GraFORTH
LANGUAGE MANUAL

Notice

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

Disclaimer of all Warranties And Liabilities

Insoft Company 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 Company, 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 Company, 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.

TABLE OF CONTENTS

CHAPTER FIVE: DELVING DEEPER. . .

CHAPTER SIX: TWO-DIMENSIONAL GRAPHICS

CHAPTER SEVEN: CHARACTER GRAPHICS

CHAPTER EIGHT: THREE-DIMENSIONAL GRAPHICS

CHAPTER NINE: MUSIC

CHAPTER TEN: FINAL WRAP

PART IV: APPENDICES. . .

A: GraFORTH][DICTIONARY Definitions

B: DATA: GraFORTH TECHNICAL DATA

C: FILES: A LISTING OF DISKETTE FILES

D: ASCII CODES

E: INDEX: GraFORTH System Manual

CHAPTER ONE: PREVIEW

CHAPTER TABLE OF CONTENTS:

Introduction to GraFORTH

The Apple computer has some potentially powerful graphics capabilities. One of the most impressive of these is the presence of high-resolution color graphics. While there has been a large number of programs written which use this capability, sometimes in a most dramatic way, and there have been several outstanding graphics utilities written to ease the task of adding Apple Graphics to programs, until now, no computer languages have been specifically created for the purpose of fully exploiting these features. GraFORTH is just such a language.

A Family of Languages

GraFORTH is the latest member of a powerful new "family of languages" developed for Insoft by Paul Lutus. The first of these related languages to be released was TransFORTH. While TransFORTH and GraFORTH are related, each of these languages has different functions and capabilities, and is designed to meet different needs. They are related in the ways members of a family are related - they have the same parentage, that of the FORTH language. In a moment, we'll take a look at that heritage, and discuss the differences between GraFORTH and other FORTH implementations. But first, let's look at the capabilities of GrafORTH you'll very soon be learning!

Features

GraFORTH provides many features not seen before on small computers. The system can draw three-dimensional images, in color, at rates that make animation possible. A sophisticated music synthesizer, a part of the language, allows the addition of music as well as sound to GraFORTH programs. Text display may be in any size, color, or typeface, and mixed with graphics images on any part of the screen. Personalized character fonts may be created, and fonts full of different two-dimensional images may be block printed to any screen location under full program control. Clearly, this is a programming language designed for applications where fast, sophisticated graphics capability is important, such as the development of games and entertainment software.

Comparison with Standard FORTH

The above features are embodied in a very fast, fully compiled version of FORTH. Nearly all other Apple languages (both BASICs, UCSD Apple Pascal, Apple FORTRAN, and most other FORTHs) are interpreted while they are running. This is often done to provide what is called 'code transportability', the ability to take programs from one computer and run them on another with either no or few modifications. Unfortunately, this drastically reduces the speed of your programs. GraFORTH (and TransFORTH) have been designed for the computer you own, the Apple. They have been specifically written to make maximum use of the features built into your machine, and therefore no attempt has been made to create transportable code. By compiling directly to 6502 machine language, speed was greatly increased over nearly every other language - a must for smooth, fast, animation quality graphics. Even though GraFORTH is fully compiled for the purpose of increased speed, commands may still be typed directly at the keyboard, rather like an interpreted language. As implemented, then, GraFORTH has both the speed of a compiled language and the immediate feedback of an interpreted language, the best of both worlds. Finally, GraFORTH, unlike standard implementations of FORTH, uses standard Apple DOS commands and file structures, to retain compatibility with the work you have already done with your computer, and to reduce the time it will take to learn GraFORTH.

If you are already familiar with another version of FORTH, you will find many similarities and many differences between GraFORTH and other FORTH versions, as GraFORTH is only loosely related to these other languages. The general structure of the language has been retained (at least outwardly), but the implementation of that structure is vastly different. These changes have been made for very specific reasons. In short, the intended usage of GraFORTH is very different from that for which FORTH was originally designed. GraFORTH is a computer graphics language, and this in and of itself brought about many changes. Further, it was our intention to make GraFORTH as easy to learn and as similar to existing Apple environments as possible. Therefore, if you already know FORTH, we hope you will bear in mind that this language has been designed for those who do not share your knowledge of FORTH-like environments and who want a fast, easy to learn graphics language. For those of you who do not know FORTH, dive in! You will find GraFORTH to be a powerful, yet intuitive language. Very soon you will be using your Apple to do things you never thought were possible before!

Comparison with TransFORTH

By way of contrast, while GraFORTH is a powerful graphics programming language, restricted to whole number (integer) calculations for the purpose of graphics speed, TransFORTH is a scientific and business oriented language with floating-point arithmetic and a much more extensive operating system. TransFORTH also has two-dimensional line-drawing and TURTLEGRAPHICS capabilities, but no three-dimensional graphics, and character graphics are limited to selection of pre-defined character sets. Thus, TransFORTH has much more calculating ability, but less graphics, while just the opposite is true of GraFORTH.

Program Editing and Storage

Programs, subroutines, or 'words', as they are known in FORTH, can be written in the language editor and stored in text files for later modification or use. Because these files are standard DOS text files, any editor of the user's choosing which creates such files may be used. Because program segments may be saved in this way, the accumulation of proven program modules is encouraged, which in turn encourages the practice of good programming techniques.

Manual Overview

Structure

The text portion of this manual is divided into three parts - an introductory or context-setting section (Chapters 1 and 2), a tutorial-based content section of seven chapters to help you understand and put to use the GraFORTH language system (3 through 9), and a section of appended reference material, including the GraFORTH Word Library Listings, Technical Data, and Index. Throughout these chapters, diagrams are used to support the text. These illustrations and the abundant use of headings should make it possible for you to skim the text, get a sense of the subject matter, find general topic areas in the body of text, and never lose your sense of where you are. The Index should help you find specific topics quickly.

Review of Content

Part II, the content of the manual (that is, that material which is about the language itself) is presented in seven major chapter divisions. Chapter 3 is primarily an introduction to the FORTH language aspects of GraFORTH, including an explanation of the definition of words, stack operation, and control structures. (In addition to being a good introduction to GraFORTH, much of the material covered in this chapter pertains to other FORTHs as well, making it an excellent FORTH overview.) Chapter 4 covers text entry, special characters, and the supplied text editor. It shows how to write and modify GraFORTH programs or "words" and how to compile them into memory from the editor buffer or from disk. Chapter 5 presents extended GraFORTH capabilities and describes how it operates, how it relates to and uses the DOS 3.3 disk operating system, and how its data structures - variables and strings - are created and used. Chapter 6 introduces GraFORTH's two-dimensional graphics capabilities including plotting and line drawing, color selection/filling, and the TURTLEGRAPHICS commands. Chapter 7 describes character graphics, particularly a program called CHAREDITOR, which allows the design of new character fonts and images that can be block printed to the screen. Chapter 8 reveals the GraFORTH 3-D graphics system, including moving and manipulating objects in 3-D space. The program IMAGEDITOR, which allows the creation and modification of 3-D objects, and another, called PROFILE, which speeds up the process for the particular class of objects which rotate or revolve around a central axis, are introduced. Another program, named PLAY, winds up the discussion of 3-D graphics by allowing you to "play" with an object in space, as you will discover in a short exercise at the end of this chapter. Chapter 9 describes how to add music (as opposed to sounds) to your programs, and Chapter 10 concludes Part II with a discussion of marketing software developed using GraFORTH. That's a lot of content, which you surely must be eager to get to, but first perhaps we should talk about the manual for a bit.

How to Use This Manual

Differences of Style

It is important to realize that everyone uses manuals according to his or her own individual learning styles and skill levels. There are those of us who start from the beginning and carefully read every word, and there are others who bound ahead looking for just enough information to "get on with it". Still others like to live on the edge, boot the disk first, and only use the manual if they have to look something up later. Furthermore, even the same reader will have differing moods and levels of interest, and will use a technical manual in different ways at different times according to his or her current understanding of the product.

Tutorial Learning

This manual is set up to be, first of all, a tutorial to guide you gradually through the steps you need to take to learn the GraFORTH language and begin to put it to use. 'Tutorial learning' has become the primary method of microcomputer instruction. Actually, it's a bit of a misnomer. There is really no tutor, unless a technical manual can be considered such. For the most part, it will be just you and the manual and whatever other resources you can pull together. Be advised, however, that there are many differences between GraFORTH and other FORTH implementations. Because of these differences (we think of them as improvements), we advise you, even if you know FORTH already, to read the manual carefully at the beginning.

Later, of course, you will be using the manual more as a reference guide than as a tutorial, and will need to be able to find specific items of information quickly. There is nothing more frustrating than knowing that you saw something someplace, but can't quite remember where. We'll help you find it, after all, you may be living with this manual for a few weeks. In either case, tutorial or reference, we have tried to accommodate all styles of learning.

Reference Aids:

Multiple Tables of Contents

As mentioned above, there are various reference aids which should allow you to find what you want quickly when using the manual as a reference guide. At the beginning of the manual, there is a comprehensive table of contents which presents the major topics of the manual, with page numbers, in the order in which they appear. Each chapter has a similar, but more complete table of contents for that chapter.

The Word Library Definitions List

Appendix A, in the back of the manual, contains an alphabetically arranged list of annotated definitions of all the GraFORTH words which come with the system. Because this is an important source of information about the language to which you will be referring frequently, we placed it first, and have also included an additional cross listing of the words by subject groupings.

Index

In Appendix E, at the end of the manual, there is a comprehensive index which lists the major topics and terms of the manual once again, but this time alphabetically.

Conventions Used

Several standard conventions are used to simplify the descriptions. All commands which you are to type in are printed in upper-case type. All 'system' responses are shown as they appear on the screen. 'Control character' entries are denoted by CONTROL-X, where X would be replaced with the actual character entered. Control character entries are made by holding down the CONTROL key while depressing the indicated key.

Request for Feedback

Let us know what you liked and didn't like about this manual. We have tried to make it as complete and friendly as possible, but we know that something, somewhere may be confusing. Let us know if we omitted a useful tip, or explained something poorly. Also, let us know what worked for you so we can continue to produce high quality manuals for future products.

Start-up Procedures

Product Information Card and Replacement Policy

The warranty of this diskette is covered in general by the statement at the bottom of the warranty and disclaimer page in the front of the manual. Since its message is hidden in legalese, let's just say that roughly what is meant is that we did our best to ship the diskette in perfect condition, but we have no control over what happens to it enroute to your disk drive. If, for some reason, it will not 'boot' (come up on the screen when the machine is turned on), then you should take or send it back to the place where you purchased it. If they cannot get it to boot, then we will replace it at no additional cost to you, for a period of 30 days after you purchased it. (Thereafter, a nominal replacement fee may be charged.) Once you have a disk that boots and runs, then it is your responsibility to protect it by using it only for the purpose of making duplicate work disks and backups (see next section).

In the meantime, we would appreciate it if you would fill out the Product Information Card. This card gives us valuable information about our customers and helps us design our products and product line to better serve you. If everyone who buys GraFORTH turns out to be retired and living in Florida, then this manual will have to be rewritten with a different set of jokes. The card also allows us to keep you up to date. If we decide to send out an updated GraFORTH diskette, then you would probably want to know about that.

Making and Using Backup Copies

If you have not yet made a backup copy of the GraFORTH diskette, then now is a good time to do so. Never use the original as a work disk, not even for a few minutes. Particularly, never use an original disk to try to solve a problem which blew up your work disk. Make a new backup if you can, and use that to experiment. Because GraFORTH is compatible with DOS 3.3, any copy program you normally use to copy your 16-sector Apple DOS disks will work to copy this diskette. The COPYA program which came with your DOS 3.3 System Master diskette is a particularly reliable one, and we recommend using it. In fact, it is recommended that you have two backup copies so that if one goes down, you won't have to open your lead-lined vault to get at the original.

A PLAYful Review

An Introductory Tutorial

We suggest that you study the Table of Contents and the Manual Diagram for a few minutes to get an idea of where we are and where we have to go, and then, because we know you are itching to get your hands back onto that machine and create a few three-dimensional forms to rotate in free-floating and free-wheeling space, we'll give you a preview of what's to come in future chapters...

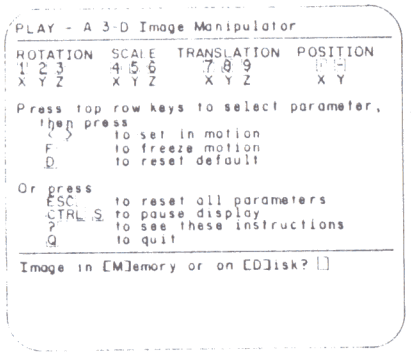
If you catalog your disk, you'll find the text file PLAY on it. PLAY is a set of routines (or "words"), which when compiled and run, allows you to pull up a three-dimensional form off the disk (several are provided), and play with it in 3-D space. Later on, we'll tell you how to use PLAY to understand better the 3-D images you are creating. But for now, we are just going to have some fun using PLAY. If you have not yet made a backup copy of your disk, we'll just have to insist that you do so now. From now on, when we speak of your GraFORTH diskette, we will actually be referring to the copy you use as a work diskette.

Running the PLAY Program

To run PLAY, boot your disk and respond with an 'N' or 'NO' to the demonstration question. When the "Ready" prompt comes on, type

READ " PLAY " <return>

Be sure to type it exactly as you see it, including the spaces between PLAY and the quotation marks. The word READ is a command that tells the GraFORTH system to read a file on the disk and compile it into the word library, that is, turn it into machine language for the machine to use. When the "Ready" prompt reappears, type 'RUN' and a set of instructions will be displayed on the screen, as illustrated in diagram below:



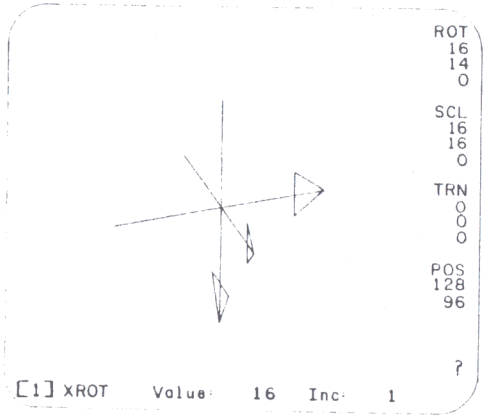
The words, ROT, SCALE, TRANS, and POS refer to the four parameters you may use to manipulate the image in space. ROT stands for the ability to rotate the object around any of three axes, SCALE stands for the ability to change the scale or size of the object, TRANS stands for the ability to translate or move the image in its 'space envelope', and POS stands for the ability to move the position of the image on the screen.

The characters, 123 456 789 :-, are pressed to activate any of the above parameters along any of the axes, X, Y, and Z, indicated below them. The commands in the middle of the screen start and stop the selected action, or reset the parameters to their starting positions (called 'defaults'). You are to press just those keys which are high-lighted in inverse. If the action ever gets too fast for you or you see something you'd like to study, pressing ConTRoL-S will stop the action until you press another key. Similarly, 'D' will reset the currently active parameter to its default position, <ESC> will put you back at the beginning, and 'Q' will put you out in the cold at the "Ready" prompt.

At the bottom of the screen, you are being asked to answer a question as to where the image is which you would like manipulated. The quickest way to understand the program is to dive in and try it, pressing the various keys along the way to see their effects. But first, we need an image to play with.

Unless you're way ahead of us, you do not have a 3-D image in memory yet, so select 'D' to answer the question at the bottom of the screen and to begin the image loading process. Next, hit <return> to default the address to 2816 (more on that later), and enter 'XYZ' as the image filename. Again, hit <return>. Your screen should now show a picture of a vertical line crossed by a horizontal arrow. In a moment you'll see that these are really three intersecting arrows. On the right side of the screen are the movement commands, ROT, SCALE, TRANS, and POS. Ignore the latter two for the purposes of this short trial run. Now the fun begins. Press '2', and then the right arrow key. Next press '1', then the right arrow key. Observe the numbers changing over on the right. See if you can figure out what they do as you select keys to press from the previous diagram. Try the left arrow keys, and watch the action and the numbers change. You may freeze the selection last changed with 'F', and also by using the arrow keys to get the parameters back to zero.

At this point, you should have a screen which looks something like the one on the next page.



Now press 'D' to reset your current parameter to its default; get the idea? The more you press the arrow keys, the faster the image will turn. If you are working on a color screen, you will see that each axis is a different color, which may help to keep them straight. Remember, pressing <esc> will set all parameters to their starting (default) positions, which may be needed if they start getting out of hand. In particular, if SCALE, TRANS, and POS get beyond a certain size, they will no longer fit on the screen, and they will begin to "wrap around", appearing quite unexpectedly on the opposite side of the screen. It will look as if you have lines bouncing off the walls, but it is really only wraparound. If you like that effect, then fine; but if not, just keep the numbers smaller.

That's enough fun. We have to get back to work and learn the rest of what GraFORTH has to offer. We'll come back to PLAY in Chapter 8, and learn what TRANS and POS actually do. But if you just can't quite quit yet, we'll mention (while the boss is out of the room) that the way to bring up another 3-D image to PLAY with is to type 'Q' and then RUN again, repeating all steps except the one where you enter the filename (try HOUSE).

CHAPTER TWO: BACKGROUND

CHAPTER TABLE OF CONTENTS:

What You'll Need to Have

Hardware Requirements

GraFORTH requires that you have the following minimum hardware components:

An Apple or Apple + computer with 48K RAM
One DOS 3.3 Apple disk drive with controller
A black and white (or green) video monitor, and/or
A color monitor or color TV with an RF modulator

Recommended Peripheral Options

In addition to the above (including the color display), it is highly recommended that you have a 16K RAM or language card, to provide more available memory, and a second disk drive.

Software Requirements

GraFORTH is written in 6502 machine language using the ALD SYSTEM Assembler which was written by Paul Lutus and is also available from Insoft. All graphics are internal and are therefore completely independent of either Apple BASIC (INTEGER or APPLESOFT). GraFORTH boots from the 'monitor', without a BASIC 'HELLO' program, as you will notice by the presence of 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 Apple II and Apple II+ machines which are sometimes troublesome to software. It also means, however, that it is not possible to add your own special HELLO program to the disk to have it do your favorite tricks on bootup. But don't despair; we will show you later how to have GraFORTH automatically run any program you wish on bootup. Further, that program can be written directly in GraFORTH.

What You'll Need to Know

What You'll Need to Know about Your Machine

While it is intended that this manual serve as a tutorial in the use of the GraFORTH language, it is not intended to cover material already covered quite thoroughly and thoughtfully in the set of manuals distributed by the Apple Computer Company. If you are a new user, unfamiliar with how to use your Apple computer, we suggest that you take the time to go through the Apple Reference Manual, which came with your machine. You will not need to know everything in it to use your Apple successfully, but the more you know, the easier it will be to understand operations which might otherwise seem puzzling.

What You'll Need to Know about the DOS

With the exception of certain small changes (see below), GraFORTH uses the standard Apple Disk Operating System, Version 3.3, known affectionately as DOS 3.3. If you are at all unfamiliar with how to use your disk operating system, we suggest you take the time now to study the DOS Manual which came with your disk drive(s). It will be time well spent.

Minor Modifications In DOS 3.3

Minor modifications have been made in the disk operating system to make it run smoothly with GraFORTH. Most of these changes will be 'user-transparent', or not noticeable, and using DOS from GraFORTH is the same as using DOS from either of Apple's BASICs. Both create TEXT type data files, and GraFORTH even uses TEXT files for saving program 'source code'. The DOS on the supplied diskette has been modified, however, to take advantage of an existing language card or RAM card. If you have such a card, DOS will be loaded automatically into the language card, leaving much more room (almost 10K) in main memory for program development. To take advantage of this additional memory, two editors have been provided on the disk; OBJ.EDITOR1 for systems without language cards and OBJ.EDITOR2 for systems with language cards. Note that GraFORTH requires the DOS it is supplied with. You can not transfer GraFORTH to a disk with a different DOS!

Making Space on the Disk

The GraFORTH diskette, as delivered, is nearly full. Not only does the disk contain all the system files needed to use GraFORTH, it also contains many demonstration files as well as some specialty files. After you have copied the diskette and exhausted your interest in the demos, you may want to trim your work disk down a bit to make room for your own files. The demonstration programs will probably be the first to go. Appendix C lists the files on the disk, indicating those which may be deleted without danger to the GraFORTH system by a ">". See the sections which follow for help on how to delete files from your work disk.

Alternatively, you might want to leave your work disk intact and set up another disk for program development. The GraFORTH system would not need to be on such a disk; you could use, instead, a standard DOS diskette. If so, you will need to copy the editor file or files onto that disk as the GraFORTH word, EDIT, looks for the editor program on the 'current drive'. If you are using a language card, copy OBJ.EDITOR2 onto your program development disk, otherwise copy OBJ.EDITOR1 onto that disk.

Deleting Files

There are three simple ways to delete files from the disk. One way is to boot an Applesoft disk, then catalog the GraFORTH diskette and delete the files you want to remove as you would on a standard DOS disk. Alternatively, you could use your favorite file utility, such as FID on your DOS 3.3 System Master Disk, or else boot GraFORTH and enter your DOS commands from the program itself. If you are already in GraFORTH, the latter method is the method of choice. To delete files directly from the program, you will need to take the following steps:

1. Boot GraFORTH and you will see the prompt

   Demonstration (Y/N)?

2. Answer 'N' and the "Ready" prompt will appear.

3. Respond with:

   EDIT <return>

The drive will whirl a bit, loading the editor, and then the editing title will appear along with a flashing cursor.

4. To enter a DOS command, type:

   ConTRoL-D <return>

and the following prompt will appear:

   Enter DOS Command :

5. Respond with:

   CATALOG <return>

   (or CATALOG,D1 <return> for two drive systems)

and the catalog will be listed.

6. Select the files to be deleted and type:

   DELETE filename <return>

The drive will run briefly, make its usual scratching sounds and the file's name will be deleted from the disk directory. You may confirm that fact with another CATALOG command. Then repeat the procedure to delete the other files you wish to remove from the disk. To return to the editor, press the <return> key twice without entering any DOS commands, and you will see the blinking cursor of the editor once again. To return back to GraFORTH, type 'BYE', then press <return>. The GraFORTH header and the "Ready" prompt should reappear.

Entering Other DOS Commands

The above steps represent the procedure to be followed to enter any standard DOS 3.3 command from GraFORTH itself. Later on, we'll describe another method which enables you to use DOS commands from the "Ready" prompt directly without entering the editor.

What You'll Need to Know about Disk Care

We assume that by now you have made a copy of the original diskette, have stored it in some safe place, have had some fun with PLAY and are anxious to get down to "work". Bear with us for one more cautionary remark (admittedly unnecessary for almost all of you). In case you are not familiar with the care and feeding of floppy diskettes, what we mean by "safe place" is that the disk is stored vertically, is not bent or folded or exposed to magnetic fields or to temperatures outside of the range 50 to 125 degrees F., and that the "naked" portion of the disk (as seen through the small oval opening in the plastic covering) is not exposed to dust, fingerprints, or cigarette ashes. We recommend that you always keep your disk in its protective sleeve and box whenever it is not actually in a disk drive. Never attempt to write on it with a pencil or ball-point pen. If treated in this way, your diskettes should give you years of devoted service, and perhaps even become collector's items of considerable value to your grandchildren (well, at least curiosities).

What You'll Need to Know about Programming

It is not necessary to know how to program to learn programming in GraFORTH. It is our position that both TransFORTH and GraFORTH are simple enough to learn that novices can take them on as beginning languages. We also believe that they are so powerful that advanced programmers can use them in a full range of commercial applications. While it is not necessary to learn programming prior to starting in on GraFORTH, if you are already familiar with BASIC or another high-level language, you will, of course, learn GraFORTH much faster. In particular, a familiarity with Applesoft and/or Apple Pascal will speed the learning of the control structures, data structures, and the file handling portions of the language. Familiarity with FORTH will give you a head start on the operation of the stacks, postfix notation, and the word library.

What You'll Need to Know about Graphics

Here again, prior experience in graphics programming is helpful to learn programming in GraFORTH, but it is not required. Graphics is the heart of GraFORTH - all kinds of graphics - standard two-dimensional graphics, TURTLEGRAPHICS, color graphics, block printing of image fonts, three-dimensional graphics, all at speeds which will support animation, and set to music if you like. If you do not intend to do a lot of graphics programming with GraFORTH, then you may have the wrong language. (Perhaps you really need TransFORTH...)

With GraFORTH, powerful graphics editors allow your images to be created with considerable ease. A powerful command set allows them to be put in motion. Routines can be set up as independent words, then tested out and stored, to be used again and again. But you do not need to know it all before you start. We'll take you through it a step at a time.

However, if you are a beginner at graphics, you will learn faster if you draw upon several sources at once and approach the subject from all sides. The Applesoft Tutorial has a good introduction to Apple graphics, as does the Apple User's Guide by Lon Poole, et al. The Apple Pascal Language Reference Manual has a good chapter on TURTLEGRAPHICS, and if you really want to get into the whole subject, try Graphic Software for Microcomputers by B. J. Korites (Kern Publications, 1981).

What You'll Need to Know about Music

As mentioned above, one of the features of GraFORTH is a music synthesizer which enables you to add music to the programs you write in the language. Operation is straightforward, and a note table is provided to make use of the music synthesizer as simple as possible. We think you will be amazed at the added dimension it will give to your programs.

What You'll Need to Do

Get an Overview

One of the most time-saving things you can do right now is to get an overview of the manual and the structure of GraFORTH. Time spent on the demos, and studying the table of contents and diagrams will give you a general framework which then just needs to be filled in with detail.

In between this chapter and Chapter 9 are the chapters which explain in detail how to use GraFORTH. Chapter 3 gives an introduction to the use of GraFORTH. It is something of a mini-manual in itself, and even those of you who know FORTH may find it a useful review of how GraFORTH differs from other FORTH languages. The next six chapters build somewhat on one another and should be taken in order, with the possible exception of Chapter 9 on music, which could be read and used anytime after Chapter 5.

Run the Demos

The set of demo programs on the diskette will give you a good sense of what GraFORTH can do. To run a demo, just answer 'Y' to the demo question which appears after bootup, and then simply select from the menus which follow. Later we shall tell you how to remove the demo question.

Plunge In

At this point, there is very little left to do but to load your work copy of GraFORTH in the drive, boot it up, and plunge in. Start at a place in the manual appropriate for your skills and knowledge, read that section, turn to the program, work the examples, and then see if you can amaze yourself with a few examples of your own. That's all there is to it. Remember, the chapters, like the language, tend to build sequentially, so it may not be wise to skip around too much.

What You'll Need to Be

Confident, fearless, and fun-loving. Willing to take risks, make mistakes, and learn from those mistakes. Willing to ask stupid questions and make a fool of yourself to find out what you need to know. Willing to let yourself enjoy life and turn work into play. In short, just your average, run-of-the-mill, Apple owner.

CHAPTER THREE: STARTING GraFORTH

Chapter Table of Contents:

Purpose and Overview

As you'll soon see, GraFORTH is a complete, structured language, with all of the interesting nuances of such a language. In this chapter, we'll introduce GraFORTH as a language. We'll discuss the GraFORTH system, the word library (sometimes called the dictionary), and the concept of 'words'. We'll show you how to use the stack to do arithmetic using Reverse Polish Notation, and then define your own words in terms of existing ones. We'll discuss the looping and control features of GraFORTH, then tie up the chapter with some rules of thumb for writing programs in GraFORTH.

This chapter (as well as the others) contains numerous examples to help you understand the GraFORTH system. We strongly encourage you to try these examples on your computer. And as you gain experience with the concepts, we encourage you to experiment further, so that you become truly comfortable working with GraFORTH.

First Things First

Insert your GraFORTH disk in the drive and boot it. After a few seconds you'll see:

```
GraFORTH ][ (C) P. Lutus 1981

Demonstration (Y/N) ?
```

If you haven't yet seen the GraFORTH system demonstration, you might want to do that now. The demonstration includes explanations of what GraFORTH is and what it does. As we go on, however, we'll ignore this question, assuming that you've either already seen the demo or are no longer interested. Later, we'll show you how to remove the demo question entirely... Now let's get into the language. Type an 'N' to the demonstration prompt, and you will see:

```
GraFORTH ][ (C) P. Lutus 1981

Ready
```


The word "Ready" appears whenever the system is ready for your input. (Makes sense...) If at any time you do not see the word "Ready" when you think you're supposed to, then it may be time to start wondering... With the word "Ready" beckoning you on, let's back up for a few moments to discuss GraFORTH.

The System

The language can be divided into two main parts. The first part contains the compiler and low-level system routines. For most applications, the internal workings of these routines can be ignored. They usually do the things which need to be done without a lot of fanfare. The second part of the system is the 'word library'. The word library is the "visible" part of the GraFORTH system, and is the basis for writing programs.

Words

The word library is made up of a large number of GraFORTH 'words'. You can see this list of words at any time by typing the word "LIST". LIST is a GraFORTH word that lists all of the GraFORTH words. (LIST will display 20 words at a time. To see the entire list, press <return> at each pause. Press ConTRoL-C if you want to stop the listing.)

Each GraFORTH word accomplishes a particular task. For example, the word "BELL" beeps the Apple speaker, the word "+" adds two numbers together, and the word "DRAW" draws a three-dimensional image on the screen. Nearly everything in GraFORTH is either a word or a number. Words can be programs, subroutines, variables, or strings. Programs are written, not by entering "program lines", but by stringing words together.

The name of a word can be any string of ASCII characters that does not include a space or carriage return. The space acts as a divider between words, and a carriage return tells the system to compile the entered line into machine language and, in most cases, execute it. Since GraFORTH uses spaces to determine when one word ends and another begins, putting spaces between GraFORTH words is very important.

The Data Stack

Words are executedd in the order they are entered. When the word "+" is executed, iit wants to add two numbers together, right then and there. This mmeans that both of the numbers to be added must already be availabble for "+" when it is executed. Where do the numbers wait beforre they are added? They are on the 'data stack', placed theere by you before entering "+".

All numbers in GraaFORTH are routed through the data stack, which we'll usually justt call the 'stack'. The stack is simply a stack of numbers, one onn top of another, much like a deck of cards, or a stack of dinner plates. When you enter a number, it is put on the top of the staack, above any numbers which might already be there. Some wordss place numbers on the stack. Some words remove numbers from the sstack. Some words do both. The word "+" is an example of this; iit removes two numbers from the stack, adds them, and places tthe sum back on the stack. If the stack is empty, and a word tries to remove a number from the stack, a phenomenon called 'stack underflow' occurs. Stack underflow will be discussed in grreater detail at the end of this chapter.

Numbers

GraFORTH is an intteger language. It uses numbers in the range -32768 to +32767. You can enter numbers outside of this range, but they will be ""folded" back into the range (e.g. the number 32769 will be storred as -32767). Certain operations, such as division, will truuncate decimal numbers back into integers. For example, 7/3=2.3333333, but GraFORTH will evaluate 7/3 as 2.

Hands-On Exppperience

Nearly every entryy in GraFORTH is ended by pressing the <return> key. For the exampples below, and throughout the rest of the manual, press the <return> key after every entry unless we tell you otherwise.

As you step through these 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, don't worry: The power switch was put on the Apple for a good reason! Just turn the power off and reboot again, then try to figure out what went wrong. We'll help you along the way.

Enough theory. Let's try some examples. Type:

```
Ready 3 4 5
```

The numbers 3, 4, and 5 have been put onto the stack. If you have any doubts, just type the word STACK.

```
Ready STACK

[3]
[4]
[5]
Ready
```

Typing STACK turns on the stack display, so you can see what numbers are on the stack. The stack display stays on until you type STACK again. This display is toggled on or off whenever you type STACK. You may want to try this a bit, but as we go on, have the stack display on. Now type:

```
Ready 6 7

[3]
[4]
[5]
[6]
[7]
Ready
```

The numbers 6 and 7 have been added to the top of the stack. Notice that the stack display is "upside-down": What we've been calling 'top of stack' is shown as beinq below the other numbers. Here's why: stacks and 'top of stack' are both standard computerese conventions, and we didn't want to break tradition by calling it the "bottom of stack". But the GraFORTH stack can hold up to 128 numbers while the Apple screen can only display 24 lines. With the stack display turned upside-down, then the 'top of stack' (the most accessible number) will always be the number closest to the "Ready" prompt, instead of beinq scrolled off the screen.

Now that we have some numbers on the stack, what can we do with them? One thing we can do is print them. The word "." (period) removes a number from the stack and prints it. Type a period:

```
Ready .
7
[3]
[4]
[5]
[6]
Ready
```

The 7 was removed from the stack and printed. Now type "+":

```
Ready +

[3]
[4]
[11]
Ready
```

The numbers 5 and 6 were removed from the stack by the word "+", added together, and the sum placed back on the stack. Now type three periods, separated by spaces:

```
Ready . . .
1143

Ready
```

The 11, 4, and 3 were all printed, without any spaces between them. We'll show you how to position the printing of both numbers and text in a bit.

You now know how to put numbers on the stack, add them together, and remove them by printing them. Since most words in GraFORTH use the stack, it's important to know exactly what's happening on the stack when a word is executed. Let's introduce a notation for the effect of a word on the stack. We'll list the word, followed by a "before and after" representation of the stack, then a brief description of what the word does. The stack numbers are shown as letters, with a dash to the right indicating top of stack. Remember, the top of stack is the dash on the right. An empty stack is indicated by three dashes. Using this notation, here are the four GraFORTH words we've shown so far:

Word	Before	After	Description
LIST	- - -	- - -	Lists the words in the GraFORTH word library.
STACK	- - -	- - -	Toggles the stack display on and off.
.	n -	- - -	Prints n.
+	m n -	p -	Takes m and n off the stack, adds them and places their sum, p, back on the stack (p=m+n).

Note that there may be other numbers on the stack below those shown in the before and after diagrams, but these are not affected by the word.

More Words

Stack Words

Here are some GraFORTH words which manipulate the numbers on the stack:

DUP duplicates (makes a copy of) the top number on the stack.

SWAP swaps the position of the top two stack entries.

DROP removes the top number from the stack. The number is lost.

OVER makes a copy of the number immediately beneath top of stack, placing the copy on the top of the stack.

PICK uses the top number on the stack to select a number from within the stack, then the number is copied to top of stack. For example, 1 PICK is equivalent to DUP, and 2 PICK is equivalent to OVER.

Here are the same words defined using the stack diagram:

Word	Before	After	Description
DUP	n -	n n -	Duplicates n.
SWAP	m n -	n m -	Swaps m and n.
DROP	n -	- - -	Drops (forgets) n.
OVER	m n -	m n m -	Copies n to top of stack.
PICK	...m n -	...m q -	Copies nth item to top of stack.

Keeping an eye on these definitions, some more examples may be helpful here:

```
Ready 1 2 3
[1]
[2]
[3]
Ready SWAP          (Exchange positions of the 2 and 3.)
[1]
[3]
[2]
Ready DUP           (Make a copy of the 2.)
[1]
[3]
[2]
[2]
Ready DROP          (Remove the copy just made.)
[1]
[3]
[2]
Ready OVER          (Copy the second from top of stack.)
[1]
[3]
[2]
[3]
```



```
Ready 4 PICK        (Copy the fourth position down stack.)
[1]
[3]
[2]
[3]
[1]

Ready DROP DROP .   (Remove 3 and 1, then print the 2.)
2
[3]
[1]
Ready DROP DROP     (Remove the remaining 3 and 1.)

Ready               (The stack is now empty.)
```

You will probably want to experiment further with each of these words with the stack display on. While their functions may not be terribly exciting, you'll find they will be very useful later on for placing numbers where they need to be at the right time.

Arithmetic Words

You've seen how "+" works; on the next page is a listing of the GraFORTH arithmetic words.

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 -	r -	remainder	(modulo)
CHS	n -	m -	m=-n	(change sign)
ABS	n -	m -	m=ABS(n)	(absolute value)
SGN	n -	m -	m=1 if n>0, 0 if n=0, -1 if n<0	(sign)
SIN	n -	m -	-128<m<127	(sine)
MIN	m n -	p -	p=m if m<n, n if n<m	(minimum)
MAX	m n -	p -	p=m if m>n, n if n>m	(maximum)
RND	- - -	n -	-32768<n<32767	(random number)
RNDB	- - -	n -	0<n<255	(random byte)

Here are some examples of the GraFORTH arithmetic words in action:

```
Ready 23 5 / .
4
Ready 23 5 MOD .
3
```

(23 divided by 5 leaves 4, and a remainder of 3.)

```
Ready 6 CHS
[-6]
Ready ABS .
6

Ready 18 19 MIN
[18]
Ready SGN .
1

Ready -7 SGN .
-1

Ready RND .
-22317
```

RND leaves a random number on the stack. (Of course, the number displayed will most likely be different from the one shown above.)

Using Words

Now that we've introduced a whole slurry of words, let's put them to use.

For these examples, we'll assume the stack is empty before beginning. There are a few ways to empty the stack. With the stack display on, you can type either DROP or "." repeatedly until the stack display shows the stack is empty.

Another way to clear everything is to type the word ABORT. ABORT restarts GraFORTH, resetting things back to their initial conditions. ABORT can be handy when used from the keyboard, but if executed from a running program, it stops the program immediately. (There is an exception to this which will be discussed in Chapter 5.)

As you've already seen, the way to add two numbers is to enter the numbers first,, then type "+".

```
Ready 3 4 + .
7
Ready
```

This notation, wheere the numbers precede the operator, is called Postfix, or Reversse Polish Notation, and is used in all versions of Forth, as well as in most Hewlett-Packard calculators. Its main advantage oveer "standard" notation is that complicated expressions can bee evaluated without having to use parentheses. For example, if ycou wanted to add 3 and 5 together, add 7 and 9 together, then mulltiply their sums in a language like Basic, you would type:

```
X=(3+5)*(7+9)
```

Note that since Bbasic always multiplies before adding, parentheses were ineeded to group the sums together. In GraFORTH, you can solve thee problem this way:

```
Ready 3 5

[3]
[5]
Ready +

[8]
Ready 7 9

[8]
[7]
[9]
Ready +

[8]
[16]
Ready *

[128]
Ready .
128
Ready
```

This example wass "unfolded" so you can see exactly what is happeninq on thee stack. Usually, the entire expression is entered on one lline:


```
Ready 3 5 + 7 9 + * .
128
Ready
```

To find the cube of a number, you can type the number three times and multiply:

```
Ready 3 3 3 * * .
27
```

Another way is to type the number once and use DUP to duplicate it:

```
Ready 3 DUP DUP * * .
27
```

DUP allows you to use any number without having to enter it repeatedly. This will be very useful for general purpose operations inside programs.

Printing Text

Printing text in GraFORTH is straightforward: type the word PRINT, the word " (quote), the text to be printed, then another quote:

```
Ready PRINT " SUPER ZAPPO SPACE GAME "
SUPER ZAPPO SPACE GAME
Ready
```

Since the quote is a GraFORTH word, the spaces between the quotes and the text are required. Note that you can use quotes within the quoted text, as long as it is not separated on both sides with spaces:

```
Ready PRINT " THIS IS THE "BEST" GAME EVER! "
THIS IS THE "BEST" GAME EVER!
Ready
```

Since PRINT does not automatically print a space or a carriage return at the end of the text, two other handy words to know are SPCE and CR. SPCE prints a space, and CR issues a carriage return. Notice the difference in the following three examples:

```
Ready PRINT " FIRE " PRINT " ONE "
FIREONE

Ready PRINT " FIRE " SPCE PRINT " TWO "
FIRE TWO

Ready PRINT " FIRE " CR PRINT " THREE "
FIRE
THREE
```

Printing text is not very useful if the system only prints the text immediately then forgets it. Fortunately, GraFORTH can do much more than that.

Defining New Words

The power of GraFORTH as a language lies in the ability to define new words in terms of old ones. In fact, writing "programs" in GraFORTH is done by simply defining a series of new words which accomplish the desired task. These new words are added to the word library and can be seen by typing the word LIST. In this way, the GraFORTH 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 directly at the keyboard. The semicolon marks the end of the colon definition, and causes the word to be compiled into machine language and added to the word library.

As an example, let's define a word that adds two numbers then prints their sum along with a short message:

```
Ready : SUM PRINT " THE SUM IS  " + . ;
```


Following the form for colon definitions, SUM is the name of the new word, and

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

is executed whenever the word SUM is entered. The word PPRINT causes the phrase "THE SUM IS" to be printed, the + adds the top two numbers on the stack, and the period prints the sum. (Note that there are two spaces between the word IS and the quoote, so that a space will appear between the text and the number..) Now let's try our new word:

```
Ready 25 31 SUM
THE SUM IS 56
Ready
```

LIST the word library, and you'll see that the word SUM haas been added:

```
Ready LIST

SUM
CHS
SGN
CALL
.
.
```

A nice addition to this word would be to reprint the numbeers being added. But before we commit ourselves to a colon definition, let's try it "live", where we can watch thingss one step at a time:

```
Ready STACK

Ready 25 31

[25]
[31]
```

We need to make copies of the two numbers: one set will bee reprinted on the screen, and other sett will be added togethher. (Remember that many GraFORTH words comsume numbeers from thee stack, so we need to have the numbers ready to "feed" them!!) The quickest way to copy a pair of numbers is by using OVER OVEER:

```
Ready OVER .

[25]
[31]
[25]
Ready OVER

[25]
[31]
[25]
[31]
```

Now let's reprint the first set of numbers along with some informative text:

```
Ready PRINT " THE SUM OF  " .
THE SUM OF 31
[25]
[31]
[25]

Ready PRINT "  AND  " . PRINT "  IS  "
 AND 25 IS
[25]
[31]
```

Now let's add the numbers...

```
Ready +
[56]
```

...and print the sum:

```
Ready .
56
```

Now let's put it into a colon definition, with a different name. Note that you can enter the definition over several lines (if you like).

```
Ready : SUM1

Ready OVER OVER PRINT " THE SUM OF  " .

Ready PRINT "  AND  " .

Ready PRINT "  IS  " + . ;
```


After entering the definition, the word SUM1 is also on the word library:

```
Ready LIST

SUM1
SUM
CHS
ABS
.
.

Ready 25 31 SUM1
THE SUM OF 31 AND 25 IS 56
```

SUM1 can now be called at any time, from either the keyboard or another word definition, as easily as any of the original GraFORTH words in the word library.

Note: As you write and enter colon definitions, be sure to enter a semicolon to finish the definition! If you don't, GraFORTH will assume that everything you type is part of a word to be executed at a later time. If GraFORTH ever responds to words like LIST with only a "Ready" prompt, you've probably left a semicolon out of colon definition.

Forgetting Words

You can see that if we keep on defining new words, the word library will continue to grow until we use up all of the memory available. 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:

```
Ready 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, to delete both SUM and SUM1, type:

```
Ready FORGET SUM
```

```
Ready LIST

CHS
ABS
SGN
.
.
```

Notice that both SUM and SUM1 were removed from the word library. Had there been more words above them, they would also have been removed.

Note: You will not get an error message if you try to FORGET a word that is not in the word library. This makes implementing program 'overlays' easier. (Overlays will be discussed in Chapter 5.) However, if you misspell the word you want to forget, then no words will be deleted from the word library. Thus, it's a good idea to use LIST to verify that the right word or words have been deleted.

Looping Structures

The GraFORTH DO - LOOP construct is available for repetitive tasks where the number of repetitions is known ahead of time. The form for a DO - LOOP is:

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

The word DO removes two values from the stack. The top number is used as an 'initial value' and the next number is used as an '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 similiar to Applesoft's "FOR -- NEXT" looping structure.

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" sets up the looping structure for 5 loops. Inside the loop, the phrase "HERE IS NUMBER" is printed, then the loop value is retrieved by I, then printed with ".". CR causes the carriage return to put each number on its own line, and LOOP 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 less than 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. Typing DO or LOOP alone can have nasty and unpredictable results.

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

Loops can be nested inside one another. The loop value for the current innermost loop is always accessed 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 0 DO

Ready    PRINT " OUTER LOOP:  " I . CR

Ready    3 0 DO

Ready       J . SPCE I . CR

Ready    LOOP

Ready LOOP ;
```



```
Ready DOUBLELOOP
OUTER LOOP: 0
0 0
0 1
0 2
OUTER LOOP: 1
1 0
1 1
1 2
OUTER LOOP: 2
2 0
2 1
2 2
OUTER LOOP: 3
3 0
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 refferenced from the inner loop with "J". Just remember that "I" allways 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 inside the innermost loop with the word "K".

The Return Stack

DO - LOOPs make use of another stack in the GraFORTH system, similar to the data stack, known as the 'return stack'. The return stack can also hold 128 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 GraFORTH 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 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 and places it on the data stack. When the loop is finally exited, the two return stack values are removed.

There are a few words in GraFORTH 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. (In Chapter 5 we'll show you 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.

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]
Ready SWAP

[2]
[3]
Ready PULL

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


Comparing Numbers

A number of GraFORTH 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 0 if the comparison is false. Here are a few examples:

```
Ready 5 5 = .
1

Ready 5 7 = .
0

Ready -32 -6 < .
1

Ready 45 46 >= .
0
```

A couple of other words related to the comparison words are AND and OR. These words remove two numbers from the stack and perform a logical operation between each of the 16 bits of the numbers, returning another number to the stack.

AND performs a bitwise "AND" between the two stack values; OR performs a bitwise "OR". Don't worry if you're unfamiliar with the relationships between numbers and their bits. Usually the importance of AND and OR is between two zero or nonzero numbers:

If both the top stack value and the second stack value are nonzero (representing "true"), then the AND of the two numbers will also be nonzero. If either or both numbers are zero, then the AND will also be zero.

If either the top stack value or the second stack value are nonzero, then the OR of the two numbers will be also nonzero. Only when both numbers are zero will the OR operation be zero.

AND and OR are useful for combining the results of two or more tests. The following example tests whether or not a given number is greater than 5 and less than 10. We'll test with two numbers, 7 and 3:

```
Ready 7
[7]
Ready DUP 5 >
[7]
[1]
Ready SWAP
[1]
[7]
Ready 10 <
[1]
[1]
Ready AND
[1]
```

7 is greater than 5 and less than 10.

```
Ready 13
[13]
Ready DUP 5 >
[13]
[1]
Ready SWAP
[1]
[13]
Ready 10 <
[1]
[0]
Ready AND
[0]
```

13 is not 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 off the test. GraFORTH 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 above 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.

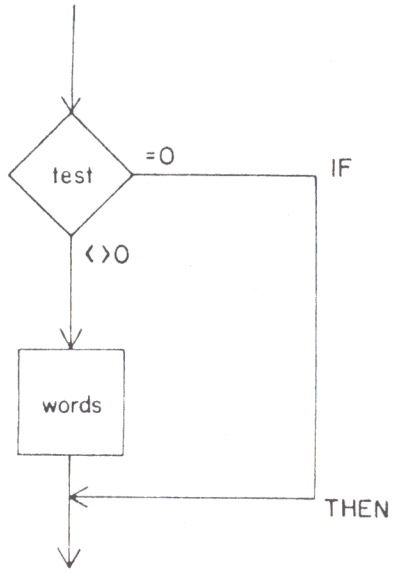
IF - THEN

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

```
<stack test 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 continues on after the word THEN. The flowchart for IF - THEN follows on the next page:



Let's use 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 the IF and THEN, is executed. If the number is zero, then it is not executed.

```
Ready 5 TEST1
THE NUMBER IS NOT ZERO

Ready 0 TEST1
THE NUMBER IS ZERO
```

IF - THEN constructs can be used with number comparison words. Remember that these words return either one or zero, 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 one on the stack if the number is greater than 25, or a zero 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.

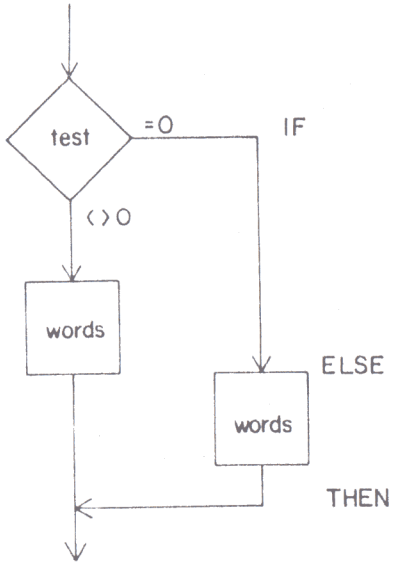
```
Ready 16 UPPERLIMIT .
16
Ready 37 UPPERLIMIT .
25
Ready
```

IF - ELSE - THEN

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

```
<test stack value>
IF
   <words executed if nonzero>
ELSE
   <words executed if zero>
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. The flowchart for IF - ELSE - THEN follows on the next page.




```
Ready : TEST2

Ready DUP 100 > IF

Ready     . PRINT "  IS GREATER THAN 100 "

Ready ELSE

Ready     . PRINT "  IS LESS THAN OR EQUAL TO 100 "

Ready THEN ;
```

Again, we've duplicated the number before comparing so that we could print it later, using one of the two periods inside the IF - ELSE - THEN. Also note that the controlled words are indented. This is certainly not a requirement, but it greatly improves the readability of the word definition. (In the next chapter, we'll 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
Ready
```

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, we DUPlicate the number because we don'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.

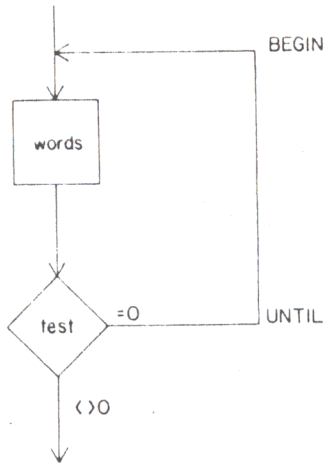
```
Ready -62 TWOLIMITS
LESS THAN 10

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 1 to it, then loops back until the number equals 8:

```
Ready 0 BEGIN DUP . CR 1 + DUP 8 = UNTIL
0
1
2
3
4
5
6
7
[8]
Ready
```

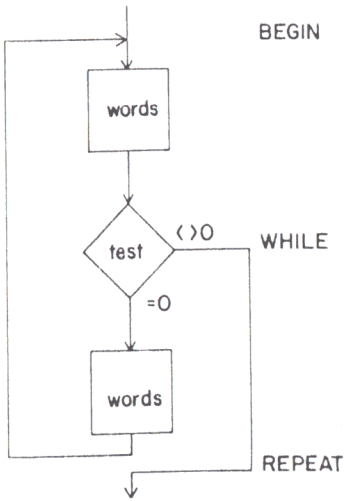
The words "DUP . CR" print the number without losing it and issue a carriage return; "1 +" increments the number; and "DUP 8 =" determines if the number equals 8. Notice that this loop leaves a copy of the number on the stack when it finishes. Adding DROP to the end of the line takes care of this.

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 is 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 for BEGIN - UNTIL. 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 0 BEGIN DUP 8 <> WHILE DUP . CR 1 + REPEAT
0
1
2
3
4
5
6
7
[8]
Ready
```

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:

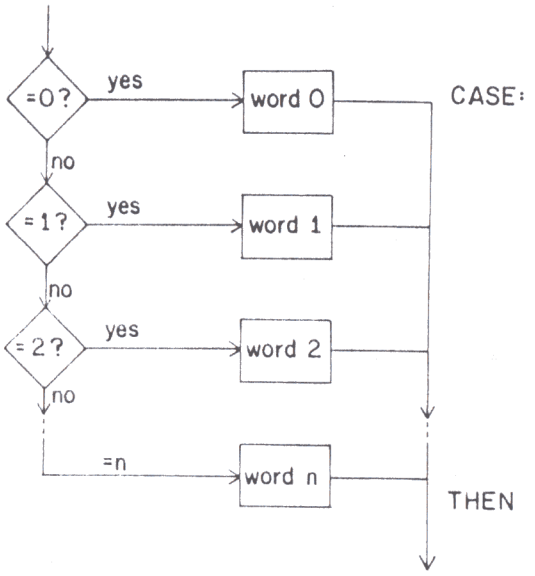
<stack value>

CASE:

<word 0>
<word 1>
<word 2>
.
.
.
<word n>

THEN

The word CASE: removes a number from the stack and uses this number to select and execute a single word from a list of words. A zero selects word 0, a one selects word 1, etc. The word THEN marks the end of the CASE: construct, and is required. The flowchart for CASE: follows on the next page:



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 0 through 3. Zero selects X, 1 selects Y, 2 selects Z, and 3 selects BELL:

```
Ready 0 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 0, 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 0 >= AND

Ready IF

Ready    CASE.TEST

Ready ELSE

Ready    PRINT " THE NUMBER IS NOT BETWEEN 0 AND 4 "

Ready    DROP

Ready THEN ;
```

SAFE.CASE first checks the number to see that it is between 0 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 0 >= 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 0 AND 4
```



```
Ready -6 SAFE.CASE
THE NUMBER IS NOT BETWEEN 0 AND 4
```

Program Structure and Other Miscellaneous Thoughts

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 GraFORTH 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 the programmer helplessly wading through miles and miles of computer print-out trying to find the elusive "bug" in a program.

Another advantage is that with separate word definitions, you can 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.

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

Word References

Words in GraFORTH can only be defined in terms of already existing words, which reside in the GraFORTH 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 unknown word will be ignored:

```
Ready 5 0 DO I . CR STRANGE LOOP
```

```
STRANGE Not Found (Press Return)

0
1
2
3
4
Ready
```

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 (Return)
```

With two words with the same name in the word library, 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 accessible from the keyboard! In general, defining words with existing word names is not a good idea and should be avoided.

Programmers who like to dabble with recursion will be happy to hear that GraFORTH 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 in certain recursive applications, but should be avoided if not needed. (Your programs can get hard for people to follow!)

Speed and Flexibility vs. Error Checking

GraFORTH is a very fast language. It has to be to manipulate 3-D images at the speeds it does. GraFORTH is also very flexible. As you'll see in Chapter 6, GraFORTH gives you direct control of your Apple.

You may be asking, "What's the catch?" The "catch" is that GraFORTH has little built-in error checking. In terms of speed, if your program works correctly, then repetitive error checking schemes can only slow your program down.

In terms of flexibility, if you're allowed to do nearly anything, then there is nothing "to protect you from". GraFORTH follows the Forth convention that if you want error checking, you'll write it into your programs. If you don't need error checking, you don't have to include it.

One example is 'stack underflow and overflow'. Stack underflow is where a word tries to remove a number from the stack and the stack is empty. If this happens, GraFORTH will merely return the number that was last on the stack. Stack overflow is caused by trying to place more than 128 numbers on the stack. If this happens, the extra numbers are ignored. If a stack underflow occurs when the stack display is on, a long stream of stack numbers may be displayed. If this happens, just type ABORT to clear the stack. (The key to avoiding stack problems is to be aware of what is happening on the stack at all times. Sometimes "single-stepping" through a list of words with the stack display on can help.)

Another example of error checking is with words that "expect" a number in a given range. We've seen this already with the word CASE:. Many words in GraFORTH use numbers in a specified range. Some words don't mind the excess; they "fold" the number back into an appropriate range and ignore the difference. Other words (like CASE:) do not fold back, and must be given a valid number. As we introduce words, we'll include any valid ranges.

Words Which Look Forward

Most words in GraFORTH look to the stack for any data or information they might need. Some words, like PRINT or FORGET, 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! The word PRINT looks for the text to be printed as it is compiled, 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.

Text Vs. Graphics

Since the Apple graphics screen is used for the normal GraFORTH display, mixed text and graphics, changeable character sets, and lower case displays can all be used in GraFORTH. However, text scrolling is not as fast as it would be on a standard text display. GraFORTH includes two words, GR and TEXT, which enable you to switch between the graphics display and a text-only display. The only advantage to using the text display is for faster scrolling, which can occasionally come in handy when editing files from the editor.

Memory Considerations

Because of the large number of features implemented in GraFORTH, and the fact that both graphics screens are being used, free memory for program development is somewhat limited. The presence of a language card or RAM card eases this limitation considerably. The memory map in Appendix F shows the available free memory with and without a language card, and with or without the text editor in memory. Memory considerations when using the text editor will be discussed in the next chapter.

The way to keep memory free is to always FORGET words that are no longer needed. Loading one large program onto the word library above another is a sure way to run out of memory. Be aware of what is on the word library, and how much memory is being used.

There are two words to help you:

The word PRGTOP 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 PRGTOP .
-32256
```


For people who "think" in hexadecimal, the word $LIST can also be very useful. $LIST is identical to LIST, except that it also displays the hexadecimal addresses 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 a $LIST:

```
Ready $LIST

$8254 CHS
$8246 ABS
$8224 SGN
$81F4 CALL
$81E9 PREG
$81DE YREG
  .    .
  .    .
  .    .
```

Since $LIST displays the address at which each word begins, the first address shown is 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

$826E IT
$8254 CHS
$8246 ABS
  .    .
  .    .

Ready FORGET IT
```

$826E is the hex address of the top of the word library.

Conclusion

Let's take a break here, and digest some of this information. This might be a good time to grab a pizza, take a nap or come out of hiding and visit someone who hasn't seen you in a few days! Anyway, when you come back we'll move into the text aspects of GraFORTH and introduce you to the supplied GraFORTH text editor. (We'll also show you some wonderful special characters to make your Apple a little more friendly...)

CHAPTER FOUR: TEXT MAGIC

Chapter Table of Contents:

Purpose and Overview

In Chapter 3, we learned (among other things) 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 the definition; to go back to the string of words which defined it.

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

GraFORTH 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 GraFORTH programs from the editor or from disk. We'll also give you some pointers to keep both system and editor memory happy. But first, we should discuss some of the special characters used in GraFORTH, both in and out of the editor, and how they can help both your programming and your programs.

Strange and Wonderful Characters

Upper and Lower Case

If you've looked at the GraFORTH demonstration, you've seen all these lower case characters on your Apple screen, but until now, we haven't told you how to enter lower case characters yourself. There's really no magic, as we'll soon see!

Upper and lower case can be set in a number of ways, and each is a two-key process.

While entering a line, type ConTRoL-O, then

"E": Subsequent entries will be in lower case unless ESC is pressed in advance. If ESC is pressed first, the following character will be in upper case.

"S": Entries will be shifted to upper case if your Apple][has the one wire shift key modification. (A wire running from the shift key to the game paddle AN3 input).

"U": All entries will be in upper case.

"L": All entries will be in lower case.

"Hidden Characters"

Although the Apple][keyboard won't accept all the ASCII characters, GraFORTH will. Here are the keys to press to get the "hidden characters":

ConTRoL-Shift-N gives a left bracket
ConTRoL-Shift-M gives an underline
ConTRoL-Shift-P gives a reverse slash
Shift-M gives a right bracket unless one of the lower case shift options has been set.

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 GraFORTH as if it were being typed again directly from the keyboard.

The Apple ESCape codes for moving the cursor also work from GraFORTH. These can be handy for making fast corrections from the GraFORTH 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.

Note: If any of the lower case shift modes have been set, then the ESCape key cannot be used to move the cursor. To move the cursor using ESCape, first set upper case only (ConTRoL-O, U) shift mode.

Line Insertions

Insertions can be made into the middle of a line using ConTRoL-I. Pressing ConTRoL-I pushes any characters to the right of the cursor one more space to the right.

To make an insertion using ConTRoL-I, first use the Apple ESCape codes to move the cursor to the beginning of the line to be changed. Use the retype key to move the cursor to the point of insertion, then press ConTRoL-I enough times to open up a space in the line for insertion. Now enter the additional text, then use the retype key to move the cursor to the end of the line, and press <return>.

Note: The ConTRoL-I feature works for editing only one 40-character line at a time. Pressing ConTRoL-I too many times can push text off the right end of the screen and into Never-Never Land...

The Text Editor

There are actually two text editors on the GraFORTH system disk, named OBJ.EDITOR1 and OBJ.EDITOR2. The first is used on systems that do not have a language card or RAM card and can edit about 2000 characters without changing the default settings. The second is used with systems that have language cards and can edit about 11,500 characters. Otherwise, the two editors are identical.

Note: GraFORTH and the GraFORTH editor both use standard DOS text files for program storage. If you have a text editor that you are accustomed to that also uses DOS text files, you may use it instead of the GraFORTH editor. Large programs will be more manageable in a text editor such as Apple Writer 2.0. Compiling programs into the GraFORTH system from disk is the same regardless of what editor is used to create the file.

For the editor examples in this chapter, we will use English sentences for text instead of GraFORTH programs. The editor doesn't know the difference, and it makes things easier to read. The editor is of course usually used for writing GraFORTH programs. The GraFORTH word MEMRD, discussed in the next section, allows text to be read and compiled directly from the editor.

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

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

The first command to know in the editor is "?", the question mark. Entering a question mark gives you the Editor Command Index, 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

Entries to the text editor are preceded by line numbers. These line numbers have no meaning to GraFORTH, and are not retained in the program file when it is saved to disk. They simply serve as an index to the file while it is in memory. The editor 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, and should be entered in upper case.)

```
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 accomplished. We're not going to show it in all of our examples, though.)

Inserting lines in the text editor is much the same as in 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. Let's insert a line between line 10 and line 20 by giving it a line number of 15:

```
15 With some important exceptions,
```


Now let's 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
34 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 80,150   lists lines 80 through 150.
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 our 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 codes and the right-arrow key can be used to retype a line that is on the screen, and ConTRoL-I can be used to make insertions within the line.

Simply entering a line number followed by <return> won't delete a line, as is true for Basic. Instead this will create a blank line, very useful in its own right for separating program segments and word definitions. To make a blank line while the Autonum feature is in use, enter a space, then press <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 80,150   deletes lines 80 through 150.
D 80,      deletes from line 80 to the end of the file.
D ,80      deletes from the beginning of the file to line 80.
```

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, we used Autonum 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.
```


We can start an insertion by entering the first line number of the insertion ourselves:

```
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. Let's 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 is used to specify the starting line number. Let's delete the lines we just entered, and re-enter them, this time using INSERT.

```
D 20,50

Done

L
10 The first line in the file...
20 The last line.
```

We want to insert between lines 10 and 20, so 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 beginning to line 80)
(Filename) : TESTFILE/80,     (Saves line 80 to end of the file)
```


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 and, if desired, the disk slot and drive at which 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 destination line number in the current file. This example will insert the file TESTFILE into the current editor file between lines 110 and 120:

```
(Filename) : TESTFILE/115
```

Note: "GET" always inserts the file into the present memory contents. The file contents are not erased by "GET". To erase the present file and get a new one, "ERASE" the present file and then "GET" a new one. Seems simple enough.

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, then press <Return> twice to remove the DOS prompt.

With the printer enabled, 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 GraFORTH's special graphic output, press Reset to turn the printer off and return to a normal display. The next chapter includes a discussion on how to access peripherals and return to GraFORTH in a normal manner.

Memory Considerations

As the GraFORTH word library grows, it can begin to use the same memory that is used from the editor. If the word library is large enough, adding words can erase a part of the editor file, or even the editor program itself. Conversely, using the editor can destroy the top of the word library, requiring the system to be rebooted.

In addition, the amount of usable editor file memory is determined by the presence or absence of a language card. We suggest you study the memory map in Appendix D and become generally familiar with areas of memory used by the GraFORTH language, the editor program and the editor file in your system.

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.

You may want to adjust the amount of memory used by the text editor, to avoid conflict with GraFORTH. To accomplish this, you may position the file either up or down in memory. To do this, press "P". A display will appear:

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

The length, position (starting address of the editor file area), and memory labels will be followed by their present numeric values. To change the editor file position, enter "Y" to this option. You will be prompted:

Enter New Position :

On a language card system, the file position can be moved somewhat higher to make more room for the GraFORTH word library. On a non-language card system, it's often best to use the editor without regard to keeping the word library intact, save the edited file to disk, and reboot GraFORTH from scratch. This method will be outlined in the next section.

Leaving the Text Editor

To leave the text editor and return to GraFORTH, simply type "B" for "Bye".

Program Compilation

GraFORTH normally accepts its input from the keyboard. Each line is compiled immediately and acted upon if necessary.

GraFORTH can also read lines from the editor file or from a disk file, treating the lines as if they were typed from the keyboard. GraFORTH programs can be written in the editor and saved to disk, then read and compiled into the system.

The word to read and compile text from the editor buffer is MEMRD. MEMRD removes a number from the stack, interprets this number as an address, and begins reading text from memory starting at this address. It reads and compiles until it either reads a zero byte (marking end-of-file) or encounters an error. Control is then returned back to the keyboard.

The address of the editor file buffer is 34817, unless changed with the Program Position option in the editor. To read the text from the editor, type:

```
Ready 34817 MEMRD
```

To read and compile directly from a text file, the word READ is used. The form for READ is:

```
READ " <filename> "
```

READ reads to the end of a file, or until an error is encountered.

MEMRD and READ are usually used to compile word definitions into the word library, but immediate-execution lines can also be included.

Comments

Usually, the GraFORTH Editor is used for writing and editing GraFORTH programs instead of the text used earlier in this chapter. Comments in the source file of a GraFORTH program can often be very helpful for understanding and keeping track of long programs.

The GraFORTH word "(" is available for inserting comments into program files. In compiling the program, when GraFORTH sees a "(" set off with a space on either side, it ignores the rest of the text on the line until it sees a ")". Comments can be inserted freely in the source file. Here is an example of such a comment line:

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

Using the Editor with GraFORTH

When smaller programs are being developed, the editor and the GraFORTH system can be used closely together. Load the editor and enter the program, then return to GraFORTH and compile the program with MEMRD. If the program has bugs or needs further changes, simply return to the editor and make those changes. When returning to GraFORTH, FORGET the original word definitions before compiling the new ones, to prevent "Not Unique" errors from occurring. (Unless you're testing a very short program, you should also save the program to disk after each edit.)

When larger programs are being developed and GraFORTH/editor memory conflicts are likely, it's best to separate editing and compiling. Use the editor to write the program, then save the program to disk. Then return to GraFORTH and compile the program with READ or MEMRD. If the program needs to be changed, FORGET the words before returning to the editor, so that editor usage won't erase the top of the word library. From the editor, reload the program from disk and continue editing.

Understanding and following the above guidelines will protect you from memory conflicts, and will make programming in GraFORTH much easier.

As you become more comfortable with programming in GraFORTH, 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 GraFORTH.

CHAPTER FIVE: DELVING DEEPER. . .

Chapter Table of Contents:

Purpose and Overview

Chapter 4 introduced GraFORTH as a language. In this chapter, we'll round out the language and give you some of the background you need before moving on to the graphics features ("What? You mean this language has graphics too?!") in the next three chapters.

We'll start off by introducing the GraFORTH standard text manipulation words (not to be confused with the fancy ones we'll show you in Chapter 7). Then we'll discuss storing data in memory, and the various words used to accomplish this. We'll talk about the two other kinds of words in GraFORTH (variables, and strings), and how they can be used to set aside memory for data storage in very convenient ways. Following this will be a discussion of the string operators built into the system and on a disk file.

Next, we'll talk about using DOS from GraFORTH, and introduce SAVEPRG, the word that makes your work permanent. We'll tie up the loose ends with a number of words which are extremely useful, but evade strict categorization.

Text Formatting Words

These are the words which are used to position text and characters on the screen, and clear the screen, or portions of it. Each of these words is straightforward.

Review

You have seen how to use PRINT, SPCE, and CR already in Chapter 3. For a quick review...

PRINT prints following quoted text 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.

New Text Positioning 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.

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 ranges for HTAB and VTAB depend on the current character size (CHRSIZE), which will be discussed in Chapter 7. For the normal character size we are using now, the range for HTAB is 0 to 39, and the range for VTAB is 0 to 23.

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. The form for WINDOW is:

<left> <width> <top> <bottom> WINDOW

Left, top and bottom are actual margins for the window. Width specifies the right margin as the number of characters from the left margin. The bottom margin number 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 the 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.

CLEOP (CLear to End Of Page) erases the screen from the current cursor position to the end of the text window.

CLEOL (CLear to End Of Line) erases from the current cursor position to the end of the line.

ERASE erases the entire screen, regardless of the setting of the text window. ERASE is usually faster than HOME.

Data Storage and Retrieval

GraFORTH has the capability of examining and changing the value stored in any location in memory. If desired, the actual decimal memory address can be entered from the keyboard for storage or retrieval. We'll show you data access in this way first, and then discuss an easier technique using named variables.

GraFORTH Memory Addresses

The Apple][contains 65536 addressable "locations". These locations are usually numbered from 0 to 65535. Most of them are used for RAM memory, which can be either read from or written to. Each memory location can store one 8-bit 'byte', representing a number from 0 to 255. Two locations, or two bytes, can store a number from 0 to 65535. Since two bytes can only reference positive numbers in the range 0 to 65535 and people sometimes like to use negative numbers, one 'bit' of the number is used to tell us the numbers sign, positive or negative. Therefore, GraFORTH uses a number range of -32768 to 32767. Since it is desirable that zero in both systems be zero, a "wrap-around" scheme is used: Addresses above 32767 are treated as negative numbers, and continue to increase until they again reach zero. (This is identical to the system used by Apple's Integer Basic, where a call to enter the system monitor must be done with a negative number: CALL -151.) A diagram will best explain this:

Positive Decimal Addresses	GraFORTH Decimal Addresses
0	0
1	1
2	2
.	.
.	.
32766	32766
32767	32767
32768	-32768
32769	-32767
32770	-32766
.	.
.	.
65533	-3
65534	-2
65535	-1

Notice that both address ranges continually increase, except that the GraFORTH addresses have a transition in the middle from positive to negative numbers. The memory map in Appendix D includes GraFORTH decimal addresses and hexadecimal addresses.

Storage and Retrieval Words

To store a number directly into a desired memory location, simply place the number you want to store and the address where you want it stored on the stack. Then type "POKEW". The word "POKEW" stands for "poke-word" and removes two numbers from the stack, interpreting them as value and address, and stores the data value at the given location. Since GraFORTH numbers occupy two bytes (commonly called a 'word', not to be confused with GraFORTH words), it actually uses the given location and the one immediately after it.

This example stores the number 12345 at location 2816 (which happens to be the beginning of a large free area of memory in GraFORTH):

```
Ready 12345 2816

[12345]
[2816]
Ready POKEW

Ready
```

To recall a number from memory and place it on the stack, place the address of the desired memory location on the stack and type "PEEKW". The word "PEEKW" stands for "peek-word" and removes a number from the stack, interprets it as an address, retrieves a number from that address, and places the retrieved number on the stack. The following example recalls the number we just stored in memory:

```
Ready 2816

[2816]
Ready PEEKW

[12345]
Ready
```

To store a single-byte value to one memory location, the word "POKE" is used instead of "POKEW". The form is the same. This example stores the number 255 to location -28721:

```
Ready 255 -28721 POKE
```

The word "PEEK" is used to retrieve single bytes from memory. The form for "PEEK" is the same as for "PEEKW". This example reads a special Apple location that contains the current horizontal cursor position:

```
Ready PRINT " Demonstrating PEEK "  36 PEEK
Demonstrating PEEK

[18]
Ready
```

Printing the phrase "Demonstrating PEEK" moved the cursor out to position 18. Reading location 36 retrieved this position as a number.

To summarize, here is a table of the four storage and rerieval words:

Word	Before	After	Description
POKEW	m n -	- - -	Puts two byte m into location n
PEEKW	n -	m - -	Reads two byte m from location n
POKE	m n -	- - -	Puts one byte m into location n
PEEK	n -	m - -	Reads one byte m from location n

Variables

GraFORTH allows you to set aside space for number storag through the word "VARIABLE". VARIABLE creates a new word and paces it on the GraFORTH word library. VARIABLE has two forms; te first one is:

VARIABLE <variable name>

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

```
Ready LIST

CHS
ABS
SGN
 .
 .
 .

Ready VARIABLE TEMP

Ready LIST

TEMP
CHS
ABS
SGN
 .
 .
```

The new word TEMP consists of two parts: a two-byte space set aside for storing a number, and a call to an internal GraFORTH routine that either places the value of the variable on the stack or stores the stack value into the variable.

To store the number 12345 in TEMP, type:

```
Ready 12345
[12345]

Ready -> TEMP

Ready
```

The GraFORTH word "->" is a special word that says "store into". It is created by typing a minus sign '-' followed by a right arrow '>'. This word sets an internal flag used by variables to determine if a "store" or a "recall" operation is to take place. When the "->" word is executed it sets this flag so the next referenced variable will do a store, rather than a recall. Note that the variable will clear this flag so no special operator is needed when doing a recall.

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

```
Ready TEMP
[12345]
```

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

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 other 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
```

We should bring up something important here. The word VARIABLE (as well as STRING, 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 GraFORTH words.

To be specific, a VARIABLE declaration cannot appear inside of a colon definition. It must be alone on its own line, not mixed with other GraFORTH 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 can't be a computed number. For example, the following line will **not** work:

```
Ready 25 7 * VARIABLE THING
```

Strings

Strings in GraFORTH are words with space set aside for storing characters or text, rather than numbers. Strings are used whenever input is requested from the keyboard, or text has to be manipulated in any way. String words are created with the word STRING, and a number of words devoted to manipulating strings and character data are included in GraFORTH. Additional words, for more complex string tasks, can be found on a disk file called "STRING WORDS", and can be compiled into the word library at any time.

Defining Strings

The word "**STRING**" is used to create words in the GraFORTH word library that are used for string storage. The form for the word STRING is:

<string size> **STRING** <string name>

The string name is the name of the word to be added to the word library. The string size is a number specifying the number of bytes, or characters, the string will hold. Remembering how precious computer memory is, the string size should be just large enough to hold whatever string data is expected to go into the string. On the other hand, sufficient room must be allotted in the string for any value ever stored into it. If you attempt to store too much text into a string, you will actually damage the GraFORTH word library. This will force you to reboot the entire system from scratch! To increase speed, FORTH implementations (GraFORTH included) typically do very little error checking. Therefore it is up to you to determine beforehand the size string you will need.

Similar to variables, string declarations draw both their string name and string size from the input line, and have the same restrictions for use as variable declarations.

The following example creates a new word called TESTSTRING which can store a string up to 45 characters long:

```
Ready 45 STRING TESTSTRING

Ready LIST

TESTSTRING
CHS
ABS
SGN
 .
 .
```


GraFORTH strings are indexed from 0 to the string size-1. When a **string** word is executed, the word removes a number from the **stack**, adds this number to the address of the beginning of the **string**, then places this address on the stack. Note that strings differ from variables in that a variable actually places its value on the stack, while a string places the address of the beginning of the string plus the specified index on the stack. Getting the address instead of the value of the string may not seem like much fun, but in a moment we'll show you some powerful words to move string information around!

In the following example, entering "0 TESTSTRING" will place the address of the beginning of the string on the stack. Entering "5 TESTSTRING" will place the address of character number 5 in TESTSTRING on the stack. The last character position of TESTSTRING is accessed with "44 TESTSTRING". Any portion of the string can be accessed quickly in this way.

```
Ready 0 TESTSTRING .
-32241

Ready 5 TESTSTRING .
-32236

Ready 44 TESTSTRING .
-32197
```

Notice the addresses returned are negative. If you don't understand why, be sure to turn back a few pages to the discussion of GraFORTH memory addresses!

Note: The addresses we show are for example purposes. The actual values may be slightly different.

Using Strings

In this section, we'll show you how to use those memory addresses that strings leave on the stack. We'll ASSIGN text to a string, and WRITE and READ lines of text to and from the Apple's screen and keyboard.

To store text directly into a string (or anywhere in memory), the word "**ASSIGN**" is used, with the form:

<string address> **ASSIGN** " <quoted text> "

ASSIGN removes a number from the stack, interprets it as a memory address, then stores the text between the quotes into memory starting at that address. Usually the address is supplied by entering the name of a string before typing ASSIGN. Here is an example:

```
Ready 0 TESTSTRING
[-32241]

Ready ASSIGN " SHE SELLS SEASHELLS "

Ready
```

The phrase "SHE SELLS SEASHELLS" has been stored into the string TESTSTRING.

To write the contents of a string 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:

<string address> **WRITELN**

The following example writes the contents of the string TESTSTRING to the Apple screen:

```
Ready 0 TESTSTRING WRITELN
SHE SELLS SEASHELLS
```

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 a memory address, then reads a line of text from the keyboard and stores the text in memory starting at that address. Like WRITELN, the form of READLN is:

<string address> **READLN**

Here is an example:

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

The phrase "SEASHELLS" has been read into the string TESTSTRING.

```
Ready 0 TESTSTRING WRITELN
SEASHELLS
```

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

```
Ready 3 TESTSTRING READLN
SHORE

Ready 0 TESTSTRING WRITELN
SEASHORE
```

The word "SHORE" was read into TESTSTRING, starting at character number 3, over the top of "SHELLS".

```
Ready 2 TESTSTRING WRITELN
ASHORE
```

The string was printed starting with character number 2, leaving only the "A" in "SEA".

When a string is stored in memory using ASSIGN or READLN, a carriage return is placed after the last character, marking the end of the string. When WRITELN writes a string from memory, it starts at the specified 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.

String Conversion

Sometimes a string will contain a number stored as text. The GraFORTH word "**GETNUM**" is used to read the number from the text, placing the value 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 0 TESTSTRING READLN
321

Ready 0 TESTSTRING 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 at the given string address, it places a zero on the stack. To determine for certain whether or not the string-to-number conversion 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 0 TESTSTRING READLN
555
Ready 0 TESTSTRING GETNUM .
555
Ready VALID .
253
```

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

```
Ready 0 TESTSTRING READLN
YOU CALL THIS A NUMBER??
Ready 0 TESTSTRING GETNUM .
0
Ready VALID .
0
```

(VALID is zero since GETNUM failed to find a number.)

PAD: The System String

GraFORTH includes a predeclared temporary string space of 124 characters called PAD. PAD is convenient for reading keyboard input without having to define a string first.

Actually, PAD is two things: a 124-byte free area of memory used for storing string data, and a word in the GraFORTH word library named PAD which places the address of this free area of memory on the stack. Note that the usual string indexing is not used with PAD:

```
Ready PAD
[812]
```

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

```
[812]
Ready READLN
Goin' back to my pad.

Ready PAD WRITELN
Goin' back to my pad.
```

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

```
Ready PAD
[812]
Ready 6 +
[818]
Ready WRITELN
back to my pad.
```

Note: PAD is considered a temporary string space because the same space is used by the GraFORTH system when compiling words onto the word library, overwriting the previous contents of PAD. Predeclared strings should be used for more permanent string storage.

Accessing Individual Characters in Strings

Since each character in a string occupies one memory location, individual characters in strings can be accessed using PEEK and POKE. In this example, a line of text is placed in TESTSTRING, then the ASCII value of the first character is read onto the stack:

```
Ready 0 TESTSTRING ASSIGN " String pickings "

Ready 0 TESTSTRING PEEK
[211]
```

211 is the ASCII value for the letter "S". "0 TESTSTRING" placed the address of the first character of the string on the stack, then PEEK read the value from this address. In the next example, the "i" in "string" is overwritten with the letter "o" by storing its ASCII value:

```
Ready 239  3 TESTSTRING POKE

Ready 0 TESTSTRING WRITELN
Strong pickings
```


String Words on Disk

There is a file on the GraFORTH system diskette called "STRING WORDS". This file contains additional words for manipulating strings in more complicated ways. To make the string words active, simply compile the file into memory by typing:

```
Ready READ " STRING WORDS "
```

Here are the words in the file "STRING WORDS":

END? is called by a few of the other words to determine if the end of a string has been reached. It removes an address from the stack, reads the value from that address, and returns a 1 if the value is 0 or 141 (the ASCII value for a carriage return), or returns 0 otherwise.

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]
```

Remember that string indexing starts at 0 and ends at the string length-1, so the last character of the above string is character number 13.

**LEFT$** is similar to the Applesoft "LEFT$" function. The form for LEFT$ is:

<source> <destination> <# of characters> **LEFT$**

LEFT$ copies the given number of characters from the source string to the destination string. In the following example, the string TESTSTRING is read, then the first 5 characters of TESTSTRING are assigned to PAD:

```
Ready 0 TESTSTRING READLN
ELIZABETH

Ready 0 TESTSTRING  PAD  5  LEFT$

Ready PAD WRITELN
ELIZA
```

**RIGHT$** is similar to Applesoft's "RIGHT$". The form is the same as for LEFT$, however the given number of characters are copied from the right end of the string. Continuing from the previous example, 4 characters from the right end of TESTSTRING are now assigned to PAD, overwriting its previous contents:

```
Ready 0 TESTSTRING  PAD  4  RIGHT$

Ready PAD WRITELN
BETH
```

Notice that with GraFORTH's string indexing, the Applesoft function "MID$" can be duplicated with LEFT$. This example reads 3 characters from TESTSTRING starting with the character number 1 (not 0):

```
Ready 1 TESTSTRING  PAD  3  LEFT$

Ready PAD WRITELN
LIZ
```

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

<source> <destination> **MOVELN**

The following example copies the contents of TESTSTRING to PAD:

```
Ready 0 TESTSTRING  PAD  MOVELN

Ready PAD WRITELN
ELIZABETH
```

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 TESTSTRING, then CONCAT is used to combine the strings in PAD:


```
Ready PAD READLN
GraFORTH:

Ready 0 TESTSTRING READLN
The Apple Graphics Language

Ready PAD  0 TESTSTRING  CONCAT

Ready PAD WRITELN
GraFORTH: The Apple Graphics Language
```

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 0. Here is an example:

```
Ready PAD ASSIGN " LIST "

Ready 0 TESTSTRING ASSIGN " LOST "

Ready PAD  0 TESTSTRING  COMPARE
[-1]
```

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

Words Manipulating Individual Characters

GraFORTH also contains words that print individual characters to the screen, and get individual characters from the keyboard. These words interpret numbers as the ASCII values for characters. (A table of ASCII characters can be found in Appendix D.)

The GraFORTH word "**PUTC**" (PUT Character) prints a single character to the screen. PUTC removes a number from the stack, interprets it as the ASCII number for a character, and prints the character at the current cursor position:

```
Ready 193       (193 is the ASCII value for the letter "A".))
[193]
Ready PUTC
A
```

PUTC removed the 193 from the stack and printed the characteer "A".

The GraFORTH word GETC (GET Character) places a flashing cursor on the screen, waits for a character from the keyboard to bee entered, then places its ASCII value on the stack:

```
Ready GETC
```

(Type the character "B".)

```
[194]
```

(GETC returns 194, the ASCII value for the character "B".)

To print a character read in with GETC, simply DUPlicate thee value read, and write it to the screen with PUTC:

```
Ready GETC DUP PUTC
```

(Type the character "Y".)

```
Y
[217]
```

(217 is the ASCII value for the character "Y".)

To check if a key has been pressed without stopping to wait,, "GETKEY" and "CLRKEY" are used. GETKEY and CLRKEY directlyy use the Apple's special keyboard memory location.

When a key is pressed, its Apple ASCII value is stored in thhe Apple keyboard location. If a key has been pressed, the nummber in this location is always 128 or greater. GETKEY reads thiis location and places its value on the stack. Executing CLRKEEY forces the value in the keyboard location to less than 128.. The next keypress after CLRKEY is executed will again bring thee value to 128 or greater.

Thus, to read the keyboard using GETKEY and CLRKEY, first execute CLRKEY to make the keyboard location less than 128, then use GETKEY until the returned value is 128 or greater. This number will be the ASCII value for the key that is pressed. GETKEY can be interspersed with other tasks so that other things can occur while simultaneously reading the keyboard. Here is a simple example that uses GETKEY and CLRKEY to "grab a character":

```
: GRAB.CHAR
  CLRKEY
  BEGIN
    GETKEY DUP
    128 <
  WHILE
    DROP
  REPEAT
  CLRKEY ;
```

Using Numbers in Other Bases

GraFORTH can accept and display number in bases other than base ten. Four words (**HEX**, **BINARY**, **DECIMAL**,and **BASE**) allow you to select what base GraFORTH uses.

The word "**HEX**" causes GraFORTH to read and print numbers in hexadecimal, base 16. In this example, a number is placed on the stack, then base 16 is selected using HEX.

```
Ready 45
[45]

Ready HEX
[2D]
```

Similarly, the word "**BINARY**" selects base two:

```
Ready BINARY
[101101]
```

The GraFORTH word **DECIMAL** gets us back to familiar territory:

```
Ready DECIMAL .
45
```

The word "**BASE**" can be used to select any base. BASE acts as a variable: the word "->" is used to assign the base. The following selects base 8 (octal):

```
Ready 8 -> BASE
```

Note that since BASE is a variable, its current value can be read and displayed. However, any base value displayed in its own base is "10". For example, a 2 in base 2 is 10, and a 16 in hexadecimal is also 10. Thus, to print the base, you must place its value on the stack, change BASE to some other base, then print the stack value. In this short example, the base selected above is displayed before and after changing back to decimal:

```
Ready BASE
[10]

Ready DECIMAL
[8]
```

Because hexadecimal and some other base numbers use letters of the alphabet as digits, possible conflicts between numbers and word names may occur. For example,, in hexadecimal, is "ACE" a GraFORTH word name or a number? To help prevent this, GraFORTH allows dollar signs ("$") to precede numbers:

```
Ready HEX

Ready $ACE
[ACE]
```

Note: All of the examples in this manual have assumed that base ten is selected. In addition, some of the programs on the GraFORTH system disk have number formatting that requires base ten. You are free to use other bases, but the results may be quite unpredictable!

Using DOS From GraFORTH

DOS Commands

Using the Apple Disk Operating System from GraFORTH is much the same as from Basic. DOS commands can be called directly from GraFORTH, either from the keyboard or in a word definition. DOS responds to a command that has been preceded by a carriage return and a ConTRoL-D (ASCII number 132). (See the Apple DOS manual for more information on disk access in general.) The form for a DOS command from GraFORTH is:

```
CR 132 PUTC PRINT " <DOS command> " CR
```

"CR" prints a carriage return and "132 PUTC" prints a ConTRoL-D. The DOS command is printed next, and the line is ended with another carriage return. Here is an example that prints a catalog:

```
Ready CR 132 PUTC PRINT " CATALOG " CR
```

Using Data Files

Text file access is also similar to Basic. The file is opened using standard DOS commands, and data can be read from or written to the file using READLN or WRITELN. File access can be simplified by defining file words ahead of time. For example, to begin reading from a text file, you can use a word like OPEN.READ. (The filename has been stored in PAD.):

```
: OPEN.READ
  CR 132 PUTC PRINT " OPEN " PAD WRITELN CR
  CR 132 PUTC PRINT " READ " PAD WRITELN CR ;
```

After executing this word, the file will be opened for reading, and data can be read in using READLN. At the end of the text, the file can be closed by simply using the GraFORTH word "CLOSE". CLOSE closes any open file.

Since GraFORTH does not have a function similar to Applesoft's "ON ERROR GOTO", DOS errors, including End Of Data, will produce an error message and stop the program. This means that either the length of the file must be known ahead of time, or there must be a special marker at the end of the file so that no more data will be read by the program. The last character in the file must also be a carriage return.

Here is a sample file that makes use of a special End Of File marker. The marker used here is an asterisk on the last line:

```
This is my test file.
Each of these lines will be printed
by the program below.
The last line must be a special marker
to end the file.  Here it is:
*
```

Let us say that we have saved this file with the name "TEST". Here is a program that will read and print each line in the file, and will stop when it encounters the end marker "*":


```
: READER
  PAD ASSIGN " TEST "     (Place filename in PAD and call)
  OPEN.READ               (OPEN.READ from above to open file.)
  BEGIN
     PAD READLN           (Read a line from file.)
     PAD PEEK             (Get first character from line.)
     170 <>
  WHILE                   (WHILE this character is not "*": )
     PAD WRITELN          (Write the line to the screen, and)
  REPEAT                  (REPEAT back for the next line.)
  CLOSE ;                 (Close the file.)
```

As the special GraFORTH DOS allots only one file buffer, only one file can be open at a time. The DOS commands "PR#n" and "IN#n" (where n is a number from 1 to 7) can be used from GraFORTH to route data to and from peripheral cards in the back of the Apple. In this way, program text or data can be sent to a printer or other peripheral. After using "PR#n" or "IN#n", either the GraFORTH word GR or TEXT can be typed to re-establish the standard GraFORTH I/O. Do not attempt to use "PR#0" as it will not leave GraFORTH intact.

The following word will print the text in the editor buffer to a printer in slot 1. It reads the characters one at a time and prints them out until it finds a zero byte, marking the end of the editor file.

```
: PRINT.BUFFER
  CR 132 PUTC PRINT " PR#1 " CR
  34817
  BEGIN
     DUP PEEK DUP
     0 <>
  WHILE
     PUTC
     1 +
  REPEAT
  GR ;
```