack value, print the
second, and DROP the last two. )
5
Ready
```

Since so many TransFORTH words affect the stack, a

general-purpose diagram for showing exactly how a word uses the stack would be handy. Below is such a diagram. Each word is listed, followed by a "before" and "after" picture of the stack, with any numbers shown as letters. The top of stack is to the right of each list of letters. For example, here is the stack diagram for "+":

Word	Before	After	Description
+	m n	p	p = m + n

This diagram shows that two numbers, m and n are removed from the stack, and one number, p, is placed on the stack. If a before or after picture does not contain any numbers (empty or unused stack), then a dash will be shown instead. (Remember that there may be other stack values beneath the ones used.) Here are the stack diagrams for the words discussed so far:

Word	Before	After	Description
LIST	-	-	Lists the words in the word library.
STACK	-	-	Toggles the stack display on and off.
+	m n	p	p = m + n
*	m n	p	p = m * n
.	n		Prints n.
DUP	n	n n	Duplicates n.
DROP	n	-	Removes n from stack.
SWAP	m n	n m	Swaps position of m and n on stack.
OVER	m n	m n m	Copies m to top of stack.

This same stack diagram is used in Appendix A in describing the operation of every word in the TransFORTH word library.

More Words

There are three other words for manipulating numbers on the data stack. These words are a little more complicated, and are not used as often:

PICK removes a number (call it "n") from the stack, then uses this number n as an index into the stack, copying the nth number to the top of stack. For example, 4 PICK would copy the 4th item on the stack to the top of stack. 1 PICK is equivalent to DUP, and 2 PICK is equivalent to OVER. To return meaningful values, the index number used with PICK should not be greater than the number of values on the stack.

ROLD "rolls" the top three stack values down, copying the 3rd

number to the 4th, the 2nd to the 3rd, the top number to the 2nd, and the 4th value up to the top. ROLU rolls the stack up, in the opposite direction. If less than 4 numbers are on the stack, the operation of ROLU and ROLD can be unpredictable!

Here are examples of PICK, ROLU, and ROLD:

```
Ready 1 2 3 4 5

[ 1 ]
[ 2 ]
[ 3 ]
[ 4 ]
[ 5 ]
Ready ROLD

[ 1 ]
[ 3 ]
[ 4 ]
[ 5 ]
[ 2 ]
Ready ROLD

[ 1 ]
[ 4 ]
[ 5 ]
[ 2 ]
[ 3 ]
Ready ROLU

[ 1 ]
[ 3 ]
[ 4 ]
[ 5 ]
[ 2 ]
Ready DROP DROP

[ 1 ]
[ 3 ]
[ 4 ]
Ready 2 PICK

[ 1 ]
[ 3 ]
[ 4 ]
[ 3 ]
```

```
Ready 4 PICK

[ 1 ]
[ 3 ]
[ 4 ]
[ 3 ]
[ 1 ]
```

Mathematical Operations

There are also a large number of mathematical functions included in TransFORTH's word library. Here is a summary of those functions. Some of these will be described in greater detail in Chapter Seven:

Word	Before	After	Description
+	m n	p	p = m + n (addition)
-	m n	p	p = m - n (subtraction)
*	m n	p	p = m * n (multiplication)
/	m n	p	p = m / n (division)
MOD	m n	p	p = m MOD n (modulo, or remainder after division)
^	m n	p	p = m ^ n (power function)
NEGATE	n	m	m = -n (negate)
ABS	n	m	m = absolute value of n
SIGN	n	m	m = 1 if n > Ø (sign) Ø if n = Ø -1 if n < Ø
SQRT	n	m	m = square root of n
SIN	n	m	m = sine of n (n in radians)
COS	n	m	m = cosine of n (n in radians)
TAN	n	m	m = tangent of n (n in radians)

Word	Before	After	Description
ATN	n	m	m = arctangent of n (m in radians)
LOG	n	m	m = natural logarithm of n
EXP	n	m	m = e ^ n (where e = 2.71828...)
INT	n	m	m = integer portion of n
FRAC	n	m	m = fractional portion of n
PI	-	n	n = pi (where pi = 3.14159...)
RND	n	m	m = random number where Ø <= m < 1. n < Ø generates new random sequence, n = Ø repeats last random number, n > Ø generates new random value.

These words can be combined using the rules of Postfix notation to evaluate complicated expressions. The problems at the end of the chapter provide a few examples of evaluating expressions with TransFORTH.

Quite complicated expressions can usually be evaluated using only the stack, though the gymnastics (with SWAP, DUP, etc.) required to keep track of all of the values can sometimes become confusing. Keep in mind that named variables will be introduced in Chapter Six. By using variables, you can keep stack manipulations simple. For any application, you can decide how much data you want to keep on the stack and how much you want to put into variables.

Overflow/Underflow

The TransFORTH data stack can hold up to 55 numbers. If you intentionally or accidentally attempt to place more numbers on the stack than it can hold, TransFORTH will abort what it is doing and print:

```
R Error : STKO (Press RETURN)
```

This cryptic message stands for "Run Error : STacK Overflow", meaning that the last word or operation has "overflowed" the stack. Since this is never a desirable occurrence, it creates a "fatal" error, which stops whatever command or program TransFORTH is executing. Press the return key, and the TransFORTH system

will reset itself.

If the stack is empty and you execute any words that attempt to remove numbers from the stack, this message will be displayed:

R Error : STKU (Press RETURN)

This stands for "Run Error : STacK Underflow", and is also a fatal error.

Note: In the interests of speed, a few TransFORTH words bypass the usual error checking routines, and rather than producing a stack underflow error, will fool TransFORTH into believing its stack contains 256 values. For example, with an empty stack and the stack display on, type "DROP". This will try to remove a number from the empty stack, and 256 stack values will scroll up the screen. You can type "ABORT" or press Reset to recover. (ABORT will be discussed in greater detail later.)

If the stack ever (apparently) becomes filled with 256 values, you have probably produced an unchecked stack underflow.

Printing Text

So far we've been dealing heavily with numbers and the data stack, but TransFORTH is equally adept at manipulating text information. Printing text is staightforward: Type the word PRINT, the word " (a quote), the text to be printed, and another quote:

```
Ready PRINT " A SAMPLE PHRASE "
A SAMPLE PHRASE
```

Note that since the quote is a TransFORTH word, spaces must appear between the word PRINT and the quote, and between the quotes and the text. A quote can appear in the text to be printed, as long as it is not set apart with spaces on both sides:

```
Ready PRINT " A "BREAKTHROUGH" IN SCIENCE! "
A "BREAKTHROUGH" IN SCIENCE!
```

Two other words are useful when printing text or numbers: CR causes a carriage return to be printed, and SPCE prints a space. Notice the differences in these three examples:


```
Ready PRINT " TRANS " PRINT " FORTH "
TRANSFORTH

Ready PRINT " TRANS " SPCE PRINT " FORTH "
TRANS FORTH

Ready PRINT " TRANS " CR PRINT " FORTH "
TRANS
FORTH
```

Remember that the word "." prints numbers without any spaces. SPCE and CR can be used to provide the spacing:

```
Ready 21 . 45 .
2145

Ready 21 . SPCE 45 .
21 45
```

Summary

Every TransFORTH command is called a "word", and resides in the "word library". The words in the word library can be listed by typing the word LIST.

Most TransFORTH words are executed in the order they are read by the system. A data stack is used to store numbers temporarily. Most TransFORTH words either remove numbers from the top of the stack, place numbers on the top of the stack, or do both. Using Postfix notation, expressions can be evaluated without using parentheses, and the user can completely control what functions the computer performs.

The words introduced in this chapter can be broken into four categories:

Stack Manipulation:
   DUP DROP SWAP OVER PICK ROLU ROLD

Arithmetic:
   + - * / ^ MOD SIN COS TAN ATN LOG EXP SQRT SIGN ABS RND INT FRAC PI

```
Text and Printing:
   . PRINT SPCE CR

Miscellaneous:
   LIST STACK
```

Problems

Translate these lines of Basic code to equivalent TransFORTH program lines. (There may be more than one "correct" solution.)

```
(1)
PRINT 5

(2)
PRINT 6 / 7

(3)
PRINT SQRT(81)

(4)
PRINT (5 + 10) / 2     (The average of 5 and 10.)

(5)
PRINT (68 - 32) * 5 / 9
(This uses the formula  C = (F - 32) * 5 / 9  to convert 68
degrees Fahrenheit to degrees Celsius.)

(6)
PRINT INT( RND(1) * 6) + 1
(A random number between 1 and 6.)
```

Translate these TransFORTH lines to Basic:

```
(7)
Ready 89 12 - .

(8)
Ready 30 9 * 5 / 32 + .
(Converting 30 degrees Celsius to degrees Fahrenheit.)

(9)
Ready 2 3 + 4 5 + * .
```



```
(10)
Ready 2 3 4 + * .

(11)
Ready 2 3 4 * + .

(12)
Ready 3 3 3 * * .

(13)
Ready 3 DUP DUP * * .

(14)
Ready 4 DUP DUP * * .

(15)
Ready 17 SQRT DUP * .
```

Solutions to Problems

```
(1)
Ready 5 .

(2)
Ready 6 7 / .

(3)
Ready 81 SQRT .

(4)
Ready 5 10 + 2 / .

(5)
Ready 68 32 - 5 * 9 / .

(6)
Ready 1 RND 6 * INT 1 + .

(7)
PRINT 89 - 12
```

(8)
PRINT 30 * 9 / 5 + 32

(9)
PRINT (2 + 3) * (4 + 5)
(Parentheses are needed to force Basic to perform the additions first.)

(10)
PRINT 2 * (3 + 4)

(11)
PRINT 2 + 3 * 4
(Parentheses are not needed here since Basic will do the multiplication first automatically.)

(12)
PRINT 3 * 3 * 3

(13)
PRINT 3 * 3 * 3
(Notice that problems 12 and 13 accomplish the same task in two different ways.)

(14)
PRINT 4 * 4 * 4
(The TransFORTH line for this problem performs the same "function" on the number 4 as the last example performed on the number 3. Only one number was changed.)

(15)
PRINT SQRT(17) * SQRT(17)

In this example, the word DUP provided a copy of the square root of 17. Another way to write this in Basic uses named variables:

X = SQRT (17) : PRINT X * X

CHAPTER THREE: DEFINING NEW WORDS

CHAPTER TABLE OF CONTENTS: Page

Immediate Mode Execution ***3-1***

Defining New Words ***3-1***

The Process of Defining Words 3-8

Forgetting Words ***3-9***

Miscellaneous Thoughts on Word Definitions ***3-10***

Word References 3-11
Words Which Look Forward 3-11
Keeping Track of Memory Usage 3-12

Summary ***3-14***

Problems ***3-14***

Solutions to Problems 3-15

Immediate Mode Execution

All of the examples in the previous chapter were done in "immediate" mode. Each line entered was compiled, executed, then forgotten. To be more specific, here are the basic tasks TransFORTH performs for a line entered in "immediate" mode:

1. TransFORTH looks for spaces to separate the line into individual word names.

2. Each word name is compared with the word names in the word library until a match is found. (If no match is found, TransFORTH attempts to read the characters as a number.)

3. For each word, TransFORTH creates (compiles) a machine language "call" to the word in memory above the top of the word library. Therefore, a line containing 6 TransFORTH words will create a total of 6 "calls" in memory, one to each word.

4. The system then directly executes this new string of code that it just created, calling each word one at a time to perform the desired task.

The next line entered will then compile another string of code directly over the previous code, overwriting it.

(Note: If this discussion of compiling and executing seems a bit mystifying, don't worry. Understanding this material is helpful in using TransFORTH, but not necessary.)

Defining New Words

One of TransFORTH's great strengths lies in the ability to define new words in terms of old ones. When defining new words, TransFORTH compiles the lines entered (as in immediate mode), then simply saves the code rather than executing and discarding it. This code then becomes a new TransFORTH word which is added to the word library. In this way, the TransFORTH language itself (of which the word library is a part) "expands" to become your program!

New words are created with "colon definitions" (so named because they begin with a colon). The form for a colon definition is:

```
:  <word name>  <string of defining words>  ;
```

The colon tells the system to begin a new word definition. The name that immediately follows the colon will be the name of the new word. The words that follow the name make up the "definition" of the word; they are the words to be executed whenever the defined word is typed. These words behave just as if they had been typed in "immediate" mode each time. The semicolon marks the end of the word definition, and causes the word to be compiled into machine language and added to the word library.

Suppose you want to print the sum of two numbers along with the message "THE SUM IS ". In immediate mode, you would enter:

```
Ready 17 24     ( Two numbers to be added. )
[ 17 ]
[ 24 ]

Ready PRINT " THE SUM IS " SPCE
THE SUM IS
[ 17 ]
[ 24 ]

Ready + .
41
```

Or on one line:

```
Ready 17 24
[ 17 ]
[ 24 ]

Ready PRINT " THE SUM IS " SPCE + .
THE SUM IS 41
```

Rather than typing the line:

```
PRINT " THE SUM IS " SPCE + .
```

every time, the words can be placed in a colon definition:

```
Ready : SUM

Ready PRINT " THE SUM IS " SPCE

Ready + . ;
```

The colon marks the beginning of the word definition, and SUM is the name of the new word. The following words make up the definition of SUM. The semicolon ends the word definition, returning the system to the immediate mode. SUM is now a new TransFORTH word, and can be seen by typing LIST:

```
Ready LIST

SUM
NOTE
NEGATE
ABS
SIGN
.
.
.
```

The word SUM is now as much a part of the TransFORTH system as the original TransFORTH words. The system literally expanded to include the word SUM. Executing SUM is exactly the same as executing the words which defined it:

```
Ready 17 24 SUM
THE SUM IS 41

Ready 99 2 SUM
THE SUM IS 101
```

As you can see, TransFORTH words, including new colon definitions, are somewhat similar to Basic GOSUB subroutines in that they are self-contained procedures which can be called from immediate mode or a running program. However, TransFORTH programs are made up entirely of new word definitions, whereas Basic programs often do not use subroutines at all.

Here is a more involved example of a colon definition:

To find the length of the diagonal (hypotenuse) of a right triangle, you square the lengths of the two sides, add them together, then take the square root. For example, if you walk 5 miles north, then 12 miles east, the distance from the starting

point to the destination can be found:

```
Ready 5 5 *        ( Square the 5. )
[ 25 ]

Ready 12 12 *      ( Square the 12. )
[ 25 ]
[ 144 ]

Ready +            ( Add the squares together. )
[ 169 ]

Ready SQRT         ( Take the square root. )
[ 13 ]

Ready .            ( Print the result. )
13
```

(We could have used the "^" (power) function to square the numbers, but multiplication is much faster.)

By making use of the stack manipulation words, you only need to enter the initial values once:

```
Ready 5 12        ( Enter the two values. )
[ 5 ]
[ 12 ]

Ready DUP         ( Duplicate the 12 )
[ 5 ]
[ 12 ]
[ 12 ]

Ready *           ( and square it. )
[ 5 ]
[ 144 ]

Ready SWAP        ( Swap the two values to move the 5 to top of
                    stack. )
[ 144 ]
[ 5 ]

Ready DUP *       ( Duplicate and square the 5. )
[ 144 ]
[ 25 ]

Ready +           ( Add the two squares. )
[ 169 ]
```

```
Ready SQRT .     ( Take the square root and print it. )
169
```

We now have a set procedure for finding the diagonal distance:

```
DUP * SWAP DUP * + SQRT
```

These words can be used in a new word definition:

```
Ready : DIAGONAL

Ready DUP *

Ready SWAP DUP *

Ready + SQRT ;
```

Note that the definition can extend over several lines. Using the colon definition format, the colon begins the definition, DIAGONAL is the name of the new word, the following words define what DIAGONAL will do when it is executed, and the semicolon ends the word definition. DIAGONAL has been added to the word library:

```
Ready LIST

DIAGONAL
SUM
NOTE
NEGATE
ABS
.
.
.
```

Executing the new word DIAGONAL is equivalent to executing the words which defined it:

```
Ready 5 12
[ 5 ]
[ 12 ]

Ready DIAGONAL

[ 13 ]

Ready .
13
```



```
Ready 10.63 5.007 DIAGONAL .
11.7501893

Ready 5 7 DIAGONAL .
8.60232527
```

The word DIAGONAL can of course be called from inside yet another new word definition. In this next example, the numbers to be used with DIAGONAL are printed with some explanatory text, the diagonal is found, then the answer is printed. (Note that to separate the printed text and numbers with spaces, either SPCE can be used or extra spaces can be inserted between the text and the quotes as shown below.)

Since printing numbers removes them from the stack, copies of the numbers must be made so that they will be available both for printing and for the word DIAGONAL. The best way to copy a pair of numbers is to use the word OVER twice:

```
Ready 5 12          ( Numbers to find diagonal of. )
[ 5 ]
[ 12 ]

Ready OVER          ( Copies the 5 to top of stack. )
[ 5 ]
[ 12 ]
[ 5 ]

Ready OVER          ( Copies the 12 to top of stack. )
[ 5 ]
[ 12 ]
[ 5 ]
[ 12 ]

Ready PRINT " IF YOU WALK  " .
IF YOU WALK 12
[ 5 ]
[ 12 ]
[ 5 ]

Ready PRINT "  MILES NORTH, THEN  " . CR
 MILES NORTH, THEN 5

[ 5 ]
[ 12 ]
```

```
Ready PRINT " MILES EAST, THE DIAGONAL DISTANCE IS " CR
MILES EAST, THE DIAGONAL DISTANCE IS

[ 5 ]
[ 12 ]

Ready DIAGONAL .
13

Ready PRINT "  MILES. "
 MILES.
```

All of this can be placed in a colon definition:

```
Ready : HOW.FAR

Ready OVER OVER

Ready PRINT " IF YOU WALK  " .

Ready PRINT "  MILES NORTH, THEN  " . CR

Ready PRINT " MILES EAST, THE DIAGONAL DISTANCE IS " CR

Ready DIAGONAL .

Ready PRINT "  MILES. " ;
```

(Note the period in the word name "HOW.FAR". A space could not be used because that would break the name into two words, "HOW" and "FAR". The period is simply a convenient convention: "HOW.FAR" is easier to read than "HOWFAR".)

```
Ready 5 12 HOW.FAR
IF YOU WALK 12 MILES NORTH, THEN 5
MILES EAST, THE DIAGONAL DISTANCE IS
13 MILES.

Ready 5 5 HOW.FAR
IF YOU WALK 5 MILES NORTH, THEN 5
MILES EAST, THE DIAGONAL DISTANCE IS
7.07106781 MILES.
```


The Process of Defining Words

As mentioned above, in immediate mode the TransFORTH system carries out a number of tasks for each line:

1. Input the line.
2. Compile the line into machine language code on the top of the word library.
3. Execute the code.
4. Discard the code (allow it to be overwritten by the next line compiled).

When TransFORTH sees the word ":" (colon), it switches into a "compile-and-save" mode. The word name which follows is placed directly onto the word library. The following code is then compiled onto the word library as in immediate mode, but not executed. When the semicolon is read, the system marks the end of the definition on the word library, then switches back to immediate mode.

When entering a colon definition, nothing can be executed until after a semicolon is read. If the system ever responds to a command with only a Ready prompt (without doing anything), you've probably forgotten a semicolon and are still inside of a word definition. The command you tried to execute immediately has instead been added to the word. Type a semicolon to return to immediate mode.

If you are not inside of a word definition and you type a semicolon, TransFORTH will print the error message:

```
C Error : UNEQ  (Press RETURN)
```

This stands for "Compile Error : UNEQual word balance", which means that the semicolon could not be matched with a corresponding colon. There are also other TransFORTH words which must be used in conjunction with another word. Any mismatch will produce the UNEQ error. These other words will be discussed in the next chapter.

The actual word definition as stored in the word library is made up of 6502 machine language code, and the lines you typed to define the word cannot be read back or modified. This has two ramifications: 1) After a word has been defined, and the definition typed in has scrolled off the screen, there is no direct way to know how the word was defined. At first this might

sound somewhat limiting, but in Chapter Five we'll show you how to save the text of a word definition before it is added to the word library. 2) If you want to change the definition of a word on the word library, you must first delete the word (using FORGET, which is described below), then redefine the word.

Forgetting Words

If you were to continue defining words, the word library would expand until eventually all available memory was filled. Sometimes words are no longer needed, or a word might contain a mistake. In either case, to delete one or more words, the word FORGET is used. It takes the form:

FORGET <wordname>

FORGET cannot selectively remove words from the middle of the word library. It only truncates off the top, deleting the specified word and every word above it. In our example, the word HOW.FAR (on the top of the word library) can be deleted by typing:

```
Ready FORGET HOW.FAR

Ready LIST

DIAGONAL
SUM
NOTE
NEGATE
ABS
.
.
.
```

HOW.FAR has been removed from the word library. Since FORGET deletes the specified word and every word above it, a number of words on the word library can be deleted with one FORGET. For example, both SUM and DIAGONAL can be deleted by typing:


```
Ready FORGET SUM

Ready LIST

NOTE
NEGATE
ABS
.
.
.
```

Anytime you need to delete a large group of words that are on the top of the library, simply type "FORGET" followed by the word in the group lowest in the LIST. Also remember that the top "standard" TransFORTH word is NOTE. To clean every additional word from the word library, LIST the library, find the word above NOTE, and FORGET that word. Everything above NOTE will be deleted.

Some of the words at the top of the supplied TransFORTH word library are not frequently used or can be duplicated with other words. If you find that you don't need a number of words at the top of the library, you can FORGET these too, to give you a little more memory for your own words. The rest of the TransFORTH system will behave normally.

Miscellaneous Thoughts on Word Definitions

Defining new words is the heart of writing programs with TransFORTH. Each new word is a self-contained subroutine that can call any previously defined subroutines. Writing programs is simply a matter of breaking the task into a number of smaller tasks, then writing word definitions (subroutines) that accomplish each of these tasks. A TransFORTH program, then, is simply a collection of one or more word definitions on the word library that work together to perform a task.

With separate word definitions, you can also have more than one "program" in memory at a time. Words can be defined completely independently of each other, and used as individual programs or routines. This is very different from Basic, where only one "program" can reside in memory at one time.

An example at the end of the next chapter will demonstrate how word definitions can be grouped together to form larger programs.

....Which brings us back to some specific points on TransFORTH.

Word References

Words in TransFORTH can only be defined in terms of already existing words, which reside in the TransFORTH word library at the time. In fact, any reference to a word that is not currently in the word library will produce an error message, and the line with the error will be ignored:

```
Ready 5 DUP Ø SWAP STRANGE CR

"STRANGE" Not Found (Press RETURN)
```

Another source of trouble is defining a word with the same name as an already existing word. If this happens, the new word is added to the word library, but a warning message is printed:

```
Ready : OVER PRINT " OVER THE RIVER AND THRU THE WOODS " ;

"OVER" Not Unique (Press RETURN)
```

If two different words in the word library have the same name, how does the system choose between them? For our example, any words that referenced OVER before the new definition was added will still reference the earlier word. Any new references to OVER will reference the new definition. That means that the original definition is no longer available from the keyboard! In general, defining words with existing word names should be done with care if done at all.

Programmers who like to dabble with recursion will be happy to hear that TransFORTH words can call themselves. Word definitions can also be nested one definition inside another, allowing the inside and outside words to call each other. These capabilities are very useful for certain recursive applications, but should be avoided if not needed. The looping and branching structures discussed in the next chapter provide for sophisticated nested loops without having to resort to recursive looping. (Recursion is also discussed in Appendix C.)

Words Which Look Forward

Most words in TransFORTH look to the stack for any data or information they might need. Some words, like PRINT or FORGET, instead look forward down the input line for further data. You

might be tempted to build a colon definition like the following:

```
Ready : TESTWORD  CR CR PRINT ;

Ready TESTWORD " HI THERE "
```

Don't try it! (...at least not with expectations of success.) The word PRINT looks for the text to be printed <u>as it is compiled</u>, not when it is executed. The above example will not work, and it may cause the system to go off the deep end.... The other words (introduced in later chapters) which look to the input line for data work the same way, and should be used as described.

Keeping Track of Memory Usage

Because of the word library structure, a number of different "programs" can reside in memory at one time. The only limitation is the amount of memory available for programs. For 64K Apples, the word library can expand by over 36,9ØØ bytes. For 48K systems, the library can expand by over 27,ØØØ bytes. The memory map in Appendix B shows how TransFORTH uses Apple memory. When working with larger programs, it is generally a good idea to remain aware of what is on the word library, and how much memory is being used.

There are two words to help you:

The word HERE places the address of the top of the word library on the stack. This can let you know how large things are getting. This example was done with no additional words on the word library. (The addresses printed here are for example purposes only. The address numbers displayed may be slightly different.)

```
Ready HERE .
12611
```

For people who like to "think" in hexadecimal, the word $LIST can also be very useful. $LIST is identical to LIST, except that it also displays the hexadecimal address of each word in the word library. By comparing adjacent numbers, you can determine how much memory each word takes. Here is a sample of $LIST:

```
Ready $LIST

$3121 NOTE
$310B NEGATE
$30F8 ABS
$30DD SIGN
$30AC CALL
$30A2 PREG
  .    .
  .    .
  .    .
```

Since $LIST displays the address at which each word begins, the first address shown in the $LIST is for the beginning of the top word, not the top of the word library at the end of the word. To determine the address of the top of the word library in hexadecimal, you can define a "dummy" word and then use $LIST. The top address will be the top of the word library after the dummy word is deleted:

```
Ready : IT ;      ( "IT" does not execute anything. )

Ready $LIST

$3143 IT
$3121 NOTE
$310B NEGATE
  .    .
  .    .

Ready FORGET IT
```

$3143 is the hex address of the top of the word library.

If the word library is filled until there is no more free memory, the line or word currently being compiled will be forgotten, and the following error message will appear:

```
C Error : PRGO (Press RETURN)
```

PRGO stands for PRoGram Overflow. If you get this message, you can either work with what you have in memory, or FORGET some of the words from the word library to make more room.

(For more information on the internal structure of the word library, see Appendix C.)

Summary

New words can be defined and added to the word library. The word library expands as new words are added, and these words follow the same conventions as the original TransFORTH system words. All words behave as subroutines: they perform a particular task, then return to whatever program or line called them.

A new word is created with a "colon definition". It consists of a colon, the name of the new word, a string of words making up the definition, and a semicolon. Executing a new word is exactly equivalent to executing the words which defined it.

Once a word is defined, it cannot be examined or modified, because it has been compiled into low-level machine language instructions. Chapter Five will describe how to save the text of a word definition so that changes can be made at any time.

Words can be removed from the word library with the word FORGET. FORGET cannot selectively remove words from the middle of the word library; it can only truncate off the top.

"Programs" in TransFORTH are written by defining one or more new words that work together to accomplish a particular task. Since programs are simply collections of word definitions, more than one program can reside in memory at a time. The only limitation is in available word library space. The words HERE and $LIST can be used to help find the amount of free memory.

Problems

Note: For these problems, there may be more than one "right" way of doing things. Actually trying out the word definitions with TransFORTH is always the best way to check your answers.

(1)
Write a word definition called AVERAGE which removes two numbers from the stack, finds their average, and places the result back on the stack.

(2)
The formula for converting degrees Fahrenheit to degrees Celsius is: C = (F - 32) * 5 / 9. Write a word definition called F-C which performs this conversion.

(3)
The formula to convert back (Celsius to Fahrenheit) is F = C * 9 / 5 + 32. Write a word called C-F which converts degrees Celsius to degrees Fahrenheit.

(4)
Write a word called C.AVERAGE that removes two Fahrenheit temperature values from the stack, averages them, then converts the average to degrees Celsius. This word should make use of two of the three previous word definitions.

(5)
With these four new words on the word library, what single command will delete the words F-C, C-F, and C.AVERAGE, leaving only AVERAGE?

Solutions to Problems

(1)
```
: AVERAGE
+ 2 / ;
```

(2)
```
: F-C
32 - 5 * 9 / ;
```

(3)
```
: C-F
9 * 5 / 32 + ;
```

(4)
```
: C.AVERAGE
AVERAGE F-C ;
```

(5)
```
FORGET F-C
```


CHAPTER FOUR: LOOPS AND TESTS

CHAPTER TABLE OF CONTENTS: Page

DO-LOOP 4-1
The Return Stack 4-4
Comparing Numbers 4-5
Decision and Branching Words 4-8
IF—THEN 4-8
IF—ELSE—THEN 4-10
BEGIN—UNTIL 4-13
BEGIN—WHILE—REPEAT 4-14
CASE:—THEN 4-15
Program Structure 4-18
The Set of Decision and Branching Words 4-19
Summary 4-20
Problems 4-21
Solutions to Problems 4-22

TransFORTH includes a number of words for performing tasks repetitively, testing certain conditions, and making program decisions on the basis of those tests. Most of these constructs are similar to ones used in Pascal, and there are also some similarities to Applesoft Basic.

Do-Loop

The TransFORTH DO - LOOP structure is used for performing tasks repetitively when the number of repetitions is known ahead of time. The form for the DO - LOOP is:

<ending value> <initial value> DO <words to be repeated> LOOP

The word DO removes two numbers from the stack. The top stack number specifies an "initial" value; the next number is the "ending" value. The words between DO and LOOP are executed, then the initial value is incremented by one. If this incremented value (which we'll call the "loop value") is still less than the ending value, the program loops back to execute the words between DO and LOOP again. This cycle is repeated as long as the loop value is less than the ending value.

If you are familiar with Applesoft Basic, you will notice that DO - LOOP is similar to Applesoft's "FOR -- NEXT" looping structure. Here is an example that prints the phrase "HI THERE" 3 times:

```
Ready 3 0 DO PRINT " HI THERE " CR LOOP
HI THERE
HI THERE
HI THERE
```

"3 0 DO" sets up the looping structure for 3 loops. Inside the loop, the words PRINT " HI THERE " CR are executed. LOOP marks the end of the loop.

It is often handy to retrieve the current loop value. Inside the DO - LOOP, the word "I" retrieves the loop value and places it on the stack. Here is an example:

```
Ready 5 0 DO PRINT " HERE IS NUMBER  " I . CR LOOP
HERE IS NUMBER 0
HERE IS NUMBER 1
HERE IS NUMBER 2
HERE IS NUMBER 3
HERE IS NUMBER 4
```


"5 0 DO" starts a loop that will repeat 5 times. Inside the loop, the phrase "HERE IS NUMBER " is printed, then the loop value is retrieved by I and printed with ".". CR causes the carriage return to put each number on its own line, and LOOP again marks the end of the loop, causing the loop value to be incremented and compared with the ending value. Note that the loop continues only as long as the loop value is <u>less than</u> the ending value. That's why the loop stops at 4, not 5 as in Applesoft.

The words DO and LOOP work as a pair and must always be matched up, either on the same line together or entered in a colon definition. This is so that one of the words will not be executed unless the other word is also present in the code. If the DO and LOOP are not together, the "C Error : UNEQ" error will occur.

To make a loop with an increment other than 1, use +LOOP instead of LOOP. +LOOP removes a number from the stack to use as the increment. This number can be either positive or negative (for loops that count backwards). Here is an example:

```
Ready 10 0 DO I . CR 2 +LOOP
0
2
4
6
8
```

The 2 was used by +LOOP as the increment.

```
Ready 150 200 I . CR -10 +LOOP
200
190
180
170
160
```

```
Ready 1.3 1.9 DO I . SPCE -.1 +LOOP
1.9 1.8 1.7 1.6 1.5 1.4
```

Note that DO - LOOPs that count backwards continue as long as the loop value is <u>greater than</u> the ending value. If this seems confusing, just remember that every DO - LOOP (either forwards or backwards) will stop before it reaches the ending value.

Loops can be nested inside one another. The loop value for the

current innermost loop is always access by "I", and the loop value for the next outer level is accessed with the word "J" as in this colon definition:

```
Ready : DOUBLELOOP

Ready 4 Ø DO

Ready    PRINT " OUTER LOOP:  " I . CR

Ready    3 Ø DO

Ready      J . SPCE I . CR

Ready    LOOP

Ready LOOP ;

Ready DOUBLELOOP
OUTER LOOP: Ø
Ø Ø
Ø 1
Ø 2
OUTER LOOP: 1
1 Ø
1 1
1 2
OUTER LOOP: 2
2 Ø
2 1
2 2
OUTER LOOP: 3
3 Ø
3 1
3 2
```

The inner loop is cycled three times for each cycle of the outer loop. Note that the outer loop value is referenced in the outer loop with "I", but is referenced from the inner loop with "J". Just remember that "I" always references the loop value for the current innermost loop.

If more than two nested loops are being used, the loop value of the third loop out can be accessed from the innermost loop with the word "K".

The Return Stack

DO - LOOPs make use of another stack in the TransFORTH system, similar to the data stack, known as the "return stack". The return stack can also hold 55 numbers, though for most programs it rarely contains more than a few. (Most versions of Forth, because they are interpreted, use the return stack for a variety of purposes. Because TransFORTH is compiled directly into machine language, the Apple's processor itself takes care of these things.)

When the word DO is encountered, the top two values on the data stack are moved over to the return stack, with the loop value on the top and the ending value underneath. The word LOOP increments the loop value on the return stack. The word "I" places a copy of the top return stack value on the data stack. When the loop is finally exited, the two return stack values are removed.

There are a few words in TransFORTH that enable you to use the return stack directly. The return stack can be a handy place to put numbers for a moment while playing games with other numbers on the data stack. (Chapter Six will explain how to declare variables for more permanent storage.) Care should be taken to avoid disturbing the value and placement of existing return stack entries when using DO - LOOPs. (In other words, if you're not sure, don't!) Here are the words that directly control the return stack:

PUSH moves the top data stack entry to the return stack.

PULL moves the top return stack entry back to the data stack.

POP removes the top return stack entry. The number is lost.

If there are numbers on the return stack, the stack display will show these too, as in the following example. Suppose there are three numbers on the stack and you want to reverse the order of the bottom two. Here is one way to do it:

```
[ 3 ]
[ 2 ]
[ 1 ]
```

```
Ready PUSH
[ 3 ]
[ 2 ]
< 1 >        ( The 1 is now on the return stack. )

Ready SWAP
[ 2 ]
[ 3 ]
< 1 >

Ready PULL
[ 2 ]
[ 3 ]
[ 1 ]
```

The return stack display cannot be used to see the loop values inside of a DO - LOOP. This is because the entire loop is executed and completed before the system returns to the immediate command mode, where the stack is displayed. (Note: There is a utility file on the system disk which includes a routine for viewing the stack from within a running program. It is described in Appendix D.)

Comparing Numbers

A number of TransFORTH words are devoted to comparing numbers. These words are:

```
<>     ( not equal to )
=      ( equal to )
>      ( greater than )
<      ( less than )
>=     ( greater than or equal to )
<=     ( less than or equal to )
```

Each of these words removes two numbers from the stack, comparing the second stack number down with the top stack number, and returns on the stack either a 1 if the comparison is true, or a Ø if the comparison is false. Here are a few examples:

```
Ready 5 5 = .
1

Ready 5 7 = .
Ø
```



```
Ready -32 -6 < .
1

Ready 45 46 >= .
Ø
```

There are four other words related to the comparison words. These words treat nonzero numbers as "true" and zeros as "false". The first three remove two numbers from the stack, then return either a 1 or a Ø.

AND returns a 1 if the top stack value <u>and</u> the next value are both "true" (nonzero). Otherwise it returns a Ø.

OR returns a 1 if the top stack value <u>or</u> the next value are nonzero. It returns a Ø only if both values are zero.

XOR (which stands for eXclusive OR) returns a 1 only if the top stack value or the next stack value (but not both) are nonzero. In other words, it returns a 1 if the true/false status of the two numbers is different, if one number is zero and the other is nonzero. If both numbers are zero or both are nonzero, XOR returns a Ø.

NOT removes one number from the stack. If the number is zero, NOT returns a 1. If the number is nonzero, NOT returns a Ø.

Here are diagrams outlining the way each of these words works:

AND

zero	zero	Ø
zero	nonzero	Ø
nonzero	zero	Ø
nonzero	nonzero	1

OR

zero	zero	Ø
zero	nonzero	1
nonzero	zero	1
nonzero	nonzero	1

XOR

zero	zero	Ø
zero	nonzero	1
nonzero	zero	1
nonzero	nonzero	Ø

```
NOT
zero      1
nonzero   0
```

These words are useful for combining or rearranging the results of tests. The following example tests whether or not a number is greater than 5 and less than 10. The same test is done with two numbers, 7 and 13:

```
Ready 7             ( Number to be tested. )
[ 7 ]

Ready DUP 5 >       ( Greater than 5? )
[ 7 ]
[ 1 ]               ( The "1" means "true".)

Ready SWAP
[ 1 ]
[ 7 ]

Ready 10 <          ( Less than 10? )
[ 1 ]
[ 1 ]

Ready AND .         ( Are both results true? )
1                   ( Yes. )
```

7 is greater than 5 and less than 10.

```
Ready 13            ( Number to be tested. )
[ 13 ]

Ready DUP 5 >
[ 13 ]
[ 1 ]

Ready SWAP
[ 1 ]
[ 13 ]

Ready 10 <
[ 1 ]
[ 0 ]
```



```
Ready AND .         ( Are both results true? )
0                   ( No. )
```

13 is not both greater than 5 and less than 10.

Decision and Branching Words

An essential part of a computer language is the ability to test a condition, then make a decision on the basis of the test. TransFORTH has five different constructs that accomplish this. Each of the constructs contains a word which removes a number from the stack. In most cases, the "decision" is made on the basis of whether the number is zero or nonzero. Any nonzero number represents a condition being true, and a zero represents false. (Note that the comparison words place a one on the stack if the comparison is true, and zero if the comparison is false.)

A simple flowchart is included with each of the following constructs, showing the "flow" of the program. The arrows indicate what is executed in what order. The boxes represent a group of words to be executed. The diamonds represent a test, usually for a zero or nonzero number.

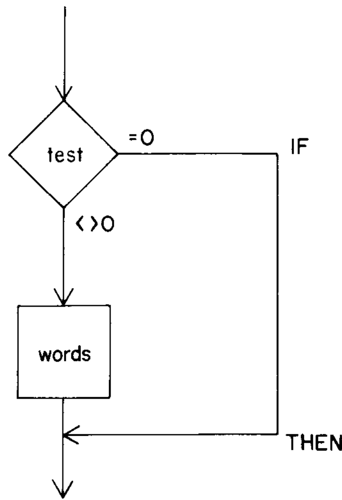
Note: Each of these constructs is made up of two or more words. Like DO - LOOP, these decision words work together, and cannot be entered alone. They must be entered either on one line or from within a colon definition.

IF-THEN

The simplest decision construct is IF - THEN. The form for IF - THEN is:

```
<test stack value>
IF
   <words to be executed>
THEN
```

The word IF removes a number from the stack. If the number is not zero, then the words between IF and THEN are executed. If the number is zero, then the words between IF and THEN are skipped over. In either case, the program then continues on after the word THEN. Here is the flowchart for IF - THEN:



Here are IF and THEN in a couple of colon definitions:

```
Ready : TEST1

Ready PRINT " THE NUMBER IS  "

Ready IF PRINT " NOT  " THEN

Ready PRINT " ZERO. " ;
```

The first and third PRINT words are executed every time. The word IF removes a number from the stack (which we'll supply before we execute TEST1). If the number is nonzero, then PRINT " NOT ", which is sandwiched between IF and THEN, is executed. If the number is zero, then it is not executed.

```
Ready 5 TEST1
THE NUMBER IS NOT ZERO

Ready Ø TEST1
THE NUMBER IS ZERO
```

IF - THEN constructs can be used with number comparison words. Remember that these words return either 1 or Ø, depending on the success or failure of the comparison. Suppose that for some application, you want to set a limit on the size of numbers. The following word will let any number less than 25 pass through "unharmed", but any number over 25 will be replaced with a 25:


```
Ready : UPPERLIMIT

Ready DUP

Ready 25 > IF

Ready    DROP 25

Ready THEN ;
```

The word DUP makes a copy of the top stack value. The word ">" compares the copy with the number 25, leaving a 1 on the stack if the number is greater than 25, or a Ø if it is not. The word IF removes the one or zero from the stack to decide whether or not to execute the following words. Remember that the original number is still on the stack. If the comparison is false, then the words between IF and THEN are not executed, and the number is left intact. If the comparison is true, then DROP 25 is executed, which removes the original number from the stack and replaces it with 25.

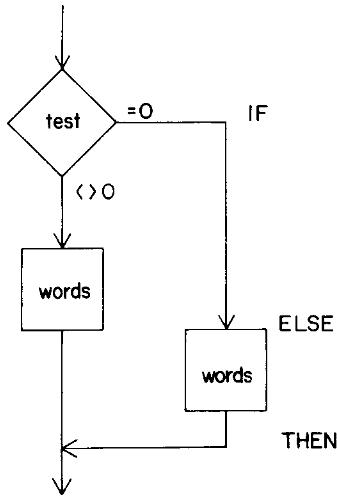
```
Ready 16 UPPERLIMIT .
16

Ready 37 UPPERLIMIT .
25
```

IF–ELSE–THEN

Another version of the IF - THEN construct is IF - ELSE - THEN. The form is:

```
<test stack value>
IF
   <words executed if "true">
ELSE
   <words executed if "false">
THEN
```

As before, the word IF removes a number from the stack. However, if the number is nonzero, then the words between IF and ELSE are executed. If the number is zero, then the words between ELSE and THEN are executed. The program then continues after the word THEN. Here is the IF - ELSE - THEN flowchart:



This word definition determines whether or not a number is greater than 100:

```
Ready : TEST2

Ready DUP .

Ready 100 > IF

Ready    PRINT "  IS GREATER THAN 100  "

Ready ELSE

Ready  PRINT " IS LESS THAN OR EQUAL TO 100  "

Ready  THEN ;
```

Again, the word definition duplicated the number before printing, so that the number could be used for the comparison. Also note that the controlled words are indented. This is certainly not a requirement, but it greatly improves the readability of the word definition. (The next chapter will show you how to use the text editor to save the text of the word definitions.)

```
Ready 106 TEST2
106 IS GREATER THAN 100

Ready 54 TEST2
54 IS LESS THAN OR EQUAL TO 100
```


As with loops, IF - THEN constructs can be nested. This example puts checks for both upper and lower limits on a number:

```
Ready : TWOLIMITS

Ready DUP 25 > IF

Ready    PRINT " GREATER THAN 25 "

Ready    DROP

Ready ELSE

Ready    10 < IF

Ready       PRINT " LESS THAN 10 "

Ready    ELSE

Ready       PRINT " BETWEEN 10 AND 25 "

Ready    THEN

Ready THEN ;
```

One IF - ELSE - THEN is placed between the ELSE and THEN of another one. Note that before the first comparison, the number is DUPlicated because the program doesn't know yet whether or not it will be needed for the second comparison. If the number is greater than 25, then it is not needed again, and is DROPped.

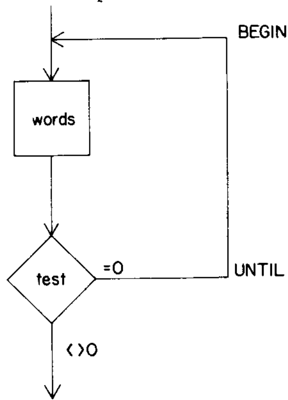
```
Ready -62 TWOLIMITS
LESS THAN 10

Ready 19 TWOLIMITS
BETWEEN 10 AND 25

Ready 684 TWOLIMITS
GREATER THAN 25
```

BEGIN–UNTIL

Another construct that allows repeated execution is BEGIN - UNTIL. The form is:

```
BEGIN
   <words to be repeated>
   <test stack value>
UNTIL
```

The word BEGIN marks the beginning of the construct. The words between BEGIN and UNTIL are executed, then the word UNTIL removes a number from the stack. If the number is zero, then the program branches back and the words between BEGIN and UNTIL are executed again. This loop is repeated until the stack value is nonzero, then the program continues past the UNTIL. This is the flowchart for BEGIN - UNTIL:



The following example starts with a zero on the stack, then prints the number, adds one to it, and loops back until the number equals 8:


```
Ready Ø BEGIN DUP . CR 1 + DUP 8 = UNTIL
Ø
1
2
3
4
5
6
7
[8]
```

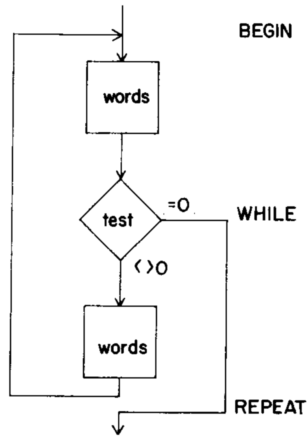
The words "DUP . CR" print the number without losing it and issue a carriage return, "1 +" increments the number, and "DUP 8 =" determines if the number equals 8. Notice that this loop leaves a copy of the number on the stack when it finishes. Adding DROP to the end of the line takes care of this.

BEGIN–WHILE–REPEAT

The BEGIN - WHILE - REPEAT construct is similar to BEGIN - UNTIL. The form is:

```
BEGIN
   <words to be repeated>
   <test stack value>
WHILE
   <controlled words>
REPEAT
```

The word BEGIN again marks the beginning of the construct. The words between BEGIN and WHILE are executed, then WHILE removes a number from the stack. If this number is nonzero, then the controlled words between WHILE and REPEAT are executed, then execution jumps back again to the words after the BEGIN. If the number is zero, then the program jumps directly past the word REPEAT and continues on. The key to remembering this is that the controlled words are REPEATed while the stack value remains nonzero. This the flowchart for BEGIN - WHILE - REPEAT. Note that the test is at the beginning of the controlled part:



The following example is similar to the previous example. The number is tested first this time. While it is not equal to 8, it is printed and incremented, and the cycle is repeated:

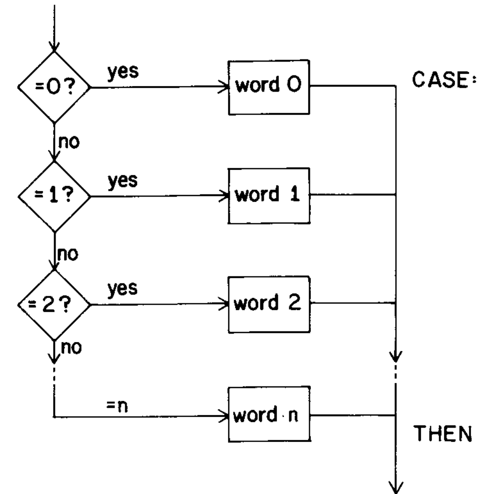
```
Ready Ø BEGIN DUP 8 <> WHILE DUP . CR 1 + REPEAT
Ø
1
2
3
4
5
6
7
[8]
```

CASE:–THEN

Sometimes a choice needs to be made from a range of possible numbers. The CASE: construct allows you to do this. The form is:

```
<stack value>
CASE:
   <word Ø>
   <word 1>
   <word 2>
      .
      .
      .
   <word n>
THEN
```

The word CASE: removes a number from the stack and uses this word to select and a execute a single word from a list of words. A zero selects word Ø, a one selects word 1, etc. The word THEN marks the end of the CASE: construct, and is required. Here is the flowchart for CASE:.



The following example shows how CASE: works:

```
Ready : X PRINT " THE NUMBER IS ZERO " ;

Ready : Y PRINT " THE NUMBER IS ONE " ;

Ready : Z PRINT " THE NUMBER IS TWO " ;
```

```
Ready : CASE.TEST

Ready CASE:

Ready    X

Ready    Y

Ready    Z

Ready    BELL

Ready THEN ;
```

X, Y, and Z are words we have defined and are called by the word CASE.TEST. The CASE: list in CASE.TEST contains four words, so the construct uses the numbers Ø through 3. Zero selects X, 1 selects Y, 2 selects Z, and 3 selects BELL:

```
Ready Ø CASE.TEST
THE NUMBER IS ZERO

Ready 1 CASE.TEST
THE NUMBER IS ONE

Ready 2 CASE.TEST
THE NUMBER IS TWO

Ready 3 CASE.TEST
(The Apple speaker beeps.)
```

Warning: If the number which CASE: removes from the stack is too large or is less than zero, something strange and probably not-so-wonderful will happen. For example, the system may hang up. (In the above example, the only acceptable numbers for CASE.TEST are Ø, 1, 2, and 3.) The key to avoiding trouble is to simply not let numbers out of the CASE: range go into the word CASE:. There are a number of ways to do this. Here is one for the above example:


```
Ready : SAFE.CASE

Ready DUP DUP 3 <= SWAP Ø >= AND

Ready IF

Ready    CASE.TEST

Ready ELSE

Ready    PRINT " THE NUMBER IS NOT BETWEEN Ø AND 3 "

Ready    DROP

Ready THEN ;
```

SAFE.CASE first checks the number to see that it is between Ø and 4 before passing it on to CASE.TEST. If it is out of range, a message is printed. (You may want to try the words "DUP DUP 3 <= SWAP Ø >= AND" directly from the keyboard to see how they work together.)

```
Ready 2 SAFE.CASE
THE NUMBER IS TWO

Ready 7 SAFE.CASE
THE NUMBER IS NOT BETWEEN Ø AND 3

Ready -6 SAFE.CASE
THE NUMBER IS NOT BETWEEN Ø AND 3
```

Program Structure.

Notice that in the last example for CASE: above, we began by defining three short words: X, Y, and Z. Then we defined the word CASE.TEST, which calls one of those three words. Finally we defined SAFE.CASE, which calls CASE.TEST.

This "chain" of definitions is the way long programs in TransFORTH are built up. The "low-level" words, which usually do rather menial tasks, are defined first. Then the next level of words, which call the first set of words, are defined. This process builds layer by layer until one last word is added to the top of the word library, which "coordinates the show". The entire program can be run by simply typing the name of this top word.

The beauty of this scheme is that each level of words can be thoroughly tested and debugged before moving on to the next higher level. This helps to prevent the all-too-familiar scene of a programmer helplessly wading through miles and miles of computer print-out trying to find the elusive "bug" in a program.

The Set of Decision and Branching Words

The decision and branching words make TransFORTH a "structured" language. This structure is complete, in the sense that any type of looping or branching can be accomplished using combinations of these constructs. In addition, the words always clearly mark the beginning and ending of any loop or branch (e.g. a BEGIN - UNTIL loop always begins with BEGIN and ends with UNTIL). This improves readability over languages like Basic, where loops and branches can become confusing with an overabundance of GOTOs and line numbers. Using the TransFORTH words, GOTO is not needed. In fact, TransFORTH does not even include a GOTO word.

Proper use of the decision and branching words can make programming simpler and programs more readable. Following are a few thoughts on how TransFORTH words can be best used for certain applications.

BEGIN - WHILE - REPEAT and BEGIN - UNTIL are very similar. The main difference is that the BEGIN - UNTIL is always executed at least once, because the test is not performed until the end of the loop. The test is performed first with BEGIN - WHILE - REPEAT, and if the test fails, the controlled words between WHILE and REPEAT are not executed at all.

An often-used construct is the endless loop. In Basic this is usually written as several lines of code followed by a GOTO back to the top. In TransFORTH, the same loop can be written with BEGIN - UNTIL. Remember that the loop repeats until the stack value removed by the UNTIL is nonzero. By placing a zero on the stack immediately before the UNTIL, this condition is never satisfied, and the loop repeats endlessly. This example simply loops round and round, with no way out:

```
Ready BEGIN  Ø UNTIL
```

Pressing RESET is the only way to recover. In this next example, the loop counts while repeating forever:

```
Ready 1 BEGIN  DUP . CR  1 +  Ø UNTIL
```


The language Logo is similar to TransFORTH in that new words (or "procedures" in Logo) can be defined in terms of old ones. Endlessly looping procedures in Logo are often created by defining procedures that "call themselves". This uses recursion in an odd way, where a "GOSUB"-like call is used to replace a "GOTO"-like jump. This technique can be duplicated with TransFORTH:

```
Ready : TEST

Ready PRINT " ENDLESS LOOP " CR

Ready TEST ;

Ready TEST       ( Press Reset to stop. )
```

This does work, but a BEGIN - UNTIL loop provides a much more readable way to accomplish the same thing, because both the beginning and end of the loop are clearly marked with BEGIN and UNTIL.

Summary

TransFORTH includes a number of decision and branching constructs. DO - LOOP is used for performing tasks repetitively when the number of repetitions is known ahead of time. DO - LOOP is similar to Basic's FOR...NEXT loop. Inside the loop the word I retrieves the value of the loop. DO - LOOPs can be nested. J retrieves the value of the next outer DO - LOOP, and K retrieves the loop value for the third outer loop.

DO - LOOPs use another stack called the return stack. The words PUSH, PULL, and POP can be used to access the return stack directly.

There are several words for comparing numbers: <>, =, >, <, >=, and <=. Each of these removes two numbers from the data stack, makes the comparison, and returns a 1 if the comparison is true, or Ø if false.

The words AND, OR, and XOR remove two numbers from the stack and perform functions that depend on whether the numbers are zero or nonzero. They return either a 1 or a Ø. NOT removes one number from the stack, and also returns a 1 or a Ø.

The following constructs each remove a zero or nonzero number from the stack to perform a branching or looping function:
IF - THEN
IF - ELSE - THEN
BEGIN - UNTIL
BEGIN - WHILE - REPEAT

The CASE: - THEN construct removes a number to select and execute one of the words in a list between the CASE: and THEN.

By using combinations of these constructs, complicated tests and branches can be performed. A GOTO-like word is not needed, and is not included in TransFORTH.

Problems

(1)
Write a word definition called RND.LOOP which prints 5 random values.

(2)
Write a word called ANY.LOOP that removes a number from the stack, then prints that number of random values.

(3)
Write a word called GRID that prints the following pattern. Use two nested DO - LOOPs, printing one "A" at a time:

```
AAAAA
AAAAA
AAAAA
AAAAA
```

(4)
Write a short word definition called TEST which removes a number from the stack. If the number is greater than 25, the word should place a 1 on the stack. If the number is less than or equal to 25, a Ø should be returned.

(5)
Write a word called TEST1 that removes a number, returning a 1 if the number is between 14 and 84, or returning Ø otherwise.

(6)
Write a word called TEST2 that, like TEST1, checks whether or not a number is between 14 and 84. If it is, print a -999; if not, print the number. (Have TEST2 call TEST1.)

(7)
What does this TransFORTH word definition do? How many numbers does it remove from the stack? How many does it leave on the stack?

```
: PUZZLE
* PUSH * PULL + ;
```

(8)
The problems in the previous chapter included routines for converting between degrees Fahrenheit and degrees Celsius. Write a word definition called CONVERT that removes two numbers from the stack. If the top number is 1, convert the second number from Fahrenheit to Celsius; if the top number is Ø, convert from Celsius to Fahrenheit. Call F-C and C-F to do this. Write the definition in two ways: using IF - ELSE - THEN, and using CASE: - THEN.

Solutions to Problems

(1)
```
: RND.LOOP
5 Ø DO
   1 RND . CR
LOOP ;
```

(2)
```
: ANY.LOOP
Ø DO
   1 RND . CR
LOOP ;
```

(3)
```
: GRID
4 Ø DO
   5 Ø DO
      PRINT " A "
   LOOP
   CR
LOOP ;
```

```
(4)
: TEST
25 > ;

(5)
: TEST1
DUP 14 >
SWAP 84 <
AND ;

(6)
: TEST2
DUP TEST1
IF DROP -999 THEN ;
```

(7)
PUZZLE removes 4 numbers from the stack, and leaves one number on the stack. This TransFORTH line:

```
Ready 2 3 4 5 PUZZLE .
26
```

is equivalent to this Basic expression:

```
PRINT 2 * 3 + 4 * 5
26
```

```
(8)
CONVERT
IF F-C
ELSE C-F
THEN ;

: CONVERT1
CASE:
   C-F
   F-C
THEN ;
```


CHAPTER FIVE: THE TEXT EDITOR

CHAPTER TABLE OF CONTENTS: Page

Cursor Movement 5-1

Introduction: Using the Text Editor 5-1

The Text Editor 5-2

Line Entries 5-4
LIST 5-4
AUTONUM 5-5
DELETE 5-6
ERASE 5-6
Automatic Insertions 5-6
INSERT 5-7
SAVE 5-8
GET 5-8
DOS Commands 5-9
Printing Files 5-9
Leaving the Text Editor 5-10

Program Compilation 5-10

Comments 5-11

Examples 5-11

Memory Considerations 5-16

Summary 5-18

Chapter Three discussed how to define new words in terms of existing ones. The words were added to the dictionary and could be called at any time. However, there was no way to save the text of a definition, to go back to the string of words which defined it.

Enter the TransFORTH text editor. This is a straightforward general purpose line-oriented editor. Text can be created here, modified, saved to disk, read back in, and more.

TransFORTH includes words to compile text into the system from the editor or directly from the disk. If any defined words need to be modified, they do not have to be completely re-entered. They can be changed from the editor, then recompiled by the system.

In this chapter, we'll discuss how to use the text editor and how to compile TransFORTH programs from the editor or from disk. We'll also give you some pointers to keep both system and editor memory happy.

Cursor Movement

As you may have discovered by now, the Apple arrow keys work as they do in most Apple applications. The left arrow is a "backspace" key that enables you to back up on the line to correct mistakes. The right arrow is a "retype" key. If you use the right arrow key to move the cursor over text on the screen, the text will be treated by TransFORTH as if it were being typed again directly from the keyboard.

The Apple][and //e ESCape codes for moving the cursor also work from TransFORTH. These can be handy for making fast corrections from the TransFORTH text editor. If you're unfamiliar with the Apple ESCape codes, we suggest you consult one of the Apple manuals. Most of the manuals discuss these codes.

Introduction: Using the Text Editor

As discussed in Chapter Three, when the TransFORTH system reads a line typed at the keyboard, it compiles the line directly into machine language code. The actual characters typed are discarded.

The text editor provides a place to save the characters you type. The editor allows you to enter text, modify or change it, and save or retrieve it from disk. The editor doesn't care whether you're typing TransFORTH programs or quotes from "MacBeth". It simply stores the text in memory.

TransFORTH includes some special commands for reading this text. When these commands are given, the TransFORTH system reads the text in memory exactly as if it were being typed from the keyboard line by line. If the text includes immediate-mode commands, the system will read them and execute them. If the text includes new word definitions, the system will compile them and add them to the word library.

TransFORTH can also read textfiles directly from disk, again treating the text as if it were being typed at the keyboard. This is very similar to the DOS command "EXEC".

Here is a quick overview for using the text editor for creating word definitions: Enter the editor, and type in the text of the word definitions. If you make any mistakes, you can easily make changes to the text. While in the editor, you can also save the text to disk as a DOS textfile. Return to TransFORTH, then enter the command for reading the text in memory. (This command will be described later in the chapter.) The system will read each line from memory, and compile the word definitions. You can then test and run the new words, as before.

If you find that you want to change one or more word definitions, first FORGET the words to remove them from the word library, then re-enter the text editor. Make the desired changes in the text, then return to TransFORTH. Type the command to read the text in memory again, and this new corrected text will be compiled.

The Text Editor

There are actually two text editors on the TransFORTH system disk, named OBJ.EDITOR1 and OBJ.EDITOR2. The first is used on 48K Apples and can edit (without changing the 'default' settings) over 12,000 characters (12 Kbytes). The second is used with 64K systems and can edit over 22,000 characters. Otherwise, the two editors are identical. When the word EDIT is entered, TransFORTH automatically loads the appropriate editor.

Note: TransFORTH and the TransFORTH editor both use standard DOS textfiles for program storage. If you already have a text editor

that can use DOS textfiles, you may want to use it for editing longer programs. The TransFORTH text editor does not have all of the features found in some larger editors and word processors, but it has the advantages of 1) residing in the Apple memory at the same time as the TransFORTH system and 2) being compatible with any standard 80-column card. Compiling programs into the TransFORTH system from a textfile on disk is the same regardless of what editor is used to create the file.

For some of the editor examples in this chapter, we will use English sentences for text instead of TransFORTH programs. The editor doesn't know the difference, and it makes the examples easier to read. The editor is of course usually used for writing TransFORTH programs. Later in the chapter, we'll give examples of editing and compiling actual TransFORTH programs.

To enter the editor from TransFORTH, type EDIT. The appropriate editor will automatically be loaded. In a few seconds you should see the TransFORTH editor header:

```
TransFORTH ][ Editor (C) 1981 P. Lutus
```

The first command to know in the editor is "?", the question mark. Entering a question mark (followed by Return) provides you with a list of all the other editor commands:

```
?

Save
Get
Insert
Delete
Program
Memory
List
Write
Erase
Autonum
Bye
ConTRoL-D=DOS
```

We'll discuss each of these commands in turn, but first let's find out how to enter text into the text editor.

Line Entries

The TransFORTH text editor uses line numbers to identify each line of text. These line numbers are only used within the editor, and are not read by the TransFORTH system or kept when the file is saved to disk. They are simply used for specifying certain lines while in the editor. The line numbers are in steps of 10, and whenever insertions or deletions are made, the file is renumbered automatically, in steps of 10 again.

To enter a line, simply type a line number followed by the line. Here are some example lines to enter:

```
10 MY VERY FIRST EDITOR LINE!
20 ENTERING LINES IN THE EDITOR IS
30 SIMILAR TO ENTERING LINES IN BASIC.
```

LIST

To see that these text lines have been stored, they can be listed by typing "LIST" or simply the letter "L". (All of the editor commands are single letters.)

```
L
10 MY VERY FIRST EDITOR LINE!
20 ENTERING LINES IN THE EDITOR IS
30 SIMILAR TO ENTERING LINES IN BASIC.

Done
```

(The "Done" message is printed whenever an editor command is successfully completed. We're not going to show it in all of our examples, though.)

Inserting lines in the text is much like from Basic. Simply enter a line number between the line numbers you want the text inserted into. Remember that after the insertion is made, however, the lines will be renumbered in steps of 10. Insert a line between line 10 and line 20 by giving it a line number of 15:

```
15 WITH SOME IMPORTANT EXCEPTIONS,
```

Now list the file again to see that the line was inserted and the following lines were renumbered:

```
L
10 MY VERY FIRST EDITOR LINE!
20 WITH SOME IMPORTANT EXCEPTIONS,
30 ENTERING LINES IN THE EDITOR IS
40 SIMILAR TO ENTERING LINES IN BASIC.
```

If the file being edited gets rather long, you don't have to list the entire file every time. The listing automatically stops every 16 lines. If you press ConTRoL-C during the pause, the listing will stop. If you press any other key, the listing will continue.

You can also use "List" to list a single line or a range of lines. Assuming a file contains at least 15 lines (numbered 10 to 150):

```
L 80      lists line 80 only.
L 80,120  lists lines 80 through 120.
L 80,     lists from line 80 to the end of the file.
L ,80     lists from the beginning of the file to line 80.
```

AUTONUM

The editor also provides automatic line numbering. Going back to the original example, list the file, then press "A" for "Autonum". The next line number, line 50, will appear for you. Enter a couple of lines with Autonum on:

```
A
50 THIS IS MUCH NICER THAN HAVING
60 TO ENTER THE LINE NUMBERS MYSELF.
70
```

To stop the Autonum feature, just press Return at the beginning of the line after the line number.

To change a line already in the editor file, simply retype the line number followed by the corrected line. The ESCape keys and the right-arrow key can be used to retype a line that is on the screen.

Entering a line number with no text creates a blank line; it does not delete the line as in Basic. Blank lines between word definitions are recommended, as they can make programs much more readable. If the Autonum feature is being used, a blank line can be created by typing a space followed by a Return.

DELETE

The "D" ("Delete") command is used for deleting a line or range of lines. Its format is identical to "List" (though its effects are very different!):

```
D 80      deletes only line 80.
D 80,120  deletes lines 80 through 120.
D 80,     deletes from line 80 to the end of the file.
D ,80     deletes from the beginning of the file to line 80.
```

After any lines are deleted, the remaining lines are again renumbered in steps of 10.

ERASE

To erase the file in memory, press "E" for "Erase". A prompt will appear:

```
Erase (Y/N) :
```

This prompt prevents inadvertent file erasure. Enter "Y" and press Return to erase the file.

Automatic Insertions

In a previous example, Autonum was used to add to the end of the file. When used in the middle of a file, Autonum also automatically inserts the text, making room for the text and renumbering later lines. For these examples, let's start with a new file. Erase the file in memory, then enter a couple of lines:

```
10 THE FIRST LINE IN THE FILE...
20 THE LAST LINE.
```

An insertion can be started by entering the first line number manually:

```
15 MUST SURELY BE FOLLOWED BY OTHERS.
```

Now, pressing "A" will cause automatic line numbering that starts following the last entered line (line 15) and insert this text into the file. Since line 15 is renumbered to become line 20,

the next line number, printed with the Autonum feature, is line 30:

```
A
30 AUTONUM DOES MORE THAN GENERATE
40 LINE NUMBERS.  IT ALSO INSERTS
50 INTO THE MIDDLE OF A FILE.
60
```

Again, Autonum is turned off by pressing Return with no text. List the file now:

```
L
10 THE FIRST LINE IN THE FILE...
20 MUST SURELY BE FOLLOWED BY OTHERS.
30 AUTONUM DOES MORE THAN GENERATE
40 LINE NUMBERS.  IT ALSO INSERTS
50 INTO THE MIDDLE OF A FILE.
60 THE LAST LINE.
```

INSERT

The "I" ("INSERT") command can also be used to initiate insertions into a file. Instead of typing the first inserted line before using Autonum, INSERT can be used to specify the starting line number. For this example, delete the lines just entered, then re-enter them, this time using INSERT.

```
D 20,50

Done

L
10 THE FIRST LINE IN THE FILE...
20 THE LAST LINE.
```

To insert between lines 10 and 20, enter:

```
I 15
```

Autonum will use this line number as the point of insertion, instead of the last accessed line.


```
A
20 MUST SURELY BE FOLLOWED BY OTHERS.
30 AUTONUM DOES MORE THAN GENERATE
40 LINE NUMBERS.  IT ALSO INSERTS
50 INTO THE MIDDLE OF A FILE.
60
```

List the file again, and you will see that these lines have been re-inserted into the file.

SAVE

To save a file to disk, press "S". A prompt will appear:

```
S
(Filename) :
```

Enter the file name you want the file to be saved under. If desired, you can also specify a disk slot and drive number here, separated by commas using the standard DOS format. Here are a couple of examples:

```
(Filename) : TESTFILE
(Filename) : TESTFILE,S6,D1
```

If you want to save only a portion of the file to disk, enter a slash after the filename, followed by the range of line numbers to be saved:

```
(Filename) : TESTFILE/80,150     saves lines 80 to 150
(Filename) : TESTFILE/,80        saves from the beginning of the
file to line 80
(Filename) : TESTFILE/80,        saves from line 80 to the end o
the file
(Filename) : TESTFILE,S6,D1/80,  slot and drive numbers can also
be used
```

GET

To get a file from disk and load it into the editor memory, press "G". A prompt will appear:

```
G
(Filename) :
```

Enter the name of the file to be loaded an , if desired, the disk slot and drive where it is located, using the same format as SAVE.

To get a file and insert it at a particular location in the existing file, enter a slash after the filename, followed by the line number where the file is to be inserted. This example will insert the file TESTFILE into the current editor file between lines 110 and 120:

(Filename) : TESTFILE/115

If you want to begin work on a new file without regard to the text currently in memory, simply use "E" to Erase the text memory before Getting the new file from disk.

Note: Since "GET" and "SAVE" use slashes to specify certain lines in a file, filenames that contain slashes cannot be used with the text editor.

DOS Commands

To enter a DOS command directly from the editor, press ConTRoL-D and Return. A prompt will appear:

Enter DOS Command :

From this prompt, you can enter any DOS command, to get a catalog, delete files, lock files, etc. The prompt repeats after each DOS command so that you can execute several commands without having to press ConTRoL-D every time. To return to the editor prompt (a flashing cursor with no prompt line), simply press Return twice.

Printing Files

Editor files can be printed directly from the editor. Type ConTRoL-D and Return to get the DOS prompt, then type "PR#1". (If your printer is in another slot, substitute that number.) The printer will be activated; press Return twice to remove the DOS prompt.

With the printer activated, you can type "L" to list the file to the printer, pressing Return when the listing stops every 16 lines. A better way is to type "W" for "Write". This option writes the editor file out without any pauses.

Since "PR#0" does not reconnect TransFORTH's special upper/lower case output, press Reset to turn the printer off and return to a normal display. Chapter Eight includes a discussion on how to access peripherals, print files, and return to TransFORTH in a normal manner under program control.

Leaving the Text Editor

To leave the text editor and return to TransFORTH, simply type "B" for "Bye". (TransFORTH is reinitialized when it is reentered. The words in the word library are untouched, but the stacks are cleared and any screen settings are returned to normal.)

Program Compilation

There are TransFORTH commands for reading the text from the editor memory or from a disk file, rather than the keyboard. They are actually general-purpose I/O routines, and will be discussed fully in Chapter Eight. Their use in compiling text from memory or disk is described below.

To read text stored in the editor memory into the TransFORTH system, type:

Ready PROGRAM MEMORY INPUT

Any text in memory will be read in and compiled. If the text contains immediate-mode commands, they will be executed as each line of text is read. If the text contains word definitions, the new words will be added to the library. If any errors occur during compilation, an error message will be displayed and the system will stop reading memory. The "Ready" prompt will appear when the system is finished reading.

Note that if you want to, you can define PROGRAM MEMORY INPUT as a single word to save typing:

```
: P
PROGRAM MEMORY INPUT ;
```

To read text stored in a textfile on disk, type:

Ready DISK> " <filename> " INPUT

where the name of the file is substituted for <filename>. The text will be read in the same way text in memory is read.

Comments

When PROGRAM MEMORY INPUT or DISK> " <filename> " INPUT are used for reading text, the TransFORTH system expects to read only valid TransFORTH words. However, comments and remarks in the file can often be very helpful for understanding and keeping track of long programs.

The TransFORTH word "(", a left parenthesis, is available for inserting comments into program files. In compiling the program, when TransFORTH sees a "(" set off with a space on either side, it ignores everything that follows until it sees a ")", also set apart with spaces. Comments can be inserted freely in the source file, and do not use any space on the word library when the program is compiled. Here is an example of a comment line:

```
10  (  PARENTHESES AROUND A COMMENT  )
```

A comment can extend over several lines, using one pair of parentheses. You should be careful to always finish comments with a ")". If you do not close a comment, TransFORTH will merrily accept the rest of the text, and whatever you type, without doing anything. This is similar to what can happen if you forget a semicolon at the end of a word definition. If this happens, try typing a right parenthesis ")" to return TransFORTH to normal. Pressing Reset will work, too.

Later examples in this manual include many comments. These comments are included to better explain the examples, but do not need to be entered with the programs.

Examples

Suppose you want to write and save a word definition that prints the squares of the numbers 1 through 10. One word definition to do this is:


```
: SQUARES
11 1 DO
   I I * . CR
LOOP ;
```

To create this word with the editor, first type "EDIT" to enter the editor. If there is already some text stored in the editor, type "E" then "Y" to erase it. Now type "A" to turn on the Autonum feature, and enter the word definition:

```
A
10 : SQUARES
20 11 1 DO
30    I I * . CR
40 LOOP ;
50
```

Press Return at line 50 to exit the Autonum feature. To save the text to disk, type "S" (for Save), then enter a filename:

```
S
(Filename) : DOSQUARES
```

(Note that filenames and word names are totally unrelated. You can use the same name for the word name and the filename if you like. If many word definitions are included in the text, giving the file the same name as the last word definition can be a handy way to keep track of what is in the file.)

The disk will whir as the file is saved. Now type "B" to exit the editor and return to TransFORTH. To compile the editor text into the system, type:

Ready PROGRAM MEMORY INPUT

The system will compile the text and return almost immediately with another "Ready" prompt. You can see that SQUARES has been added to the word library by typing LIST:

```
Ready LIST

SQUARES
NOTE
NEGATE
ABS
.
.
.
```

To run SQUARES, type:

```
Ready SQUARES
1
4
9
16
25
36
49
64
81
1ØØ
```

The squares of the numbers 1 through 1Ø are printed. Suppose that you now want to change the definition to print the numbers being squared along with the squares. You must first remove the old definition from the word library:

```
Ready FORGET SQUARES
```

Now enter the editor again:

```
Ready EDIT
```

and list the program:

```
L
1Ø : SQUARES
2Ø 11 1 DO
3Ø    I I * . CR
4Ø LOOP ;

Done
```


The following addition will change the definition to print the numbers being squared:

```
25    I . SPCE

L
1Ø : SQUARES
2Ø 11 1 DO
3Ø    I . SPCE
4Ø    I I * . CR
5Ø LOOP ;

Done
```

Now exit the editor, then recompile the text from the TransFORTH system:

```
Ready PROGRAM MEMORY INPUT

Ready SQUARES
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
1Ø 1ØØ
```

Since the original definition of SQUARES (not the latest version) is saved on the disk, you can replace the new version with the old again:

```
Ready FORGET SQUARES

Ready DISK> " DOSQUARES " INPUT
```

The disk will whir as the textfile is compiled.

```
Ready SQUARES
1
4
9
16
25
36
49
64
81
100
```

Of course, the editor text can include more than one word definition, or a combination of word definitions and immediate-mode lines. Try this step-through example. A description of how it works follows.

```
Ready EDIT

(Now in the editor:)

E

Erase file (Y/N) : Y

Done

A
10 PRINT " NOW COMPILING WORD DEFINITIONS " CR
20
30 : AVERAGE
40 + 2 / ;
50
60 : SEE.AVERAGE
70 OVER OVER
80 . SPCE . SPCE
90 AVERAGE . CR ;
100
110 PRINT " WORDS HAVE BEEN COMPILED " CR
120
130 5 3 AVERAGE . CR
140 5 3 SEE.AVERAGE
150

B
```



```
(Now back in TransFORTH:)

Ready PROGRAM MEMORY INPUT
NOW COMPILING WORD DEFINITIONS
WORDS HAVE BEEN COMPILED
4
3 5 4
```

The editor file entered is a mixture of immediate commands and word definitions. Remember that the TransFORTH system reads the editor file line by line. The first line prints the message "NOW COMPILING WORD DEFINITIONS", then the two word definitions, AVERAGE and SEE.AVERAGE are compiled onto the word library. AVERAGE removes two numbers from the stack and averages them, leaving the result on the stack. SEE.AVERAGE removes two numbers, copies and prints them, calls AVERAGE to find their average, then prints the result to the screen.

After the colon definitions are compiled, line 110 is read and executed, printing "WORDS HAVE BEEN COMPILED". Line 130 finds and prints the average of 5 and 3 by calling the new word AVERAGE, and line 140 uses SEE.AVERAGE to do both the averaging and printing.

If you were to try using PROGRAM MEMORY INPUT to read the text again, the system would compile the two word definitions a second time, producing "Not Unique" errors.

Memory Considerations

The amount of usable editor file memory is determined by the presence or absence of the extra 16K of memory. In a 48K system, the editor can store up to 12,285 characters. In a 64K Apple, the editor can store up to 22,013 characters.

When editing programs, to find the amount of free memory left in the editor file area, press "M" for "Memory". You will see:

```
Free Memory
```

followed by the number of bytes (or characters) of memory left.

The TransFORTH system, Apple DOS, the text editor program, and the text being edited all reside in the Apple memory simultaneously. As long as the TransFORTH word library has not grown too large, there aren't any problems with memory conflicts.

However, if the word library grows large enough, it can begin to overwrite the editor text in memory. Conversely, if the word library is already large, using the text editor can overwrite the top of the word library, crashing the system.

With large programs, it is a good idea to remain generally aware of what areas of memory in the Apple are being used. The memory map in Appendix B shows how TransFORTH uses Apple memory.

In some cases you may have a very large word library, but not need much space for editing. In other situations you may want to edit a very large textfile, but not use much word library space. To accomodate these situations, the text editor includes a command that allows you to change the starting address of the text buffer in memory. By making this address higher, you can leave more room for the word library to grow. By making it lower, you can create more space for editing text.

To change the position of the editor file, press "P". A display will appear:

Program Length
Position
Free Memory
Change Position (Y/N) :

The Program Length, Position (starting address of the editor file area), and Free Memory labels will be followed by their present numeric values. To change the editor file position, enter "Y". You will be prompted:

Enter New Position :

Enter the address where you want the editor text to begin. Note that this must cover a valid address area. If you enter an invalid address (zero for example), any editing will most likely crash the TransFORTH system.

(Note: The Position shown will always be 1 greater than the position value you type in. (This is because the first byte of the editor buffer is a special beginning marker.) For example, if you enter a position of 30000, then run the ("P") Program Position command again, it will show a Position value of 30001.)

For extremely large programs on the word library, there may not be enough memory available to support both the word library and the editor file (or even the editor program itself). In this case, extra care must be taken: Before entering the editor, use

FORGET to remove any extra words you've defined from the word library. Enter the editor, adjust the text position if necessary, erase any stray characters left in the buffer, load the text from disk, and make any additions or changes. Then be sure to save the text back to disk before leaving the editor. Return to TransFORTH, and use either PROGRAM MEMORY INPUT or DISK> " <filename> " INPUT to compile the text. If you have to return to the editor, again use FORGET to clear your words from the word library. This will prevent the top of the library from being destroyed by subsequent editing.

As you become more comfortable with programming in TransFORTH, you will probably want to use the editor to list some of the program files on the system diskette. We encourage you to do this. The system files provide excellent programming examples in TransFORTH. Appendix D provides a discussion of some of these files. You can use the editor to catalog the disk, then Get and List the files, or compile the files onto the word library and run them.

Program examples later in the manual are printed in a simple list form. You can enter the examples using whatever method you prefer. For short examples, you may want to type them directly at the keyboard. For longer examples, you will probably want to use the TransFORTH text editor or another editor if you have one. Each method works equally well, and we leave the decision to you.

Summary

The TransFORTH system does not allow you to modify word definitions once they have been compiled. They must be deleted with FORGET, then re-entered. Using a text editor to save the text of the word definitions can make this process much simpler. TransFORTH includes words for reading and compiling text stored in memory or in a textfile on disk. Any DOS-compatible editor can be used; the TransFORTH text editor is convenient because it can reside in memory along with TransFORTH.

To enter the TransFORTH editor, type EDIT. Inside the editor, lines are indexed with line numbers. After any insertions or deletions, lines are automatically renumbered in steps of 10.

All editor commands can be entered as single letters. The commands include:

```
?              Lists the editor commands
Save           Save the text in memory to disk
Get            Reads a textfile on disk into memory
Insert         Specifies a line number for insertion with Autonum
Delete         Deletes one or more lines
Program        Allows adjustment of the text buffer position
Memory         Displays amount of free memory in the editor
List           Lists one or more lines of text
Write          Lists without pauses
Erase          Erases text in memory
Autonum        Provides line numbers and allows insertions
automatically
Bye            Exits the editor
ConTRol-D=DOS  Allows DOS commands to be entered
```

The TransFORTH command PROGRAM MEMORY INPUT reads and compiles the text residing in editor memory, treating it as if it were being typed from the keyboard. The command DISK> " <filename> " INPUT reads a textfile on disk in the same way. Comments can be included in a file if surrounded by parentheses.

CHAPTER SIX: DATA STRUCTURES

CHAPTER TABLE OF CONTENTS: Page

Variables **6-1**

Number Format and Storage **6-5**

Storage and Retrieval Words 6-5

Arrays **6-7**

Accessing Array Elements 6-8
Clearing Arrays 6-11
Array Error Checking 6-11
Array Sizes 6-12
Memory Arrays 6-12
Accessing Arrays from Loops 6-13
Sequential Memory Access 6-15

Strings **6-17**

String to Number Conversion 6-19
"Arrays" of Strings 6-21
PAD: The System String 6-22
String Manipulation Words 6-23
Accessing Individual Characters 6-26
Character Input and Output 6-27

Combining Text and Numerical Data in an Array **6-29**

Summary **6-31**

Problems **6-32**

Solutions to Problems 6-35

All of the numbers used in previous examples have been stored on the stack. The data stack provides an excellent temporary storage for a few numbers at a time, but is not well suited for storing and manipulating many values at once.

TransFORTH includes words that allow you to set aside separate variables, arrays, and strings for storing numerical and text data. In addition, TransFORTH includes capabilities for reading or writing values to any location in the Apple memory. This chapter will discuss how these words are used for more convenient and complete data manipulation.

Variables

TransFORTH allows you to set aside space for number storage through the word "VARIABLE". VARIABLE is a special word, in that it creates a new word and places it on the TransFORTH word library. In this way it is similar to the TransFORTH word ":" (colon). VARIABLE has two forms; the first one is:

VARIABLE <variable name>

The variable name is the name of the word created and placed on the word library. For example:

```
Ready LIST

NOTE
NEGATE
ABS
.
.
.

Ready VARIABLE TEMP

Ready LIST

TEMP
NOTE
NEGATE
ABS
SGN
.
.
.
```


The new word TEMP (created with VARIABLE) consists of two parts: a five-byte space set aside for storing a floating-point number, and a call to an internal TransFORTH routine that either places the value of the variable on the stack or stores the stack value into the variable.

To recall the value stored in the variable TEMP, just type its name:

```
Ready TEMP
[ Ø ]

Ready DROP
```

When TEMP was created with the word VARIABLE, it was given an initial value of zero. By calling TEMP, this value was copied onto the stack. With the value of the variable on the stack, it can be used by any appropriate TransFORTH word.

To store the number 123.45 into TEMP, type:

```
Ready 123.45
[ 123.45 ]

Ready -> TEMP

Ready
```

The TransFORTH word "->" is a special word that says "store into". It is created by typing a minus sign "-" followed by a greater-than sign ">". This word sets an internal flag used by variables to determine if a "store" or "recall" operation is to take place. When the "->" word is executed it sets this flag so that the next referenced variable will perform a store, rather than a recall. Calling the variable will also clear the flag so that subsequent variable accesses will do a recall unless the "->" word is executed again.

Whenever you need to recall the value of a variable, simply type its name. To store a value into a variable, always type the TransFORTH word "->" before typing the variable name.

For convenience, we'll call the "->" word a "store-arrow" and read the example:

```
Ready 123.45 -> TEMP
```

as "123.45 store-arrow TEMP", or simply "123.45 is stored into TEMP". You can see that 123.45 has been stored, by recalling the value of TEMP again:

```
Ready TEMP .
123.45

Ready -67 -> TEMP

Ready TEMP .
-67
```

Unless otherwise specified, when a variable is first created and compiled using the word VARIABLE, the initial value of the variable is zero. To give a variable a different initial value, the second form of VARIABLE is used, where the initial value is entered on the line with the declaration:

```
<initial value>  VARIABLE  <variable name>

Ready 35 VARIABLE COUNT
```

COUNT will contain the value 35 until another value is stored over it:

```
Ready COUNT .
35

Ready 87 -> COUNT

Ready COUNT .
87
```

If you need to use the same number many times in a program, you can put the number in a variable that has a descriptive name, then call the variable rather than entering the number every time. This saves program space and can often help make a program more readable.

We should bring up something important here. The word VARIABLE (as well as ARRAY and MARRAY, which we'll discuss shortly) is a compiling word, in that it produces new words itself. It is also a word that looks forward down the input line for the word name.

It therefore must be used with more care than most TransFORTH words.

To be specific, a VARIABLE declaration cannot appear inside of a colon definition. It should be alone on its own line, not mixed with other TransFORTH words. Any initial value provided when the variable is declared is taken directly from the input line, not from the stack. Since the initial value is not from the stack, it cannot be a computed number. For example, the following line will not work:

```
Ready 25 7 / VARIABLE THING
```

Using variables, TransFORTH expressions are often easier to read. For example, this line of Basic:

```
Q = P^2*3 + P*5 + 4
```

can be translated into TransFORTH (after defining the variables) as:

```
P 2 ^ 3 *  P 5 * +  4 +  -> Q
```

An equivalent TransFORTH expression that doesn't use variables would be:

```
DUP 2 ^ 3 * SWAP 5 * + 4 +
```

Here is a short program that removes 5 numbers from the stack, then prints the largest of the numbers. A variable is used to keep track of the maximum value:

```
VARIABLE X

: MAX5
-> X            ( Put 1st number into variable as "maximum" )
4 Ø DO          ( Loop to get next 4 numbers )
   DUP X >      ( Copy number, then compare to current "maximum" )
   IF -> X      ( If greater, save it as new maximum )
   ELSE DROP    ( Otherwise, don't need it )
   THEN
LOOP
X . ;           ( Print maximum )
```

```
Ready 7 4 -26.2 107 .003 MAX5
107

Ready 96 0 1 -200.1 .6 MAX5
96
```

Number Format and Storage

The Apple][memory space is divided into 65,536 separate locations. Each location has a numbered "address", ranging from 0 to 65,535. Most locations contain RAM; some are used for ROM or I/O. The memory map in Appendix B shows how these areas are divided, and how TransFORTH uses Apple memory. (For more information on Apple memory locations, consult the Apple Reference Manual.)

Any single RAM location contains one byte of memory, and can store an integer number between 0 and 255. Two consecutive locations used together can hold a number as large as 65535. TransFORTH internally uses a floating-point format that uses 5 bytes of memory. All values on the stack and in variables are stored in the floating-point format.

You can store numbers anywhere in the Apple memory in any one of these three formats: Numbers between 0 and 255 can be stored as single bytes in integer format. Values between 0 and 65535 can be stored in the two-byte integer format. (TransFORTH automatically makes the conversion between integer and floating-point format when moving numbers between the stack and other memory.) Any numbers out of the integer range must be stored using the floating-point format, which uses five bytes of memory per value.

Storage and Retrieval Words

There are six separate words for storing and retrieving numbers in memory: PEEK, PEEKW, PEEKN, POKE, POKEW, and POKEN. The "POKE" words are used for storing numbers into memory; the "PEEK" words are used for retrieving numbers:


```
PEEK   - Reads 1 byte (0 to 255) from memory
PEEKW  - Reads 2 bytes (0 to 65,535) from memory
PEEKN  - Reads 5 bytes (floating-point number) from memory

POKE   - Stores 1 byte (0 to 255) into memory
POKEW  - Stores 2 bytes (0 to 65,535) into memory
POKEN  - Stores 5 bytes (floating-point number) into memory
```

PEEKW and POKEW stand for "PEEKWord" and "POKEWord", because a number stored as two bytes is sometimes called a "word" (not to be confused with TransFORTH words). PEEKN and POKEN stand for "PEEKNumber" and "POKENumber".

The format for the three POKE words is:

```
<value>  <address>  POKE
```

The POKE words remove two numbers from the stack, the value to be stored and the address it is to be stored into. They then poke the value into that address. For POKEW, the two-byte number is stored into given location <u>and</u> the one immediately after it. For POKEN, the given location <u>and</u> the next four are used to store the floating-point value.

For example, this line will store the number 697 into locations 20000 and 20001 as a two-byte integer:

```
Ready 697 20000 POKEW
```

The next line pokes the number 2.3 in floating-point format into the 5 locations starting with address 128:

```
Ready 2.3 128 POKEN
```

(POKEW will only poke numbers up to 65535. If you try to POKEW a number greater than this, TransFORTH will instead poke a 0. Also, if you try to POKE a number greater than 255, TransFORTH will poke only the first byte of the number.)

The format for the three PEEK words is:

```
<address>  PEEK
```

The PEEK words remove one number from the stack, interpret this number as an address, read a value from that address, then place the value on the stack. This example reads the number that was poked into location 20000:

```
Ready 20000 PEEKW .
697
```

Note that a number stored into memory in one format usually cannot be read correctly in another format. This example tries to read the floating-point number at location 128 using the one-byte integer format:

```
Ready 128 PEEK .
51
```

PEEK returned a wrong value, 51 rather than 2.3, because different formats were used for POKEing and PEEKing.

To summarize, here is a table of the six storage and retrieval words:

Word	Before	After	Description
POKE	n a	-	Puts single-byte n into address a
POKEW	n a	-	Puts two-byte n into address a
POKEN	n a	-	Puts floating-point n into address a
PEEK	a	n	Reads single-byte n from address a
PEEKW	a	n	Reads two-byte n from address a
PEEKN	a	n	Reads floating-point n from address a

Arrays

Arrays in TransFORTH are words with space set aside for storing many numbers. Each number, or element, stored in an array is accessed by entering the array name along with one or more index numbers (called "subscripts"). Arrays can be declared with any number of dimensions, limited only by available memory. A one-dimensional array uses one subscript to access each element, a two-dimensional array uses two subscripts, etc. Arrays can also be created for storing text and string data, or a combination of text and numbers.

TransFORTH arrays provide an extremely versatile method of data storage, but require a little extra care to use. We'll introduce the simpler uses of arrays first, then move on to the more sophisticated applications.

Arrays are created with the TransFORTH word ARRAY. ARRAY is a word that, like VARIABLE, places a new word on the word library. The form for ARRAY is:

<element size> <dimension1> <dimension2>. . . <# of dimensions> ARRAY <array name>

<array name> is the name of the new array word placed on the word library. Working backwards from the word ARRAY, <# of dimensions> is a number specifying how many dimensions the array will contain. Before this number, there should be one or more numbers designating how many elements lie along each dimension, one number per dimension. The first value on the line, <element size>, specifies how many bytes each element contains.

The following examples should clarify this:

```
Ready 5 10 20 2 ARRAY GRID
```

This line creates a new TransFORTH word, an array named GRID. The array has two dimensions, and is 10 elements "wide" by 20 elements "deep", for a total of 200 elements. Each element of the array can hold 5 bytes, or one floating-point number.

```
Ready 2 200 1 ARRAY NUMBERS
```

This example creates an array named NUMBERS. NUMBERS is a one-dimensional array with 200 elements (see "Array Sizes" below), each 2 bytes in length.

```
Ready 5 8 8 8 3 ARRAY CUBE
```

This line creates a three-dimensional array named CUBE, with 8 elements on each side, storing 5 bytes per element.

It should be pointed out that, like VARIABLE, ARRAY is a compiling word, and draws all of its information from the input line, not the stack. This means that the same restrictions that were discussed for variables also apply for arrays: Array declarations must be on separate lines outside of colon definitions, and the numbers used on the line cannot be computed values.

Accessing Array Elements

You can access an individual array element by placing one or more subscript numbers on the stack, then entering the array name. A

one-dimensional array requires one subscript, a two-dimensional array requires two, etc. For example, "6 14 GRID" will access element (6,14) from the array GRID. "199 NUMBERS" will access the 199th element in NUMBERS, and "1 2 3 CUBE" accesses element (1,2,3) in CUBE. Note that the array words "know" how many dimensions they contain, and will pull the appropriate number of subscript values from the stack when executed. The correct values must therefore be waiting on the stack.

Accessing an array element as just described places the Apple address of that array element on the stack. With the address available, the PEEK and POKE words can be used to store and retrieve values from the array. Here is an example:

```
Ready 25 NUMBERS
[ 13966 ]
```

The Apple address of the 25th element of NUMBERS is 13966. (Note: This address is shown for example purposes. Depending on what other words lie on the word library, you may get a different address.) With the address of the array element on the stack, you can now use POKEW to store a value into that element:

```
Ready 256
[ 13966 ]
[ 256 ]

Ready SWAP
[ 256 ]
[ 13966 ]

Ready POKEW
```

POKEW removes the two numbers from the stack, storing the 256 into location 13966, which is element 25 in NUMBERS. POKEW is the appropriate word to use, since it stores two bytes into memory, and NUMBERS was defined as an array of 2-byte elements. By placing the values on the stack in the order needed by POKEW, the SWAP shown above is not needed:

```
Ready 256
[ 256 ]

Ready 25 NUMBERS
[ 256 ]
[ 13443 ]

Ready POKEW
```


The above example can also be duplicated on one line:

```
Ready 256  25 NUMBERS  POKEW
```

The number can be retrieved using PEEKW:

```
Ready 25 NUMBERS PEEKW .
256
```

Here is another example. A two-dimensional array is declared, with an element length of one byte. Two values are stored into the array, then read back out:

```
Ready 1 6 7 2 ARRAY THING

Ready 208  6 7 THING POKE

Ready 212  6 6 THING POKE

Ready 6 7 THING PEEK .
208

Ready 6 6 THING PEEK .
212
```

Arrays with five-byte elements are accessed in the same way:

```
Ready 987.654  3 4 5 CUBE  POKEN

Ready 1.05946309  7 7 4 CUBE  POKEN

Ready 3 4 5 CUBE PEEKN
[ 987.654 ]

Ready 7 7 4 CUBE PEEKN
[ 987.654 ]
[ 1.05946309 ]

Ready DROP DROP
```

Clearing Arrays

When an array is declared, all of its elements are initially cleared to zeros. You can also clear an array to all zeros at any time with the word ERASE. ERASE has the form:

ERASE <array name>

To clear the array THING to zeros, enter:

Ready ERASE THING

The word ERASE can be used freely inside of colon definitions, but is only intended for erasing arrays. Following the word ERASE with something other than an array name will confuse, and probably hang, the TransFORTH system.

Array Error Checking

When array elements are accessed, TransFORTH does not check if the index numbers are in bounds for the array. For example, TransFORTH will not prevent you from trying to access element (300,300) in the array THING, even though THING is only a 6 by 7 array. Reading an out-of-bounds array element will simply return an invalid number from somewhere in the Apple memory. However, writing to an out-of-bounds element will store a number into Apple memory, overwriting its previous contents. If this memory is in an important part of the TransFORTH system or the disk operating system, the computer will most likely hang. Therefore, do not write to an element outside of the array.

Array error checking was left out of TransFORTH on purpose. The reason for this is that error checking is a time-consuming process that contributes nothing to a program if the program already works correctly. If you need error checking, you can write a routine that checks the numbers before they are passed on to the array name to select an element. If you don't need error checking, you don't have to include it.

Array Sizes

Arrays have slightly more storage capacity than previously suggested. A subscript number of zero will select a valid element for any array. This means that an array declared as:

5 10 10 2 ARRAY EXAMPLE

can index along each dimension from 0 to 10, which is 11 elements. The entire array actually contains 11 * 11 = 121 elements, not 100 elements. (For the sake of continuity, we will continue to call this a "10 by 10" array.)

Memory Arrays

Another kind of array, called a memory array, can be created using the word MARRAY. This array is nearly identical to a normal array. The difference is that the array data itself can reside anywhere in the Apple memory. The form for MARRAY is similar to ARRAY, except that an extra number is included which specifies the starting address for the array.

Ready 5 8 8 2 32768 MARRAY STUFF

This example creates a memory array named STUFF. It is a two-dimensional (8 by 8) array of 5-byte floating-point values, and the array elements themselves begin at location 32768 in the Apple memory.

As will be discussed in Chapter Seven, TransFORTH can call machine language routines that lie in free areas of memory. A memory array provides one way of passing data to and from these routines, since the array can be kept in a fixed place in the Apple memory, regardless of the size of the word library. (It is up to the user make certain that the memory array resides in a free area of memory.)

Accessing Arrays From Loops

DO - LOOPs provide an excellent way of printing the contents of arrays. In the following example, a few values are stored one at a time into a one-dimensional array, then the the elements are printed from a loop. The loop value ("I") is used to select each element in turn:

```
Ready 5 1Ø 1 ARRAY SHOW

Ready 243  2 SHOW  POKEN

Ready 1.1  6 SHOW  POKEN

Ready -19  7 SHOW  POKEN

Ready 11 Ø DO I SHOW PEEKN . SPCE LOOP
Ø Ø 243 Ø Ø Ø 1.1 -19 Ø Ø Ø
```

All eleven elements of the array (numbered Ø to 1Ø) were printed out from the loop. The following word definition also prints the subscript number along with the value of each element. (You can enter the definition either "live" from the keyboard or using the TransFORTH text editor.):

```
: SEE.SHOW
11 Ø DO
   I . SPCE
   I SHOW PEEKN . CR
LOOP ;

Ready SEE.SHOW
Ø Ø
1 Ø
2 243
3 Ø
4 Ø
5 Ø
6 1.1
7 -19
8 Ø
9 Ø
1Ø Ø
```

An array with two or more dimensions can be accessed using nested loops. This example includes a word definition that fills an

array with numbers, and two word definitions for printing the values in the array:

```
5 3 4 2 ARRAY SHOW2

: STUFFIT
4 Ø DO                       ( Loop for first index )
   5 Ø DO                    ( Loop for second index )
      J 1Ø * I +             ( Outer loop value * 1Ø + inner loop
                               value )
      J I SHOW2 POKEN        ( is stored into element of SHOW2 )
   LOOP
LOOP ;

: SEEIT1
5 Ø DO
   4 Ø DO
      I J SHOW2 PEEKN .
      SPCE
   LOOP
   CR
LOOP ;

: SEEIT2
4 Ø DO
   5 Ø DO
      J I SHOW2 PEEKN .
      SPCE
   LOOP
   CR
LOOP ;
```

STUFFIT places a number in each element of the array, using the loop values as indices into the array. Executing the nested loop is equivalent to:

```
 Ø   Ø Ø SHOW2  POKEN
 1   Ø 1 SHOW2  POKEN
 2   Ø 2 SHOW2  POKEN
 3   Ø 3 SHOW2  POKEN
 4   Ø 4 SHOW2  POKEN
1Ø   1 Ø SHOW2  POKEN
11   1 1 SHOW2  POKEN
12   1 2 SHOW2  POKEN
 .          .
 .          .
32   3 2 SHOW2  POKEN
33   3 3 SHOW2  POKEN
34   3 4 SHOW2  POKEN
```

Note that the only difference between SEEIT1 and SEEIT2 is the order of the loops. SEEIT1 increments the first index 4 times for every increment of the second index. SEEIT2 increments the second index more often than the first. After compiling the word definitions, they can be executed:

```
Ready STUFFIT

Ready SEEIT1
Ø 1Ø 2Ø 3Ø
1 11 21 31
2 12 22 32
3 13 23 33
4 14 24 34

Ready SEEIT2
Ø 1 2 3 4
1Ø 11 12 13 14
2Ø 21 22 23 24
3Ø 31 32 33 34
```

Notice that the orientation of the array simply depends on the order of the loops. By using the correct order, you can display arrays with either: first index=columns, second index=rows; or second index=columns, first index=rows.

Sequential Memory Access

For most applications, it doesn't matter what order TransFORTH actually stores the array elements in memory. For some string applications, however, this information can be helpful.

In arrays with 2 or more dimensions, the elements are stored with

the first index being incremented most often. For example, if an array is defined as:

```
Ready 1 3 4 2 ARRAY IT
```

then TransFORTH stores the elements in the following order:

```
Ø Ø IT
1 Ø IT
2 Ø IT
3 Ø IT
Ø 1 IT
1 1 IT
2 1 IT
  .
  .
1 4 IT
2 4 IT
3 4 IT
```

The next section on strings will discuss when this order is important to TransFORTH programs.

Strings

Strings are used when text data, rather than numerical data, need to be manipulated. Some languages, such as Applesoft Basic, have separate string data types (e.g. "A$") for handling strings. In TransFORTH, arrays are used for storing string data. Each text character uses one byte in the array. A number of TransFORTH words are included for reading, writing, and manipulating strings in memory.

(Note: If you have been entering and running the examples of the previous section, there are now a number of arrays and colon definitions on the word library. Now would be a good time to FORGET them, to clear space for the next set of examples.)

Before a string can be stored, an array must be declared. The following example creates a one-dimensional array with 50 single-byte elements. When a string is stored in this array, one character will be stored as an ASCII value in each byte, or each element of the array:

```
Ready 1 50 1 ARRAY STR1
```

To store text into an array, the word ASSIGN> is used. The form for ASSIGN> is:

```
<address>  ASSIGN> " <text> "
```

ASSIGN> removes a number from the stack, interprets this number as an address, then places the following quoted text into memory starting at that address. Usually the address is supplied by entering the name of a string array before typing ASSIGN>. Here is an example:

```
Ready 0 STR1
[ 12628 ]

Ready ASSIGN> " SHE SELLS SEASHELLS "
```

The phrase "SHE SELLS SEASHELLS" has been stored into the array STR1. Each character occupies one element of the array.

To write the contents of a string array to the screen, the word WRITELN is used. WRITELN removes a number from the stack, interprets it as a memory address, then writes the text starting

at that address to the screen. The form of WRITELN is:

```
<address>  WRITELN
```

The following example writes the contents of the string STR1:

```
Ready 0 STR1 WRITELN
SHE SELLS SEASHELLS
```

Note: WRITELN, like "." (period), doesn't print any spaces or carriage returns after writing the string. You can use SPCE or CR to add these if you want.

Text can be read in from the keyboard and stored in a string (or anywhere in memory) using the word READLN. READLN removes a number from the stack, interprets it as an address, then reads a line of text from the keyboard and stores the text into memory starting at that address. Like WRITELN, the form for READLN is:

```
<address> READLN
```

Here is an example:

```
Ready 0 STR1 READLN
SEASHELLS                ( You type this line )
```

The word "SEASHELLS" has been read into the string STR1, overwriting its previous contents.

```
Ready 0 STR1 WRITELN
SEASHELLS
```

Of course, assigning, reading, and writing don't have to start at the beginning of a string array. Strings can be modified by reading into the string, but starting with an array element in the middle of the string:

```
Ready 3 STR1 READLN
SHORE

Ready 0 STR1 WRITELN
SEASHORE
```

The word "SHORE" was read into STR1, starting at element 3 (character number 3), over the top of "SHELLS".

```
Ready 2 STR1 WRITELN
ASHORE
```

The string was printed starting with character number 2, bypassing the "SE" in "SEASHORE".

Here is a short example word definition which shows a simple use of READLN and WRITELN:

```
: ASK.NAME
PRINT " WHAT IS YOUR NAME?  "
Ø STR1 READLN                     ( Read name into STR1 )
PRINT " HI,  "                    ( Print "HI" )
Ø STR1 WRITELN                    ( then name from STR1 )
PRINT " .  ALL DONE... " ;        ( then "ALL DONE...". )

Ready ASK.NAME
WHAT IS YOUR NAME? FREDDY
HI, FREDDY.  ALL DONE...
```

When a string is stored into memory using ASSIGN> or READLN, a single-byte zero is placed after the last character, marking the end of the string. When WRITELN writes a string from memory, it starts at the specified string address and continues until it finds either a carriage return or a byte containing a zero. Either of these mark the end of a string for WRITELN. (WRITELN also recognizes a user-definable End-Of-File-CHaRacter, which will be introduced in Chapter Eight.)

Note: STR1 was defined to store 5Ø elements, or 5Ø characters. If a string longer than 5Ø characters (including the end-of-string marker) is read into STR1, the extra characters will be stored past the end of STR1 over a part of the TransFORTH system, causing the system to hang! Storing strings that are too large for an array is equivalent to writing numbers into an out-of-bounds array element. The moral of the story is: Make certain that the string array is large enough to accommodate anything that might be read into it. There are a couple of techniques to make this a little easier; these will be discussed shortly.

String to Number Conversion

Sometimes a string will contain a number stored as text. You can use the TransFORTH word GETNUM to read the number from the text, placing the number on the stack. GETNUM removes a number from the stack, again interpreting it as a memory address. It then reads the text starting at that address and attempts to find a number, which it places on the stack.

In the following example, the number 321 is first read into a string as text, then converted to a stack value with GETNUM:

```
Ready Ø STR1 READLN
321

Ready Ø STR1 GETNUM
[ 321 ]
```

When using GETNUM, nonnumeric characters may follow the number without interfering with the conversion, but the number must begin as the first character of the string.

If GETNUM cannot find a number in the text at the given string address, it places a zero on the stack. To determine for certain whether or not the string-to-number converion was successful, the word VALID is used. VALID leaves a number on the stack. If the last GETNUM was successful, the number will be nonzero; if the conversion failed, VALID will return zero:

```
Ready Ø STR1 ASSIGN> " -55 "

Ready Ø STR1 GETNUM .
-55

Ready VALID .
254
```

VALID is nonzero since GETNUM was able to convert the number.

```
Ready Ø STR1 ASSIGN> " YOU CALL THIS A NUMBER?? "

Ready Ø STR1 GETNUM .
Ø

Ready VALID .
Ø
```

VALID is zero since GETNUM failed to find a number.

Whenever you want to read a number from the keyboard in a program, simply READLN it into a string as text, then convert it to a number with GETNUM. For convenience, you can define this operation as a single TransFORTH word:

```
: READ.NUMBER
Ø STR1 READLN
Ø STR1 GETNUM ;

Ready READ.NUMBER
33.6                    ( You type this. )

[ 33.6 ]                ( READ.NUMBER returns this. )
```

"Arrays" of Strings

Because arrays can be created with any practical number of dimensions, an array can be used for storing more than one text string. One method for storing a number of strings is to define a two-dimensional array:

```
Ready 1 5Ø 15 2 ARRAY STR2
```

This creates a string array with 15 "rows" of 5Ø characters. Because the characters in a string array are stored in sequential bytes, the proper index must be used to access the 15 rows. (See "Sequential Memory Access", discussed earlier.) For example:

```
Ready Ø Ø STR2 READLN
HE IS INNOCENT

Ready 3 Ø STR2 READLN
KILLED IT

Ready Ø Ø STR2 WRITELN
HE KILLED IT
```

In the above example, 3 Ø STR2 is only three locations away from Ø Ø STR2. Thus, the second string partially overwrote the first. This can be avoided by incrementing the second index instead of the first index:

```
Ready Ø Ø STR2 READLN
HE IS INNOCENT

Ready Ø 3 STR2 READLN
THE MAID DID IT

Ready Ø Ø STR2 WRITELN
HE IS INNOCENT
```

(The first string is now unharmed by the second.)

Probably the best way to create an array of strings is to use an array element length that is at least as long as the longest string. Then the string will be written as a sequence of bytes within a single element of the array. This example creates a string array with 15 strings of 5Ø characters each:

```
Ready 5Ø 15 1 ARRAY STR3

Ready Ø STR3 READLN
WHAT WAS THE MOTIVE?

Ready 3 STR3 READLN
IT WAS THE MONEY

Ready Ø STR3 WRITELN
WHAT WAS THE MOTIVE?
```

(Again, the second string does not overwrite the first.)

The middle of a string can still be accessed by adding an offset to the address:

```
Ready 3 STR3
[ 13811 ]

Ready 7 +
[ 13818 ]

Ready WRITELN
THE MONEY
```

PAD: The System String

TransFORTH includes a predeclared temporary string space of 144 characters called PAD. PAD is convenient for reading keyboard input without having to define a string first.

Actually, PAD is two things: a 144-byte free area of memory used for storing string data, and a word in the TransFORTH word library named PAD which places the address of this free area of memory on the stack. Note that the usual array indexing is not used with PAD:

```
Ready PAD

[ 832 ]
```

(832 is the address of the PAD string buffer.)

[832]
Ready READLN
WHY AM I HERE?

Ready PAD WRITELN
WHY AM I HERE?

To access the middle of the PAD buffer, simply add an offset to the address:

Ready PAD
[832]

Ready 4 +
[836]

Ready WRITELN
AM I HERE?

Note: PAD is considered a <u>temporary</u> string space because the same memory is used by the TransFORTH system when creating arrays, overwriting the previous contents of PAD. Predeclared string arrays should be used for more permanent string storage. In addition, since PAD is not an actual array, ERASE cannot be used to erase the contents of PAD.

String Manipulation Words

TransFORTH also includes a number of words for manipulating string data in various ways.

The TransFORTH word LENGTH removes a string address from the stack and returns the length (number of characters) of that string:

Ready PAD ASSIGN> " HOW LONG AM I? "

Ready PAD LENGTH
[14]

The LENGTH value returned does not include the end-of-string marker. Remember that array indexing starts at element Ø and ends at the string length-1. If a string is 3 characters long, it is stored at character positions Ø, 1, and 2, with the zero

end-of-string marker at character position 3. (A string of length 3, then, actually requires 4 bytes of memory.)

Pressing only Return when READLNing a string returns a "null" string with a length of zero:

Ready PAD READLN
(Press Return.)

Ready PAD LENGTH
[Ø]

MOVELN simply copies a string from one address to another. The form is:

<source> <destination> MOVELN

The following example reads a string into PAD, then copies the contents of PAD to STR1:

Ready PAD READLN
ONE GOOD STRING LEADS TO ANOTHER

Ready PAD Ø STR1 MOVELN

Ready Ø STR1 WRITELN
ONE GOOD STRING LEADS TO ANOTHER

CONCAT concatenates two strings together. The form for CONCAT is:

<string1> <string2> CONCAT

CONCAT copies the contents of <string2> to the end of <string1>. The contents of <string2> are unchanged. In this example, strings are read into both PAD and STR1, then CONCAT is used to combine the strings in PAD:

Ready PAD READLN
STUCK-

Ready Ø STR1 READLN
TOGETHER

Ready PAD Ø STR1 CONCAT

```
Ready PAD WRITELN
STUCK-TOGETHER
```

COMPARE makes an alphabetical comparison between two strings, returning a value on the stack. The form for COMPARE is:

```
<string1>  <string2>  COMPARE
```

If <string1> is greater than <string2> (in alphabetical order, <string1> comes after <string2>), COMPARE returns a 1. If <string1> is less than <string2>, COMPARE returns a -1. If the two strings are equal, COMPARE returns a Ø. Here is an example:

```
Ready PAD ASSIGN> " BAD "

Ready Ø STR1 ASSIGN> " BOLD "

Ready PAD  Ø STR1  COMPARE
[ -1 ]
```

The word COMPARE placed a -1 on the stack because the contents of PAD is "less than" the contents of STR1.

Here is a sample program that uses some of these string features. The program asks the user to type in his/her name, then a password. If the password entered is "TOP SECRET", then the program congratulates the person by name. If the password entered is not correct, "INVALID PASSWORD" is printed.

```
1 15 1 ARRAY PASSWORD
1 1ØØ 1 ARRAY NAME

: EXAMPLE
Ø PASSWORD ASSIGN> " TOP SECRET "    ( Set up correct password. )
PRINT " WHAT IS YOUR NAME? "
SPCE  Ø NAME READLN                  ( Input name. )
PRINT " WHAT IS THE PASSWORD? "
SPCE  PAD READLN                     ( Input password. )
PAD  Ø PASSWORD  COMPARE             ( Is password correct? )
IF                                   ( No: COMPARE was nonzero. )
   PRINT " INVALID PASSWORD "
ELSE                                 ( Yes: COMPARE was zero. )
   PRINT " HI, " SPCE
   Ø NAME WRITELN                    ( Write message with name. )
   PRINT " .  YOU GUESSED THE PASSWORD! "
THEN ;
```



```
Ready EXAMPLE
WHAT IS YOUR NAME? FREDDY
WHAT IS THE PASSWORD? RUBBER DUCKIE
INVALID PASSWORD

Ready EXAMPLE
WHAT IS YOUR NAME? FREDDY
WHAT IS THE PASSWORD? TOP SECRET
HI, FREDDY.  YOU GUESSED THE PASSWORD!
```

Two more string manipulation words, LEFT$ and RIGHT$, can be found in the textfile "UTILITIES". These words work much like the LEFT$ and RIGHT$ functions in Applesoft Basic. See Appendix D for more information.

Accessing Individual Characters

As mentioned earlier, strings are stored as ASCII character values, one ASCII value for each byte in the array. (If you are used to Applesoft Basic, you should know that the ASCII values used by TransFORTH, and by the Apple internally, are different from the Applesoft CHR$ values. Add 128 to the Applesoft CHR$ values to obtain the equivalent TransFORTH ASCII values.) A table of ASCII characters and their equivalent values can be found in Appendix E.

Strings can be accessed one character at a time by PEEKing or POKEing the individual ASCII values in memory. For example:

```
Ready Ø STR1 READLN
A DOG IS A MAN'S BEST FRIEND

Ready Ø STR1 PEEK .
193
```

The last line PEEKed the ASCII value of the first character of STR1. 193 is the ASCII value for the letter "A". Every character of the string can be printed as an ASCII value by using a loop:

```
Ready 29 Ø DO I STR1 PEEK . SPCE LOOP
193 16Ø 196 2Ø7 199 16Ø 2Ø1 211 16Ø 193
16Ø 2Ø5 1Ø3 2Ø6 167 211 16Ø 194 197 211
212 16Ø 198 21Ø 2Ø1 197 2Ø6 196 Ø
```

The loop PEEKed each byte of the string in turn, printing the

ASCII value. You can (if you feel so inclined) verify that the numbers printed are the ASCII values for all of the characters in the string.

You can also change individual characters. The ASCII value for the letter "H" is 200. This example POKEs the letter "H" over the "D" in "DOG":

```
Ready 200  2 STR1  POKE

Ready 0 STR1 WRITELN
A HOG IS A MAN'S BEST FRIEND
```

Writing a zero (which is the end-of-string marker) into a string ends the string at that point:

```
Ready 0  5 STR1 POKE

Ready 0 STR1 WRITELN
A HOG
```

Character Input and Output

Individual characters can be printed using the word PUTC. PUTC removes a number from the stack, interprets the number as an ASCII character, then prints this character. Here is an example:

```
Ready 193 PUTC
A
```

The word PUTC removed the 193 from the stack and printed its equivalent ASCII character, "A". The next example prints several characters:

```
Ready 200 PUTC  197 PUTC  204 PUTC  204 PUTC  207 PUTC
HELLO
```

This next example PEEKs and prints each character of STR1 in turn, including characters past the end-of-string marker itself. The Apple II 40-column display will print the ASCII characters for the zero bytes as inverse "@" signs as shown below. The zero bytes may or may not be printed by 80-column cards.

```
Ready 29 0 DO I STR1 PEEK PUTC LOOP
A HOG@IS A MAN'S BEST FRIEND@
```

Individual characters can be read from the keyboard using the

TransFORTH word GETC. GETC flashes the cursor, waits for a keypress, then places the ASCII value of the character typed on the stack.

(Note: The operation of both PUTC and GETC are actually more "general purpose" than described here. The details will be explained in Chapter Eight.)

Here is an example:

```
Ready GETC
```

(The cursor flashes. Type the letter "A".)

```
[ 193 ]
Ready
```

GETC placed the ASCII value for the letter "A", 193, on the stack. Note that GETC flashes the cursor, but does not print the character typed. PUTC can be used to print characters as they are entered. This next example reads 10 characters from the keyboard, printing each one:

```
Ready 10 0 DO GETC PUTC LOOP
```

PUTC removed the values that GETC placed on the stack. If you want to print a character <u>and</u> use it for something else, you must DUPlicate its ASCII value. The following example accepts and prints characters until the letter "A" (ASCII 193) is typed:

```
Ready BEGIN GETC DUP PUTC 193 = UNTIL
```

In some cases, you may want to monitor the keyboard for a keypress without actually stopping program execution. A couple of word definitions and an understanding of how the Apple software reads the keyboard make this simple.

The keyboard uses two address locations in the Apple, which can be called the "keyboard data" location and the "clear keyboard strobe" location. The ASCII value for the last key pressed is always stored in the keyboard data location. If a key has been pressed, the number in this location is 128 or greater. By PEEKing this location, you can retrieve this ASCII value. Accessing the clear keyboard strobe location (PEEKing or POKEing) will reset the keyboard data location, forcing its value to be less than 128. The next keypress after this access will again bring the value to 128 or greater.

Thus to read the keyboard, first access the clear keyboard strobe location to make the value in the keyboard location less than 128, then periodically PEEK the keyboard data location until the returned value is 128 or greater. This number will be the ASCII value for a key that has been pressed. The PEEKing can be interspersed with other tasks so that other things can occur while simultaneously reading the keyboard.

The address of the keyboard data location is 49152; the address of the clear keyboard strobe location is 49168. Using the following short word definitions can make keyboard access more readable. GETKEY reads the keyboard data location, and CLRKEY clears the keyboard strobe:

```
: GETKEY
49152 PEEK ;          ( Read keyboard data location )

: CLRKEY
49168 PEEK DROP ;     ( Access clear keyboard strobe location )
```

Here is a simple example that uses GETKEY and CLRKEY to "grab" a character without displaying a cursor:

```
: GRAB.CHAR
CLRKEY          ( Clear keyboard strobe before reading keyboard )
BEGIN           ( Loop to: )
   GETKEY DUP   ( Read keyboard data location and make a copy of
                  value )
   128 <        ( Continue as long as value is less than 128 )
WHILE
   DROP         ( Don't need copy of value; loop back to get
                  another )
REPEAT
CLRKEY ;        ( Clear keyboard strobe for next keypress )
```

Combining Text and Numerical Data in an Array

Since accessing an array element simply places the address of the element on the stack, TransFORTH arrays have great flexibility in data storage. As discussed above, arrays can be used for storing single-byte integers, two-byte integers, five-byte floating point numbers, and strings. By choosing the right element lengths and offsets, you can store both text and numerical data in a single array.

For example, suppose you want to write a checkbook register

program, to keep track of checks written and deposits made. For each check or deposit, you need to store 1) a "category" number describing whether the entry is a (1) check or (2) deposit, 2) the check number, 3) amount, and 4) who the check is payable to (or a description of the deposit). All of this information can be stored in a single element of an array if each element is many bytes wide:

1. The first byte of each element (byte 0) contains the category number as a (0 to 255) one-byte integer.
2. The next two bytes of each element (bytes 1 and 2) contain the check number as a (0 to 65535) two-byte integer.
3. The next five bytes (bytes 3 through 7) contain the check amount as a floating-point number.
4. The last 40 bytes of each element (byte 8 and up) store the check/deposit description as a 40-byte string.

This comes to a total of 48 bytes per element. To store up to 500 checks/deposits in this way, the following array declaration can be used:

```
Ready 48 500 1 ARRAY CHECKS
```

As described earlier, the middle of an array element can be accessed by adding an offset to the address. Following this idea, a set of simple word definitions can be written to access any part of any entry. Before calling one of these words, the array subscript (0 to 500) should be waiting on the stack.

```
: GET.CATEGORY  CHECKS PEEK ;
: PUT.CATEGORY  CHECKS POKE ;
: GET.NUMBER  CHECKS 1 + PEEKW ;
: PUT.NUMBER  CHECKS 1 + POKEW ;
: GET.AMOUNT  CHECKS 3 + PEEKN ;
: PUT.AMOUNT  CHECKS 3 + POKEN ;
: GET.DESCRIP  CHECKS 8 + WRITELN ;
: PUT.DESCRIP  CHECKS 8 + READLN ;

Ready 25 GET.CATEGORY .
1                         ( Category 1 means this is a check. )

Ready 25 GET.DESCRIP
JOE'S FEED STORE ON 2/8/83

Ready 25 GET.AMOUNT .
13.95
```

```
Ready 26 GET.CATEGORY .
2                       ( This is a deposit. )
```

```
Ready 26 GET.DESCRIP
PAYCHECK ON 2/16/83
```

To record a payment, for example:

```
Ready 250 27 PUT.AMOUNT  ( Put $250 into amount for entry 27. )
```

```
Ready 27 PUT.DESCRIP
RENT ON 2/20/83         ( You type this. )
```

etc...

This is simply one example of how arrays can be used for storing a combination of text and numerical data. Many other options and uses are possible.

Summary

In addition to being manipulated on the stack, numbers can be stored in predeclared variables, created with the word VARIABLE. Variables are TransFORTH words, like colon definitions, and can be found on the word library with LIST. Executing a variable usually places the value of the variable on the stack. However, if the store-arrow ("->") was executed since the last variable access, executing a variable instead removes a number from the stack and stores it into the variable.

Numbers can be stored and recalled from any location in the Apple memory. PEEK, PEEKW, and PEEKN remove an address from the stack, read the value at the location(s), and place the value on the stack. POKE, POKEW, and POKEN remove both value and address from the stack, storing the value at the address. PEEK and POKE access single bytes, PEEKW and POKEW access two bytes, and PEEKN and POKEN access 5 bytes.

Arrays are used for storing many values. Arrays can be declared with any number of dimensions, elements, and bytes per element, limited only by available memory. Memory arrays can also be created, which place the array data anywhere in Apple memory. The word ERASE can be used to clear every element of an array to zero.

Executing an array word removes one or more subscript numbers

from the stack (depending on how many dimensions the array has) and returns the address of that element. The PEEK and POKE words can then be used to store or retrieve a number from that location. Array accesses are not checked for subscripts that are out of bounds.

Arrays are also used for storing string data in TransFORTH. A string is stored as a number of single-byte ASCII values followed by a zero end-of-string marker. A string array must be at least as long as any text that might be stored in it. The TransFORTH word ASSIGN> is used to store text into memory starting at a given address. READLN reads a line of text from the keyboard into memory, and WRITELN prints a line of text from memory.

GETNUM can be used to convert a number stored as text in a string to an actual number. VALID is used to determine whether or not the string-to-number conversion was successful.

PAD returns the address of the 144-byte system string space.
LENGTH determines the length of (number of bytes in) a string.
MOVELN copies a string from one location to another.
CONCAT concatenates two strings together.
COMPARE compares two strings by alphabetical order.

Individual characters can be PEEKed or POKEd into strings as ASCII values.

PUTC removes a number from the stack and prints its ASCII character. GETC reads the keyboard for a keypress, then returns the ASCII value for the key pressed.

Problems

Note: Some of the problems in this section are somewhat more difficult than the other problems in the manual. However, they provide good examples of common programming applications. Whether you "work on" these problems or not, we encourage you to look over the solutions carefully.

(1)
Write a line of TransFORTH code that creates a variable named ZEBRA with an initial value of -100.

(2)
Write a short word definition named INC that increments ZEBRA by 5 when executed.

(3)
What is printed when the following example is executed?

```
43 VARIABLE X

: QUIZ
X . SPCE SPCE
X 100 * X + DUP . -> X CR
X 100 / . ;
```

(4)
Write a routine called MIN5 that removes 5 numbers from the stack, then prints the smallest value of the 5. (See MAX5 in the text above.)

(5)
In the following array definition, how many dimensions does the array THING have? How many elements in each dimension? How many bytes per element? How many elements can the entire array store? (Don't forget zero as a valid subscript.) Is PEEK, PEEKW, or PEEKN the appropriate word for reading numbers from this array?

```
Ready 2 12 4 6 3 ARRAY THING
```

(6)
Write a word definition called PRINT.THING which prints the value of every element in the ARRAY THING using nested loops. (Don't worry about formatting; just print the values.)

(7)
Write a routine that reads a line of text from the keyboard, attempts to convert the text into a numeric value, then prints the value. Write the program so that it will duplicate the example below:

```
Ready NUMBER
NUMBER, PLEASE? 22.7
YOUR NUMBER IS 22.7

Ready NUMBER
NUMBER, PLEASE? MICHIGAN
'MICHIGAN' IS NOT A VALID
TRANSFORTH NUMBER
```


(8)
Write an "adding machine" program that keeps and displays a running total, and allows you to add values to this total. (Checking for invalid numbers is optional.) An entry of no number (only return) ends the program:

```
Ready ADDER
TOTAL: 0
ADD? 4
TOTAL: 4
ADD? 19
TOTAL: 23
ADD? 101.6
TOTAL: 124.6
ADD?                 (Press RETURN.)

Ready
```

(9)
Create a one-dimensional 8 element floating-point array named NUMS, then write a routine called LOADNUMS for filling NUMS with values from the keyboard. Check for invalid numbers.

```
Ready LOADNUMS
NUMBER 0? 3.3
NUMBER 1? -10
NUMBER 2? FIVE
INVALID ENTRY
NUMBER 2? 5
NUMBER 3? 53
NUMBER 4? 1.6E26
NUMBER 5? -4321
NUMBER 6? 99.999
NUMBER 7? 8.5
```

(10)
Write a word which finds the largest value stored in NUMS, then prints the subscript number for that element.

(11)
Write a word which searches for the first occurrence of a given value in NUMS, then prints the subscript number for that element. Have the word read the desired value from the stack.

Solutions to Problems

(1)
```
Ready -100 VARIABLE ZEBRA
```

(2)
```
: INC
ZEBRA 5 + -> ZEBRA ;
```

(3)
```
Ready QUIZ
43  4343
43.43
```

(4)
```
VARIABLE X

: MIN5
-> X
4 0 DO
   DUP X <
   IF -> X
   ELSE DROP
   THEN
LOOP
X . ;
```

(5)
THING is a three-dimensional 12 by 4 by 6 array. Including the zero index, it actually stores 13 times 5 times 7 equals 455 elements. Each element is two bytes long, and PEEKW is the appropriate word for reading numbers from THING.

(6)
```
: PRINT.THING
13 0 DO
   5 0 DO
      7 0 DO
         K J I THING PEEK . SPCE
      LOOP
   LOOP
LOOP ;
```


(7)
```
: NUMBER
PRINT " NUMBER PLEASE?  "
PAD READLN PAD GETNUM
VALID IF
   PRINT " YOUR NUMBER IS  " .
ELSE
   DROP
   PRINT " ' " PAD WRITELN
   PRINT " ' IS NOT A VALID " CR
   PRINT " TRANSFORTH NUMBER "
THEN ;
```

(8)
```
VARIABLE TOTAL

: ADDER
0 -> TOTAL                            ( Start with zero total. )
BEGIN                                 ( Loop for repeated additions: )
   PRINT " TOTAL:  " TOTAL . CR          ( Print total. )
   PRINT " ADD?  "
   PAD READLN                    ( Read value from keyboard as text. )
   PAD LENGTH                    ( While length of line not zero )
WHILE                            ( Something more than RETURN typed, )
   PAD GETNUM                    ( Convert to a number )
   TOTAL + -> TOTAL              ( and add into total. )
REPEAT ;                         ( Loop back for more. )
```

(9)
With this particular solution, two colon definitions are used. GRABNUM reads one line from the keyboard and checks for a valid number. LOADNUMS calls GRABNUM 8 times to get the numbers, and stores them into the array NUMS.

```
5 8 1 ARRAY NUMS

: GRABNUM
BEGIN
   PRINT " NUMBER  " I .     ( Print subscript )
   PRINT " ?  "
   PAD READLN PAD GETNUM     ( Get the number )
VALID NOT WHILE              ( If the GETNUM failed: )
   DROP                      ( Forget the (wrong) number returned )
   PRINT " INVALID ENTRY " CR      ( Print the error )
REPEAT ;                           ( and try again )
```

```
: LOADNUMS
8 0 DO                        ( Loop for 8 elements: )
   GRABNUM                    ( Get the value )
   I NUMS POKEN               ( and store it into NUMS. )
LOOP ;
```

(10)
The routine must keep track of both the maximum value and the element number of this value:

```
VARIABLE MAX
VARIABLE INDEX

: MAXNUM
0 -> INDEX                       ( Start by assuming first element )
0 NUMS PEEKN -> MAX              ( is the maximum. )
8 1 DO                           ( Loop to read next 7 elements: )
   I NUMS PEEKN MAX >            ( If current element is )
   IF                            ( greater than maximum: )
      I -> INDEX                 ( Save both element number )
      I NUMS PEEKN -> MAX        ( and new maximum )
   THEN
LOOP
PRINT " GREATEST VALUE IS  " MAX . CR        ( Print maximum )
PRINT " STORED AT ELEMENT  " INDEX . ;       ( and element number )
```

(11)
This problem is one example of a common need: searching for an item in an array. In a language like Applesoft Basic, a FOR-NEXT loop would be used to scan through the array, with a jump out of the loop when the proper element was found. This technique does not work in TransFORTH, since a DO - LOOP cannot be "jumped" out of at will.

When searching, there are two conditions that can end the loop: finding the desired element, or reaching the end of the array. The routine below uses a BEGIN - UNTIL loop that exits when either of these conditions are met. Then the index number is tested. If the index is past the last element, then the entire array was read and the desired element was never found.

```
VARIABLE KEY
VARIABLE INDEX
```



```
: SEARCHNUM
-> KEY                     ( Read the desired value from the stack. )
-1 -> INDEX                ( The index starts before element 0. )
BEGIN
   INDEX 1 + -> INDEX          ( Increment the index )
   INDEX NUMS PEEKN KEY =      ( Does the next element = the key? )
   INDEX 7 > OR                ( Or is index past end of array? )
UNTIL                          ( End the loop for either one. )
KEY .                          ( Print the key )
INDEX 8 = IF                       ( If index past end of array: )
   PRINT "  NOT FOUND IN NUMS "        ( then key was not found. )
ELSE                                   ( Otherwise: )
   PRINT "  AT ELEMENT  " INDEX .      ( key was found. )
THEN ;
```

CHAPTER SEVEN: MISCELLANEOUS WORDS AND FUNCTIONS

CHAPTER TABLE OF CONTENTS: Page

Screen Display Words 7-1

Number Formatting 7-2

Program Control Words 7-4

ABORT 7-4
RUN 7-4
AUTORUN 7-4

Saving the TransFORTH System 7-6

Miscellaneous Words 7-8

Notes and Sound Effects 7-8
Moving Memory 7-10
Retrieving Word Addresses 7-10
Calling Machine Language Routines 7-11
Reading the Game Paddles and Buttons 7-11
Compiling Bytes into Memory 7-12
Leaving TransFORTH (gently) 7-13

Scientific Functions 7-14

Summary 7-18

Problems 7-19

Solutions to Problems 7-21

Included in TransFORTH are a large number of assorted words for controlling number formatting, positioning characters on the screen, creating a "turnkey" system, and more. These words are broken down below into the general categories of Screen Display Words, Number Formatting Words, Program Control Words, and Miscellaneous Words. In addition, a section at the end of this chapter describes TransFORTH's scientific functions in greater detail.

Screen Display Words

There are a number of TransFORTH words that either print characters or control the format of what is printed on the screen. Some of these words have already been introduced. For a quick review:

. (a period) removes a number from the stack and prints it. One of several number formats can be used; they are discussed below.

PRINT prints the quoted text that follows, starting at the current cursor position.

CR issues a carriage return, moving the cursor to the beginning of the next line.

SPCE prints a space.

WRITELN removes an address from the stack, and prints the string at that address.

PUTC removes an ASCII value from the stack and prints the equivalent ASCII character.

Here are the new words:

HTAB removes a number from the stack, interprets it as a horizontal cursor position, and tabs to that cursor position. The cursor remains in the same vertical position. The valid range for HTAB is Ø (left margin) to the window width. (See WINDOW below.)

(Note: On the Apple //e 8Ø-column display, Ø HTAB does not always work correctly. This is due to a problem in Apple's 8Ø-column firmware.)

VTAB removes a number from the stack, interprets it as a vertical

cursor position, and tabs to that cursor position. The cursor remains in the same horizontal position. The valid range is 0 (screen top) to 23 (screen bottom).

WINDOW removes four numbers from the stack to establish a text window. The text window is a rectangular area on the screen designed to protect other parts of the screen from being overwritten. All text scrolling will occur inside the window, leaving the rest of the screen unaffected. (If WINDOW is not executed, then the text window is considered to be the entire screen.) The form for WINDOW is:

```
<left>  <width>  <top>  <bottom>  WINDOW
```

<Left>, <top> and <bottom> are actual margins for the window. <width> specifies how many characters wide the window is. The bottom margin should reference the line immediately below the window. For example, a window 10 characters wide by 5 lines high in the lower right corner of a 40-column screen would be set by:

```
Ready 30 10 19 24 WINDOW
```

(The left margin is at position 30, the window width is 10 characters, the top margin is at line 19, and the bottom margin is above line 24.)

HOME erases the screen inside the text window. (HOME actually prints a CTRL-L. The TransFORTH system then interprets this character as an erase-window command. The end effect is the same.)

INVERSE causes TransFORTH to print characters in inverse (i.e. black on white).

NORMAL switches the character display back to normal (white on black).

HEXPRT removes a number from the stack and prints it as two hexadecimal digits. Any integer between 0 and 255 can be represented as two hex digits; a number out of this range is first "folded" back (e.g. 256 becomes 00, 257 becomes 01, etc.).

Number Formatting

TransFORTH has four possible display formats for numbers, set with the words FIX, SCI, ENG, and $.

FIX displays numbers in the usual floating-point format with the decimal point fixed to the right of the "one's" place. If the number is less than 1E-2 (.01) or greater than or equal to 1E9 (1 billion), then the number is instead displayed using scientific notation. FIX is the default display mode, the one used by TransFORTH until another is selected.

SCI displays numbers using scientific notation, with one digit, followed by an optional decimal point and up to 8 more digits, then a one or two digit exponent.

ENG (engineering notation) is similar to SCI, except that the format is adjusted so that the exponent is always a multiple of 3. This allows for easy conversion to the metric prefixes used in engineering or electronics applications. Here is a table of metric prefixes:

tera-	1E12	trillions
giga-	1E9	billions
mega-	1E6	millions
kilo-	1E3	thousands
-	1	ones
milli-	1E-3	thousandths
micro-	1E-6	millionths
nano-	1E-9	billionths
pico-	1E-12	trillionths

For example, 12,345 (say a length in meters) would be displayed with ENG as 12.345E3, and is the same as 12.345 kilometers.

$ selects TransFORTH's "dollar" notation that uses a dollars-and-cents format with aligned decimal points. Every number printed uses exactly 10 characters: one to seven digits with leading spaces, the decimal point, and two more digits. When numbers are printed in columns using the dollar format, the decimal points will line up. (Numbers less than .01 or greater than 9999999.99 will not fit in this format and are displayed in scientific notation.)

Program Control Words

ABORT

Executing the word ABORT restarts the TransFORTH system, closing any open files or I/O, clearing the stack, and resetting most of the conditions currently set. If called from a running program, ABORT stops the program immediately, returning to immediate mode. (For the only exception, see AUTORUN below.) ABORT can often be used as a "fast" way out of a program.

RUN

The TransFORTH word RUN automatically executes the top word on the dictionary. This can be a great convenience when loading and running programs from disk. By using RUN, you don't have to check what the top word on the dictionary is after compiling a file in order to run it. In addition, if the top word has a name something like:

SOCIO.ECONOMIC.TREND.PATTERN.FORECASTER.AND.BLACKJACK.PROGRAM,

using RUN can save a bit of typing, too....

(For users interested in convoluted programming practices, RUN can accomplish something that no other TransFORTH word can: call a word which hasn't been defined yet. RUN executes whatever word is on the top of the word library at runtime, even when called from a lower library word. Unless used carefully, this technique can cause havoc with words calling other words, only to wind back around on themselves. However, there is a program on the TransFORTH disk which uses RUN to call the top library word, and will be discussed later in the chapter.)

AUTORUN

The word AUTORUN goes a step beyond RUN. AUTORUN removes a number from the stack. If this number is nonzero, then TransFORTH will automatically execute the top word on the word library every time control is returned to the TransFORTH system level (i.e. whenever you expect to see a "Ready" prompt). DOS

errors, TransFORTH or machine language errors, executing the word ABORT, or pressing the Reset key with the AUTORUN option on all will cause the top library word to be executed. Here is an example to give you a feel for the way AUTORUN works:

```
Ready : TEST PRINT " AUTORUN IS ON!!! " ;
```

This word is added to the top of the word library so that AUTORUN will have a very visible effect.

```
Ready 1 AUTORUN
AUTORUN IS ON!!!     ( The word is automatically executed. )

Ready 3 5
AUTORUN IS ON!!!
[ 3 ]
[ 5 ]

Ready SWAP
AUTORUN IS ON!!!
[ 5 ]
[ 3 ]

Ready ABORT
```

(The screen clears.)

```
AUTORUN IS ON!!!
Ready
```

Fortunately, the AUTORUN option can be turned off by typing:

```
Ready Ø AUTORUN

Ready
```

If the top dictionary word runs a "closed" program which never exits to the system level, the AUTORUN option effectively makes the TransFORTH language itself inaccessible. Any errors or ABORTs simply restart the program. If you don't mind rebooting, enter the following lines:

```
1 VARIABLE N

: COUNT
BEGIN                ( Endless loop: )
   N . CR            ( Print N, then )
   N 1 + -> N        ( Add 1 to N. )
Ø UNTIL ;

Ready 1 AUTORUN
1
2
3
4
5
.
.
.
```

COUNT begins counting, and with no way to turn the AUTORUN option off, it can't be stopped. Reboot from scratch....

Saving the TransFORTH System

The TransFORTH language is stored on the system disk as an executable binary file with the name "OBJ.FORTH". As mentioned in Chapter One, when the disk is booted, this file is automatically loaded and run.

The TransFORTH word SAVEPRG is used to create TransFORTH binary files similar to OBJ.FORTH. SAVEPRG saves the complete current TransFORTH system, including any new words added to the word library, as a binary file. Once created, this file can be BRUN at any time, bringing the modified TransFORTH system back into memory.

SAVEPRG is a powerful tool. You can save "customized" systems, with your favorite special-purpose words already in the word library when the system is booted. You can also save finished applications programs, in such a way that the program automatically starts up when booted. This is ideal when the obvious presence of a "language" is neither needed nor desirable. In addition, the contents of variables and arrays remain intact when a system is saved with SAVEPRG.

To use SAVEPRG, first compile the words to produce the "finished" system you want to save, then type SAVEPRG:


```
Ready SAVEPRG

Enter Program Name:
```

This prompt asks for the filename you want the new system saved as. The TransFORTH system disk automatically BRUNs the file "OBJ.FORTH", so if you want this new system to boot automatically, you should name <u>your</u> file "OBJ.FORTH" too. Your file will then overwrite the supplied TransFORTH system on disk. (Make sure you're using a copy of the disk and not the original!) You are then prompted:

```
Autorun (Y/N) :
```

This prompt asks whether or not you want the saved system to boot with the AUTORUN option on. If you answer Yes to this question, then the new system will automatically run the top word on the word library, starting a program in motion. If desired, your program can later turn the AUTORUN option back off, returning access of the TransFORTH language to the user. If you answer the AUTORUN question with No, the new system will display the "Ready" prompt on boot-up, with immediate access to the language.

After answering this question, the disk drive whirs for a bit, saving this new system to disk.

The TransFORTH system as supplied includes an additional word on the top of the word library which asks the demonstration prompt on boot-up. The source text for this extra word can be found in the disk file "QUERY". The system was saved with the AUTORUN option on so that the demonstration prompt would come up automatically. When you answer No to the demo question, the word turns AUTORUN off (freeing the language), then FORGETs itself! This leaves the system in its "usual" state.

The TransFORTH system can be saved to disk without the demo prompt simply by using SAVEPRG with no additional words on the word library. (This should be done to a <u>copy</u> of your disk, in case lightning decides to strike while the system is being written to disk.) Boot the disk, answer No to the demo question, then type:

```
Ready SAVEPRG

Enter Program Name :OBJ.FORTH

Autorun (Y/N) :N
```

After the disk stops whirring, reboot the TransFORTH disk. When the system boots, the demo prompt will be gone.

You can also put the demo prompt back into the system. Type:

```
Ready DISK> " QUERY " INPUT
```

This adds the word that asks the demo question to the top of the word library. Now type:

```
Ready SAVEPRG

Enter Program Name :OBJ.FORTH

Autorun (Y/N) :Y
```

The system will be saved with the demo prompt back in.

(Note: The TransFORTH demo uses high-resolution graphics. When compiled, the file QUERY turns off 80-column card recognition so that the saved TransFORTH system will not turn the 80-column card on unless you answer "No" to the demonstration prompt. However, after following the above example, if you're using an 80-column card, TransFORTH no longer "knows" the card is there. If you want to continue with the system without rebooting, you should press Reset to turn the 80-column card completely off. See also the discussion on 80-column cards in Appendix D.)

Miscellaneous Words

Notes and Sound Effects

The TransFORTH word NOTE plays a note of a given pitch and duration through the Apple speaker. NOTE removes two numbers from the stack. The form is:

```
<pitch> <duration> NOTE
```

Larger <pitch> values produce notes lower in pitch. Larger <duration> values produce notes longer in duration. Both pitch and duration values should be integers from 1 to 255. For example, the following line will sound two notes, the second one an octave higher than the first:


```
Ready 150 100 NOTE  75 100 NOTE
```

The word NOTE will play notes, but not rests. The following word, REST, removes a duration value from the stack and simply waits that amount of time. The duration is approximately equivalent to a NOTE duration:

```
: REST
1.8 * 0 DO LOOP ;
```

Notes and rests can be combined to play short tunes, as in this word definition:

```
: SHAVE&HAIRCUT
104 120 NOTE
139 40 NOTE
147 40 NOTE
139 40 NOTE
131 120 NOTE
139 120 NOTE
120 REST
110 60 NOTE
60 REST
104 60 NOTE ;
```

Song generating programs can be written using TransFORTH. Rather than using repeated NOTE commands, reading pitch and duration values from an array using a loop would be more efficient for longer songs. Some sound effects are also possible. While complicated sound effects are beyond the scope of this book, here are a couple of program ideas to use as a starting point:

```
: ZIP
0 75 DO
   I 1 NOTE
-1 +LOOP ;

: OVERLAP
256 1 DO
   I 2 NOTE
   256 I - 3 NOTE
LOOP ;
```

Moving Memory

MOVMEM simply moves a block of memory from one location to another. MOVMEM removes three numbers from the stack. The form for MOVMEM is:

```
<source>  <destination>  <# of bytes>  MOVMEM
```

The <source> number is the starting address of the data to be moved. The <destination> is the address of where the block is to be moved to. <# of bytes> specifies how many bytes are to be moved. For example, to move 256 bytes from address 16384 to address 16896, enter:

```
Ready 16384 16896 256 MOVMEM
```

Array or string data can be moved and rearranged with MOVMEM. However, addresses and memory lengths should be chosen carefully. MOVMEM will not prevent data from being accidentally written over important system locations. Also, be careful when copying overlapping areas of memory. (MOVMEM copies the bytes in ascending order.)

Retrieving Word Addresses

The word ' (an apostrophe, also called a "tic") places on the stack the address of the word that follows it, and prevents that word from being executed. Here is an example:

```
Ready ' BELL
[ 12222 ]
```

The tic placed the address of the word BELL on the stack, and prevented BELL from being executed. (The tic word retrieves the address at runtime, not at compile time as the other "non-RPN" words do. Tic cannot retrieve the addresses of the looping and branching words or the compiling words such as VARIABLE or ":".)

The address returned by "tic" is always greater than the hexadecimal address shown with $LIST. That is because the $LIST address indicates the beginning of the word definition, and "tic" returns the address of the executable portion of the word. See Appendix C for more information on the word library structure.

Calling Machine Language Routines

Machine language programs in memory can be called directly from TransFORTH with the word CALL. CALL removes a number from the stack, interprets it as a memory address, then calls the machine language routine at that address. (The routine should end with an RTS (ReTurn from Subroutine) instruction to return to TransFORTH properly.) Machine language programs can be loaded from disk into any free area of memory, then called from TransFORTH. Accessing the disk through Apple DOS is discussed in the next chapter.

Before a machine language CALL is made, values can be placed in the Apple processor's A, X, Y, and P registers using the TransFORTH variables AREG, XREG, YREG, and PREG. Before making the machine language CALL, simply place the desired values into AREG, XREG, YREG, and PREG as you would any other variable. When CALL is executed, it loads the processor registers with the values of these variables before doing the call. After the routine has executed, the values of the registers are loaded back into the variables and can be read from TransFORTH, just as any other variable. Read on for an example of CALL....

Reading the Game Paddles and Buttons

Reading the values of the Apple game paddles provides an excellent example of using CALL. The Apple System monitor contains a routine at location 64286 for reading the game paddles. It expects to see the number of the game paddle (0 to 3) in the processor's X register. It returns a number from 0 to 255 (based on the position of the paddle) in the Y register. The following word reads the value of the game paddle by storing the top stack value into XREG, calling the paddle routine, then reading the value of YREG:

```
: READ.PADDLE
-> XREG
64286 CALL
YREG ;
```

(The Apple manuals warn that two consecutive readings of a game paddle can produce incorrect results, and suggest a short wait loop between readings.)

The two directions of a joystick are read by the Apple as two

paddle values. The first joystick reads as paddles Ø and 1, and the second joystick reads as paddles 2 and 3.

While we're on the subject of game paddles: To read the pushbuttons with the paddles, all that is needed is a PEEK into the proper memory location. The locations to PEEK are:

Button	Location
1	49249
2	4925Ø
3	49251

(The various Apple manuals number the three pushbuttons in comflicting ways. Some manuals number the buttons 1, 2, and 3 as shown above, while others use the numbers Ø, 1, and 2.)

The value returned will be a number between Ø and 255. If the number is 128 or greater, then the button is being pushed. If the number is less than 128, the button is not being pushed. Enter the following line, and hold down button 1 on the paddle or joystick while pressing Return to execute the line:

```
Ready 49249 PEEK
[ 255 ]
```

Since the button is being pushed, the value returned is greater than 127.

On an Apple //e, the Open Apple key corresponds to Button 1, and the Closed Apple key corresponds to Button 2. You can find whether or not the Apple keys are being pressed by PEEKing their corresponding button locations.

Compiling Bytes into Memory

The word "," (comma) causes a number to be compiled as a byte directly into TransFORTH. Numbers compiled as bytes can be used in a variety of ways. Small assembly language routines can be placed into executable TransFORTH words by translating the machine language code into decimal numbers, then compiling the numbers into TransFORTH with commas. Straightforward tables of numbers can also be made, though these words cannot be "executed".

Here is an example of a word that contains a number table of the Apple's visible high-resolution colors. The numbers are stored as individual bytes following the word name in memory:


```
: COLOR.TABLE
1 , 2 , 3 , 5 , 6 , 7 , ;
```

These numbers correspond to the colors green, violet, white, red/orange, blue, and another brand of white. (Graphics colors will be discussed in greater detail in Chapter Nine.) Each number can be accessed by using the tic to retrieve the address of COLOR.TABLE, adding an offset (Ø to 5), then picking out the appropriate number with PEEK. The following word definition retrieves and prints each of the color numbers in turn:

```
: SEE.COLORS
6 Ø DO              ( Loop for 6 values: )
   ' COLOR.TABLE    ( Get the address of COLOR.TABLE )
   I + PEEK         ( Add offset and PEEK color value )
   . SPCE           ( Print the value )
LOOP ;

Ready SEE.COLORS
1 2 3 5 6 7
```

Once PEEKed and put on the stack, you can use the numbers with any appropriate TransFORTH words.

Since the comma places bytes directly into the word library, a few guidelines must be followed. A comma should only be used with a number from Ø to 255, and inside a word definition. If a number is assembled with a comma as the <u>first</u> byte of a word definition, the number must be less than 128 and not equal to 1Ø. (For the reasons why, see Appendix C for technical information on TransFORTH's word library structure.)

Leaving TransFORTH (gently)

Executing the word BYE exits the TransFORTH system and enters the Apple][system monitor, with DOS active. The Apple monitor can be useful when interfacing TransFORTH with machine language programs, etc.

The TransFORTH language begins at hex location $CØØ. To restart TransFORTH from the monitor, type "CØØG".

Scientific Functions

TransFORTH's floating-point scientific functions were briefly introduced in Chapter Two. In this section, we'll look at the appropriate ranges for these functions, show how to graph them using a program on disk, and provide word definitions for functions that are not built into TransFORTH.

TransFORTH does not flag errors for number "wrap-around"; that is, if a computation produces a number larger (or smaller) than TransFORTH can handle, an incorrect answer will be returned, but an error message will not be printed. For example, squaring the number 5E2Ø should return 25E4Ø, or 2.5E41, but TransFORTH can't store a number this large:

```
Ready 5E2Ø DUP * .
2.159Ø4213E-36
```

This shouldn't cause any problems, since most practical computations never approach TransFORTH's number limit.

The four trigonometric functions, SIN, COS, TAN, ATN, express angles in radians rather than degrees. There are 57.29577951 degrees in one radian. The following word definitions can be used to convert between degrees and radians:

```
: DEGRAD              ( Degrees to radians )
57.29577951 / ;

: RADDEG              ( Radians to degrees )
57.29577951 * ;
```

For example:

```
Ready 9Ø DEGRAD SIN .
1                     ( The sine of 9Ø degrees is 1. )

Ready 45 DEGRAD COS .
Ø.7Ø71Ø6781           ( The cosine of 45 degrees is Ø.7Ø7... )

Ready 1 ATN
[ Ø.785398163 ]

Ready RADDEG .
45                    ( The arctangent of 1 is 45 degrees. )
```


Another approach is to simply define new trig functions that use degrees directly:

```
: DEGSIN
DEGRAD SIN ;    ( Convert to radians, then find sine. )

Ready 9Ø DEGSIN .
1
```

Both square root and natural logarithm are mathematically undefined for negative values. The TransFORTH SQRT function produces erratic values for negative numbers; LOG returns a mirror image of its positive side.

EXP is defined only for values between -88 and 88, since this function increases rapidly to very large numbers. Beyond these limits, incorrect values are returned, or an X/Ø (division by zero) error may occur.

The RND function works very much like Applesoft's random number function. RND removes one number from the stack and returns a random number Ø <= n < 1. If the number on the stack is greater than Ø, RND returns a random number. If the given number is less than Ø, RND begins a new "pseudo-random sequence". Every subsequent random number generated is based on this random sequence, until you change it with another negative number. If you use the same negative number, the same sequence of numbers will be generated each time. If the number on the stack is Ø, then RND returns the most recent random number again.

This pseudo-random sequence method gives you random, but repeatable, numbers. If you need random numbers that don't repeat, then PEEKW the two-byte number from location 78, NEGATE it, and use this negative number with RND to start a new random sequence. The Apple creates a new random number in locations 78 and 79 every time it waits for a keypress.

A program on the TransFORTH disk is designed to graph these functions (or any function which uses one stack value and returns one stack value). The program is stored in the textfile "FUNCTION". To compile it into the word library, type:

```
Ready DISK> " FUNCTION " INPUT
```

To graph a function, you must first compile it onto the top of the word library. FUNCTION uses the RUN command (described earlier) to "call" the top library word and return a function

value. If you want to graph a built-in TransFORTH function, add a new word to the top of the library which calls it. For example, to graph the SIN function, enter this word:

```
Ready : SIN1  SIN ;
```

Now type "FUNCTION", and the FUNCTION program will ask you for the range to graph:

```
Ready FUNCTION
LOW X? -2
HIGH X? 2
LOW Y? -5
HIGH Y? 5
```

The program clears the screen, draws the X and Y axes, then plots the function. To return to the text screen, press any key after the graph is done. You can run FUNCTION again, entering new values for the graph range. If either the X or Y axis falls off screen, it is not drawn:

```
Ready FUNCTION
LOW X? -15
HIGH X? 15
LOW Y? .1
HIGH Y? 1.1
```

To graph a new function, simply add a new word to the top of the library to call that function, then run FUNCTION again.

A number of functions, while not built into TransFORTH, can be easily added with colon definitions:

```
: LOG10     ( Logarithm base 10 )
LOG 10 LOG / ;

: LOGX      ( Logarithm base X: )
SWAP LOG SWAP LOG / ;     ( form is <number> <base> LOGX )

: SEC       ( Secant )
COS 1 SWAP / ;

: CSC       ( Cosecant )
SIN 1 SWAP / ;

: COT       ( Cotangent )
TAN 1 SWAP / ;
```



```
: ARCSIN    ( Inverse sine )
DUP DUP * NEGATE 1 +
SQRT / ATN ;

: ARCCOS    ( Inverse cosine )
ARCSINE NEGATE
1.5708 + ;

: ARCSEC    ( Inverse secant )
DUP DUP * 1 - ATN
SWAP SIGN 1 - 1.5708 *
+ ;

: ARCCSC    ( Inverse cosecant )
DUP DUP * 1 - SQRT 1 SWAP / ATN
SWAP SIGN 1 - 1.5708 *
+ ;

: ARCCOT    ( Inverse cotangent )
ATN NEGATE 1.5708 + ;

: SINH      ( Hyperbolic sine )
DUP EXP SWAP NEGATE EXP
- 2 / ;

: COSH      ( Hyperbolic cosine )
DUP EXP SWAP NEGATE EXP
+ 2 / ;

: TANH      ( Hyperbolic tangent )
DUP DUP EXP SWAP NEGATE EXP
DUP PUSH + PULL NEGATE
SWAP / 2 * 1 + ;

: SECH      ( Hyperbolic secant )
DUP EXP SWAP NEGATE EXP
+ 2 SWAP / ;

: CSCH      ( Hyperbolic cosecant )
DUP EXP SWAP NEGATE EXP
- 2 SWAP / ;

: COTH      ( Hyperbolic cotangent )
DUP DUP EXP SWAP NEGATE EXP
DUP PUSH - PULL
SWAP / 2 * 1 + ;
```

```
: ARGSINH  ( Inverse hyperbolic sine )
DUP DUP * 1 + SQRT
+ LOG ;

: ARGCOSH  ( Inverse hyperbolic cosine )
DUP DUP * 1 - SQRT
+ LOG ;

: ARGTANH  ( Inverse hyperbolic tangent )
DUP 1 SWAP -
SWAP 1 + LOG 2 / ;

: ARGSECH  ( Inverse hyperbolic secant )
DUP DUP * NEGATE 1 +
SQRT 1 + LOG SWAP / ;

: ARGCSCH  ( Inverse hyperbolic cosecant )
DUP DUP DUP * 1 + SQRT
SWAP SIGN * 1 + LOG SWAP / ;

: ARGCOTH  ( Inverse hyperbolic cotangent )
DUP 1 + SWAP 1 - /
LOG 2 / ;
```

Summary

This chapter introduced a number of useful general-purpose words.

Screen Display:
HTAB tabs to a given horizontal position.
VTAB tabs to a given vertical position.
WINDOW establishes a text window on the screen.
HOME erases the screen inside the text window.
INVERSE causes characters to be printed in inverse.
NORMAL returns character printing from inverse to normal.
HEXPRT prints a given number as two hexadecimal digits.

Number Formatting:
FIX sets the usual floating-point notation with fixed decimal place.
SCI sets scientific notation.
ENG sets engineering notation.
$ sets a dollars-and-cents notation, with aligned decimal points.

Program Control Words:
ABORT stops a running program and resets the TransFORTH system, including stacks.
RUN executes the top word on the word library.
AUTORUN turns the Autorun option on or off, which automatically executes the top library word.
SAVEPRG saves the entire TransFORTH system to disk, with any modifications and additions.

Miscellaneous Words:
NOTE plays a note of a given pitch and duration.
MOVMEM moves a block of memory from one location to another.
' (tic) retrieves the address of the following TransFORTH word.
CALL calls a machine language routine.
AREG, XREG, YREG, and PREG are variables that are loaded into the Apple's processor registers when CALL is executed.
, (comma) compiles a byte directly into a TransFORTH colon definition.
BYE exits TransFORTH and enters the Apple system monitor.

Problems

(1)
The word HOME erases the screen only inside the current text window. Consider the following word definition:

```
: TEST
HOME
23 VTAB 10 HTAB PRINT " HI THERE "
0 40 12 20 WINDOW
HOME ;
```

Will the last HOME command erase the "HI THERE" from the screen?

The next three problems show numbers printed in each of the 4 TransFORTH display formats: FIX, SCI, ENG, and $. Match up the printed number with the format used to print it.

(2)
a) 1.2345678E4
b)   12345.67
c) 12345.678
d) 12.345678E3

(3)

```
a)        25.ØØ
b) 25EØ
c) 2.5E1
d) 25
```

(4)

```
a) Ø.33
b)        Ø.33
c) 3.3E-1
d) 33ØE-3
```

(5)
After entering the following lines:

```
Ready : LOOPER  BEGIN Ø UNTIL ;

Ready 1 AUTORUN
```

How can you bring the "Ready" prompt back?

(6)
For the following two NOTE commands, which one plays the higher pitched note (or are they the same pitch)? Are the durations equal, or does one sound longer than another?

```
Ready 85 85 NOTE

Ready 85 17Ø NOTE
```

(7)
Consider the following lines, remembering that MOVMEM moves a given number of bytes, and strings are stored one character per byte.

```
Ready 1 5Ø 1 ARRAY STR

Ready PAD ASSIGN> " SOUTH OF THE BORDER "

Ready Ø STR ASSIGN> " NORTH POLE "

Ready PAD  Ø STR  5  MOVMEM

Ready Ø STR WRITELN
```

What is printed when the last line is executed?

Solutions to Problems

(1)
The "HI THERE" is not erased. The text window is above that line.

(2)
a) SCI
b) $
c) FIX
d) ENG

(3)
a) $
b) ENG
c) SCI
d) FIX

(4)
a) FIX
b) $
c) SCI
d) ENG

(5)
Reboot.

(6)
Both notes are of equal pitch. The duration of the second note is twice as long as the first.

(7)
"SOUTH POLE". The 5 characters "SOUTH" were copied from PAD to STR.

CHAPTER EIGHT: INPUT AND OUTPUT

CHAPTER TABLE OF CONTENTS: Page

The I/O System 8-1

Handling I/O 8-4
Related Input/Output Words 8-6
Restrictions on DISK I/O 8-10
Number to String Conversion 8-10

Apple DOS Disk Access 8-11

Using Textfiles for Data Storage 8-12

Saving the Contents of Arrays 8-13

Overlays 8-14

Binary File Overlays 8-16

Summary 8-17

Problems 8-18

Solutions to Problems 8-19

One of TransFORTH's greatest strengths is its Input/Output operating system. Using TransFORTH I/O commands, you can print to or read from Apple peripheral cards, areas of memory, custom I/O routines, or disk files. The operating system features are used whenever any character is printed (output) or read (input), and they are compatible with the Apple Disk Operating System.

This chapter will cover the Input/Output operating system, access to Apple DOS, and overlays, which provide a method for running programs that are larger than will fit in the Apple memory.

The I/O System

Normally, TransFORTH outputs characters to the Apple screen video (or an 80-column card in slot 3 or the Apple //e auxiliary slot) and reads input characters from the keyboard. By executing one of the eight I/O commands, either input or output can be changed.

OUTPUT commands tell TransFORTH where any characters it needs to print should be sent. TransFORTH output is used with the words PRINT, WRITELN, . (period), CR, SPCE, BELL, HEXPRT, HOME, and PUTC, and TransFORTH system messages such as the "Ready" prompt and stack display. If the OUTPUT is changed, then any characters printed will go to the output specified.

INPUT commands specify where TransFORTH should read characters from. TransFORTH input is used with the words READLN and GETC, and when the system reads lines to be compiled and executed. If the INPUT is changed, then any characters read will come from the new input rather than from the keyboard.

The various commands are:

```
<address> DEVICE INPUT
<address> DEVICE OUTPUT
<address> MEMORY INPUT
<address> MEMORY OUTPUT
DISK> " <filename> " INPUT
DISK> " <filename> " OUTPUT
<address> DISK INPUT
<address> DISK OUTPUT
```

We'll first describe each of the commands, then later show some examples for using then with other I/O words:

<address> DEVICE INPUT: This is used for reading characters from a peripheral card in one of the Apple slots. The <address> on the stack should be the address of the desired slot. To find the address, use either this formula:

<address> = <slot> * 256 + 49152

or this word definition:

```
: SLOT
256 * 49152 + ;
```

For example, to tell TransFORTH to begin reading characters from an RS-232 card in slot 2 (rather than from the keyboard), you can enter either of the following lines:

```
Ready 49664 DEVICE INPUT

Ready 2 SLOT DEVICE INPUT
```

<address> DEVICE OUTPUT: This routes subsequent output characters to an Apple peripheral card. <address> is the same as for DEVICE INPUT. DEVICE OUTPUT can be used for sending text to printers, modems, etc. For example, either of the following lines will cause characters to be sent to a printer card in slot 1:

```
Ready 49408 DEVICE OUTPUT

Ready 1 SLOT DEVICE OUTPUT
```

(Note for machine language programmers: DEVICE INPUT and OUTPUT can also be used for calling any custom I/O routines stored in free areas of memory. Simply substitute the decimal address of the routine for the <address> above. The characters are passed through the 6502 accumulator.)

<address> MEMORY INPUT: This command causes characters to be read directly from memory. This memory can be a TransFORTH string, text in free memory, or the text editor program buffer. <address> is the starting address of the area of memory to be used. For example, the following line will tell TransFORTH to read characters from memory starting at address 32768:

```
Ready 32768 MEMORY INPUT
```

Recall that PROGRAM MEMORY INPUT is the command for reading and compiling text stored in editor memory. Here is how it works:

The word PROGRAM places the address of the editor program buffer on the stack. MEMORY INPUT then specifies this area of memory as a source of character input. TransFORTH continues to read lines to be compiled, but now draws them from the editor memory. The text from the editor is read just as if it were being entered from the keyboard. When all of the editor lines have been read, an End-Of-File condition occurs, and input returns to normal.

MEMORY INPUT can also be used for compiling code directly from strings, creating some unusual programming possibilities. For example, you can store the text of a TransFORTH command in a string from a running program, then use MEMORY INPUT to actually compile and execute that string later.

<address> MEMORY OUTPUT: This causes printed characters to be stored directly into memory. As with MEMORY INPUT, the <address> can be any legal Apple address, in a string, free memory, or even the editor buffer.

DISK> " <filename> " INPUT: This command tells TransFORTH to open the specified textfile on disk and read subsequent characters from the textfile. Note that this is the command used for reading and compiling programs on disk. The following line will compile the disk file BASCON:

```
Ready DISK> " BASCON " INPUT
```

DISK> " <filename> " OUTPUT: This opens a textfile and causes printed characters to be written into the textfile.

<address> DISK INPUT: This command also specifies a textfile on disk as a source of characters. In this case, <address> should be the address of a string that contains the name of the file. Note that when using DISK> " <filename> " INPUT, the filename is compiled with the command, and cannot be changed from a running program. With <address> DISK INPUT, the string containing the filename can be modified when the program is running. The following example also reads the file BASCON from disk (The array STR must be defined.):

```
Ready 0 STR READLN
BASCON

Ready 0 STR DISK INPUT
```

<address> DISK OUTPUT: This is similar to DISK> " <filename> " OUTPUT in that it opens a textfile and causes printed characters to be written into the file. It uses the same form as <address>

DISK INPUT, where the filename is stored in a string and <address> is the address of that string.

For all of the disk I/O commands, any slot, drive, or volume numbers needed can be included right with the filename in the string or in quotes. Here are a couple of examples:

```
Ready DISK> " BASCON,D2 " INPUT
```

This reads the file BASCON from drive 2.

```
Ready Ø STR ASSIGN> " TEST,S5,D1 "

Ready Ø STR DISK OUTPUT
```

This example will print characters to the textfile TEST at slot 5, drive 1.

The TransFORTH system includes a special textfile speedreader. This speedreader is accessed whenever textfiles are used as a source of input, both from TransFORTH and the TransFORTH text editor, allowing programs and data to be loaded much more quickly than is possible with Apple DOS. You'll notice a great difference in speed between reading from and writing to the disk.

The speedreader also allows for file-to-file copying. Usually, Apple DOS will not allow you to read from one textfile while writing to another. The TransFORTH speedreader bypasses DOS so that direct file read and write can take place simultaneously. File copying is discussed in greater detail below.

Executing the TransFORTH word CLOSE will immediately return both input and output to normal. In addition, if an End-Of-File condition occurs while a line of text is being read, input and output will return to normal. (See also "EOF" and "MOVFILE" below.)

Handling I/O

It is important to remember that the above TransFORTH I/O routing commands do not cause any characters to be printed or read; they only specify source and destination for characters that will be printed or read by the system or a running program. For example, when you type PROGRAM MEMORY INPUT, this tells TransFORTH to use the editor program buffer as the source for reading characters. The actual reading and compiling is then done by the TransFORTH system, in the same way it usually reads and compiles from the

keyboard.

If an INPUT command is instead followed by READLN's or GETC's, then the characters read will come from that source of input. Try the following example:

Enter the editor:

```
Ready EDIT
```

Erase any text in editor memory, then enter the following text:

```
1Ø  8 Ø DO I . SPCE LOOP CR
2Ø  PRINT " AN EDITOR EXAMPLE "
```

Exit the editor, then run this line from TransFORTH:

```
Ready PROGRAM MEMORY INPUT PAD READLN PAD WRITELN CR PAD READLN
PAD WRITELN CLOSE
```

The above example reads two text lines from the editor (since PROGRAM MEMORY INPUT sets the editor buffer as the input source), and prints them directly to the Apple screen (since the output is normal):

```
8 Ø DO I . SPCE LOOP CR
PRINT " AN EDITOR EXAMPLE "
Ready
```

If you type only PROGRAM MEMORY INPUT, then the TransFORTH system itself will read the lines, and act on them accordingly:

```
Ready PROGRAM MEMORY INPUT
Ø 1 2 3 4 5 6 7
AN EDITOR EXAMPLE

Ready
```

Because of these capabilities, a small complication can sometimes arise. Suppose you have a program (a series of word definitions) in a file on disk, and you want the program to begin running automatically when it is compiled. To do this, all you have to do is place the word "RUN" as the last line of the file. When this last line is read, the program will begin running.

Now suppose that 1) the RUN command isn't on the very last line, but that a few blank lines follow the RUN at the end of the file, and 2) when the program is run, it starts by asking for keyboard

input with READLN or GETC. The problem that arises is that the disk file is still open when the RUN command is executed. This means the READLN or GETC will read the remaining characters from the disk file, not from the keyboard!

The solution is to substitute a "CLOSE RUN" for the RUN. The word CLOSE will guarantee that input and output are returned to normal after this line is read from the file. Any remaining characters in the file will be ignored. Therefore, if you want a program to begin running automatically when it is compiled, the best method is to place the words "CLOSE RUN" as the last line of the file.

Another related aspect is TransFORTH's ability to "link source files", allowing several files to be compiled with one command. If you want to compile file A, file B, and then file C, add this command to the end of file A:

```
DISK> " B " INPUT
```

Then add the following command to the end of file B:

```
DISK> " C " INPUT
```

Then, to compile all three files, simply type from the keyboard:

```
DISK> " A " INPUT
```

File A will be compiled, then the last line of A will open and begin compiling file B. Similarly, the last line of B will open and compile file C.

Related Input/Output Words

EOF: Suppose you're writing a program that, among other things, opens and reads characters from a disk file. If you try to read more characters than the file contains, the file will close and the program will start reading characters from the keyboard. What is needed is a way to determine when the End-Of-File is reached before too many characters are read.

The TransFORTH word EOF can be used to determine an End-Of-File condition from any input or output, including a disk file. EOF places a number on the stack. Usually this number is zero (false). However, if an End-Of-File is reached, executing EOF will return a nonzero value (true). EOF is then reset to zero when another INPUT or OUTPUT or an ABORT is executed.

The following word definition provides a simple example for using EOF. The routine opens the disk file TURTLE as a source of characters. It then uses a BEGIN - UNTIL loop to read characters from the file (with GETC) and print them to the screen (with PUTC) <u>until</u> EOF becomes true, when the end of the textfile is reached. CLOSE then closes both input and output, returning them to normal.

```
: READFILE
DISK> " TURTLE " INPUT
BEGIN
   GETC PUTC
   EOF
UNTIL
CLOSE ;
```

When READFILE is run, the text of the file TURTLE is printed to the screen. On a 40-column screen, the last character printed will be an inverse "@". This is the ASCII character for the zero End-Of-File marker in the file.

EOFCHR: Using the TransFORTH word EOFCHR, you can select a different character or value (than zero) to be the End-Of-File marker. EOFCHR removes a number from the stack. The number, an ASCII value, is used as the new End-Of-File marker. If this value is later input or output as a character, EOF will become true.

For example, suppose you want to print the file TURTLE only to the end of the first colon definition. You know that the text of the colon definition ends with a semicolon. By setting the ASCII value for a semicolon as the End-Of-File marker, EOF will become true when the semicolon is read:

```
Ready 187 EOFCHR    ( 187 is the ASCII value for a semicolon. )
```

```
Ready READFILE
```

As promised, the file is printed only as far as the first semicolon. Entering:

```
Ready 0 EOFCHR
```

makes the End-Of-File marker a zero again.

EOFCHR can also be used when working with strings. The end of a

string is determined by either a zero or the EOFCHR value. For example:

```
Ready PAD READLN
SAN FRANCISCO

Ready PAD LENGTH .
13                    ( PAD contains 13 characters. )

Ready 195 EOFCHR    ( This sets 195, the ASCII letter "C", as the
End-Of-File character. )

Ready PAD WRITELN
SAN FRAN              ( The "C" in "FRANCISCO" now marks the end of
the string. )

Ready PAD LENGTH .
8                     ( There are 8 characters in the string before
the "C". )

Ready Ø EOFCHR
```

Executing ABORT or pressing Reset will automatically set EOFCHR back to zero.

MOVFILE: Most of what the word definition READFILE (above) accomplishes can be done with a single TransFORTH word, MOVFILE. MOVFILE simply reads characters from the current input source and prints then to the current output destination. When an End-Of-File is encountered, the reading and printing stops and both input and output are closed. The following line:

```
Ready DISK> " TURTLE " INPUT MOVFILE
```

is equivalent to READFILE. The INPUT command sets the disk file TURTLE as the source of characters. MOVFILE reads the characters from TURTLE and prints them to the screen (the current output destination). In this application, MOVFILE provides a quick and easy way to see the contents of a file without using the text editor.

MOVFILE can be used for a wide variety of data copy and transfer operations, including file-to-file copying. This example copies the entire contents of the file BASCON into a new file named TEMP. (Any previous contents of TEMP are overwritten.)

```
Ready DISK> " BASCON " INPUT  DISK> " TEMP " OUTPUT  MOVFILE
```

This line prints the text in the editor buffer to a printer connected to slot 1:

```
Ready PROGRAM MEMORY INPUT  1 SLOT DEVICE OUTPUT  MOVFILE
```

As you can see, many combinations of input and output are possible with MOVFILE.

ECHO: When both input and output are changed, subsequent characters are routed directly from the source and to the destination, often bypassing the video screen completely. Using the word ECHO, you can <u>monitor</u> the flow of text, by ECHOing it to the video screen. ECHO removes a number from the stack. Different numbers have different effects:

1 ECHO echoes input characters to the screen
2 ECHO echoes output characters to the screen
Ø ECHO turns off screen echoing

For example, you can print text to the screen as it is compiled from the editor buffer or a disk file onto the word library:

```
Ready 1 ECHO

Ready PROGRAM MEMORY INPUT
```

You can also see the characters copied from one textfile to another by echoing output characters:

```
Ready 2 ECHO  DISK> " BASCON " INPUT  DISK> " TEMP " OUTPUT
MOVFILE  Ø ECHO
```

Because ECHO must accomodate a wide variety of input and output combinations, including the oddities of Apple DOS, there are a few restrictions: 1) If you turn on output echo when the screen is already used for output, TransFORTH will send each character to the screen twice. 2) If you echo input while DISK OUTPUT is in effect, the echoed characters will be written to the textfile. 3) You should not turn output echo on or off after a textfile has been opened with DISK OUTPUT. If you do, DOS will close the

file. 4) Lastly, some peripheral and printer cards read and change the Apple's special video cursor locations, the same ones that ECHO must use to print to the screen. If this happens, the print-out and/or the video output will become garbled, and ECHO cannot be used.

Restrictions on DISK I/O

Because disk I/O requires at least partial assistance from Apple DOS, there are a few restrictions in printing characters to a disk file using TransFORTH I/O. First, output to a disk file cannot be combined with keyboard input. If you try to use the keyboard for input, then DOS will disconnect the output to the file.

Secondly, if disk files are selected for both input and output, the INPUT command must come before the output command:

```
Ready DISK> " BASCON " INPUT  DISK> " TEMP " OUTPUT  MOVFILE
```

Lastly, the words CLOSE, ECHO, and DISK INPUT and OUTPUT include calls to DOS. These calls will sometimes print an extra carriage return, which means the cursor will advance to the beginning of the next line. If the cursor was on the bottom line of the screen, the text will scroll. If you don't want this scrolling, be sure to VTAB away from the bottom line before you call any of these words.

Number to String Conversion

Remember that GETNUM is used for converting strings to numbers. MEMORY OUTPUT can be used to do the opposite: print a number into a string. To do this, set the string as a MEMORY OUTPUT, print the number to the string output with . (period), print a zero end-of-string marker, then CLOSE the output. Here is an example which prints the the square root of 2 into STR:

```
Ready Ø STR MEMORY OUTPUT 2 SQRT . Ø PUTC CLOSE

Ready Ø STR WRITELN
1.41421357       ( The number is stored as text in the string. )
```


Apple DOS Disk Access

Apple DOS commands can be directly executed from TransFORTH. Remember that in Basic, DOS commands can be called in one of two ways: directly from the keyboard, or printed from a program with a CTRL-D. TransFORTH does not allow DOS commands to be typed directly. However, you can execute DOS commands by using the same general syntax as from a running Basic program: Print a CTRL-D, then the DOS command. With TransFORTH, the form is:

```
CR 132 PUTC PRINT " <DOS command> " CR
```

The CR prints a carriage return, starting a new output line. 132 PUTC prints a CTRL-D (132 is the Apple ASCII value for a CTRL-D). The DOS command is printed next, then another CR ends the line so that the DOS command will be executed. For example, this line will catalog the disk:

```
Ready CR 132 PUTC PRINT " CATALOG " CR
```

Any Apple DOS commands can be executed gracefully from TransFORTH, except for the following commands which use or rely on Basic:

LOAD
SAVE
RUN
FP
INT
INIT
CHAIN

In addition, TransFORTH keeps MAXFILES set at 1. Since TransFORTH has its own textfile speedreader, changing MAXFILES is usually not necessary. If you need to, you can set MAXFILES to a larger number on 48K Apples if you are not using the text editor. On 64K Apples, you can increase MAXFILES up to 2 only if you are not using the graphics module GR.TEXT.64K or the auxiliary memory features. (See the next two chapters for more information.)

PR#n and IN#n can be used for accessing peripheral cards in the Apple slots. DEVICE INPUT and OUTPUT are already available for these functions, however. If you use PR#n or IN#n, TransFORTH will not "know" what its actual inputs or outputs are, and words such as ECHO may not work correctly. There is one exception: If

you want to print out a DOS catalog of the disk, you must use PR#n to activate the printer, since DOS is doing the printing, not TransFORTH.

Note: Since TransFORTH uses its own I/O routines, executing "PR#0" will not return TransFORTH to a normal state. The word TEXT will turn off any PR#n or IN#n access, while CLOSE will close any TransFORTH input and output.

Using Textfiles for Data Storage

TransFORTH supports both sequential and random access data files. Sequential files are best handled using the DISK INPUT and OUTPUT commands. The following word definitions demonstrate the use of sequential data files. The word FILL.SEQUENTIAL fills a sequential file with 500 lines (or "fields") of text. GET.SEQUENTIAL accesses a given field by reading through all of the fields preceding it. The desired field number should be waiting on the stack:

```
: FILL.SEQUENTIAL
DISK> " SEQUENTIAL " OUTPUT  ( Open file for output. )
500 0 DO                     ( Loop for 500 times: )
   PRINT " FIELD:  " I . CR  ( Write field into file. )
LOOP
CLOSE ;                      ( Close the file. )

: GET.SEQUENTIAL
DISK> " SEQUENTIAL " INPUT   ( Open file for input. )
1 + 0 DO            ( Loop for specified number of times + 1 to )
   PAD READLN       ( read fields from file. )
LOOP                ( Last field read is desired field. )
CLOSE ;             ( Close the file. )

Ready FILL.RANDOM   ( The disk whirs as the file is filled. )

Ready 35 GET.SEQUENTIAL      ( Field 35 is read into PAD. )

Ready PAD WRITELN
FIELD: 35
```

Since TransFORTH's disk commands do not allow you to specify record and length parameters, random access files are best handled through explicit DOS commands. (The format and control of random access files are more completely described in the Apple Disk Operating System manual.)

In the following example, FILL.RANDOM fills a random file with 500 records, and GET.RANDOM accesses a given record in the file (with the record number on the stack). Even though the syntax differs, notice that TransFORTH uses the same DOS commands for accessing random files that Basic does:

```
: FILL.RANDOM
CR 132 PUTC
PRINT " OPEN RANDOM,L15 " CR         ( Open the file. )
500 0 DO                             ( Loop for 500 records: )
   CR 132 PUTC
   PRINT " WRITE RANDOM,R " I . CR   ( Set record number. )
   PRINT " RECORD:  " I . CR         ( Print text into record. )
LOOP
CR 132 PUTC PRINT " CLOSE " CR ;     ( Close the file. )

: GET.RANDOM
CR 132 PUTC PRINT " OPEN RANDOM,L15 "          ( Open the file. )
CR 132 PUTC
PRINT " READ RANDOM,R " . ( Point to record, number from stack. )
CR PAD READLN                        ( Read the record into PAD. )
CR 132 PUTC PRINT " CLOSE " CR ;               ( Close the file. )

Ready FILL.RANDOM          ( The disk whirs, filling the file. )

Ready 256 GET.RANDOM       ( Record 256 is read from the file
                             into PAD. )

Ready PAD WRITELN
RECORD: 256
```

Saving the Contents of Arrays

With a basic understanding of how much memory is used by arrays, you can save the entire contents of an array to disk as a binary file. The data can later be loaded from disk back into the array. This is very handy if you want to work with large sets of data.

In TransFORTH, an array is simply a block of memory that is segmented into many elements. The array data is saved by BSAVEing the block of memory which the array uses. Two values must be found: the starting address and the length of the array. The starting address is simply the address of the first element of the array. The length can be found as follows:

```
length = total number of elements * number of bytes per element
```

When determining number of elements, remember to include the zero elements. For example, the length of the following array:

```
Ready 5 8 9 2 ARRAY FROG
```

can be found by:

```
length = (8 + 1) * (9 + 1) * 5 = 9 * 1Ø * 5 = 45Ø bytes
```

The starting address (for our example) is:

```
Ready Ø Ø ARRAY FROG .
12219
```

With this information, the array data can be saved to disk with a BSAVE:

```
Ready CR 132 PUTC PRINT " BSAVE FROGDATA,A12219,L45Ø " CR
```

If the array remains in the same place in memory, the data can be brought back at a later time with a simple BLOAD:

```
Ready CR 132 PUTC PRINT " BLOAD FROGDATA " CR
```

If the array has been recompiled into a different area of memory, the new starting address must be found:

```
Ready Ø Ø FROG .
1228Ø

Ready CR 132 PUTC PRINT " BLOAD FROGDATA,A1228Ø " CR
```

Overlays

Sometimes a program may grow so large that it cannot fit in the Apple memory. If the program can be broken into two or more smaller sections, where one only one section needs to be in memory at any given time, then overlays can be used.

The word "overlay" refers to a portion of a program that overlays, or overwrites, another portion which is not currently needed. For example, suppose you are writing a large program that (1) stores and (2) retrieves financial information from a

file. You never need to do the store and retrieve operations at the same time, and can divide the program into two parts. However, each part relies on the same low-level common variables and routines when accessing the disk.

Using overlays, the common routines stay in memory at all times. The store part of the program is loaded and used as needed. When it is time to switch to retrieve operations, the retrieve part is then loaded over the top of the store part, overlaying and overwriting it. The retrieve operations can then be done.

Implementing overlays in TransFORTH requires a good working knowledge of how TransFORTH programs are read, compiled, and run. If you understand what is happening and why, then overlays are fairly easy to create. The process can be broken into a few steps:

Compile the common variables and routines onto the word library first, then compile the first part of the program to be run.

Run this section.

When it is time to switch to the next part, a word definition in memory should be called to: 1) FORGET the words from the first part (removing it from the system), and 2) begin compiling the second part in from disk.

When the new file has finished compiling, the second part will be in memory, ready to be run. To execute it, one of two techniques can be used: 1) Include the immediate commands CLOSE RUN as the last line in the second part's source file. This will force the file closed and begin running the newly compiled code. 2) Have the TransFORTH system in AUTORUN mode before compiling the second part. This will guarantee that the new code will begin executing immediately.

Here is a short example of an overlay. Enter the editor, type in the following lines, and save the text with the filename "PART1":

```
: COMMON
PRINT " THE COMMON ROUTINE IS IN MEMORY " ;

: PART1
PRINT " THIS IS PART 1 " CR
COMMON CR
PRINT " NOW GOING TO PART 2... "
FORGET PART1
DISK> " PART2 " INPUT ;
```

Now clear the editor buffer and enter these lines, saving them to disk as "PART2":

```
: PART2
PRINT " THIS IS PART 2 " CR
COMMON CR
PRINT " ALL DONE NOW. " ;

CLOSE RUN
```

Now return to TransFORTH, compile "PART1" from disk, and run it:

```
Ready DISK> " PART1 " INPUT

Ready RUN
THIS IS PART 1
THE COMMON ROUTINE IS IN MEMORY
NOW GOING TO PART 2      ( PART1 FORGETs itself and calls PART2. )
THIS IS PART 2
THE COMMON ROUTINE IS IN MEMORY
ALL DONE NOW.
```

Note: If a word (or group of words) on the word library is forgotten from a program, it can still be called while the program is running. However, the next line of code compiled by the system will overwrite it.

To complicate the issue further, an alternative to putting the FORGET line in the first part is to place it as the first line of the second file, outside of a word definition. When this second file is compiled, the first task executed will be to FORGET the first part.

Binary File Overlays

There is another, completely different, way to implement overlays, with its own advantages and disadvantages. For this method, compile the low-level words and one part of the program onto the word library, then call SAVEPRG to save the entire system as an executable binary program. The AUTORUN option should be set so that this part of the program will begin running immediately when BRUN.

Repeat this for each "overlay" part, compiling the common code and one overlay, then saving each new system with SAVEPRG. This will produce several executable binary files, one for each part.

Then to switch from one part to another, each part can simply make a DOS call to BRUN the next part. The new part, being another TransFORTH system, will completely overwrite the first.

The advantage to this approach is that your finished program disk has more privacy and security, since you don't have to include source files on the disk.

The disadvantages are 1) binary files sometimes take up more space on disk (especially if large arrays are declared, which also increases loading time), and 2) passing information from one part to another is much more difficult. Any data stored in variables or arrays will be lost as the new system is brought into memory. The best solution is to copy all important data into some free area of memory immediately before BRUNning the second part. Then have the second part read the data back into its variables before continuing.

Summary

TransFORTH includes a versatile Input/Output operating system for routing characters in various ways. You can specify either character INPUT or OUTPUT or both. Input and output can be a peripheral card DEVICE, an area of MEMORY, or a textfile on DISK. With various combinations, the I/O system can be used to read and compile textfiles, print to a printer, write data into memory, communicate with devices through serial cards, etc. The input/output commands do not <u>cause</u> characters to be printed or read; they only specify sources and destinations for characters that are printed or read by the system or a running program.

A textfile speedreader in TransFORTH allows files to be read much faster than is normally possible with Apple DOS. The speedreader also allows one textfile to be read while another is written, for file-to-file copying, etc.

The word CLOSE closes any open disk file and returns all TransFORTH I/O to normal. EOF becomes nonzero when an End-Of-File condition occurs. EOFCHR allows you to specify an ASCII value other than zero as the End-Of-File CHaRacter. (Both the I/O system and the string manipulation words recognize EOFCHR.) ECHO causes input or output characters to be echoed to the screen. MOVFILE simply copies characters from input to output until an End-Of-File condition occurs.

A number can be written into a string by specifying the string as

OUTPUT, then printing the number and a zero End-Of-String marker, then CLOSEing output. This feature is similar to Applesoft's STR$ function.

DOS commands can be called directly by printing a carriage return, a CTRL-D, the DOS command, and another carriage return. However, the I/O system duplicates the functions of the DOS commands PR#n, IN#n, OPEN, READ, and WRITE. When manipulating data in textfiles, sequential files are best handled with the TransFORTH I/O system, random access files with explicit DOS commands.

Array data can be saved to disk by determining the start and length of the block of array memory, then BSAVEing this block. The data can be reloaded later with BLOAD.

Overlays are used when a program is too large to fit in memory, but can be segmented into smaller parts. Textfile overlays involve running the first part of the program, FORGETting the words in memory, compiling the next part into the word library, and continuing execution. Binary file overlays are done by saving each part with SAVEPRG as an executable program, then using BRUN to switch from one part to another.

Problems

(1)
Write a TransFORTH line to print the phrase "HI, THERE" to your printer.

(2)
Print the contents of the file "BASCON" to the screen.

(3)
Print the contents of "BASCON" to the printer.

(4)
Print only the first line of the file "BASCON" to the printer.

(5)
Copy the contents of "BASCON" into a new file called "BASCON1".

(6)
Compile BASCON into memory, with the proper command set to see the characters on the screen as they are read and compiled.

(7)
The word PI places the value of pi on the stack. Write the value of pi into a string as ASCII characters, then print the string.

(8)
Delete the new file "BASCON1" from disk using a DOS command from TransFORTH.

Solutions to Problems

(1)
Assuming the printer is in slot 1:

```
Ready 49408 DEVICE OUTPUT PRINT " HI, THERE " CLOSE
```

or if the word SLOT is in the word library:

```
Ready 1 SLOT DEVICE OUTPUT PRINT " HI, THERE " CLOSE
```

(2)
```
Ready DISK> " BASCON " INPUT MOVFILE
```

(3)
```
Ready DISK> " BASCON " INPUT 49408 DEVICE OUTPUT MOVFILE
```

(4)
```
Ready DISK> " BASCON " INPUT 49408 DEVICE OUTPUT PAD READLN PAD
WRITELN CLOSE
```

(5)
```
Ready DISK> " BASCON " INPUT DISK> " BASCON1 " OUTPUT MOVFILE
```

(6)
```
Ready 1 ECHO DISK> " BASCON " INPUT
```

(7)
```
Ready 1 50 1 ARRAY STR

Ready 0 STR MEMORY OUTPUT  PI .  0 PUTC  CLOSE

Ready 0 STR WRITELN
3.14159266
```

(8)
```
Ready CR 132 PUTC PRINT " DELETE BASCON1 " CR
```

CHAPTER NINE: GRAPHICS

CHAPTER TABLE OF CONTENTS: Page

High Resolution Graphics **9-1**

In and Out of Graphics 9-1
Combined Text and Graphics 9-2
Graphics Drawing Commands 9-3
Color 9-5
ORMODE and EXMODE 9-7
Scaling Functions and Graphs 9-8
Character Sets 9-9

Turtlegraphics **9-10**

PENUP 9-11
PENDOWN 9-11
MOVE 9-12
TURNTO 9-12
TURN 9-12
MOVETO 9-13
Examples 9-14

Larger Graphics Programs **9-15**

Screen "Dumps" and Saves **9-16**

Low Resolution Graphics **9-17**

LGR 9-17
LGRF 9-18
LCOLOR 9-18
LPLOT 9-18
LHLINE 9-19
LVLINE 9-19
LSCRN 9-20
Leaving Low Resolution Graphics 9-20
An Example 9-20

Summary **9-21**

Problems **9-22**

Solutions to Problems 9-23

TransFORTH includes a variety of graphics features, including low and high resolution graphics, Turtlegraphics, and text printing on the graphics screen. Some of the commands can be found in the TransFORTH word library; others are included in source files on the TransFORTH system disk. Several graphics demonstration programs are also included on the disk.

This chapter describes in detail all of the TransFORTH graphics commands. However, it assumes that you're at least a little familiar with Apple graphics from either Applesoft or Pascal. For more detailed information, see the Apple manuals.

Note: The graphics features will not work if an 80-column card is active. You may need to remove the card while using the graphics features. (Alternatively, you can use a special routine which causes TransFORTH to ignore the 80-column card. See Appendix D for more information.)

High Resolution Graphics

In the Apple memory are two 8192-byte screen areas which can be used for storing and displaying high-resolution images. Each screen can hold up to 53,760 dots, or "pixels", arranged in a grid 280 pixels wide by 192 tall.

TransFORTH can display graphics on the second of the two graphics screens. (The TransFORTH language itself overlaps the first screen area.) The graphics words use a display area approximately 9% narrower than usual: 256 pixels horizontally rather than 280. (The missing 9% is from the right side of the screen.) This reduction in size significantly increases the speed of TransFORTH graphics.

In and Out of Graphics

There are two separate but similar ways of using high-resolution (or "hi-res") graphics. Both of these can be used for drawing points, lines, and rectangular areas on the graphics screen. The first uses only built-in TransFORTH commands, and cannot combine text with the graphics. The second makes use of a "graphics module" from the disk, and allows you to print text anywhere on the graphics screen, as well as draw points, lines, and areas.

To enter the high-resolution graphics-only mode, type the TransFORTH word GR. GR erases the second graphics screen, then

switches to display this screen on the Apple video display. If you type GR from immediate mode, the screen will go blank, since there is nothing drawn on the graphics screen yet:

```
Ready GR
```

On the now-invisible text screen, a "Ready" prompt is still waiting your command. To return to text mode, type TEXT. (You will have to type it "blind".) The text screen will reappear, showing the commands you've typed:

```
Ready GR

Ready TEXT
```

Combined Text and Graphics

For many graphics applications, switching between text and graphics modes works fine. However, in most cases, and to better show you here how the graphics routines work, it is much more convenient to be able to display text and graphics at the same time. The TransFORTH system disk contains two machine language routines designed to accomplish just that, by "drawing" the text characters onto the graphics screen. Once the appropriate routine is active, all screen printing will automatically go to the graphics screen. Both upper and lower case characters are supported.

The two modules are stored as binary files on the disk. One is designed for 48K Apples, the other for 64K systems. The methods for loading and running each are different, but similar:

The name of the graphics module used with 48K systems is "GR.TEXT.48K". To bring the module into memory, enter the following:

```
Ready CR 132 PUTC PRINT " BLOAD GR.TEXT.48K " CR
```

The disk will whir as the module is loaded. To turn the text-and-graphics on at any time, type:

```
Ready 37888 CALL
```

The name of the graphics module used with 64K (or larger) systems is "GR.TEXT.64K". To bring this module into memory, enter:

```
Ready CR 132 PUTC PRINT " BLOAD GR.TEXT.64K " CR
```


The disk will whir as the module is loaded. To turn the text-and-graphics on at any time, type:

```
Ready 47616 CALL
```

The appropriate graphics module can also be loaded and run with one command by substituting "BRUN" for the "BLOAD" in the line(s) above. Once in memory, it can later be CALLed without being reloaded every time.

Calling the module puts TransFORTH into graphics mode, and directs character input and output to its own special routines. Whenever a character is printed, it is then routed to the character graphics routines to be drawn on the screen.

You should now see the word "Ready" displayed (in graphics) on the Apple high-resolution screen. The cursor is a steady white block, rather than blinking or flashing, since the graphics screen cannot display flashing characters. Type LIST. The usual list of TransFORTH words is displayed, though the characters on the graphics screen scroll much more slowly than characters in text mode. (Any graphics images in the text window will also scroll. This can be used for some interesting effects.) All of the usual TransFORTH screen display words (HTAB, VTAB, WINDOW, etc.) can be used in the text-and-graphics mode.

When you want to return to the normal text display, type TEXT. TEXT switches the special character I/O back to normal in addition to returning to text mode.

Note: GR can be used when in either hi-res mode to erase the entire graphics screen. As in text mode, the word HOME erases only inside the text window.

Another Note: The graphics module uses the same area of memory as the text editor. Because of this overlap, you must return to text mode before calling the editor. The TransFORTH system will recognize that the editor has been overwritten and will reload it from disk. If you want to return to mixed text-and-graphics after editing, you will have to load the graphics module into memory again.

Graphics Drawing Commands

To specify points on the graphics screen, TransFORTH uses "Cartesian coordinates". This is a common method which selects a

point by naming the column and the row the point is in. The horizontal position is the X coordinate and the vertical position is the Y coordinate.

The range of screen coordinates for TransFORTH graphics is:

X from 0 (screen left) to 255 (screen right)

Y from 0 (screen top) to 191 (screen bottom)

Thus, the upper left corner of the screen can be represented with X=0 and Y=0, or simply the X,Y pair (0,0).

The text character display from the graphics module still uses all 280 dots across the screen for 40 characters per line.

For these examples, a text window is used to keep text from scrolling all over the beautiful graphics. The examples that follow will keep the graphics above the text window and away from harm. To enter text-and-graphics mode and establish a window, do the appropriate graphics CALL (see above), then type:

```
Ready 0 40 18 24 WINDOW
```

This sets a 40-column wide window from line 18 to the bottom of the screen. Now type:

```
Ready GR
```

This clears the screen.

The three main drawing words are PLOT, LINE, and FILL. The TransFORTH word PLOT removes two numbers from the stack, interprets them as X and Y coordinates, and plots a point at those coordinates on the screen. The form for PLOT is:

```
<X-coordinate>  <Y-coordinate>  PLOT
```

This example will plot a point in the upper left corner of the screen:

```
Ready 0 0 PLOT
```

Here is another point, near the upper right corner of the screen:

```
Ready 200 25 PLOT
```

The word LINE, like PLOT, removes two numbers from the stack and

interprets them as X and Y coordinates. LINE then draws a straight line from the last plotted point to the given coordinates. To draw a line, you simply use the last point plotted as one of the endpoints, then give LINE the coordinates of the other endpoint:

```
Ready 50 100 LINE
```

This draws a diagonal line from the point (200,25) to (50,100). You can draw another line by starting at the endpoint of the last line:

```
Ready 50 10 LINE
```

This draws a vertical line up from the end of the first line.

TransFORTH can also fill in rectangular areas with the word FILL. FILL removes X and Y coordinates from the stack. It treats the last plotted point as one corner of an area, and the given coordinates as the opposite corner. This example fills in a rectangular area on the right side of the screen:

```
Ready 120 125 PLOT

Ready 200 75 FILL
```

For both LINE and FILL, the "last plotted point" is always the point last used by a plotted word, whether it was PLOT, LINE, or FILL.

Color

Of course, TransFORTH can draw in colors, too! The color is set with the word COLOR. COLOR removes a number from the stack and uses it to select a color. The eight color numbers (0 through 7) are the same as those used by Applesoft Basic. Here is a listing of the graphics colors:

Color Number	Color		
0	Black	(1)	
1	Green	(1)	
2	Violet	(1)	
3	White	(1)	
4	Black	(2)	
5	Orange	(2)	(depends on monitor)
6	Blue	(2)	(depends on monitor)
7	White	(2)	

White dots are always plotted two pixels wide to appear as a true white on color monitors. The orange and blue colors may appear different shades on different monitors. The colors can be divided into two groups. The numbers in parentheses represent the "group number" (either 1 or 2). Because of some Apple hardware constraints, it may be desirable to use colors from the same group when drawing lines or areas close together. We'll show you an example of this in a bit. (The Apple][and //e Reference Manuals contain more information on the strange internal details of these constraints.)

If you don't mind a bit of typing, this example will display 6 diagonal lines in each of the visible colors:

```
Ready GR

Ready 1 COLOR 0 0 PLOT 100 100 LINE

Ready 2 COLOR 20 0 PLOT 120 100 LINE

Ready 3 COLOR 40 0 PLOT 140 100 LINE

Ready 5 COLOR 60 0 PLOT 160 100 LINE

Ready 6 COLOR 80 0 PLOT 180 100 LINE

Ready 7 COLOR 100 0 PLOT 200 100 LINE
```

With your color monitor properly adjusted, the colors of these lines (from left to right) should be green, violet, white, orange, blue, and another brand of white.

Lines and points can be drawn over FILLed areas, but the colors will be affected:

```
Ready GR

Ready 5 COLOR

Ready 0 0 PLOT 100 100 FILL
```

This draws an orange rectangle in the upper left portion of the screen. To draw a line of a different color through it, type:

```
Ready 6 COLOR

Ready 0 0 PLOT 150 150 LINE
```


Note that 6 COLOR specifies blue, but the line appears white when drawn over the orange background. Now try the same example again, this time using colors from different color groups:

```
Ready GR 5 COLOR

Ready 0 0 PLOT 100 100 FILL

Ready 1 COLOR

Ready 0 0 PLOT 150 150 LINE
```

Whoops! You should see a series of small green rectangles along the diagonal. This is the result of the Apple hardware limitations. The solution to avoid this trouble is to simply use colors of the same group when lines or areas are superimposed or placed close together.

While the two whites and the four colors place dots on the screen, the two blacks erase dots. Here are a couple of examples:

```
Ready GR 3 COLOR 0 0 PLOT 100 100 FILL

Ready 0 COLOR 0 0 PLOT 100 100 LINE

Ready 1 COLOR 240 20 PLOT 120 120 LINE

Ready 0 COLOR 240 20 PLOT 120 120 LINE
```

ORMODE and EXMODE

TransFORTH has two different "drawing modes", called "ormode" and "exmode". Amazingly enough, these modes are set with the TransFORTH words ORMODE and EXMODE. The "default" mode (the mode TransFORTH uses when one is not specified) is ORMODE. The philosophy behind ORMODE is that the plotting words put dots of the specified color on the screen regardless of what is already on the screen. With EXMODE, however, a drawing command will put points on the screen only where points are not already plotted. If some points to be plotted are already plotted, those points will instead be turned off.

A couple of examples will be helpful here. First FILL an area, then draw an overlapping line in ORMODE:

```
Ready GR 3 COLOR 100 50 PLOT 150 100 FILL
```

```
Ready 50 50 PLOT 200 100 LINE
```

The line goes straight through the middle of the rectangle. In ORMODE, the way to erase a line is to draw a black line over it:

```
Ready 0 COLOR 50 50 PLOT 200 100 LINE
```

The line was erased, but it neatly chopped the rectangle in half, too. Using EXMODE, anything that can be done can also be undone. Try the same example again, this time in EXMODE:

```
Ready GR EXMODE
```

```
Ready 3 COLOR 100 50 PLOT 150 100 FILL
```

```
Ready 50 50 PLOT 200 100 LINE
```

The line is white, except where it passes over the white background of the rectangle. Here it is changed to black. Now to erase the line, you want to make white sections black, and the black trace through the rectangle white. And this is exactly what happens with any plotting in EXMODE. You can erase the line by telling TransFORTH to draw it again:

```
Ready 50 50 PLOT 200 100 LINE
```

The line is erased, and the rectangle is again intact. The key to understanding EXMODE is that if something is drawn once, it appears on the screen. If it is drawn again, it disappears, leaving the screen as if the object had never been drawn.

Scaling Functions and Graphs

The difficulty in plotting functions is usually in scaling the points so that the graph fills the Apple screen. Here is an example that should give you an idea how to deal with plotting and scaling in general.

Two cycles of a sine wave range from X=0 to X=4*pi (about 12.57) and Y=-1 to Y=1. This can't be plotted without at least some scaling, because TransFORTH can't plot negative numbers.

Here is a word that plots a sine curve that fills the screen:

```
: SINE.CURVE
GR 0 96 PLOT            ( Set graphics & plot first point. )
256 1 DO                ( Loop for 255 more points. )
   I 255 / PI * 4 *     ( Scale the X-coordinate to the sine. )
   SIN                  ( Find the sine. )
   -95 * 96 +           ( Scale the result back to the screen. )
   I SWAP LINE          ( Draw the line from the last point. )
LOOP
GETC DROP TEXT ;        ( Wait for a keypress, then end. )
```

The first scaling (divide by 255 and multiply by 4*pi) brings the X=0 to X=255 down to X=0 to X=4*pi to be used by the sine function. The second scaling (multiply by -95 and add 96 to center it) brings the result of Y=-1 to Y=1 back to the screen range of Y=1 to Y=191. The negative number (-95) is used to turn the points "upside-down", since TransFORTH's Y-coordinate (like Basic's) increases downward, not up.

By running SINCE.CURVE, you can see how the graph is scaled to fill the screen. The program FUNCTION, introduced in Chapter Seven, is designed to scale any range onto the screen. You may wish to adapt it to your own needs.

Character Sets

As discussed earlier, when the graphics module is in use, text characters are literally "drawn" on the graphics screen. The shapes of these characters, however, are not fixed. A portion of the graphics module contains the "table" of character shapes used in drawing the characters. This table is called a "character set" or "font".

There are five different character sets stored as binary files on the TransFORTH system disk:

CHR.SYS
CHR.STOP
CHR.SLANT
CHR.GOTHIC
CHR.BYTE

Each character set uses a different "style". You can load a character set into memory so that it overwrites the set built into the graphics module. Then the new character set (and style) will be used when characters are printed on the hi-res screen.

Like the graphics modules, the character sets are loaded into different areas of memory, depending on how much memory is available. (The graphics module should already be in memory.)

To load a character set into an Apple][48K system, type:

```
Ready CR 132 PUTC PRINT " BLOAD <filename>,A38656 " CR
```

where <filename> is the name of one of the character sets listed above. To load a character set into a 64K Apple, type:

```
Ready CR 132 PUTC PRINT " BLOAD <filename>,A48384 " CR
```

As soon as the new character set is loaded, any character printing on the high resolution screen (including scrolling) will use that character set. With the graphics module active, try loading a few of the character sets into memory to see what they look like. (The CHR.SYS "system" character set is the same one that is built into the graphics module, and its character shapes look like the standard Apple text characters.) Call the graphics module, then BLOAD the file CHR.SLANT, using either the 48K or 64K format, whichever is appropriate. Try CHR.BYTE, then CHR.SYS.

(Note: The TransFORTH character sets use the same format as GraFORTH character sets. If you have GraFORTH, you can create new character sets to be used with TransFORTH. Many of the Apple DOS Toolkit character sets are also compatible, and can be used with TransFORTH.)

Turtlegraphics

Turtlegraphics is also available from TransFORTH. Turtlegraphics is a somewhat different way of specifying how to draw lines. Imagine a tiny turtle sitting on the middle of the screen with a pen tied to his tail. Wherever he moves he draws a line behind him. You can tell him to turn to the left or the right, to walk forward a given distance leaving a straight line behind him, or lift the pen so that a line will not be drawn as he moves. (For the mathematicians among us, this way of drawing lines could be considered as using "relative polar coordinates".)

The Turtlegraphics words in TransFORTH are found on the system disk in a textfile called "TURTLE". These words can be compiled into the word library by typing:

```
Ready DISK> " TURTLE " INPUT
```

You can see the words added to the word library by typing LIST. A few of the words added to the TURTLE file are not used directly, but are called by other words.

Turtlegraphics can be "initialized" in one of two ways. Typing TURTLE sets Turtlegraphics in graphics-only mode, while TURTLE.TEXT calls the graphics module. More specifically:

TURTLE.TEXT first determines if the appropriate graphics module is loaded. If not, it loads it automatically. TURTLE.TEXT then CALLs the module, sets a text window along the bottom 4 lines of the screen, and calls TURTLE.

TURTLE clears the graphics screen, then positions the turtle in the center of the screen, facing up, with the pen down.

To initialize Turtlegraphics with a text-and-graphics display, type:

```
Ready TURTLE.TEXT
```

Once text-and-graphics Turtlegraphics is initialized with TURTLE.TEXT, the turtle can later be reset with either TURTLE.TEXT or TURTLE. (TURTLE does not turn off the mixed text-and-graphics.)

PENUP

The word PENUP "lifts" the turtle's pen so that the turtle can be moved without drawing a line on the screen. The pen stays up until a PENDOWN command is given or the turtle is reset with either TURTLE or TURTLE.TEXT.

PENDOWN

PENDOWN "lowers" the turtle's pen. A line will be drawn whenever the turtle is moved with the pen down. PENDOWN stays in effect until a PENUP command is given.

MOVE

The word MOVE moves the turtle a given number of pixels in the direction it is pointing. If the pen is down, a line will be drawn. If the pen is up, a line will not be drawn. The form is:

<distance> MOVE

The distance is measured in pixels, or dots. To move the turtle 50 pixels, type:

Ready 50 MOVE

TURNTO

The turtle can be turned to a certain angle with TURNTO. TURNTO has the form:

<angle> TURNTO

The angle given is in degrees, and increasing angles are in a clockwise direction. Zero is straight up, 90 is to the right, 180 is facing down, and 270 is to the left. This example turns the turtle to face to the right (to 90 degrees), then moves it 75 pixels:

Ready 90 TURNTO

Ready 75 MOVE

TURN

The word TURN turns the turtle clockwise from its current direction a given angle. The form is the same as for TURNTO, but TURN is a relative turn from the turtle's current direction. The following example now turns the turtle 45 more degrees clockwise, then moves the turtle 50 pixels:

Ready 45 TURN

Ready 50 MOVE

MOVETO

Lastly, MOVETO moves the turtle directly to a specified (X,Y) position on the screen. If the turtle's pen is down when the MOVETO command is given, a line will be drawn. If the pen is up, no line will be drawn, but the turtle's position will be updated. The form for MOVETO is:

<X-coordinate> <Y-coordinate> MOVETO

You can move the turtle to the upper-left corner of the screen, then have the turtle draw a line back to the center, with the following commands:

Ready PENUP

Ready 0 0 MOVETO

Ready PENDOWN

Ready 128 96 MOVETO

At any time, you can also find the current position and angle of the turtle through three variables:

TURTLE.X contains the current X coordinate of the turtle.
TURTLE.Y contains the current Y coordinate of the turtle.
TURTLE.ANG contains the angle the turtle is pointing, in degrees.

For example, you can find which direction the turtle is pointing by retrieving the value of TURTLE.ANG:

Ready TURTLE.ANG .
135

The turtle is still turned 135 degrees from straight up.

Examples

The advantage of Turtlegraphics is that shapes can be drawn in different sizes and facing different directions with little work. For example, to draw a square, you can type the following:

```
Ready TURTLE.TEXT

Ready 5Ø MOVE 9Ø TURN 5Ø MOVE 9Ø TURN

Ready 5Ø MOVE 9Ø TURN 5Ø MOVE
```

A faster way is to repeat the words in a loop:

```
Ready TURTLE

Ready 4 Ø DO 5Ø MOVE 9Ø TURN LOOP
```

This line can be put into a word definition and used at any time:

```
: SQUARE
4 Ø DO
   5Ø MOVE
   9Ø TURN
LOOP ;
```

Now the square can be drawn starting at any point on the screen and turned any direction:

```
Ready TURTLE

Ready PENUP Ø 1ØØ MOVETO PENDOWN SQUARE

Ready PENUP 55 1ØØ MOVETO 3Ø TURNTO PENDOWN SQUARE

Ready PENUP 12Ø 1ØØ MOVETO 6Ø TURNTO PENDOWN SQUARE

Ready PENUP 19Ø 1ØØ MOVETO 9Ø TURNTO PENDOWN SQUARE
```

The following example makes a very nice circular pattern using SQUARE:

```
Ready TURTLE

Ready 36 Ø DO SQUARE 1Ø TURN LOOP
```


This next definition can draw squares of varying sizes. The "size number" (the length of one side in pixels) should be waiting on the stack:

```
: SQUARE1
4 Ø DO
   DUP MOVE
   9Ø TURN
LOOP
DROP ;

Ready TURTLE

Ready 1Ø SQUARE1

Ready 45 TURN 2Ø SQUARE1

Ready GR 65 1 DO I SQUARE1 1Ø TURN LOOP
```

Larger Graphics Programs

The second graphics screen uses Apple addresses 16384 to 24575 (hex $4ØØØ to $5FFF). Unfortunately, if the word library becomes large enough, it can grow directly through this memory area. If you're using high-resolution graphics, program bytes will begin to appear on the hi-res screen as dots, and the next screen erase will destroy the top of the word library.

If you find that a TransFORTH graphics program is growing through address $4ØØØ (if the value of HERE is close to or above 24575), you can adjust the program to hop around the graphics screen area. This is done by declaring a large array in the appropriate place to surround the graphics screen memory. Calling the hi-res graphics words will then change screen bytes that are inside the array, but not affect the rest of the program. Here is the technique:

1. Compile your program onto the word library, then $LIST the library to see what words use what portions of memory. Find the word which has an address less than but closest to $4ØØØ. (Also note about how far from $4ØØØ it is.)

2. FORGET the program, then enter the editor and load the source file for the program. Locate the word found above. Immediately before this word, insert a line to declare an ARRAY that is at least 8192 bytes long. Here is an example declaration:

```
1 8250 1 ARRAY GRAPHICS.SPACE
```

3. Save the new source file back to disk, then compile it onto the word library again.

4. $LIST the new program. Make certain that the address of the new array is less than $4000, and the address of the next higher word is greater than $6000. If this next address is still less than $6000, repeat step 2, increasing the size of the array.

Screen "Dumps" and Saves

If you want, you can save pictures on the hi-res screen to disk as binary files. These files can later be reloaded to put the picture back on the screen.

To create a graphics file on disk, first draw the picture you want to save using TransFORTH's graphics commands. Then execute the following command:

```
CR 132 PUTC PRINT " BSAVE <filename>,A16384,L8192 " CR
```

where <filename> is the name you want the picture saved as. The disk will whir as the screen area is saved. Note: If you type the command in immediate mode while using mixed text-and-graphics, the characters typed will be saved with the screen. It's usually better to put the command in a word definition to be called without affecting the graphics screen.

To load the picture back into the hi-res screen memory, type:

```
CR 132 PUTC PRINT " BLOAD <filename> " CR
```

Many printers include the capability to do a high-resolution "screen dump", printing the contents of the screen on paper. Different printers use different techniques to accomplish this. You should carefully read your printer manual to determine exactly what it "wants" in order to do the dump.

Some printers require you to initialize them with "PR#n", then print a few special characters. For example, the Grappler

interface card uses a CTRL-I followed by the letters "G2". This can be done directly from TransFORTH, as in this example word definition:

```
: DUMP1
CR 132 PUTC PRINT " PR#1 " CR     ( Or use "49408 DEVICE OUTPUT" )
137 PUTC                                         ( Prints CTRL-I )
PRINT " G2 " CR              ( Prints "G2" and a carriage return )
TEXT ;                ( Turns the printer off after the screen dump )
```

Other printers require a special routine to be loaded from disk. If this routine uses the same memory that TransFORTH uses, then the printer dump cannot be called directly from TransFORTH. (As usual, see the memory map in Appendix B.) If this is the case, simply save the screen image to disk as a binary file. From Basic, you should then be able to follow the instructions that came with the printer to load the file into memory and do the screen dump.

Low Resolution Graphics

Low resolution ("lo-res") graphics capabilities are added to the TransFORTH system by compiling the textfile "LO.RES" from disk:

```
Ready DISK> " LO.RES " INPUT
```

Low resolution graphics allow for 40 points horizontally by 48 vertically, for a total of 1920 points. Each point can be one of 16 possible colors. Four lines of text can also be added to the bottom of the display.

Here are the words included in the "LO.RES" file:

LGR

The word LGR (which stands for "Lo-res GRaphics") initializes lo-res graphics mode with a 4-line text window at the bottom of the screen:

```
Ready LGR
```

LGRF

LGRF (which stands for Lo-res GRaphics, Full screen) also initializes the entire screen to low resolution graphics, but without the 4 text lines. LGRF can be used when text is not needed.

LCOLOR

The color of lo-res points is set with the word LCOLOR. The form is:

<color #> LCOLOR

<color #> should be a number from 0 to 15. The Apple lo-res colors are:

0	Black	8	Brown
1	Magenta	9	Orange
2	Dark Blue	10	Grey 2
3	Purple	11	Pink
4	Dark Green	12	Light Green
5	Grey 1	13	Yellow
6	Medium Blue	14	Aquamarine
7	Light Blue	15	White

The following line sets the low resolution graphics color to medium blue:

Ready 6 LCOLOR

Note: Calling either LGR or LGRF always sets the color to black. A color must be set with LCOLOR before plotting any points.

LPLOT

A single point is plotted with the word LPLOT. The form for LPLOT is:

<X-coordinate> <Y-coordinate> LPLOT

The range of low-resolution coordinates is:

X from 0 (screen left) to 39 (screen right)

Y from 0 (screen top) to either 40 (for LGR) or 48 (for LGRF).

Here are a couple of examples:

Ready 28 12 LPLOT

Ready 10 12 LPLOT

LHLINE

LHLINE is used to draw a horizontal line in the current color. The form is:

<left> <right> <Y> LHLINE

<left> and <right> are the X-coordinates for the left and right endpoints of the line. <Y> is the Y-coordinate the line lies on. This example draws a pink horizontal line centered slightly below the middle of the screen. (The line runs from X=9 to X=29 with Y=30.)

Ready 11 LCOLOR

Ready 9 29 30 LHLINE

LVLINE

The word LVLINE is used to draw a vertical line in the current color. The form for LVLINE is similar to LHLINE:

<top> <bottom> <X> LVLINE

<top> and <bottom> are the Y-coordinates for the top and bottom endpoints of the line. <X> is the X-coordinate for the line. The following example draws a line down the center of the screen:

Ready 8 LCOLOR

Ready 5 20 19 LVLINE

If you've been following all of these examples, you should now have a simple silly-looking face drawn on the screen!

LSCRN

You can determine the current color of any point on the low resolution screen with LSCRN. The form is:

<X-coordinate> <Y-coordinate> LSCRN

LSCRN removes the X and Y coordinates from the stack, looks at the given point, and returns on the stack the color number for that point. For example, this line reads the color of the top of the nose on the silly face:

```
Ready 19 5 LSCRN
[ 8 ]
```

Leaving Low Resolution Graphics

To exit the lo-res graphics mode and return to the text screen, type:

```
Ready TEXT HOME
```

An Example

Low resolution graphics, by its very nature, does not have enough resolution for many tasks such as plotting function curves, etc. However, lo-res graphics is faster and has a much wider range of colors that hi-res. The following example program creates a rapidly changing colorful display. It simply draws random horizontal and vertical lines in random colors (except black). (The file "LO.RES" must be compiled before this can be run.)

```
: RANDOM        ( Select a random integer less than the number )
1 RND * INT ;   ( given on stack. )

: GETKEY        ( Read ASCII value of key last pressed. )
49162 PEEK ;
```



```
: WILD
LGRF                                    ( Set full-screen lo-res )
BEGIN                                   ( Start loop: )
   15 RANDOM 1 + LCOLOR                          ( Random color )
   39 RANDOM 39 RANDOM 47 RANDOM LHLINE  ( Random horiz. line )
   15 RANDOM 1 + COLOR                           ( Random color )
   47 RANDOM 47 RANDOM 29 RANDOM LVLINE  ( Random vert. line )
   GETKEY 160 =                         ( Has space bar been pressed? )
UNTIL
TEXT HOME ;                             ( Return to text mode )
```

To run the program, simply type:

```
Ready WILD
```

The screen goes "wild".... Press the space bar to exit.

Summary

TransFORTH displays high-resolution graphics on the Apple's second hi-res screen. An 80-column card may have to be removed or disabled for graphics to work. Two hi-res "modes" are available.

The graphics-only mode is entered with the TransFORTH word GR.

Graphics with text printing on the graphics screen is made available by loading and compiling one of the two graphics modules on disk. 48K Apple]['s use the "GR.TEXT.48K" module called at location 37888; 64K systems use "GR.TEXT.64K" called at location 47616. These routines route TransFORTH's I/O to "draw" output characters on the screen. TEXT is used with either graphics mode to return to a normal text screen.

The word PLOT plots a point on the hi-res screen. LINE draws a line from the last plotted point. FILL fills a rectangular area using the given coordinates and the last plotted point as opposite corners of the area. The range of coordinates for these words is X = 0 to 255 and Y = 0 to 191.

COLOR sets the plotting color for these words. The eight standard hi-res colors are used. If either "black" is used, points are erased rather than plotted. In ORMODE, points are plotted regardless of what is currently on the screen. In EXMODE, if a point is plotted over a pixel that is already on, that pixel is instead turned off. Drawing a line a second time

in EXMODE erases the line.

There are several character sets on the TransFORTH disk, each with a different character "style". When using the graphics module, loading one of these over the original character set will change the style of the characters printed.

Turtlegraphics is included in TransFORTH, which uses an imaginary turtle on the screen for drawing lines in different directions and lengths. TURTLE initializes Turtlegraphics in graphics-only mode. TURTLE.TEXT also loads the appropriate graphics module (if necessary), then calls it. The words used to manipulate the turtle are PENUP, PENDOWN, MOVE, MOVETO, TURN, and TURNTO. The turtle's current position and angle are stored in TURTLE.X, TURTLE.Y, and TURTLE.ANG.

If a graphics program grows large enough to overlap the graphics screen memory, a dummy array must be declared in the appropriate place to surround the screen memory, protecting the rest of the program from harm.

High resolution screen images can be sent to a printer that has screen-dump capabilities, but the methods depend on what printer is used. The screen can always be saved to disk with a simple BSAVE. The instructions with the printer should then explain how to load and print this picture file from Basic.

Low resolution graphics are available by compiling the textfile "LO.RES". LGR initializes lo-res graphics with a 4-line text window; LOGRF initializes full screen graphics. The lo-res plotting words are LCOLOR, LPLOT, LHLINE, LVLINE, and LSCRN. TEXT HOME returns TransFORTH to text mode.

Problems

Note: As usual, there is more than one possible solution to each of these problems. Because of the visual aspect involved, feel free to experiment with TransFORTH to find the answers. Then compare your answers with the solutions below.

(1)
Write a colon definition that draws a triangle on the high-resolution screen (any triangle!).

(2)
After executing this word

```
: WHAT'S.LEFT?
GR EXMODE
0 0 PLOT 100 100 LINE
0 0 PLOT 99 99 LINE ;
```

What lines or points will be left on the screen?

(3)
Write a word definition to print the phrase "MIXED TEXT AND GRAPHICS" on the hi-res screen, and surround it with a rectangular border.

(4)
Write a word definition that will draw an equilateral triangle using Turtlegraphics.

(5)
Using low-resolution graphics, plot 16 vertical bars side by side, each bar in a different color. Start with black at the left and go through the colors to white on the right.

Solutions to Problems

(1)
Here is one triangle:

```
: TRIANGLE
GR  100 0 PLOT  200 100 LINE
0 100 LINE  100 0 LINE ;
```

(2)
Because EXMODE is set, drawing the second line erases most of the first line. Only a single point at (100,100) remains on the screen.

(3)

```
: PHRASE
11 VTAB 9 HTAB
PRINT " MIXED TEXT AND GRAPHICS "
20 VTAB
55 112 PLOT 231 112 LINE 231 88 LINE
55 88 LINE 55 112 LINE ;
```

```
(4)
: TRIANGLE1
50 MOVE 120 TURN
50 MOVE 120 TURN
50 MOVE ;
```

or

```
: TRIANGLE2
3 0 DO
   50 MOVE
   120 TURN
LOOP ;
```

(Notice that in TRIANGLE2, the turtle is turned three times for a a total of 360 degrees (a full circle). Any regular polygons drawn this way (including SQUARE in the text above) will divide the 360 degrees by the number of sides in the polygon.)

```
(5)
: COLOR.BARS
LGR
16 0 DO
   I LCOLOR
   0 30 I LVLINE
LOOP ;
```


CHAPTER TEN; APPLE //e AUXILIARY MEMORY

CHAPTER TABLE OF CONTENTS: Page

Installing the Auxiliary Memory Features ***10-1***

Understanding Auxiliary Memory ***10-1***

Using Auxiliary Memory ***10-2***

Single Address Words 10-3
Arrays in Auxiliary Memory 10-4
Double Address Words 10-6
Character Input and Output 10-7

Saving High-Resolution Pictures in Auxiliary Memory ***10-8***

Summary ***10-9***

If you have an Apple //e with an extended 80-column text card, you can use the auxiliary memory on this card with TransFORTH. TransFORTH fully supports the use of the card for up to 46 Kilobytes of extra data storage. You can store individual numbers, strings, arrays, or even entire textfiles in the auxiliary memory. For rapid graphics effects, you can also save up to five high-resolution pictures on the card, then display any of them on the screen with a single command.

TransFORTH's auxiliary memory commands are not difficult to use, but they do require a little forethought. When storing or retrieving information, you will often be dealing directly with memory addresses on the card. This means you need to allot enough memory for everything you want to store, and choose your addresses accordingly. In addition, auxiliary memory can provide you with an unsurpassed opportunity for crashing the system, since a free area in auxiliary memory may correspond to a part of your valuable program in main memory. (Discovering that the monthly payroll program has just been overwritten with a high-res picture of Mickey Mouse can ruin anybody's day....) Plan ahead, keep back-ups, and you'll find that the extra memory can be extremely helpful.

Installing the Auxiliary Memory Features

First make certain that the extended 80-column text card is installed in your Apple //e. The TransFORTH language already includes part of the necessary software for using the memory on the card. To bring in the remainder and to activate the auxiliary memory features, simply compile the file "AUXILIARY":

```
Ready DISK> " AUXILIARY " INPUT
```

This loads a special machine language module, and adds six words to the word library. These words are AUX, SETAUX, SETMAIN, FIRST, SECOND, and AUXMEM. TransFORTH is now ready to use auxiliary memory. (Once the file is compiled, the features remain active until you reboot your Apple.)

Understanding Auxiliary Memory

Before using auxiliary memory from TransFORTH, it's necessary to have a basic understanding of how auxiliary memory "relates" to main memory and to the Apple processor. The Apple //e Reference

Manual and Extended 80-Column Text Card Manual provide much more complete information, but here are the basics: As discussed in Chapter Six, the Apple processor is only "aware" of 64K (65,536) different locations. Each location has a fixed address, a number from 0 to 65,535. This means that even with 128K of memory installed in your computer, the processor can only see 64K of it at any one time. On the Apple //e, the 128K is divided into two 64K blocks, main memory and auxiliary memory. Main memory is built into the Apple; auxiliary memory is provided on the extended text card.

The Apple //e uses special software switches to "map in" either main memory or auxiliary memory. Whatever memory is mapped in is the memory that the processor "sees". The processor then uses the same 65,536 addresses when accessing either memory. The only difference as far as the processor is concerned is the setting of the software switches. The Apple can also select a few combinations of main and auxiliary, so that some addresses access main memory, and others access auxiliary.

TransFORTH handles all of the complicated switching between main and auxiliary memory. The only things you need to be concerned about are:

1) What is the address?
2) Is it in main or auxiliary memory?

The addresses in auxiliary memory that you can use are 2048 through 49151, for a total of 47,104 (46K) locations. Trying to access addresses outside of this range in auxiliary memory may have unpleasant (that's the nice word) results. The memory map in Appendix B shows how all of memory is used.

Using Auxiliary Memory

The TransFORTH auxiliary memory words can be divided into three groups:

1) AUX, SETAUX, and SETMAIN affect the TransFORTH "single-address" words: PEEK, PEEKW, PEEKN, POKE, POKEW, POKEN, MARRAY, ERASE, READLN, WRITELN, ASSIGN>, LENGTH, and GETNUM.

2) FIRST and SECOND affect the TransFORTH "double-address" words: MOVMEM, CONCAT, COMPARE, and MOVELN.

3) AUXMEM affects the TransFORTH I/O words INPUT and OUTPUT.

Single-Address Words

The "single-address" words listed above all remove one address from the stack, then store to or retrieve from that address. The words AUX, SETAUX, and SETMAIN allow you to select whether you want that address to be in main memory or auxiliary memory.

The single-address words, of course, usually access main memory. To make one of the words access auxiliary memory instead, simply precede it with the word AUX. For example,

```
Ready 8000 PEEK
```

reads a value from location 8000 in main memory, while

```
Ready 8000 AUX PEEK
```

reads a value from location 8000 in auxiliary memory. The next PEEK (unless you precede it with another AUX) will then access main memory again. All of the single-address words work the same way. Here are a few more examples:

```
Ready 1555 8000 POKEW
```

(The number 1555 is poked into location 8000 in main memory.)

```
Ready 1212 8000 AUX POKEW
```

(1212 is poked into location 8000 in auxiliary memory.)

```
Ready 8000 PEEKW .     ( The value in main memory is read back. )
1555

Ready 8000 AUX PEEKW . ( The value in auxiliary memory is read. )
1212

Ready 8400 AUX ASSIGN> " I'M IN AUXILIARY MEMORY! "
```

(A string is assigned directly into address 8400 in auxiliary memory.)

```
Ready 8400 ASSIGN> " I'M IN MAIN MEMORY. "
```

(Another string is assigned to the same address in main memory.)


```
Ready 8400 AUX WRITELN
I'M IN AUXILIARY MEMORY!

Ready 8400 WRITELN
I'M IN MAIN MEMORY.
```

Whenever the word AUX is executed, it sets a flag that indicates that the next single-address word that is executed will use auxiliary memory. Every single-address word checks the flag for itself, then clears the flag.

If most of the data you're manipulating is in auxiliary memory, then including AUX in front of every single-address word can become tedious. That's why SETAUX and SETMAIN are available. SETAUX forces <u>all</u> subsequent single-address words to use auxiliary memory. SETAUX stays in effect until SETMAIN is called. SETMAIN tells all single-address words to return to using main memory. (ABORT or pressing Reset also turn off SETAUX, returning access to main memory.)

```
Ready SETAUX
```

(All single-address words will now use auxiliary memory.)

```
Ready 8000 PEEKW .
1212

Ready 8400 WRITELN
I'M IN AUXILIARY MEMORY!

Ready SETMAIN
```

(Single address words are returned to main memory.)

```
Ready 8400 WRITELN
I'M IN MAIN MEMORY.
```

Arrays in Auxiliary Memory

By using MARRAY, you can create an array in auxiliary memory. Remember that when an array is first declared, its contents are first cleared to zeros. If auxiliary mode is set ahead of time (with either AUX or SETAUX), then the array area cleared will be in auxiliary memory. Auxiliary mode must be set before beginning the MARRAY declaration. Here is an example:

```
Ready AUX

Ready 1 200 1 16384 MARRAY BEST
```

This declared a one-dimensional 200-element array named BEST starting at location 16384 in auxiliary memory, clearing the array to zeros. This next example is not correct, since the AUX appears in the middle of the definition:

```
Ready 1 200 1 16384 AUX MARRAY BOZO
```

Important: Even when a memory array is declared with auxiliary memory set, the array doesn't "know" that it uses auxiliary memory. That information is not stored with the array. Calling the array name (with subscripts) returns the address of an element in the array as before. Once on the stack, that address is simply a number, and there's no way to determine from that number whether the next single-address word should use main or auxiliary memory. This means that you must still set auxiliary mode whenever you want to access or ERASE the array. Here are a few examples, using the array BEST as defined above:

```
Ready 0 BEST
[ 16384 ]
```

(Retrieve the address of the first element in BEST.)

```
Ready AUX      ( Set auxiliary mode. )
[ 16384 ]

Ready ASSIGN> " THE BEST AUXILIARY ARRAY AROUND... "
```

(Assign a string into BEST.)

```
Ready 0 BEST AUX WRITELN    ( Print the string in BEST. )
THE BEST AUXILIARY ARRAY AROUND...

Ready AUX 0 BEST LENGTH
[ 34 ]
```

(You can put the AUX before the array name if you like.)

```
Ready AUX ERASE BEST
```


Double Address Words

The TransFORTH words MOVMEM, CONCAT, COMPARE, and MOVELN are "double-address" words. Each of them removes two addresses from the stack to work with two different areas of memory simultaneously. MOVMEM moves a block of memory from one address to another, CONCAT concatenates two strings, COMPARE compares two strings, and MOVELN moves a string from one address to another. Obviously, AUX and SETAUX will not work with these words if one address is in main memory and the other is in auxiliary. Instead, the words FIRST and SECOND are used. FIRST refers to the first address used by these words, and SECOND refers to the second.

The format for both FIRST and SECOND is:

```
<number> FIRST
<number> SECOND
```

The word FIRST removes one number from the stack. If the number is 0, then whenever a double-address word is called, the first address will access data in main memory. If the number is 1, then the first address will access data in auxiliary memory.

The word SECOND works much the same way, but applies toward the second address used by a double-address word. If the number SECOND removes is 0, then the second address will reference main memory. If the number is 1, then the second address will reference auxiliary memory.

When the auxiliary memory features are loaded, both FIRST and SECOND are set to zero, to access main memory. If you execute an ABORT or press Reset, FIRST and SECOND will be reset to zero.

A few examples should help clarify this. The following line:

```
Ready 24576 32768 256 MOVMEM
```

moves a block of 256 bytes from address 24576 to 32768. If you want to move the bytes from 24576 in auxiliary memory to 32768 in main memory, just set the FIRST address for auxiliary and the SECOND address for main:

```
Ready 1 FIRST  0 SECOND

Ready 24576 32768 256 MOVMEM
```

To move memory from auxiliary to auxiliary, simply set both FIRST and SECOND to 1:

Ready 1 SECOND

(1 FIRST was already set in the above example.)

Ready 24576 32768 256 MOVMEM

If you want things to operate normally again, be sure to set FIRST and SECOND back to main memory!

Ready Ø FIRST Ø SECOND

Character Input and Output

With the word AUXMEM, you can use auxiliary memory as a source of character INPUT or OUTPUT, in the same way that the word MEMORY lets you use main memory for character I/O. Auxiliary memory, with 46K available, can store much larger files than any free area in main memory. The forms for AUXMEM are:

<address> AUXMEM INPUT
<address> AUXMEM OUTPUT

Once set, all character I/O will use the auxiliary memory for INPUT or OUTPUT until an End-Of-File condition occurs or a CLOSE is executed. (See Chapter Eight for more general information on I/O.)

As an example, you can specify a textfile as a source of input, an address in auxiliary memory as a destination, and move the entire texfile into auxiliary memory with one line of code!

Ready DISK> " BASCON " INPUT 2Ø48 AUXMEM OUTPUT MOVFILE

The disk whirs as the file is transferred. Now that the file is in auxiliary memory, you can compile it onto the word library, read the text one line at a time from a program, make changes, etc., all much faster in memory than on disk. To compile the file in auxiliary memory, simply type:

Ready 2Ø48 AUXMEM INPUT

Now type LIST. The BASCON words have been added to the top of

the library. If you want to print the file directly to the screen, enter:

Ready 2Ø48 AUXMEM INPUT MOVFILE

Auxiliary memory can be used much like a RAM-based "disk". Suppose you've written a program that accesses several textfiles, and the disk access is slowing things down. If you know about how many bytes long each file is, you can read all of your files into auxiliary memory when the program begins, each file starting at a different address. As the program runs, working directly with the files in memory removes the need for the slower disk access. When the program is finished, simply write the files back to disk if necessary.

Note: As shown above, the word AUXMEM allows you to move textfiles to and from auxiliary memory. However, Apple DOS cannot access this memory directly. This means you cannot readily BLOAD or BSAVE data to and from auxiliary memory. The best way to get around this is to use MOVMEM to transfer the data between auxiliary and main memory, and do the disk access from main memory.

Saving High-Resolution Pictures in Auxiliary Memory

Below are two colon definitions that allow you to save up to 5 high-res screen images into auxiliary memory, then transfer them back to the display screen. Loading an image from auxiliary memory is much faster than BLOADing from disk, and can be used for interesting graphics effects.

The word definitions SAVE.PICTURE and GET.PICTURE each remove a number (1 through 5) to select which save area is to be used. For example:

Ready 2 SAVE.PICTURE

saves the contents of the TransFORTH high-res screen to the second buffer in auxiliary memory.

Ready 5 GET.PICTURE

gets the contents of the fifth buffer and moves it to the display screen.

```
: SAVE.PICTURE
0 FIRST 1 SECOND
16384 SWAP 8192 * 8192 MOVMEM
0 SECOND ;
```

```
: GET.PICTURE
1 FIRST 0 SECOND
8192 * 16384 8192 MOVMEM
0 FIRST ;
```

Summary

If you have an Apple //e with an extended 80-column text card installed, you can access up to 46K of auxiliary memory. The auxiliary features are provided by compiling the textfile "AUXILIARY". All of the TransFORTH words which access Apple main memory by address can also access auxiliary memory. The allowable auxiliary address range is 2048 to 49151.

The TransFORTH "single-address" words all remove one address from the stack to select a location in memory. If a single-address word is prefaced with AUX, then the one access will be to auxiliary memory. SETAUX causes all single-address words to refer to auxiliary memory, until cancelled by SETMAIN, ABORT, or pressing Reset.

MARRAYs can be declared in auxiliary memory for versatile data storage. However, an array doesn't "know" that whether it uses main or auxiliary memory. Each access to the array must be done with auxiliary mode set.

The TransFORTH "double-address" words remove two addresses from the stack. Whether each address refers to auxiliary or main memory is determined by the words FIRST and SECOND. FIRST and SECOND remove a number from the stack. If the number is 0, then the first/second address of all double-address words will refer to main memory. If the number is 1, then the first/second address will refer to auxiliary memory.

Auxiliary memory can be used for character I/O by using either <address> AUXMEM INPUT or <address> AUXMEM OUTPUT. If the approximate lengths of the files are known, then several textfiles can be stored in auxiliary memory simultaneously, reducing the need for frequent disk access.

APPENDIX A: TransFORTH WORD LIBRARY LISTING

TransFORTH Word Library Listing

The following is a list of the words in the TransFORTH word library. The list includes the word name, a "before and after" picture of the stack, the page number in the text where the word is first introduced, and a brief description of what the word does.

The stack picture shown represents relevant numbers on the top of the stack indicated by letters. The top of the stack is to the right. A dash represents an empty stack. How the words use the stack can often be inferred simply from the stack picture.

The word descriptions here are concise and a bit more technical. For more information on each word, we suggest you refer back to the text, using the page numbers provided and the index in the back of this manual.

Word	Before	After	Page
"	-	-	2-15

A set of quotes surrounding text causes the text to be compiled into the program. Used only with PRINT, ASSIGN>, and DISK>.

Word	Before	After	Page
$	-	-	7-2

Sets dollar display format, which causes numbers to be printed as 10 characters, with leading spaces, aligned decimal points, and two digits to the right of the decimal. Numbers greater than 9999999.99 and less than 0.01 are displayed in scientific notation.

Word	Before	After	Page
$LIST	-	-	3-12

Lists words in the word library with hexadecimal addresses. At each pause, press CTRL-C to stop the listing, or any other key to continue. If the $LIST is being written to disk (with DISK OUTPUT), the listing will not pause.

Word	Before	After	Page
'	-	a	7-10

a = address of the word that follow the ' (tic). It also prevents that word's execution. Will not work correctly with compiling words like FORGET, VARIABLE, etc.

Word	Before	After	Page
(	-	-	5-11

Indicates the beginning of a program comment, which is passed over by the TransFORTH compiler. A right-parenthesis ")" set apart by one or more spaces marks the end of the comment. A comment can extend over several lines.

Word	Before	After	Page
*	m n	p	2-7

p = m * n (multiplication)

Word	Before	After	Page
+	m n	p	2-5

p = m + n (addition)

Word	Before	After	Page
+LOOP	n	-	4-2

Marks the end of a DO - LOOP structure, using n as a loop value increment. After adding n, if the loop value is equal to or beyond the ending value, the loop ends. Otherwise, execution loops back to the corresponding DO.

Word	Before	After	Page
,	-	-	7-12

Compiles a single byte directly into the code of a word definition. The comma should follow a number from 0 to 255 inside a colon definition.

Word	Before	After	Page
-	m n	p	2-13

p = m - n (subtraction)

Word	Before	After	Page
->	-	-	6-2

(Store-arrow) causes the next variable reference to store the top stack value into the variable, rather than placing the variable value on the stack.

Word	Before	After	Page
.	n	-	2-5

(Period) prints n to the output, using the current number display format.

Word	Before	After	Page
/	m n	-	2-13

p = m / n (division)

Word	Before	After	Page
:	-	-	3-2

Marks the beginning of an executable (colon) word definition. The name that follows is the name of the new word. The next words define the new word, and a semicolon ends the colon definition.

Word	Before	After	Page
;	-	-	3-2

Marks the end of a colon definition.

Word	Before	After	Page
<	m n	p	4-5

p = 1 if m < n, otherwise p = Ø. (Less than)

Word	Before	After	Page
<=	m n	p	4-5

p = 1 if m <= n, otherwise p = Ø. (Less than or equal to)

Word	Before	After	Page
<>	m n	p	4-5

p = 1 if m <> n, otherwise p = Ø. (Not equal to)

Word	Before	After	Page
=	m n	p	4-5

p = 1 if m = n, otherwise p = Ø. (Equal to)

Word	Before	After	Page
>	m n	p	4-5

p = 1 if m > n, otherwise p = Ø. (Greater than)

Word	Before	After	Page
>=	m n	p	4-5

p = 1 if m >= n, otherwise p = Ø. (Greater than or equal to)

ABORT (not applicable) 7-4
Restarts TransFORTH. Program is stopped, screen erased, text window reset, all I/O returned to normal, any open textfiles closed, MAXFILES reset to 1, TransFORTH header printed (unless in AUTORUN mode), set ORMODE, hi-res color white 3, clear store-arrow and EOF, zero EOFCHR, reset stacks.

ABS m n 2-13
n = absolute value of m.

AND m n p 4-6
p = 1 if both m and n are nonzero, otherwise p = Ø.

AREG (variable) 7-11
Value of AREG (Ø to 255) is placed into processor Accumulator before a CALL. After CALL, contents of Accumulator are loaded back into AREG.

ARRAY - - 6-8
Declares an array with following name. Numbers before "ARRAY" are (in reverse order) number of dimensions, number of elements along each dimension, and number of bytes per element. Array is added to word library with all elements cleared to zero. Declaring an array also destroys contents of PAD.

Word	Before	After	Page
ASSIGN>	a	-	6-17

Places following quoted text into memory starting at address a.

ATN m n 2-14
n = arctangent of m (n in radians).

BEGIN - - 4-13
Marks the beginning of a BEGIN - WHILE - REPEAT or BEGIN - UNTIL construct, where progam execution can loop back to.

BELL - - 4-17
Prints a CTRL-G, which will beep the Apple speaker if normal output is active.

BYE (not applicable) 7-13
Exits TransFORTH to the Apple system monitor. (TransFORTH can be reentered by typing CØØG from the monitor.)

CALL a - 7-11
Loads processor registers (values between Ø and 255) from variables AREG, XREG, YREG, and PREG, calls machine language routine at address a, then on return stores register values back into variables.

CASE: n - 4-15
Selects and executes the nth word from the following list of words, numbered starting from Ø. THEN closes off the list of CASE: words and marks the point where execution continues. If n is greater than the number of CASE: words or less than Ø, the system may hang.

CLOSE - - 8-4
Closes any open textfile, resets MAXFILES 1, and returns TransFORTH I/O to normal.

COLOR n - 9-5
Selects the color for high-resolution graphics. Valid range is Ø to 7.

COMPARE a b n 6-25
Alphabetically compares strings at addresses a and b. n = 1 if string a > string b, n = Ø if string a = string b, n = -1 if string a < string b. (Zero or EOFCHR mark the end of either string.)

Word Before After Page

CONCAT a b - 6-24
Concatenates string at address b to the end of string at address a. String b is unchanged. (LENGTH is called to find end of string a; MOVELN is called to copy string b to string a.)

COS m n 2-13
n = cosine of m (m in radians).

CR - - 2-15
Prints a carriage return (ASCII value 141, or CTRL-M) to the current output.

DEVICE - 2 8-1
Places a 2 on stack to designate a device (peripheral card or subroutine) for subsequent INPUT or OUTPUT.

DISK - 1 8-1
Places a 1 on stack to designate a textfile on disk for subsequent INPUT or OUTPUT.

DISK> - a 2 8-1
Compiles following quoted text (hopefully a filename) into memory, places address a of filename on stack, followed by a 2 to designate a textfile on disk for subsequent INPUT or OUTPUT.

DO m n - 4-1
Initializes a DO - LOOP, using n for an initial loop value and m as an ending value, pushing them onto return stack. At corresponding LOOP or +LOOP, if incremented loop value is not equal to or beyond ending value, execution loops back to DO.

DROP n - 2-9
Discards n from the stack.

DUP n n n 2-1Ø
Makes a copy of n on the stack.

ECHO n - 8-9
If n = 1, echo input characters to the screen. If n = 2, echo output characters. If n = Ø, no screen echo.

EDIT (not applicable) 5-3
Loads from disk (if necessary) and runs the appropriate text editor (OBJ.EDITOR1 for 48K Apple]['s, OBJ.EDITOR2 for Apple //e's or 64K Apple]['s).

Word Before After Page

ELSE - - 4-1Ø
Separates the two controlled areas in an IF - ELSE - THEN construct. If stack value removed by IF is nonzero, words between IF and ELSE are executed; if number is zero, words between ELSE and THEN are executed.

ENG - - 7-2
Sets engineering display format, causing numbers to be printed as one to three digits, optional decimal and more digits, with an exponent that is a multiple of 3.

EOF - n 8-6
n = 1 if an End-Of-File condition occurs, then is reset to Ø when INPUT or OUTPUT are called again or if ABORT is executed or system restarted.

EOFCHR n - 8-7
n specifies value of ASCII character to use to flag End-Of-File. All I/O and string operations use this character to detect End-Of-File or End-Of-String.

ERASE - - 6-11
Erases (clears to zero) contents of array. Array name follows. Use ERASE only with arrays.

EXMODE - - 9-7
Sets high-resolution graphics EXclusive-or Mode, causing subsequent plotted points to turn on corresponding screen pixels that are off, and turn off pixels that are on.

EXP m n 2-14
n = e ^ m, where e = 2.71828182. (natural exponent) If m < -88 or m > 88, result is out of range and n will be indeterminate or a divide-by-zero error may occur.

FILL x y - 9-5
In hi-res graphics, fills a rectangular area with diagonal corners on the last plotted point and (x,y). x can range from Ø to 255, y from Ø to 191.

FIX - - 7-2
Sets "fixed decimal" floating-point display format, causing numbers to be printed without an exponent. Numbers greater than 999,999,999 or less than Ø.Ø1 are printed in scientific notation.

Word	Before	After	Page
FORGET	-	-	3-9

Removes following word and all words in the TransFORTH word library above it.

FRAC	m	n	2-14

n = fractional portion of m (portion to the right of the decimal point).

GETC	-	n	6-27

Gets a single character from the current input, placing its ASCII value n on the stack. (Sets EOF if character read is EOFCHR.)

GETNUM	a	n	6-19

Converts text string at address a into number n. Nonnumeric characters may follow the number, but the number must begin as the first character of the string. Unsuccessful conversions return Ø. (Ignores EOFCHR when reading string.)

GR	-	-	9-1

Clears the high-resolution screen (screen 2) to black and displays this screen on Apple video.

HERE	-	a	3-12

Returns the address of the top of the TransFORTH word library. (See Appendix C for more information.)

HEXPRT	n	-	7-2

Prints n (Ø to 255) as a pair of hexadecimal digits (ØØ to FF).

HOME	-	-	7-2

Prints a CTRL-L. If normal output is active, this erases the text screen inside the text window and moves the cursor to the upper left corner of the window.

HTAB	h	-	7-1

Sets the text column for subsequent printing. HTAB is relative to the left margin of the text window. Valid range is Ø to 39 (for 4Ø-column screen) or 79 (for 8Ø-column card), less if a text window is set.

I	-	n	4-1

Returns the top return stack value, which is the loop value for the current innermost loop if no PUSHes, PULLs, or POPs have been done inside the loop.

Word	Before	After	Page
IF	n	-	4-8

Marks the beginning of an IF - THEN or IF - ELSE - THEN construct. If n is nonzero, words between IF and THEN (or IF and ELSE) are executed, otherwise execution continues after THEN (or between ELSE and THEN).

INPUT	a n	-	8-1

Removes "I/O designator" n (for DEVICE, MEMORY, or DISK) and address a from stack to select a source for character input. (If n = Ø, input is returned to normal and n is the only number removed from stack.)

INT	m	n	2-14

n = integer portion of m (truncated toward zero).

INVERSE	-	-	7-2

Causes text characters printed to the Apple screen to be in inverse (black-on-white).

J	-	n	4-3

Returns the third return stack value, usually the loop value for the next outer loop.

K	-	n	4-3

Returns the fifth return stack value, usually the loop value for the third outer loop.

LENGTH	a	n	6-23

Returns the number of characters in string at address a (not including the End-Of-String marker: either zero or EOFCHR).

LINE	x y	-	9-4

In hi-res graphics, draws a line from the last plotted point to (x y). x ranges from Ø to 255, y from Ø to 191.

LIST	-	-	2-2

Lists the words in the TransFORTH word library. At each pause, press CTRL-C to stop the listing, or any other key to continue. If the LIST is being written to disk (with DISK OUTPUT), the listing will not pause.

LOG	m	n	2-14

n = natural logarithm of m. (Logarithms of negative numbers are mathematically undefined. If m is negative, n = logarithm of the absolute value of m.)

Word	Before	After	Page
LOOP	-	-	4-1

Marks the end of a DO - LOOP structure, incrementing the loop value. If the loop value is less than the ending value, execution loops back to the words after the corresponding DO.

MARRAY	-	-	6-12

Declares an array with the array data in a specified free area of memory. The form is similar to ARRAY, but the desired array address should be the last number before MARRAY on the line.

MEMORY	-	3	8-1

Places a 3 on stack to designate Apple memory for subsequent input or output.

MOD	m n	p	2-13

p = remainder after dividing m by n. (modulo)

MOVELN	a	b	6-24

Copies a line of text from address a to address b. (Stops at zero, EOFCHR, or carriage return; writes zero to end string.)

MOVFILE	-	-	8-8

Copies characters from input to output until End-Of-File (zero at end of textfile, or EOFCHR).

MOVMEM	a b n	-	7-10

Moves a block of n bytes from address a to address b.

NEGATE	m	n	2-13

n = - m.

NORMAL	-	-	7-2

Resets character printing on the Apple screen to normal white-on-black.

NOT	m	n	4-6

n = 0 if m is nonzero, n = 1 if m is zero.

NOTE	p d	-	7-8

Sounds a note of pitch p and duration d through the Apple speaker. Valid range for both p and d is 1 to 255. Greater pitch numbers produce lower pitches. Greater duration numbers play longer notes.

Word	Before	After	Page
OR	m n	p	4-6

p = 1 if either m or n are nonzero. p = 0 if both m and n are zero.

ORMODE	-	-	9-7

Sets hi-res graphics OR drawing mode, causing points to be plotted regardless of what screen pixels are currently on or off.

OUTPUT	a n	-	8-1

Removes "I/O designator" n (for DEVICE, DISK, or MEMORY) and address a from stack to select destination for character output. (If n = 0, output is returned to normal, and n is the only number removed from stack.)

OVER	m n	m n m	2-10

Copies m (second stack value) to top of stack.

PAD	-	832	6-22

Returns the address (832) of the 144-byte "PAD" string space.

PEEK	a	n	6-5

Reads single byte integer value n from address a.

PEEKN	a	n	6-5

Reads floating-point value n from the 5 bytes starting at address a.

PEEKW	a	n	6-5

Reads integer value n from the 2 bytes starting at address a.

PI	-	3.14159266	2-14

Returns value of pi (3.14159266).

PICK	..k m n	..k m p	2-11

Copies the nth item (p) to top of stack.

PLOT	x y	-	9-4

In hi-res graphics, plots a point with coordinates (x,y). x can range from 0 to 255, y from 0 to 191.

POKE	n a	-	6-5

Stores single byte integer value n into address a.

POKEN	n a	-	6-5

Stores floating-point value n into 5 bytes starting at address a.

Word	Before	After	Page

POKEW n a - 6-5
Stores integer value n into 2 bytes starting at address a.

POP - - 4-4
Discards top return stack value.

PREG (variable) 7-11
Value of PREG is stored into processor status register before a CALL. After CALL, value of status register is stored back into PREG. (If improper status bits are still set on return from CALL, the system may not work properly.)

PRINT - - 2-15
Prints following quoted text to the current output.

PROGRAM - a 5-10
Returns current address of editor program buffer. (If editor has not been loaded, PROGRAM returns default buffer address of 24577.)

PULL - n 4-4
Moves top return stack value n to data stack.

PUSH n - 4-4
Moves top data stack value n to return stack.

PUTC n - 6-27
Prints character with ASCII value n to current output. (Sets EOF if character printed is EOFCHR.)

READLN a - 6-18
Reads a line from current input into string starting at address a. (String ends when input character is zero, carriage return, or EOFCHR. Zero is written to mark end of string.)

REPEAT - - 4-14
Marks the end of the BEGIN - WHILE - REPEAT construct, causing execution to jump back to the words following the corresponding BEGIN.

RND m n 2-14
n is a random number: 0 <= n < 1. If m is positive, new random number. If m = 0, repeat last random number. If m is negative, begin new pseudorandom sequence.

Word	Before	After	Page

ROLD j k m n k m n j 2-11
Rolls stack down, moving top three values down one position and moving 4th value to top of stack.

ROLU j k m n n j k m 2-12
Rolls stack up, moving top stack value to 4th position, and moving next three values up one position.

RUN - - 7-4
Executes the top word on the word library.

SAVEPRG - - 7-6
Saves the current TransFORTH system, along with any changes and additions to the word library, to disk as an executable binary file.

SCI - - 7-2
Sets scientific notation display format, where numbers are printed as a single digit followed by an optional decimal point and up to 8 more digits, then an exponent.

SIGN m n 2-13
n = 1 if m > 0, 0 if m = 0, -1 if m < 0.

SIN m n 2-13
n = sine of m. (m in radians)

SPCE - - 2-15
Prints a space (ASCII 160) to the current output.

SQRT m n 2-13
n = square root of m. (n is indeterminate if m < 0.)

STACK - - 2-4
Toggles the stack display on or off.

SWAP m n n m 2-10
Swaps the position of the top two stack values.

TAN m n 2-13
n = tangent of m. (m in radians)

TEXT - - 9-2
Resets TransFORTH's internal video and keyboard routines, and displays the normal text screen on the Apple video display. TEXT is used to turn off any of the graphics modes, or a PR#n or IN#n.

Word	Before	After	Page

THEN - - 4-8
Marks the end of an IF - THEN, IF - ELSE - THEN, or CASE: - THEN construct, as the point where execution continues from.

UNTIL n - 4-13
Marks the end of a BEGIN - UNTIL construct. If n = Ø, execution jumps back to the words that follow BEGIN.

VALID - n 6-2Ø
n is nonzero if last GETNUM produced a valid number, otherwise n = Ø. (The TransFORTH system also sets VALID when reading and compiling numbers, and clears it at a "Not Found" error.)

VARIABLE - - 6-1
Declares a variable with following name, creating a new TransFORTH word. Any preceding number is used as the variable's initial value.

VTAB n - 7-1
Sets the text row for subsequent printing to the screen. The row is relative to the top of the screen (not the text window), and the valid range is from Ø to 23.

WHILE n - 4-14
Marks the decision point for a BEGIN - WHILE - REPEAT construct. If n is nonzero, execution continues after WHILE, otherwise execution jumps to word after corresponding REPEAT.

WINDOW L w t b - 7-2
Sets a text window with left margin L, width w, top margin t, and bottom margin b. b actually indicates one line below text window. Cursor jumps to top line of new window.

WRITELN a - 6-17
Writes text from string at address a to current output. (End of string is marked by zero, EOFCHR, or carriage return.)

XOR m n p 4-6
p = 1 if zero/nonzero statuses of m and n are different, Ø if they are the same.

XREG (variable) 7-11
Value of XREG is placed into processor X register before a CALL. After CALL, value of X register is stored back into XREG.

Word	Before	After	Page

YREG (variable) 7-11
Value of YREG is placed into processor Y register before a CALL. After CALL, value of Y register is stored back into YREG.

^ m n p 2-13
p = m ^ n. (m to the nth power)

APPENDIX B: TransFORTH SYSTEM MEMORY MAP

CHAPTER TABLE OF CONTENTS: Page

Memory Map B-1

48K Apples B-2
64K Apples B-2
Apple //e (in addition to above) B-3
Apple //e Auxiliary Memory B-3

TransFORTH Page Zero Map B-3

Useful Locations in Page Zero B-4

TransFORTH System Memory Map

0 to 255	$0000 to $00FF	6502 Page Zero. See Page Zero listing below.
256 to 511	$0100 to $01FF	6502 Stack
512 to 719	$0200 to $02CF	Line Input Buffer
720 to 751	$02D0 to $02EF	Textfile Speedreader (RWTS IOB) Table
752 to 767	$02F0 to $02FF	TransFORTH (Temporary Number Buffer)
768 to 831	$0300 to $033F	TransFORTH Compiler Stack
832 to 975	$0340 to $03CF	PAD String Area
976 to 1023	$03D0 to $03FF	Apple DOS Link Area
1024 to 2047	$0400 to $07FF	Text Display Screen
2048 to 2303	$0800 to $0900	Data Stack
2304 to 2559	$0900 to $0A00	Return Stack
2560 to 2815	$0A00 to $0AFF	Speedreader Data Buffer
2816 to 3071	$0B00 to $0BFF	Speedreader Track/Sector List Buffer
3072 to 12671 (approximate)	$0C00 to $317F	TransFORTH][B as supplied
12672 and up	$3180 and up	Free Memory for program or data
16384 to 24575 (Screen 2)	$4000 to $5FFF	Hi-Res Graphics Screen Memory

48K Apples

Decimal	Hex	Use
24576 to 36863	$6000 to $8FFF	Text Editor File Buffer (when used)
36864 to 39247	$9000 to $994F	Text Editor Program (when used)
37888 to 38423	$9400 to $99FF	GR.TEXT.48K with character set (when used)
38656 to 38423	$9700 to $99FF	Graphics character set (when used)
39424 to 49151	$9A00 to $BFFF	DOS 3.3
49152 to 53247	$C000 to $CFFF	Apple][hardware I/O
53248 to 63487	$D000 to $F7FF	Apple][Basic ROM
63488 to 65535	$F800 to $FFFF	Apple][System Monitor ROM

64K Apples

Decimal	Hex	Use
24576 to 46591	$6000 to $B5FF	Text Editor File Buffer (when used)
46592 to 48975	$B600 to $BF4F	Text Editor Program (when used)
47616 to 49151	$BA00 to $BFFF	GR.TEXT.LC with character set (when used)
48384 to 49151	$BD00 to $BFFF	Graphics character set (when used)
49152 to 53247	$C000 to $CFFF	Apple II hardware I/O
53248 to 53887	$D000 to $D27F	Used by TransFORTH in Apple //e (See below)
53888 to 63487	$D280 to $F7FF	DOS 3.3
63488 to 65535	$F800 to $FFFF	Apple][System Monitor ROM

Apple IIe (in addition to above)

Decimal	Hex	Use
53248 to 53759	$D000 to $D1FF	Auxiliary memory module (when used)
53760 to 53887	$D200 to $D27F	Monitor "patch" for graphics

The auxiliary memory module is the set of routines used for accessing auxiliary memory. It is loaded automatically when the file " AUXILIARY" is compiled. This area cannot be overwritten once the auxiliary memory features are added.

The monitor patch is loaded into Apple //e computers whenever the GR.TEXT.64K graphics module is run. Once loaded, this area may not be overwritten unless TransFORTH is rebooted. Appendix C contains more information on this patch.

Apple IIe Auxiliary Memory

Decimal	Hex	Use
0 to 511 words	$0000 to $01FF	Cannot be used with auxiliary words
512 to 1023	$0200 to $03FF	Free for auxiliary use
1024 to 2047	$0400 to $07FF	80-column display area
2048 to 49151	$0800 to $BFFF	Free for auxiliary use
53248 to 65535	$D000 to $FFFF	Cannot be used with auxiliary words

TransFORTH Page Zero Map

Decimal	Hex	Use
0 to 31	$00 to $1F	Used by TransFORTH
32 to 79	$20 to $4F	Apple][Monitor use
80 to 183	$50 to $B7	Not used (some DOS uses)
184 to 255	$B8 to $FF	Used by TransFORTH

Useful Locations in Page Zero

(The values at these locations can be found by PEEKing the decimal address.)

```
26    $1A  variable store-arrow flag
32    $20  monitor text window left margin
33    $21  monitor text window width
34    $22  monitor text window top margin
35    $23  monitor text window bottom margin
36    $24  monitor cursor horizontal position (40-column)
37    $25  monitor cursor vertical position (40-column)
50    $32  monitor inverse/normal text flag
54    $36  monitor character output vector (two bytes)
56    $38  monitor character input vector (two bytes)
188   $BC  hi-res last plotted X position
189   $BD  hi-res last plotted Y position
196   $C4  hi-res current color
205   $CD  memory output vector (two bytes)
210   $D2  auxiliary memory FIRST value
211   $D3  auxiliary memory SECOND value
222   $DE  memory input vector (two bytes)
237   $ED  data stack pointer
238   $EE  return stack pointer
242   $F2  auxiliary memory flag
```

The data stack pointer contains a 0 for an empty stack, and is incremented by 5 for each number on the stack. The return stack pointer contains a 255 for an empty stack, and is decremented by 5 for each number on the return stack.

The memory input and output vectors point to the address of the next character to read or write.

The auxiliary memory flag (used only on an Apple //e with an extended 80-column text card) contains $00 when main memory is accessed, $80 when AUX is set, and $40 when SETAUX is set.

The monitor cursor position values may not be valid when displaying 80 columns. If you're using an Apple //e 80-column text card, the correct horizontal cursor position is stored in location 1403 ($57B) and the vertical position is stored in location 1531 ($5FB).

APPENDIX C: TransFORTH "TECHNICALITIES"

CHAPTER TABLE OF CONTENTS: Page

Errors and Error Handling C-1

Error Trapping C-2
Word Library Structure and Compilation C-3
Memory Usage C-4
Floating Point Format C-5
Monitor Patch for Apple //e Graphics C-5
Magic Tricks with Memory Cards C-6
Recursion C-6

TransFORTH Technicalities

Errors and Error Handling

TransFORTH, like most versions of Forth but unlike many other high-level languages, does not check for valid number ranges as words are executed. For example, arrays subscripts are not checked against the size of the array. Numbers used to select a word with CASE: are not checked against the number of words in the CASE: list. If range checking is needed, it can be written into the program.

TransFORTH does print error messages for a variety of other errors. "R Error" means Runtime error; "C Error" means Compiling error. Here is a list of error messages:

STKU data STacK Underflow
STKO data STacK Overflow
RETU RETurn stack Underflow
RETO RETurn stack Overflow
PRGO PRoGram Overflow - The program being compiled is too large to fit in available memory.
UNEQ UNEQual word balance - This occurs if part of any of the following groups of words are left incomplete (e.g. semicolon without colon, IF without THEN, etc.):

```
IF - THEN
IF - ELSE - THEN
BEGIN - UNTIL
BEGIN - WHILE - REPEAT
DO - LOOP
DO - +LOOP
CASE: - THEN
: - ;
```

DOS Disk Operating System error. The appropriate error message is included.
X/0 Division by zero.
LABL LABeL missing - A word name was needed but not provided.

"<wordname> Not Found" means that a word that does not exist was called or referenced.

"<wordname> Not Unique" means that a new word was defined that has the same name as an existing word. (The new word is compiled anyway. The error is only a warning message.)

Error Trapping

In many cases, you can trap errors yourself, and take appropriate action based on the error. This is most useful for trapping DOS errors (e.g. FILE NOT FOUND because the user inserted the wrong disk.) The technique is a little tricky, and has to be designed into the program at its top level.

Whenever an error occurs, TransFORTH stores an error number into location 3079 ($C07). After the error, you can PEEK this location to read the error number. Here are the numbers and the errors they stand for:

```
 0  Stack overflow
 4  Stack Underflow
 8  Return stack underflow
12  Return stack overflow
16  Program overflow
20  Unequal word balance
24  DOS error
28  Divide by zero
32  Label missing
```

("Not Found" and "Not Unique" errors are not flagged.)

If a DOS error occurred, a 24 is stored into location 3079 and another number representing the DOS error is stored into location 3080 ($C08):

```
2 or 3  Range error
     4  Write protected disk
     5  End of data
     6  File not found
     7  Volume mismatch
     8  I/O error
     9  Disk full
    10  File locked
    11  Syntax error
    12  No buffers available
    13  File type mismatch
```

Remember that if AUTORUN is set, TransFORTH will automatically execute the top word on the word library whenever a TransFORTH

error occurs, a machine language BRK instruction is encountered, or the user presses Reset. Suppose your program runs with AUTORUN set. Begin by POKEing a 255 (which is not a valid error number) into location 3079. If the top library word begins running later, an error may have just occurred. Have the top word PEEK the value back from location 3079. If it is now a valid error number, then an error just occurred. You can use the error number to decide what action to take, if any. You should also reset location 3079 back to 255 to trap possible future errors.

If you want, you can also suppress TransFORTH's usual error messages so that only your error routines are used. The value in location 3078 ($C06) is usually an even number. To suppress TransFORTH's error messages, add 1 to this value to make it an odd number:

```
Ready 3078 PEEK 1 + 3078 POKE
```

If you want to return error handling to normal, subtract 1 to make the number even again. (Do not change this number in any other way. Other bits are used by TransFORTH for internal flags.)

Word Library Structure and Compilation

Each word in the word library consists of three parts:

1. A two-byte "pointer location" containing the address of the next lower word in the word library.

2. The word name (ASCII characters with high bit set).

3. The executable machine language code for the word. The first byte of this part should be less than 128 (high bit clear), or it will be interpreted as another character in the word name. This byte should also not equal 10 (hex $0A), as this is used as a special compile-time flag.

The hexadecimal numbers displayed by $LIST are the addresses of the pointer locations. A number returned by tic (') is the address of the executable portion of the word.

Program lines entered are compiled directly into 6502 machine language in the memory immediately above the current top of the word library. If the line is an "immediate" command, and not part of a word definition, the machine language code is executed,

then promptly forgotten. (The next line compiled will overwrite it.) If the line is part of a word definition, the code produced is saved, not executed, and the word library "boundary" expands.

During compilation, TransFORTH separates the input line by spaces into individual word names, then searches through the library for each word. For each word search, TransFORTH first reads the current value of HERE to find the top of the word library. At the top of the library is a two-byte pointer containing the starting address of the top word in the library. Beginning with this first word, the system follows the pointers from word to word down through the library. At each word, the word name and the input word are compared to see if this is the word being searched for.

If the word name is found, the first byte of executable code is examined. If this byte is a 10 (hex $0A), then the following compiler code is executed immediately (during compilation). If the byte is not a 10, TransFORTH places a 3-byte JSR (Jump-to-SubRoutine) instruction to this word into the code being compiled at the top of the word library. At runtime, the JSR will call this word.

If the word name is not found, the word library search falls through to a routine which attempts to convert the word into a number. If this routine fails, the "Not Found" error is displayed.

Since programs are compiled into machine language, TransFORTH does not use an "address interpreter", unlike most other versions of Forth. To begin execution, TransFORTH simply calls the machine code. An RTS (ReTurn-from-Subroutine) instruction placed at the end of the code returns execution to the system level.

Memory Usage

All TransFORTH words use:

2 bytes for the library link pointer
1 byte for each character in the word name

In addition, colon definitions use:

3 bytes (usually) for each word called
8 bytes for each number compiled
1 byte (an RTS instruction) to mark the end of the word

Variables use:

3 bytes for the variable handler call
5 bytes for the variable value

Numbers in the code use:

3 bytes for number handler call
5 bytes for the number value

Each array contains a 3-byte handler call followed by a table of 2-byte numbers defining the array:

Number of dimensions
Size of first dimension
Size of second dimension
etc.
Length of each element (in bytes)
Pointer to beginning of array data

The number of bytes the array data itself uses can be found by multiplying the sizes of each dimension (plus 1 to include zero elements) together, then multiplying by the number of bytes per element. With MARRAYs, the array data is located in a free area of memory. With ARRAYs, the array data immediately follows the above table.

Floating-Point Format

All numbers on the stack and in variables are stored using a 5-byte floating-point format, with a 4-byte mantissa and a 1-byte binary exponent. Relative bytes Ø to 3 make up the mantissa, with the less significant bytes first. Since the high-order bit of any nonzero binary number is always a 1, this bit is stripped off the mantissa. Bit 7 of relative byte 3 is instead used as the sign bit. Relative byte 4 is the two's-complement signed exponent, but its sign bit is reversed (1=positive, Ø=negative).

Monitor Patch for Apple IIe Graphics

When the GR.TEXT.64K graphics module is run on Apple //e computers, a patch is made into the Apple system monitor in bank-switched RAM. The patch area resides at $D2ØØ to $D27F, below DOS. The patch remains active until the Apple is rebooted, and this area of memory may not be overwritten.

The Apple //e includes routines which happen to switch from video display page 2 to page 1 momentarily whenever certain monitor calls are made, including scrolling and clear-to-end-of-line. Unfortunately, this produced a visible and annoying flicker on the screen when using TransFORTH's mixed graphics and text. The monitor patch added to the graphics module eliminates most of this flicker.

(GR.TEXT.64K checks several nearby bytes before making the patch. If Apple ever changes their //e monitor ROM, these bytes will be invalid and the patch will not be made.)

Magic Tricks With Memory Cards

On 64K Apples, TransFORTH uses only bank 2 of the upper 16K memory area. For programmers who are familiar with accessing the bank-switch locations, the 4K area $DØØØ to $DFFF in bank 1 is free for storing MARRAYs or other data. Apple][users who have 32K (or larger) memory cards may also want to access the extra memory these cards provide. The bank-switching addresses are described in your memory card manual and the Apple //e Reference Manual. You can throw the memory switches by PEEKing the appropriate addresses.

Using these high memory areas are complicated by the fact that DOS also resides in high memory and controls all normal character I/O. This means DOS must always be banked in (and the other memory banked out) whenever any character is input or output, and whenever a TransFORTH disk command is run. To be safe, you should bank DOS out only long enough to store or retrieve a data item, then bank DOS back in immediately.

There is another approach you can use if you don't want to bank DOS in and out. Rather than booting the TransFORTH disk (with its special 64K DOS), simply boot a normal 48K DOS disk, type MAXFILES 1, then BRUN the OBJ.FORTH file to start up TransFORTH. With DOS sitting in its usual 48K memory space, TransFORTH will think it is running on a 48K Apple, and the entire memory card space will be free for use. (Be sure to copy the Apple monitor over to the banked memory if you plan to keep this memory active.)

Recursion

Recursion was discussed briefly in Chapter Three, mainly in describing where it is less than appropriate. However, recursion

can be put to good use in pattern matching, back-tracking problems, etc. TransFORTH does not have "local" variables as in Pascal, but if need be, parameters can be pushed onto the stack explicitly before a recursive call, or copied from variables onto the stack.

Detailed recursive programming is beyond the scope of this manual, but the following example should provide a "taste" of how to use recursion:

The factorial of a number n is the product of all of the whole numbers up to n. Factorials can be defined using a recursive definition:

```
Factorial ( 1 ) = 1
If n > 1, Factorial ( n ) = n * Factorial ( n - 1 )
```

The following TransFORTH word uses this definition to compute factorials. The number n should be waiting on the stack when this is run:

```
: FACT
DUP             ( Copy n for following test )
1 > IF          ( If n > 1: )
   DUP          ( Keep n and )
   1 - FACT     ( Find factorial of n - 1 )
   *            ( Multiply them together )
THEN ;
```

Note that if n = 1, none of the word between IF and THEN are executed, and the DUP at the top of the definition simply passes the 1 through as the result:

```
Ready 1 FACT .
1

Ready 4 FACT .
24
```

While factorials provide an excellent example of recursion, "iteration" is more efficient than recursion for finding factorials. The following definition, FACT1, places a 1 on the stack, then multiplies it by each whole number in a loop. The number left is the product of all the whole numbers up to n:


```
: FACT1
1                 ( Place 1 on stack as "starting" product )
SWAP 1 + 1 DO     ( Loop from 1 up to (and including) n )
   I *            ( Multiply current product by loop value )
LOOP ;
```

The textfile "HILBERT" on the TransFORTH system disk is a graphics program that plots "Hilbert curves". The program uses two recursive word definitions, one nested inside the other, that call each other. HILBERT provides a good example of more complicated recursive programming. "GROWTREE" is another program that uses nested word definitions for recursion.

APPENDIX D: TransFORTH FILES

TransFORTH Files

There are a number of demonstration and utility programs stored in textfiles on the TransFORTH system disk. To compile one of these files, simply type:

```
Ready DISK> " <filename> " INPUT
```

substituting the name of the textfile for <filename>. Most programs can then be started by typing "RUN". Some of the graphics programs use identical word names, so it's usually a good idea to FORGET the words from one program before compiling the next, to prevent "Not Unique" errors from occurring. (Remember that NOTE is the top TransFORTH system word. FORGETting the word above NOTE will "clean" all additional words from the word library.)

Here is a list of the demonstration and utility files:

"LISSAJOUS" is a short graphics routine that draws the Lissajous pattern that is also shown at the beginning of the demonstration program. When the program has finished drawing the pattern, press any key to return to the text screen.

"HILBERT" draws graphics patterns known as "Hilbert space curves", using Turtlegraphics along with some rather sophisticated recursive techniques.

"GROWTREE" draws a different recursive graphics pattern that looks like branches growing from a "two-ended" tree.

"FILE.DEMO" is a program demonstrating the use of random access and sequential data files. Some of the routines this program uses are nearly identical to the word definitions that were shown as examples in Chapter Eight.

The program first asks whether you want to read a random access or sequential file. It then asks if you want to re-stuff the data file. If this is the first time you've run the program, type "Y". The data file will be created and filled with record numbers. Then enter the number of the record you wish to read back. (Notice that records in random files can be accessed more quickly than those in sequential files.) When prompted, press Return to repeat, or type "Q" to quit.

"PRIMES" is the prime number benchmark program discussed in Chapter One. Running the top word PRIMES will generate 1899 prime numbers in 94 seconds. After PRIMES has been run, you can see the list of primes generated by typing "SEE.PRIMES".

"FACTOR" is a short word definition for factoring positive numbers. FACTOR removes a number from the stack, then prints that number's factors. If the number is prime, nothing is printed.

"BASCON" allows you to convert a number from any base (from 2 to about 36) to base 2, 8, 10, and 16. Letters of the alphabet are used for digits greater than 9, just as in usual computerese hexadecimal notation. The program displays appropriate instructions.

"BUBBLE" is a program that can sort up to 200 strings into alphabetical order. (Each string must be less than 40 characters long.) To begin, the strings should be stored in a sequential textfile named "TEST". The word STUFF reads the strings from the file into the program's array. Typing SORT then sorts the array into alphabetical order. You can also type SHOW to print the strings on the screen, or WRITE to write the strings back to the file "TEST". With just a few modifications, this program can be used with different filenames and different string sizes.

"DEMO" is the source text of the TransFORTH demonstration program. This file is compiled and run when you select to see the demo. The listing also provides an excellent example in writing larger programs in TransFORTH. You can compile the file onto the word library and run it, or print the listing to a printer. (Note that the program is broken into several segments, each segment usually one "screen" of information. The top word, RUNDEMO, sets the appropriate display mode and calls each segment in turn.)

The "UTILITIES" File

The textfile "UTILITIES" contains a number of useful word definitions that act alone, rather than as part of a larger "program". You may want to use some of these definitions in your programs. You can either compile the entire "UTILITIES" file and call the words you need, or copy the desired words directly into your program.

GR.MOD (which stands for GRaphics.MODule) can be used from your TransFORTH program to activate the mixed text and graphics

feature at any time. It first checks what size Apple is being used (48 or 64K), then determines whether or not the appropriate graphics-and-text module for the system has been loaded. If it has, the word simply calls the module to turn on the mixed text and graphics. If not, it loads it from disk and runs it.

LOAD.CHRSET is used to load another character set into memory for use with mixed text and graphics. First store the character set's filename into PAD, then call LOAD.CHRSET. It will automatically load the character set into the appropriate memory area. Any character printing on the graphics screen will then use the new character set.

SEE.STACK displays the contents of the data and return stacks, just like the usual TransFORTH stack display. However, SEE.STACK can be called from within running programs, inside DO - LOOPs, etc. This word can often be a great help in debugging more complicated programs.

DATA.ITEMS determines how many numbers are currently on the data stack, then places this value on the stack.

RETURN.ITEMS determines how many numbers are currently on the return stack, then places this value on the data stack.

RETURN.PICK behaves like the TransFORTH word PICK, except that it picks numbers from the return stack. It removes a number n from the data stack, finds the nth value on the return stack, and copies this value to the data stack. 1 RETURN.PICK is equivalent to the word I, 3 RETURN.PICK to J, and 5 RETURN.PICK to K. If four or more DO - LOOPs are nested, the fourth loop value out can be retrieved with 7 RETURN.PICK.

REPLICATE can be used to fill an array with any numeric or string value. You can also use a DO - LOOP to store the value into each element of the array, but REPLICATE will fill the array much more quickly. REPLICATE uses this form:

<1st element address> <# of elements> <element size> REPLICATE

First store the desired value into the first element of the array, then place on the stack 1) the address of the first element, 2) the total number of elements, and 3) the number of bytes per element. Then call REPLICATE. Every element of the array will be filled with the value of the first element.

GETKEY reads the Apple keyboard directly, without waiting or displaying a flashing cursor. GETKEY is described in Chapter Six.

CLRKEY clears the Apple keyboard to accept another keypress with GETKEY. It is also described in Chapter Six.

EMIT removes an ASCII value from the stack, and prints the equivalent character directly to the Apple screen, even if a DEVICE or MEMORY OUTPUT has been selected. This can be useful for displaying screen messages without having to first CLOSE an open output. (EMIT will not work correctly with DISK OUTPUT, or when a peripheral card that affects Apple monitor locations is being used for output.)

LEFT$ works much like Applesoft's "LEFT$" command. It removes three numbers from the stack:

<source addr> <destination addr> <number of characters> LEFT$

then copies the given number of characters from the source address to the destination address. (It ends the destination string by writing a zero End-Of-String marker.)

RIGHT$ is like Applesoft's "RIGHT$" command. The form is the same as for LEFT$, but it checks the length of the source string then copies the characters from the right end of the string. (Note: RIGHT$ calls LEFT$, so LEFT$ must also be compiled.)

READ.PADDLE is the same definition for reading the game paddle that is described in Chapter Seven.

READ.BUTTON is the same definition for reading the status of the paddle buttons that is described in Chapter Seven.

SLOT converts an Apple slot number to a TransFORTH address, which can then be used with DEVICE INPUT or OUTPUT. SLOT is discussed in Chapter Eight.

DOS prompts you to enter a DOS command, then executes the command. Adding the word DOS permanently to the word library can save you the time and bother of having to type the CR 132 PUTC PRINT etc. sequence every time you want to catalog the disk or delete a file.

P is simply an abbreviation word for PROGRAM MEMORY INPUT, and is also a handy permanent addition to the word library.

D (for Disk) prompts you for a filename, then reads and compiles that file with DISK INPUT.

NO.80COL actually modifies the TransFORTH system so that it will no longer recognize the presence of an 80-column card. This is useful when you want to create or run graphics programs and don't want to remove the card. You can create a version of TransFORTH that never checks for the 80-column card. Simply run NO.80COL, then immediately call SAVEPRG to save this non-80-column system to disk.

Note: If the 80-column card is active when you run NO.80COL (as it probably will be), then the next time TransFORTH resets its video (with TEXT, ABORT, or pressing Reset), characters will begin printing to the invisible 40-column screen rather than the 80-column card. The system will appear to hang. There are two solutions: Either 1) type whatever character sequence the card requires to turn itself off before TransFORTH resets I/O, or 2) simply press Reset. The system will then recover properly in 40-column mode.

YES.80COL modifies the TransFORTH system to restore 80-column card recognition again. The card will take over the display the next time video is reset.

The next two words can be used with Apple][or Apple][Plus computers that have had lower case display added. Since TransFORTH normally converts Apple][40-column text to upper case only, these lower case capabilities are wasted unless a change is made to the TransFORTH system.

NO.CASECONVERT modifies TransFORTH so that it does not automatically convert 40-column text to upper case. This modified system can then be saved permanently with SAVEPRG.

YES.CASECONVERT changes TransFORTH back to normal.

APPENDIX E: DIFFERENCES BETWEEN TransFORTH] [AND TransFORTH] [B

Differences Between TransFORTH and TransFORTH B

When a variable is called, it no longer returns the address of the variable. Now, if the word "->" precedes the variable reference, the top stack value is removed and stored into the variable. If the variable is referenced without the "->", then the value of the variable is placed on the stack.

```
Ready 5 -> X    ( 5 is stored into variable X. )

Ready X .       ( The value of X is retrieved. )
5
```

Since variables now return values directly, the words CONSTANT and ? have been removed.

The store and retrieve words have new names.

Old name	New name
B!	POKE
W!	POKEW
!	POKEN
B@	PEEK
W@	PEEKW
@	PEEKN

Applesoft Basic is no longer needed for high-resolution graphics. The following graphics words have been added:

GR
TEXT
PLOT
LINE
FILL
COLOR
ORMODE
EXMODE

All white dots are plotted two pixels wide to form true white on the Apple high-resolution screen. The point (0,0) is now in the upper left corner of the screen, like Applesoft but unlike the old TransFORTH. (You can write word definitions to turn it

reverse it if you like. Subtract the Y-coordinate from 191.)

To print text on the graphics screen, the graphics modules GR.TEXT.48K and GR.TEXT.64K are used. (The appropriate module can be used with Turtlegraphics, but is not loaded automatically by the Turtlegraphics package.) Several character sets are included on the TransFORTH disk.

Other words added are:

INVERSE
NORMAL
AUTORUN
PREG

SAVEPRG now uses a single "Autorun" flag for creating turnkey programs. This flag can also be turned on and off from a running program with the word AUTORUN.

The word BYE now exits to the monitor, rather than Basic.

AREG, XREG, YREG, and PREG are now variables. Their values can be set before a CALL like a normal variable. (PREG keeps the processor status register.)

More 80-column display cards are recognized, including the Videx Videoterm.

DEVICE I/O will now work correctly with any standard Apple peripheral card.

TransFORTH will print both upper and lower case to 80-column displays and the Apple //e 40-column screen, and will convert to upper case for Apple]['s.

Automatic 80-column card recognition can be turned off for graphics programs, using a word in the "UTILITIES" file on disk.

DISK OUTPUT can now be ECHOed to the screen correctly.

If a DOS error occurs, TransFORTH will now print what kind of DOS error it was.

0 SQRT used to return a very small number. It now returns zero.

Negative numbers to odd powers now correctly return negative numbers.

The rolling directions for ROLU and ROLD have been reversed, to reflect the usual concepts about "top of stack".

Errors no longer clear the 80-column screen. Except in a couple of unavoidable cases, error messages are always written to the screen rather than the current output.

Comments can extend over several lines with only one pair of opening and closing parentheses.

If the text editor has not been loaded, PROGRAM will return the usual default editor buffer address. It used to return 0 unless the editor was loaded.

If a file is being compiled in Autorun mode, nothing will be executed until the entire file is read in. (It used to incorrectly execute each word as it was compiled.)

Negative numbers can be poked as single or double bytes into memory with POKE or POKEW. (Before, the absolute value of the number was poked.) However, PEEK and PEEKW always return a positive number. If a negative number is poked, then peeked back, TransFORTH returns the corresponding positive "two's complement" number. For PEEK, this equals the negative number + 256; for PEEKW, this equals the negative number + 65536.

Negative numbers can also now be used as memory addresses (e.g. -151 CALL).

All forward references have been removed from the built-in TransFORTH word library. If any built-in words at the top of the library are not needed, they can be safely removed with FORGET.

A small bug when COMPAREing 2 identical strings when EOFCHR is nonzero has been fixed.

With an Apple //e with the extended 80-column text card, TransFORTH can use up to 46K of auxiliary memory for data storage. The auxiliary features are added by compiling the file "AUXILIARY".

The TransFORTH text editor has a new command, Write, which works like List, but does not pause every 16 lines. The Write command is convenient when printing out text files from the editor.

If the editor program position is set to overwrite all of TransFORTH, the editor will not automatically reload TransFORTH from disk at the end of the edit.

The text editor used to occasionally hang up when its buffer memory was completely filled with text. It now works correctly.

Lowercase letters can be used for all text editor commands now, including Autonumbering.

Indented lines in the text editor (with spaclKYLLP]Z ginning of a line) are now spaced correctly.

APPENDIX F: TABLE OF ASCII CHARACTERS

Table of ASCII Characters

Set High Bit		Clear		
DEC	HEX	DEC	HEX	CHAR
128	80	0	00	ConTRoL-@
129	81	1	01	ConTRoL-A
130	82	2	02	ConTRoL-B
131	83	3	03	ConTRoL-C
132	84	4	04	ConTRoL-D
133	85	5	05	ConTRoL-E
134	86	6	06	ConTRoL-F
135	87	7	07	ConTRoL-G (Bell)
136	88	8	08	ConTRoL-H (Left Arrow)
137	89	9	09	ConTRoL-I
138	8A	10	0A	ConTRoL-J (Down Arrow)
139	8B	11	0B	ConTRoL-K (Up Arrow)
140	8C	12	0C	ConTRoL-L
141	8D	13	0D	ConTRoL-M (Return)
142	8E	14	0E	ConTRoL-N
143	8F	15	0F	ConTRoL-O
144	90	16	10	ConTRoL-P
145	91	17	11	ConTRoL-Q
146	92	18	12	ConTRoL-R
147	93	19	13	ConTRoL-S
148	94	20	14	ConTRoL-T
149	95	21	15	ConTRoL-U (Right Arrow)
150	96	22	16	ConTRoL-V
151	97	23	17	ConTRoL-W
152	98	24	18	ConTRoL-X
153	99	25	19	ConTRoL-Y
154	9A	26	1A	ConTRoL-Z
155	9B	27	1B	ESCape
156	9C	28	1C	ConTRoL-\
157	9D	29	1D	ConTRoL-]
158	9E	30	1E	ConTRoL-^
159	9F	31	1F	ConTRoL-_
160	A0	32	20	SPACE
161	A1	33	21	!
162	A2	34	22	"
163	A3	35	23	#
164	A4	36	24	$
165	A5	37	25	%
166	A6	38	26	&

Set High Bit		Clear		
DEC	HEX	DEC	HEX	CHAR
167	A7	39	27	'
168	A8	40	28	(
169	A9	41	29	)
170	AA	42	2A	*
171	AB	43	2B	+
172	AC	44	2C	,
173	AD	45	2D	-
174	AE	46	2E	.
175	AF	47	2F	/
176	B0	48	30	0
177	B1	49	31	1
178	B2	50	32	2
179	B3	51	33	3
180	B4	52	34	4
181	B5	53	35	5
182	B6	54	36	6
183	B7	55	37	7
184	B8	56	38	8
185	B9	57	39	9
186	BA	58	3A	:
187	BB	59	3B	;
188	BC	60	3C	<
189	BD	61	3D	=
190	BE	62	3E	>
191	BF	63	3F	?
192	C0	64	40	@
193	C1	65	41	A
194	C2	66	42	B
195	C3	67	43	C
196	C4	68	44	D
197	C5	69	45	E
198	C6	70	46	F
199	C7	71	47	G
200	C8	72	48	H
201	C9	73	49	I
202	CA	74	4A	J
203	CB	75	4B	K
204	CC	76	4C	L
205	CD	77	4D	M
206	CE	78	4E	N
207	CF	79	4F	O
208	D0	80	50	P
209	D1	81	51	Q
210	D2	82	52	R

Set High Bit		Clear		
DEC	HEX	DEC	HEX	CHAR
211	D3	83	53	S
212	D4	84	54	T
213	D5	85	55	U
214	D6	86	56	V
215	D7	87	57	W
216	D8	88	58	X
217	D9	89	59	Y
218	DA	90	5A	Z
219	DB	91	5B	[
220	DC	92	5C	\
221	DD	93	5D	]
222	DE	94	5E	^
223	DF	95	5F	
224	E0	96	60	`
225	E1	97	61	a
226	E2	98	62	b
227	E3	99	63	c
228	E4	100	64	d
229	E5	101	65	e
230	E6	102	66	f
231	E7	103	67	g
232	E8	104	68	h
233	E9	105	69	i
234	EA	106	6A	j
235	EB	107	6B	k
236	EC	108	6C	l
237	ED	109	6D	m
238	EE	110	6E	n
239	EF	111	6F	o
240	F0	112	70	p
241	F1	113	71	q
242	F2	114	72	r
243	F3	115	73	s
244	F4	116	74	t
245	F5	117	75	u
246	F6	118	76	v
247	F7	119	77	w
248	F8	120	78	x
249	F9	121	79	y
250	FA	122	7A	z
251	FB	123	7B	{
252	FC	124	7C	|
253	FD	125	7D	}
254	FE	126	7E	~

Set High Bit		Clear		
DEC	HEX	DEC	HEX	CHAR
255	FF	127	7F	DELETE

APPENDIX G: INDEX

TransFORTH Index

"   2-15, 5-11, 6-17, A-1
$ 7-2, 7-3, 2-9, A-1
$LIST   3-12, 9-15, 9-16, A-1
'   7-10, A-1
(5-11, A-2
)   5-11
*   2-7, 2-11, 2-13, A-2
+   2-5, 2-11, 2-13, A-2
+LOOP   4-2, A-2
,   2-9, 7-12, 7-13, A-2
-    2-13, A-2
->   6-2 to 6-4, A-2
.   2-5 to 2-6, 2-11, 7-1, 8-1, A-2
/   2-13, A-2
48K Apples   1-8, 3-12, 5-2
64K Apples   1-8, 3-12, 5-2
80-column cards   2-1 to 2-2, 5-3, 7-8, 10-1, 10-2, D-5
:   3-1, A-2
;   3-1, A-2
<   4-5, A-3
<=   4-5, A-3
<>   4-5, A-3
=   4-5, A-3
>   4-5, A-3
>=   4-5, A-3
?   5-3

A

ABORT   7-4, A-3
ABS   2-13, A-3
accessing array elements   6-8 to 6-10
accessing arrays from loops   6-13 to 6-15
accessing individual characters   6-26, 6-27
AND   4-6, 4-7, A-3
Apple ///   1-8
Apple keys   7-12
AREG   7-11, A-3
ARRAY   6-8, 6-17, 6-21, 9-16, A-3
array error checking   6-11

arrays 6-7 to 6-16, 8-13, 8-14, 9-16, D-3
arrays in auxiliary memory 10-4, 10-5
arrays of strings 6-21, 6-22
array sizes 6-12
ASSIGN> 6-17, 10-2, A-4
ATN 2-14, 7-14, A-4
automatic insertions, editor 5-6, 5-7
Autonum, editor 5-5
AUTORUN 7-4 to 7-7, A-4
AUX 10-2 to 10-5
auxiliary memory 10-1 to 10-9, B-3
AUXMEM 10-2, 10-7, 10-8

B

back-ups 1-12
BEGIN - UNTIL 4-13, 4-14, 4-19, 4-20, A-4, A-13
BEGIN - WHILE - REPEAT 4-14, 4-15, 4-19, A-4, A-11, A-13
BELL 4-17, 8-1, A-4
binary file overlays 8-16, 8-17
branching 4-8 to 4-20
buttons 7-12, D-4
BYE 7-13, A-4

C

C Error 3-8
CALL 7-11, A-4
calling machine language routines 7-11
Cartesian coordinates 9-3, 9-4
CASE: - THEN 4-15 to 4-18, A-4, A-13
character input and output 6-27 to 6-29
characters 6-26 to 6-29
character sets 9-9, 9-10, B-2
clearing arrays 6-11
CLOSE 8-4, 8-15, 8-16, A-4
CLRKEY 6-29, D-4
COLOR 9-5 to 9-7, A-4
colors 7-12, 7-13, 9-5 to 9-7
combined graphics and text 9-2, 9-3
combining text and numerical data 6-29 to 6-31

comments 5-11
COMPARE 6-25, 10-2, A-4
comparing numbers 4-5, 4-6
comparisons with other languages 1-3 to 1-5
compile 1-5, 3-1, 3-8, 5-10
compiling bytes into memory 7-12, 7-13
CONCAT 6-24, 10-2, A-5
ConTRoL-C 2-2
ConTRoL-L 7-2
copies 1-12
COS 2-13, 7-14, A-5
CR 2-15, 2-16, 7-1, 8-1, A-5
cursor movement 5-1

D

data stack 2-4 to 2-8, B-1, D-3
decision and branching words 4-8 to 4-20
defining new words 3-1 to 3-9
Delete, editor 5-6
demonstration program 1-13, 2-1, 7-7, 7-8
DEVICE 8-1, 8-2, 8-9, A-5
DISK 8-1 to 8-4, A-5
DISK> 5-11, 8-1 to 8-4, 8-6 to 8-10, A-5
DO - LOOP 4-1, A-5, A-9
DOS 3.3 1-8, 1-9, 8-4, B-2
DOS commands 5-9, 8-11 to 8-14, 9-2, 9-10, 9-16
double address words 10-6, 10-7
DROP 2-9, 2-11, A-5
dummy words 3-13
DUP 2-10, 2-11, A-5

E

E 2-9, 7-3
ECHO 8-9, 8-10, A-5
EDIT 5-2, 5-3, 8-5, A-5
editor 5-1, B-2
ELSE 4-10 to 4-12, A-6
endless loops 4-19, 4-20
end-of-string marker 6-19
ENG 7-2, 7-3, A-6
EOF 8-6, 8-7, A-6
EOFCHR 8-7, 8-8, A-6
ERASE 6-11, 10-2, A-6
Erase, editor 5-6

errors 1-6, 2-14, 2-15, 6-11, 7-14, C-1 to C-3
ESCape codes 5-1
EXMODE 9-7, 9-8, A-6
EXP 2-14, 7-15, A-6

F

feedback 1-11
FILL 9-5 to 9-8, A-6
FIRST 10-2, 10-6, 10-7, 10-9
FIX 7-2, 7-3, A-6
floating-point format C-5
fonts 9-9, 9-10
FORGET 3-9, 3-10, 5-2, 5-13, 5-14, A-7
Forth 1-4
FRAC 2-14, A-7
frisbee 1-12
functions 2-13, 2-14, 7-14 to 7-18, 9-8, 9-9

G

game paddles 7-11, 7-12, D-4
Get, editor 5-8, 5-9
GETC 6-28, 8-1, A-7
GETKEY 6-29, 9-20, 9-21, D-4
GETNUM 6-19 to 6-21, 10-2, A-7
Gosub 3-3, 4-19
Goto 4-19
GR 9-1 to 9-3, 9-6 to 9-8, A-7
GraFORTH 1-3
graphs 9-8, 9-9
graphics 7-8, 7-12, 7-13, 9-1 to 9-21, D-2, D-3
graphics drawing commands 9-3 to 9-5

H

handling I/O 8-4 to 8-6
hardware 1-7
HERE 3-12, A-7
HEXPRT 7-2, 8-1, A-7
high-resolution graphics 9-1 to 9-10, 10-8, 10-9, B-1
HOME 7-2, 8-1, 9-20, A-7
HTAB 7-1, A-7

I

I 4-1, 4-4, A-7
IF - ELSE - THEN 4-10 to 4-12, A-6, A-8, A-13
IF - THEN 4-8 to 4-10, A-8, A-13

immediate mode 3-1
INPUT 5-10, 5-11, 8-1 to 8-10, 10-2, 10-7, 10-8, A-8
input/output commands 8-1 to 8-10, 10-7, 10-8
Insert, editor 5-7, 5-8
INT 2-14, A-8
INVERSE 7-2, A-8

J

J 4-3, A-8
joystick 7-11, 7-12

K

K 4-3, A-8
keyboard input 6-28, 6-29, D-4

L

LABL error C-1
larger graphics programs 9-15, 9-16
LCOLOR 9-18, 9-19, 9-21
leaving the text editor 5-10
leaving TransFORTH 7-13
LEFT$ 6-26
LENGTH 6-23, 6-24, 10-2, A-8
LGR 9-17
LGRF 9-18, 9-21
LHLINE 9-19, 9-21
LINE 9-4 to 9-8, A-8
Line entries, editor 5-4
LIST 2-2, 2-11, A-8
List, editor 5-4, 5-5
LOG 2-14, 7-15, A-8
LOOP 4-1, A-9
lower case 2-2
low resolution graphics 9-17 to 9-21
LPLOT 9-18, 9-19
LSCRN 9-20
LVLINE 9-19, 9-21

M

machine language routines 7-11
MARRAY 6-12, 10-2, A-9
MAXFILES 8-11
MEMORY 5-10, 8-1 to 8-5, 8-9, 8-10, A-9
memory arrays 6-12
memory cards C-6
memory usage 3-12, C-4, C-5
memory usage, editor 5-16 to

5-18
miscellaneous words 7-8 to 7-13
MOD 2-13, A-9
monitor patch B-3, C-5, C-6
MOVE 9-12, 9-14, 9-15
MOVELN 6-24, 10-2, A-9
MOVETO 9-13, 9-14
MOVFILE 8-8, 8-9, A-9
moving memory 7-10
MOVMEM 7-10, 10-2, A-9

N

NEGATE 2-13, A-9
nested loops 4-2, 4-3
NORMAL 7-2, A-9
NOT 4-6, 4-7, A-9
notes and sound effects 7-8, 7-9
NOTE 7-8, 7-9, A-9
Not Found error 3-11, C-1
Not Unique error 3-11, C-2
number format and storage 6-5
numbers 2-8, 2-9
number to string conversion 8-10

O

OBJ.EDITOR1 or 2 5-2
OBJ.FORTH 2-2
OR 4-6, A-10
ORMODE 9-7, 9-8, A-10
OUTPUT 8-1 to 8-10, 10-2, 10-7, A-10
OVER 2-10, 2-11, 3-6, A-10
overflow 2-14
overlays 8-14 to 8-17

P

PAD 6-22, 6-23, A-10, B-1
paddles 7-11, 7-12, D-4
parentheses 2-6, 5-11
PEEK 6-5 to 6-7, 6-10, 6-26, 10-2, A-10
PEEKN 6-5 to 6-7, 6-10, 10-2, A-10
PEEKW 6-5 to 6-7, 6-10, 10-2, A-10
PENUP 9-11, 9-13, 9-14
PENDOWN 9-11, 9-13, 9-14
PI 2-14, A-10
PICK 2-11, 2-12, A-10
pixels 9-1
PLOT 9-4 to 9-8, A-10

POKE 6-5 to 6-7, 6-10, 6-27, 10-2, A-10
POKEN 6-5 to 6-7, 6-10, 10-2, A-10
POKEW 6-5 to 6-7, 6-9, 10-2, A-11
POP 4-4, 4-5, A-11
Postfix 2-6
PREG 7-11, A-11
PRGO error 3-13, C-1
PRINT 2-15, 7-1, 8-1, A-11
printing files 5-9, 5-10
PROGRAM MEMORY INPUT 5-10, 5-12, 8-2, 8-3, A-8, A-9, A-11
program compilation 5-10
program control words 7-4 to 7-6
program execution 2-3 to 2-4
Program position, editor 5-17
programs 3-10
program structure 4-18, 4-19
PULL 4-4, 4-5, A-11
PUSH 4-4, 4-5, A-11
PUTC 6-27, 7-1, 8-1, A-11

R

R Error 2-15
RAM cards 1-8, C-6
READLN 6-18 to 6-21, 8-1, 10-2, A-11
Ready prompt 2-1
recursion 3-11, 4-19, C-6 to C-8
registration card 1-12
REPEAT 4-14, 4-15, 4-19, A-11
restrictions on disk I/O 8-10
RETO error C-1
RETU error C-1
retrieving word addresses 7-10
return stack 4-4, B-1, D-3
Reverse Polish Notation 2-6
RIGHT$ 6-26
RND 2-14, 7-15, A-11
ROLD 2-11, 2-12, A-12
ROLU 2-12, A-12
RPN 2-6
RUN 7-4, A-12

S

Save, editor 5-8
SAVEPRG 7-6 to 7-8, A-12
Saving the TransFORTH system 7-6

to 7-8
scaling functions and graphs 9-8, 9-9
SCI 7-2, 7-3, A-12
scientific functions 2-13, 2-14, 7-14 to 7-18
screen dumps 9-16, 9-17
SECOND 10-2, 10-6, 10-7, 10-9
SETAUX 10-2, 10-4
SETMAIN 10-2, 10-4
SIGN 2-13, A-12
SIN 2-13, 7-14, A-12
single-address words 10-3, 10-4
SLOT 8-2
SPCE 2-15, 2-16, 7-1, 8-1, A-12
speed of execution 1-5
speedreader 8-4, 8-11, B-1
SQRT 2-13, 3-3 to 3-5, 7-15, A-12
STACK 2-4, A-12
stack display 2-4, 4-4, 4-5
stack overflow 2-14
stack underflow 2-15
stacks 2-4 to 2-8, B-1, D-3
STKO error 2-14, C-1
STKU error 2-15, C-1
storage and retrieval words 6-5 to 6-7
string arrays 6-21, 6-22
string manipulation words 6-23 to 6-26
strings 6-17 to 6-27, D-4
string to number conversion 6-19, 6-20
subroutines 3-10
SWAP 2-10, 2-11, A-12
system requirements 1-7
system string PAD 6-22, 6-23, B-1

T

TAN 2-13, 7-14, A-12
TEXT 9-2, 9-3, 9-20, A-12
text editor 5-1
textfiles 5-2, 5-8 to 5-11, 8-3, 8-4 to 8-13
textfile speedreader 8-4, 8-11, B-1
THEN 4-8 to 4-12, 4-15 to 4-18, A-13
tic 7-10
top of stack 2-5
TransFORTH][1-3
TransFORTH][B 1-3
TURN 9-12, 9-14, 9-15
TURNTO 9-12, 9-14
TURTLE 9-11, 9-14, 9-15
TURTLE.ANG 9-13
TURTLE.TEXT 9-11, 9-14
TURTLE.X 9-13
TURTLE.Y 9-13
Turtlegraphics 9-10 to 9-15

U

underflow 2-15
UNEQ error 3-8, C-1
UNTIL 4-13, 4-14, 4-19, 4-20, A-13
upper case 2-2

V

VALID 6-20, A-13
VARIABLE 6-1 to 6-3, A-13
variables 6-1 to 6-5
VTAB 7-1, 7-2, A-13

W

WHILE 4-14, 4-15, 4-19, A-13
WINDOW 7-2, A-13
word definitions 3-1, 3-2
word library 2-2, C-3, C-4
word references 3-11
words 2-2 to 2-4
WRITELN 6-17 to 6-19, 7-1, 8-1, 10-2, A-13

X

X/0 error C-1
XOR 4-6, A-13
XREG 7-11, A-13

Y

YREG 7-11, A-14

^

^ 2-13, 3-4, A-14