GSoft™ BASIC 1.0

A BASIC Interpreter
for the
Apple IIGS

Mike Westerfield

Byte Works®, Inc.
8000 Wagon Mound Dr. NW
Albuquerque, NM 87120-2845
Voice (505) 898-8183
FAX (505) 898-4092
E-Mail MikeW50@AOL.COM

Credits

GSoft BASIC Interpreter
    Mike Westerfield

Documentation
    Mike Westerfield

Beta Testers
    Jeff Blakeney
    Richard N. Cain
    Gareth Jones
    Glen Gunselman
    Charles Hartley
    Eric Shepherd
    Timothy Tobin

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

xiv

Table of Contents

Table of Contents

Chapter 1 – Introducing GSoft BASIC

Welcome to GSoft BASIC! GSoft BASIC is a complete programming environment for writing programs on the Apple IIGS. You get a simple, easy to use BASIC interpreter that is, in many important ways, one of the most sophisticated implementations of BASIC ever produced. There is a second version of the interpreter that executes from the popular ORCA programming environment, perfect for toolbox programming or for people who want a more complete (although harder to learn) programming environment. And, of course, there is a utility that creates GSoft BASIC applications that will run from the Finder, even on computers where GSoft BASIC is not installed.

While the GSoft BASIC environment was deliberately kept simple, it's still a cut above the old Applesoft BASIC environment. After all, GSoft BASIC is designed to run on the Apple IIGS, not an Apple II with as little as 16K of RAM and 12K of ROM. We had room to add many nice features not found in Applesoft BASIC, like a full screen editor.

GSoft BASIC is the only Apple IIGS BASIC—for that matter, the only BASIC we're aware of on any platform—that gives full, natural access to the toolbox. The reason is simple: Toolbox programming requires records and pointers. Some modern BASICs support records, but none we are aware of support pointers with the natural grace of C or Pascal. GSoft BASIC does.

After purchasing a new program, you would probably like to sit right down at your computer and try it out. We encourage you to do just that, and in fact, this manual is designed to help you. Before getting started, though, we would like to take some time to suggest how you should approach learning to use GSoft BASIC.

About the Manual

This manual is your guide to GSoft BASIC. To make it easy for you to learn about the system, this manual has been divided into three major sections. The first part is called the *User's Guide*. It is a tutorial introduction to the development environment, showing you how to create BASIC programs using GSoft BASIC. The second part is called the *Environment Reference Manual*. It is a working reference to provide you with in-depth information about the development environment you will use to create GSoft BASIC programs. Part three is the *Language Reference Manual*. It contains information about the GSoft BASIC programming language.

Regardless of your programming background, we recommend reading all of this chapter and the next, along with any portions of Chapter 3 that interest you. You should skim the major headings for the remainder of the book so you know what information is available, and approximately where to find it. Spend some time browsing as you do this; you'll find many commands and features that you probably wouldn't anticipate in a BASIC.

While this manual will teach you how to use GSoft BASIC to write and test programs, it is not a programming tutorial. It is primarily a reference manual, giving you a comprehensive guide to GSoft BASIC in a format that makes it easy to look up specific information. Basic concepts

1

about programming in GSoft BASIC are necessary to create useful, efficient programs. If you are already familiar with programming in some other language, especially another dialect of BASIC, this reference manual is probably all you will need to begin writing your own programs. If you are new to BASIC, you can start with our *Learn to Program in GSoft BASIC* course, which is written specifically for GSoft BASIC. You'll find more details about this course, and several other books that may be of interest, at the end of this chapter.

Other Books and Reference Materials

This section lists a lot of books, but don't be intimidated by the list. You don't actually need any of them to use GSoft BASIC, and very few people will ever use all of them. This list is here to give you some ideas for further exploration, not as a list of required books!

If you are new to BASIC, you will need to supplement this manual with a good beginner's book on the BASIC programming language. A companion course is available from the Byte Works that teaches you the BASIC language and some basic techniques for programming. The book is called *Learn to Program in GSoft BASIC*, and it has one distinct advantage over any other BASIC programming book: it is written specifically for GSoft BASIC running on an Apple IIGS.

Unlike C and Pascal, there is no widely accepted language standard or a common core set of extensions to a standard language for BASIC. That makes using general BASIC programming books tricky, but not impossible. In general, books that are not written for a specific dialect of BASIC should be fairly easy to use with GSoft BASIC. You'll find many such books in your local library or through on-line bookstores, or by special order from local bookstores. The recent trend for books on bookstore shelves is towards specific implementations of BASIC, though. Most of these books are not suitable for use with GSoft BASIC.

If you would like to learn to write Apple IIGS toolbox programs with windows and pull down menus, we suggest *Toolbox Programming in GSoft BASIC*, which is a complete introduction to the world of toolbox programming. You will eventually need a copy of the Apple IIGS Toolbox Reference, volumes 1 through 3, and *Programmer's Reference for System 6.0.1*, but these aren't needed right away. These books do not teach you about the toolbox, but they are essential references for advanced toolbox programming. Depending on the kind of programming you are doing, you may also need other reference books. The common ones are listed below. You will also need some way to create resource forks. One way is Apple's Rez compiler, which ships with all ORCA languages and with *Toolbox Programming in GSoft BASIC*.

Learn to Program in GSoft BASIC

Mike Westerfield

Byte Works, Inc., Albuquerque, New Mexico

As this manual is prepared, this book has not been released, but is planned. It will be based on an existing introductory programming course in programming which has been used by thousands of people to learn Pascal and C.

This introductory BASIC programming course is written specifically for GSoft BASIC running on an Apple IIGS. It contains hundreds of complete programs as examples, as well as problems with solutions.

Toolbox Programming in GSoft BASIC

Mike Westerfield

Byte Works, Inc., Albuquerque, New Mexico

As this manual is prepared, this book has not been released, but is planned. It will be based on an existing introductory programming course in programming which has been used by thousands of people to learn toolbox programming in Pascal and C.

This is the only self-paced course available for programming the Apple IIGS toolbox. Unlike the toolbox reference manuals, this is a course that teaches you how to write programs, not a catalog of the various toolbox calls available on the Apple IIGS. It includes four disks filled with toolbox source code, as well as an abridged toolbox reference manual, so you won't have to buy all of the toolbox reference manuals right away. It also comes with Apple's Rez compiler and the full version of the ORCA shell.

Celestial BASIC: Astronomy On Your Computer

Eric Burgess

Sybex, Berkeley, CA, 1982

This is one of my favorite programming books of all time. If you're at all interested in astronomy, it's worth the effort to find a copy of this classic book. It has a great collection of simple BASIC programs that perform a wide variety of calculations, like planet positions, moon phases, dates for Easter, and so forth.

BASIC Computer Games

David H. Ahl, Ed.

Workman Publishing, New York, 1978

An amazingly diverse collection of short BASIC games. This old book is worth chasing down, too.

Technical Introduction to the Apple IIGS

Apple Computer

Addison-Wesley Publishing Company, Inc., Reading, Massachusetts

A good basic reference source for the Apple IIGS.

Apple IIGS Hardware Reference

Apple IIGS Firmware Reference

Apple Computer

Addison-Wesley Publishing Company, Inc. Reading, Massachusetts

These manuals provide information on how the Apple IIGS works.

Apple IIGS Toolbox Reference: Volume I

Apple IIGS Toolbox Reference: Volume II

Apple IIGS Toolbox Reference: Volume III

Apple Computer

Addison-Wesley Publishing Company, Inc. Reading, Massachusetts

These volumes provide essential information on how the tools work—the parameters you need to set up and pass, the calls that are available, etc. You must have these books to use the Apple IIGS toolbox effectively.

Programmer's Reference for System 6.0.1

Apple Computer

Byte Works, Inc., Albuquerque, New Mexico

The first three volumes of the toolbox reference manual cover the Apple IIGS toolbox up through System 5. This book covers the new features added to the toolbox and GS/OS in System 6.

Apple IIGS GS/OS Reference

Apple Computer

Addison-Wesley Publishing Company, Inc., Reading, Massachusetts

This manual provides information on the underlying disk operating system. You will need this reference if you want to bypass GSoft BASIC's build-in file handling commands. You might want to do that for greater efficiency, more control, or to access file and disk handling features that are not built into GSoft BASIC.

ORCA/M: A Macro Assembler for the Apple IIGS

Mike Westerfield and Phil Montoya

Byte Works, Inc., Albuquerque, NM

ORCA/M is the standard macro assembler for the Apple IIGS. While you cannot mix other languages directly with GSoft BASIC, you can call user tools, known as libraries, from GSoft BASIC. ORCA/M is the ideal choice for writing user tools.

4

Chapter 2 – Getting Started

This chapter describes the three major components of GSoft BASIC, helping you decide which environment you want to use. It also describes installing and starting GSoft BASIC.

Setting Up GSoft BASIC

Backups

As with any program, the first step you should take is to make a backup copy of the original disks. To do this, you will need two blank disks and a copy program—Apple's Finder will do the job, or you can use any other copy program if you have a personal favorite. If you are unfamiliar with copying disks, refer to the documentation that came with your computer.

As always, copies are for your personal use only. Using the copies for any purpose besides backing up your program is a violation of federal copyright laws. If you will be using GSoft BASIC in a classroom or work situation where more than one copy is needed, please contact the Byte Works, Inc. for details on our licensing policies.

Registration

From time to time, we make improvements to GSoft BASIC. You should return your registration card so we can notify you when the software is improved. We also notify our customers when we release new products, often offering substantial discounts to those who already have one of our programs.

System Requirements

To use GSoft BASIC, you will need an Apple IIGS with at least 1.125M of memory for ROM 3 machines, or 1.25M of memory with ROM 1 machines. You can actually run GSoft BASIC with less memory, but we don't recommend less.

You will also need at least two disk drives, and at least one of those must be a 3.5" disk drive. To use all of the features and utilities included with GSoft BASIC, you will need a second 800K floppy disk drive or a hard drive with 1.5M of free space.

The MakeRuntime utility requires System 6.0 or System 6.0.1. GSoft BASIC itself can be used with System 5.0.4, although we recommend System 6.0.1. Programs written in GSoft BASIC require System 5.0.4 or better, although they obviously need a later operating system if the program itself makes calls that only exist in the later O/S.

GSoft BASIC supports color monitors, printers and accelerator cards, but does not require them.

Running GSoft BASIC

You do not need to do any initialization to use GSoft BASIC. After booting your computer, insert the disk labeled *GSoft BASIC* in your 3.5" disk drive and run GSoft.Sys16. After a few moments you will be in the GSoft BASIC shell, ready to write programs. If you're already familiar with Applesoft BASIC, you'll be able to write programs immediately—but be sure and scan the manual, because there's a lot more to GSoft BASIC!

Using GSoft BASIC from Floppy Disks

If you're a bare-bones minimalist, you can actually run GSoft BASIC from a single 3.5" floppy disk. Here's what you need to do:

- Make a copy of the system disk that you use now to boot your computer.
- Erase the Finder, ProDOS 8, the tools, control panels and fonts from this copy. The specific files to delete from the System 6.0.1 boot disk are shown below; this list will be slightly different for other version of the operating system. For directories, leave the directory intact, but delete the contents.

  ```
  Finder
  P8
  CDevs:
  desk.accs:ControlPanel
  Tools:
  Fonts:
  ```

- Copy GSoft.Sys16 from the *GSoft BASIC* disk to your boot disk.

  This gives you a disk that will boot directly into GSoft BASIC. It leaves plenty of room for your programs.

  You'll probably want to copy the full screen editor to your disk, too. It's not essential—you can enter programs with line numbers without it—but a full screen editor is almost too nice to consider doing without. To add the full screen editor, copy the folder named Shell and all its contents to your boot disk.

  Finally, you might want to copy the file GSoftTools.gst from the *GSoft BASIC* disk. This gives you access to the Apple IIGS toolbox. You may not want to write toolbox programs, but QuickDraw II is nice even for simple graphics programs.

The disk you've just built is a perfect system for writing short programs. If you added the tool interface file, GSoftTools.gst, you can work through *Learn to Program in GSoft BASIC* with this disk.

Installing GSoft BASIC on a Hard Disk

If you have a hard disk, you should install GSoft BASIC on your hard disk. It will run faster, and you won't have to look for utilities on the second disk.

Start by running the Installer from the *GSoft BASIC* disk. There are four installer scripts:

GSoft BASIC

Use this option if you do not have any other ORCA languages. This installs all of GSoft BASIC and the utilities that are not designed to run under the ORCA shell.

GSoft BASIC for ORCA

Many people already own an ORCA programming language or another product that includes an ORCA compatible shell. While GSoft BASIC does not come with a copy of the ORCA shell, you can install it in an ORCA compatible shell if you already have one.

Use this option if you have any ORCA language with a version number of 2.0 or greater. This option installs all of the GSoft BASIC utilities designed to run under the ORCA shell.

Installing GSoft BASIC changes your SysCmnd and SysTabs files. These files are commonly customized as you add languages and utilities. The files supplied with GSoft BASIC contain appropriate command and tab settings for ORCA/M, ORCA/Pascal, ORCA/C, ORCA/Modula-1 and ORCA/Integer BASIC, as well as GSoft BASIC. Including command and tab settings for languages you do not own does not cause problems, but installing this file could wipe out any custom changes you have made. If you have made changes, make a copy of your SysCmnd and SysTabs file before installing GSoft BASIC. After Installing GSoft BASIC, replace your original files, then make the following changes by hand.

Add these lines to your ORCA:Shell:SysCmnd file, placing them in alphabetical order compared to the existing entries:

```
BASIC        *L    260    GSoft BASIC
COMPILETOOL   U           GSoft BASIC Tool Compiler
```

You'll also need a new tab line in the ORCA:Shell:SysTabs file. Each language uses three lines. The easiest way to create the three lines for GSoft BASIC is to start by copying the three existing lines for language 4. Because the last line is long, it will look like four lines on the screen. The lines you need to copy will look like this:

7

```
4
10011001
000000010000000100000000100000000010000000
100000001000000001000000000100000000100000000010000000
100000000100000000010000000100000000010000000
100000001000000002
```

After copying these lines, paste them in your SysTabs file at the proper location for language number 260. The language number is the first line of the three line set, or 4 in the sample you see above. Change the language number to 260 and save the file.

If you are using an older version of ORCA, the second line may have fewer characters on the second line—10011001 in the example. You can use the shorter version you see in your existing SysTabs file or add all of the characters you see in the example; it won't matter at all to the existing editor.

GSoft BASIC for ORCA and Finder

This option installs both of the versions of GSoft BASIC described above. Use this option if you want to use GSoft BASIC from the Finder and from the ORCA shell.

GSoft.Sys16, the version of GSoft BASIC that runs from the Finder, is installed in the ORCA folder. This saves a little space, since the editor, samples, and some utilities are not installed twice. You should still modify the SysCmnd and SysTabs files as described above.

.PRINTER Driver

This installs the .PRINTER driver, a GS/OS driver and accompanying Init, CDev and CDA that allows you to print to any standard text printer from GSoft BASIC. Install this driver for all versions of GSoft BASIC. See Chapter 4 for a description of the driver.

The Three Worlds of GSoft BASIC

The GSoft BASIC Shell

This manual describes using GSoft BASIC from the GSoft BASIC shell, a complete, self contained programming environment that launches directly from the Finder. It's essentially the classic Applesoft BASIC programming environment on steroids. Most of the familiar old commands are there, along with some nice new ones, like the full screen editor.

Incidentally, shell is the name for a program that lets you type commands like CATALOG, then carries them out.

This is a great programming environment for hacking out quick solutions to problems, learning to program, and writing mid-sized text and graphics programs. It starts to fall a little short

if you decide to write desktop programs, mostly because you need some way to create resource forks, and you can't do that from within the GSoft BASIC shell.

Using GSoft BASIC from the ORCA Shell

The ORCA programming languages ship with a much more advanced shell. There are three advantages of this shell over the GSoft BASIC shell. First, the ORCA shell supports Apple's Rez compiler, which is one way to create resource forks for desktop programs. The second advantage is that the ORCA shell doesn't reformat your programs like the GSoft BASIC shell. (Some people may not consider that an advantage!) The program is saved in an ORCA SRC file, which saves the source code as ASCII characters; these source files are left in exactly the format you type them. Finally, if your program needs user tools, you can develop the user tool and the GSoft BASIC application from the ORCA shell.

There are several disadvantages, though. First, the ORCA shell is larger, so it needs more RAM and disk space. It's also harder to learn to use, mostly because of the sheer number of commands and features. GSoft BASIC programs start slower, because programs are stored as plain ASCII files, which must be converted to GSoft BASIC files before they can be executed. This conversion process more than doubles the amount of memory required, too.

On balance, we recommend using the GSoft BASIC shell for most GSoft BASIC programming. Switch to the ORCA shell for extremely large programs, toolbox programs, or programs that need user tools.

The ORCA shell ships with ORCA/M, ORCA/C, ORCA/Pascal, ORCA/Modula-2 and *Toolbox Programming in GSoft BASIC*.

Creating GSoft BASIC Programs that Run From the Finder

You may run most, if not all, of your programs directly from the GSoft BASIC shell, and never feel the need to launch them directly from the Finder. The MakeRuntime utility lets you convert the program to a form that will launch from the Finder, though.

Converting the program to run from the Finder adds about 90K to the size of the disk file. The RAM used to run the program actually drops, but it's not a significant difference. These programs can also run on computers where GSoft BASIC is not installed.

Programs converted to run from the Finder can't be changed. You need to write and change programs from within one of the two programming environments described above.

Chapter 3 – Programming on the Apple IIGS

The Apple IIGS is a very flexible machine. With it, you can write programs in a traditional text environment, in a high-resolution graphics environment, or in a Macintosh-style desktop environment. GSoft BASIC lets you write programs for all of these environments.

In this chapter, we will look at each of the programming environments in turn, examining how you use GSoft BASIC to write programs, what tools and libraries are available, and what your programs can do in each of the environments. This chapter assumes you are typing and running the programs as you go.

Text Programs

Text programs are by far the easiest kind of programs to write. As an example, we'll create a simple text program to show how many payments will be needed to pay off a loan for any given interest rate, loan amount, and payment. The variables are placed at the top of the program as constants, so there is no input.

This is actually the first time we have created a program from scratch in this manual, so we will go over the steps involved fairly carefully. If you aren't in the GSoft BASIC programming environment, start it now by launching GSoft.Sys16 from the Finder. You'll see a header and a } character followed by the cursor; from here you can enter various commands. The commands are described in Chapter 5, but we'll go over all of the ones you will use as we write the program.

Start by typing EDIT and pressing the return key. This puts you into the GSoft BASIC full screen editor. Type in the program shown below. The basic commands in the editor are pretty similar to most Apple IIGS text based editors, so you may not have any trouble using it. If you need to look up a specific editor command, glance through Chapter 6. Be sure and give Command-? a try—this displays an on-line help screen with common commands.

Spacing and letter case generally aren't too critical. GSoft BASIC will reformat the program and convert identifiers to uppercase when you exit the editor. Also, all of the lines that start with an exclamation point are comments; you can skip them if you'd like. The same is true of the DIM statements which end with a colon and an exclamation point. The colon lets you add a new statement to the same physical line, something this program uses to describe how the variables are used. You can leave off everything from the colon to the end of the line.

With those exceptions, though, be sure to type the program exactly as you see it, especially if you are new to programming. BASIC is flexible when it reads your program, but until you know what changes are allowed, stick with formatting that you know works!

Although the point of this example is to show you how to type in a program from scratch, it's only fair to point out that the following example is also on the GSoft disk in the folder :GSoft:Samples:Text.Samples directory. If you have installed GSoft BASIC on a hard disk, the file is in the Text.Samples folder there, too.

```
!-------------------------------------------------------------------
!
!  Finance
!
!-------------------------------------------------------------------
!
!  This program prints the balance on an
!  account for monthly payments, along with the
!  total amount paid so far.
!
!-------------------------------------------------------------------
!
LOANAMOUNT = 10000.0:! amount of the loan
PAYMENT = 600.0:! monthly payment
INTEREST = 15:! yearly interest (as %)
!
DIM BALANCE:! amount left to pay
DIM MONTHLYINTEREST:! multiplier for interest
DIM PAID:! total amount paid
DIM MONTH AS INTEGER :! month number
DIM USE$:! format string
!
! set up the initial values
!
BALANCE = LOANAMOUNT
PAID = 0
MONTH = 0
MONTHLYINTEREST = 1.0 + INTEREST / 1200.0
!
! write out the conditions
!
PRINT USING "Payment schedule for a loan of $$#####.##";LOANAMOUNT
PRINT USING "with monthly payments of $$##.##  at an";PAYMENT
PRINT USING "interest rate of #%.";INTEREST
PRINT
PRINT "   month    balance    amount paid"
PRINT "   -----    -------    -----------"
USE$ = "   ###############   $$############.##   $$############.##"
!
! check for payments that are too small
!
IF BALANCE * MONTHLYINTEREST - BALANCE >= PAYMENT THEN
   PRINT "The payment is too small!"
```

```
ELSE
  WHILE BALANCE > 0
    ! add in the interest
    BALANCE = BALANCE * MONTHLYINTEREST
    ! make a payment
    IF BALANCE > PAYMENT THEN
      BALANCE = BALANCE - PAYMENT
      PAID = PAID + PAYMENT
    ELSE
      PAID = PAID + BALANCE
      BALANCE = 0
    END IF
    ! update the month number
    MONTH = MONTH + 1
    ! write the new statistics
    PRINT USING USE$;MONTH, BALANCE, PAID
  WEND
END IF
```

Once the program is typed in, you should save it to disk. You may think you saved it when you left the editor; after all, the editor asked you if you wanted to save the changes. You only saved the working copy GSoft BASIC used, though—there is no permanent copy on disk.

There are three different save commands; you choose one based on whether you want to save the program in the natural format for GSoft BASIC, as a text file, or in the natural format for compiled programming languages. As a general rule, it's best to save the program in GSoft BASIC's preferred format using the SAVE command. To save your program in the same folder as GSoft BASIC itself, type

```
SAVE Finance
```

Of course, you can use whatever file name you prefer. If your disk is a ProDOS format disk, file names must start with a letter, and can contain up to 15 characters. The remaining characters can be letters, numbers or periods.

If you'd like to explore these commands in more depth, start with Chapter 5. The three save commands are SAVE, TSAVE and SSAVE. You'll also find details about the file formats in *Types of Files Used by GSoft BASIC*, near the beginning of the chapter. *File Names* in Chapter 14 goes into detail about valid file names. These chapters also discuss navigation through the directory structure of a typical disk, as well as how to type full and partial path names. You can safely skip all of those details until later, saving your programs in the same directory as GSoft BASIC, but even if you don't look into these topics now, it's good to know where the information is when you do need it.

With the program safely on a disk, protected from all but the most severe accident, it's time to run your program. Type:

```
RUN
```

If you typed everything correctly, you'll see the results scroll across the screen. With the figures shown, some of the information will scroll off of the screen; you can change the interest rate, the size of the payment, or the amount of the loan to see the top lines.

If you didn't type everything correctly you'll see some form of error message. Type EDIT to get back to the editor, make any necessary changes, leave the editor, then try the program again. Remember to SAVE occasionally to guard against disaster.

One of the classic interactive computer games of all time will serve as our second example, giving us a chance to explore text input and accessing the Apple IIGS toolbox. In this simple game, the computer will pick a distance to a target, and you pick a firing angle for a cannon. The computer then lets you know if you hit the target, or if you missed and by how much.

```
! ---------------------------------------------------------------
!
!            Artillery
!
! This classic interactive text game lets you
! pick the angle of your artillery gun in
! an attempt to knock out the enemy position.
! The computer picks a secret distance. When
! you fire, you will be told how much you
! missed by, and must fire again. The object
! is to hit the target with the fewest shells.
!
! ---------------------------------------------------------------
!
BLASTRADIUS = 50.0:! maximum distance from target for a hit
DTR = 0.01745329:! convert from degrees to radians
VELOCITY = 434.6:! muzzle velocity
!
! choose a distance to the target
!
DISTANCE = RND (1) * 5900.0
!
! not done yet...
!
DONE = 0
TRIES = 1
!
! shoot 'til we hit it
!
DO
!
! get the firing angle
!
INPUT "Firing angle: ";ANGLE
```

```
! compute the muzzle velocity in x, y
!
ANGLE = ANGLE * DTR
VX = COS (ANGLE) * VELOCITY
VY = SIN (ANGLE) * VELOCITY
!
! find the time of flight
! (velocity = acceleration * flightTime, two trips)
!
FLIGHTTIME = 2.0 * VY / 32.0
!
! find the distance
! (distance = velocity * flightTime)
!
X = VX * FLIGHTTIME
!
! see what happened...
!
IF ABS (DISTANCE - X) < BLASTRADIUS THEN
    DONE = 1
    PRINT "A hit, after ";TRIES;
    IF TRIES = 1 THEN
        PRINT " try!"
    ELSE
        PRINT " tries!"
    END IF
    SELECT CASE TRIES
        CASE 1
            PRINT "(A lucky shot...)"
        CASE 2
            PRINT "Phenomenal shooting!"
        CASE 3
            PRINT "Good shooting."
        CASE ELSE
            PRINT "Practice makes perfect - try again."
    END SELECT
ELSE IF DISTANCE > X THEN
    PRINT USING "You were short by # feet.";DISTANCE - X
ELSE
    PRINT USING "You were over by # feet.";X - DISTANCE
END IF
TRIES = TRIES + 1
LOOP WHILE NOT DONE
```

When you run the program, you will see a prompt for the firing angle followed by a white box. This white box is the cursor used by interactive text programs. It lets you know that input is expected by the program. If you make a mistake, you can use the delete key to back space over

15

your input. In fact, you can use all of the line editing commands available when you are typing lines in the GSoft BASIC shell. These commands are summarized in Chapter 5, *The Line Editor*.

If you get stuck in the middle of the program, or just get bored with it, you can stop the program by holding down the control key and pressing C or by holding down the command key (the one with the clover shape) and pressing the period key.

Console Control Codes

When you are writing text programs that will execute on a text screen, one of the things you should know about are the console control codes. These are special characters that, when written to the standard text output device, cause specific actions to be taken. Using console control codes, you can beep the speaker, move the cursor, or even turn the cursor off. The console control codes are covered in Appendix B.

Keep in mind that these console control codes only work with the text screen. You can write text to a variety of places, such as printers, the graphics screen, or disk files, but the console control codes only work with the text display device. There is more than one console, too. The codes shown in Appendix B apply to the console driver built into GSoft BASIC. If you are running GSoft BASIC from the ORCA shell, or from some other ORCA compatible shell, you should refer to the documentation for that shell to find out which console it uses and what control codes are available.

Stand-Alone Programs

So far, our examples have executed from GSoft BASIC's shell, or perhaps you installed GSoft BASIC in the ORCA shell and executed it from there. Either way, the program can't be used by someone who does not own GSoft BASIC.

There is a third version of GSoft BASIC that is designed to run from the Finder, but it isn't one you can use to write programs. Instead, you use the MakeRuntime utility, described in the next chapter, to create an application you can run from the Finder using a program you've already written. Of course, these programs can also run from the Finder using a program you've already written. Of course, these programs can also run from other program launchers, as long as they can run programs designed to run from Apple's Finder.

You might think that you have to write a program with pull down menus, windows, and the whole toolbox interface before it can run from the Finder, but that simply isn't true. GSoft BASIC runs from the Finder, after all, and it uses a text interface. The fact is, you can run any GSoft BASIC program from the Finder after attaching the run-time module to the program with MakeRuntime. The Artillery program you just typed in is a great program to try this with.

There is one, and only one, problem you have to keep in mind when converting text programs to run under the Finder. In a text environment like GSoft BASIC's shell, you generally end a text program by simply letting it finish, just like our Finance program did. That works well in a text environment, but not from the Finder. As soon as the program finishes, it returns to the Finder, and you can't see the text screen anymore! If a program presents information and quits, like the Finance example, be sure to use a statement like

at the end of the program.

```
INPUT "Press the RETURN key to return to the Finder."; A$
```

Graphics Programs

A large subset of programs need to display graphics information of some kind, but aren't necessarily worth the effort of writing a complete desktop program. These include simple fractal programs, programs to display graphs, slide show programs, and so forth. In this book, these programs are called graphics programs.

Your First Graphics Program

Writing a graphics program with GSoft BASIC is really quite easy. In general, all you have to do is switch to the graphics display with the HGR statement and issue QuickDraw II commands. QuickDraw II is the largest and most commonly used tool in the Apple IIGS toolbox, so it's also a good place to get started along the road to writing desktop programs. For extremely simple tasks, like drawing fractals or plotting graphs, you might even be able to get by with the commands built into GSoft BASIC. They are described in Chapter 15.

If you want to use the extended graphics commands available in QuickDraw II, you will need a copy of *Apple IIGS Toolbox Reference, Volume 2*. This book was written by Apple Computer, and is published by Addison Wesley: reprints are available from the Byte Works, Inc. While the toolbox reference manual is a reference, and thus not an easy book to read, it is essential that you have a copy to answer your specific questions about the toolbox. This section shows a couple of examples so you know how to create graphics programs using GSoft BASIC, but there is a lot more to QuickDraw II than you see here.

Our first QuickDraw II sample, which draws spirals on the graphics screen, shows the commands MOVETO, which initializes the place where QuickDraw II will start drawing from (called the pen location), and LINETO, which draws a line from the current pen location to the specified spot, moving the pen location in the process.

```
HGR
THETA = 0.0
R = 100.0
MOVETO (320, 100)
WHILE R > 0.0
THETA = THETA + 3.1415926535 / 20.0
LINETO (CINT (COS (THETA) * R * 1.6) + 160, CINT (SIN (THETA) * R) + 100)
R = R - 0.15
WEND
GET S$
```

17

Save the program as Spiral, then run it. Keep in mind that the program waits for you to press a key after it finishes; this gives you a chance to stare at the pretty picture before it goes away.

Programming on the Desktop

Most people we talk to want to write programs that use Apple's desktop interface. These programs are the ones with menu bars, multiple windows, and the friendly user interface popularized by the Macintosh computer. If you fall into that group of people, this section will tell you how to get started.

Anyone who tells you that writing desktop programs is easy, or can be learned by reading a few short paragraphs, or even a chapter or two of a book, is probably a descendent of someone who sold snake oil to your grandmother to cure her arthritis. It just isn't so. Learning the Apple IIGS toolbox well enough to write commercial-quality programs is every bit as hard as learning a new programming language. In effect, that's exactly what you will be doing. The Apple IIGS Toolbox Reference Manuals come in four large volumes. Most of the pages are devoted to brief descriptions of the tool calls—about one call per page. It takes time to learn about all of those calls. Fortunately, you don't have to know about each and every call to write desktop programs.

Learning the Toolbox

As we mentioned, learning to write desktop programs takes about the same amount of time and effort as learning to program in BASIC. If you don't already know how to program in BASIC, learn BASIC first! Concentrate on text and graphics programs until you have mastered the language, and only then move on to desktop programming.

This doesn't mean that you need to know everything there is to know about BASIC, but you should feel comfortable writing programs that are a few hundred lines long, and you should understand how to use records and pointers, since the toolbox makes heavy use of these features. There is a companion course for GSoft BASIC called *Learn to Program in GSoft BASIC*. It teaches you BASIC and some fundamental concepts like sorting, linked lists, and dealing with files. It's written specifically for this language and the Apple IIGS; it's a great place to start. If you would like more information about this course, contact the Byte Works, Inc.

The toolbox itself is very large. The *Apple IIGS Toolbox Reference Manual* is a three volume set that is basically a catalog of the hundreds of tool calls available to you. These three volumes cover the tools up through System 5.0; the additions in System 6.0 and 6.0.1 are covered in *Programmer's Reference for System 6.0.1*, available from the Byte Works. This four-volume set is an essential reference when you are writing your own toolbox programs. If your file input and output needs are advanced, you may also need to add *Apple IIGS GS/OS Reference*, available as a reprint from the Byte Works, Inc. A lot of people have tried to write toolbox programs without these manuals. I can't name a single one that succeeded.

A lot of people have been critical of the toolbox reference manuals because they do not teach you to write toolbox programs, but that's a lot like being critical of the Oxford English Dictionary because it doesn't teach you how to write a book. The toolbox reference manuals are a detailed,

technical description of the toolbox, not a course teaching you how to use the tools. *Toolbox Programming in GSoft BASIC* does teach you the toolbox, though. This self-paced course also includes an abridged toolbox reference manual, so you can learn to use the toolbox before you spend a lot of money buying the four volume toolbox reference manual. This course is also available from the Byte Works, Inc.

All of this is not meant to frighten you away. Anyone who can learn a programming language can learn to write desktop programs. Unfortunately, too many people approach desktop programming with the attitude, fostered by some books and magazine articles, that they can learn to write desktop programs in an evening, or at most a weekend. This leads to frustration and usually failure. If you approach desktop programming knowing it will take some time, but willing to invest that time, you will succeed.

Hardware Requirements

Most programming languages use individual tool interface files, one per tool; these interface files are often plain typed text. This takes a lot of room, and as a result, you really need a hard drive to write toolbox programs with those languages. GSoft BASIC is different. You can write toolbox programs on any computer with two 800K floppy disk drives—text programs are easy to write on a computer with just one 800K floppy disk drive. We still recommend a hard drive, though. It makes your computer boot faster, and gives you lots of workspace.

The biggest problem with toolbox programming using GSoft BASIC is that you need some way to create resource forks. One way to do this is with Apple's resource compiler, which ships with all of the ORCA languages and with *Toolbox Programming in GSoft BASIC*. The big problem with Apple's resource compiler is that it requires the ORCA shell, so you're forced to use this larger, more complicated programming environment. It also comes with *Toolbox Programming in GSoft BASIC*.

The other thing you will need to write any large program is more than 1.25M of memory. You can squeak by with 1.25M of memory for small toolbox programs, but you'll run into memory problems before long. We recommend 2M or more of memory for toolbox programming.

Chapter 4 – GSoft BASIC Utilities

This chapter describes the various utilities and support tools that come with GSoft BASIC. You'll probably never use some of these utilities, while others will be useful from almost your first program.

The .PRINTER driver is used both by BASIC itself and by the programming environment to print program listings and other pages that only contain text. You may want to look at the information about the .PRINTER driver right away.

MakeRuntime is a utility you'll probably use eventually. It takes a BASIC program and converts it into an executable program that launches directly from the Finder, even on computers where GSoft BASIC is not installed. You can safely skip it until you have a program you want to convert.

Most people will never use CompileTool, which is an advanced programmer utility used to create tool interface files for tools and user tools; and only people who already own another ORCA language can use the ORCA Shell version of GSoft BASIC. Skip these sections entirely if they do not apply to you.

Printing With the .PRINTER Driver

The operating system on the Apple IIGS gives you a number of ways to write to a printer, but none of them can be used with standard file write commands, which is the way you would write text to a printer on many other computers. On the other hand, GS/OS does allow the installation of custom drivers, and you can use GS/OS file output commands to write to a custom driver. Our solution to the problem of providing easy to use text output to a printer is to add a custom driver called .PRINTER.

There are two ways to use the .PRINTER driver from GSoft BASIC. You can print directly from your BASIC programs by opening .PRINTER as a file and writing text; this method is described in detail in *Printing*, found in Chapter 14. You can also list files to the printer with the PR command, described in Chapter 5. This section describes how to install and configure the printer driver for your particular printer.

Installing .PRINTER

.PRINTER is a RAM based driver, so it must be installed on your boot disk before you can use the driver. There is an installer script on the GSoft BASIC install disk that will copy the correct file for you.

Configuring .PRINTER

All printers are not created equal, so any printer driver must come with some method to configure the driver. By default, our printer driver is designed to handle a serial printer installed in slot 1. It prints a maximum of 80 characters on one line, after which it will force a new line, and put any remaining characters on the new line. After printing 60 lines a form feed is issued to advance the paper to the start of a new page. When a new line is needed, the driver prints a single carriage return character ($0D). If any of these options are unsuitable for your printer, you can change them using either a CDev or a CDA. Both of these programs produce a configuration file called Pinit.Options, which will be placed in your System directory, so you need to be sure your boot disk is in a drive and not write protected when you configure your printer. This file is read by an init called TextPrinterInit at boot time to configure the text printer driver, which is itself a GS/OS driver called TextPrinter.

The figures show the screens you will see when you use the .PRINTER CDev from Apple's Control panel or when you select the .PRINTER CDA from the CDA menu. The options that you can select are the same for both configuration programs; these are described below.

Control Panels

Text Printer

.PRINTER Driver Configuration 1.0.1

✓ Slot: 1
✓ Lines per page: 60
✓ Columns per line: <BRAM default>
✓ Add LF after CR: <BRAM default>
✓ Delete first LF after CR: <BRAM default>
✓ Turn on MSB: Yes
✓ Init string: <none>

Select: ← → ↑ ↓   Cancel: Esc   Save: ↵

Option	Description
Slot	This entry is the physical slot where your printer is located.
Lines per page	This entry is a single number, telling the printer driver how many lines appear on a sheet of paper. Most printers print 66 lines on a normal letter-size sheet of paper; it is traditional to print on 60 of those lines and leave the top and bottom 3 lines blank to form a margin. When the printer driver finishes printing the number of lines you specify, it issues a form-feed character ($0C), which causes most printers to skip to the top of a new page.

Columns per line

If you set this value to 0, the printer driver will never issue a form-feed character.

This option is a single number telling the printer driver how many columns are on a sheet of paper. Most printers print 80 columns on a normal letter-size sheet of paper. If you use a value of -1, the printer driver will never split a line. (Using the CDA configuration program, the value before 0 shows up as BRAM default; you can use the normal control panel printer configuration page to set the line length to unlimited.) What your printer does with a line that is too long is something you will have to determine by trial and error.

Delete LF

Some printers need a carriage-return line-feed character sequence to get to the start of a new line, while others only need a carriage-return. Some programs write a carriage-return line-feed combination, while others only write a carriage-return. (GSoft BASIC writes a simple carriage-return.) This option lets you tell the printer driver to strip a line-feed character if it comes right after a carriage-return character, blocking extra line-feed characters coming in from programs that print both characters.

You can select three options here: Yes, No, or BRAM Default. The Yes option strips extra line-feeds, while the No option does not. The BRAM Default option tells the printer driver to use whatever value is in the BRAM; this is the same value you would have selected using the printer configuration program in the control panel.

Add LF

Some printers need a carriage-return line-feed character sequence to get to the start of a new line, while others only need a carriage-return. This option lets you tell the printer driver to add a line-feed character after any carriage-return character that is printed.

You can select three options here: Yes, No, or BRAM Default. The Yes option adds a line-feed, while the No option does not. The BRAM Default option tells the printer driver to use whatever value is in the BRAM; this is the same value you would have selected using the printer configuration program in the control panel.

Turn on MSB

This line is a flag indicating whether the printer driver should set the most significant bit when writing characters to the printer. If this value is Yes the printer driver will set the most significant bit on all characters before sending the characters to the printer. If this value is No, the most significant bit will be cleared before the character is sent to the printer.

Init string

This option sets a printer initialization string. This string is sent to the printer when the driver is used for the first time. With most printers and interface cards there is some special code you can use to tell the printer that the characters that follow are special control codes. These codes are often used to control the character density, number of lines per page, font, and so forth. This initialization string, sent to the printer by the

.PRINTER driver the first time the printer is used, is the traditional way of setting up your favorite defaults.

You will find many cases when you will need to send a control character to the printer as part of this initialization string. To do that using the CDev configuration program precede the character with a tilde (~) character. For example, an escape character is actually a control-[, so you could use ~[to send an escape character to the printer. The printer driver does not do any error checking when you use the ~ character, it simply subtracts $40 from the ASCII code for the character that follows the ~ character and sends the result to the printer. For example, g is not a control character, but ~g would still send a value, $27, to the printer. Just type the control character in the normal way from the CDA configuration program; it will show up as an inverse character on the display.

The manual that comes with your printer should have a list of the control codes you can use to configure the printer.

The .PRINTER driver is a copyrighted program. As an owner of GSoft BASIC, you may include .PRINTER with any program you distribute that is written in GSoft BASIC, so long as our copyright information is not removed. There is no licensing fee.

MakeRuntime

The MakeRuntime utility is a very simple program that creates a version of your GSoft BASIC program that doesn't need GSoft BASIC to run.

As you develop a program, whether you use the GSoft BASIC shell or the ORCA shell, you use GSoft BASIC itself, perhaps some compiled tool interface files, and perhaps some user tools. GSoft BASIC is a copyrighted program which you cannot upload or give away, but you may want to distribute your program. The MakeRuntime utility makes this possible. It places your GSoft BASIC program, a special version of the GSoft BASIC interpreter known as the GSoft BASIC Runtime Module, and any tool or library interfaces your program uses into a single executable file. This file is an S16 program, suitable for execution from Apple's Finder or any other program launcher that launches S16 programs.

The process is so simple that MakeRuntime may be confusing at first because there are so few options! When you run MakeRuntime from the Finder you will see a standard Apple IIGS open dialog. Select any GSoft BASIC program, either a tokenized file like the ones you normally create with the GSoft BASIC shell or an ORCA source file, and open the file. After the program loads you will see a standard Apple IIGS save dialog. Type the program name, pick the destination folder, and press Save. After a few moments you'll have a GSoft BASIC program you can run from the Finder. Once the program is saved you'll get another open dialog, giving you a chance to convert another program. This process continues until you press Cancel.

Once you've pressed Cancel, you're in a very simple desktop application. You can convert more programs by selecting Make Runtime from the File menu, use desk accessories, check the version of MakeRuntime from the About box, or quit the program.

What MakeRuntime Does

MakeRuntime creates an S16 program that can be executed from pretty much any program launcher. There are several pieces that go into this completed application file.

The first piece is your GSoft BASIC program itself. If it starts out as an ASCII file, it is tokenized. The tokenized source file is placed in the resource fork of the completed application file. This is stored as resource type $7FFF, resource ID $0000001.

Your program might make use of tool calls; GS/OS calls; calls to the Talking Tools tool set; or calls to user tools, also known as libraries. MakeRuntime scans your program, creating a catalog of all of the interfaces needed, then writes a single special interface file to the resource fork. This is an abbreviated version of the complete header files contained in the various .gst files. This abbreviated version only contains the interfaces used by your program, and only contains the parts of the interfaces your program needs when it runs. These are stored as resource type $7FFF, resource ID $0000002.

Your program might use resources of it's own. Any resources used by your program are copied from your program's resource fork to the resource fork of the application file. Since the GSoft BASIC program is stored in the resource fork as resource type $7FFF, ID $0000001; and the tool interfaces are stored in resource type $7FFF, ID $0000002, you should avoid that combination for any of your own resources. In general, avoid using any resource with a resource ID of $7FFF in your GSoft BASIC programs.

The last piece that your program needs is GSoft BASIC itself, or at least enough of GSoft BASIC to execute your program. MakeRuntime adds a special version of GSoft BASIC known as the GSoft BASIC Runtime Module to the data fork of your application file. The original copy of the GSoft BASIC Runtime Module is imbedded in MakeRuntime itself; it is copied to each of your application files.

The finished file will run from the Finder. If your program worked from the GSoft BASIC shell or from the ORCA shell it should also work from the Finder—there are no additional requirements.

Things That Can Go Wrong

There are a few things that can go wrong as you convert your GSoft BASIC program with MakeRuntime. Most of these are the obvious problems that can go wrong with any program—out of memory errors, disk full errors, or disk input or output errors. All of these are rare, they are flagged by appropriate error messages, and as an experienced computer user, you already know what to do about them.

There is one error that is unique to MakeRuntime, though. If your program needs tool interface files, MakeRuntime needs to find the originals. It looks in the same folders as GSoft BASIC, examining the ORCA libraries folder, the folder containing MakeRuntime, and the

folder containing your GSoft BASIC program. You need to make sure the .gst files used by your program are in at least one of these locations when you use MakeRuntime. The easiest way to do this is to leave MakeRuntime in the same folder as GSoft.Sys16 if you are using the GSoft BASIC shell, or the same folder as ORCA.Sys16 if you are using the ORCA shell. Also, if you are using the ORCA shell, be sure you run MakeRuntime from the ORCA shell and not the Finder—that way the ORCA libraries prefix is properly set when MakeRuntime executes.

MakeRuntime will display an error dialog with the message "Interface for tool $00, call $00 not found" if it can't find a tool, user tool or GS/OS interface for a call used in your program. The $00 fields are filled in with the appropriate tool or GS/OS call numbers, and the message changes slightly to identify GS/OS calls and user tools. That's your cue that you need to make sure all of the interface files are available. The error message even tells you which tool interface is missing, although you have to identify the missing .gst file by tool number, not by the file name. After all, MakeRuntime couldn't find the file, to it doesn't know the name of the file.

Including Libraries with GSoft BASIC Programs

If your program makes calls to a library (also called a user tool), those libraries are required for your program to function. If you distribute your program, be sure to send the library files with the program. For example, if your program makes calls to the Game Paddle Tool, you must include the file UserTool001 from the *:System:Tools: folder, and if your program makes calls to the Time Tool, you must include the file UserTool002. Tell the people who use the program that these files must be placed in the Tools folder of their System folder, and that the disk must be available when your program runs; better yet, send along an installer that puts all of the files in the correct location.

Licensing

The libraries contained in UserTool001 and UserTool002, as well as the GSoft BASIC Runtime Module, are Copyright 1998 by the Byte Works, Inc. As an owner of GSoft BASIC, the Byte Works, Inc. grants you a royalty free license to distribute these two libraries and programs that have the GSoft BASIC Runtime Module attached, so long as the GSoft BASIC Runtime Module is not modified in any way. The source code for the two libraries is included with GSoft BASIC; you may distribute modified versions of these libraries so long as the Byte Works, Inc. copyright information is not removed.

As a condition of this royalty free license, any documentation, disks, or about boxes where copyright information is normally displayed should bear the statement

Portions copyright 1998, Byte Works, Inc. All rights reserved.

This royalty free license is limited to these two files and the GSoft BASIC Runtime Module. No other files or documentation may be distributed in any way without the express written permission of the Byte Works, Inc.

CompileTool

Before describing this utility, there are two caveats.

First, the average BASIC programmer does not need to use or understand CompileTool. It is used by relatively advanced programmers who need to create interface files for user tools, or who have some reason to modify the tool interface files that ship with GSoft BASIC.

Second, and partly due to the first, this section is written for advanced programmers. Most of this manual is written with the beginning or intermediate programmer in mind. The writer worked hard to make it easy to read, accessible, and to provide lots of examples. This section is written for the advanced programmer. An advanced programmer writing user tools already has a pretty good idea what this section is all about, and can guess many of the details. Unlike a beginning or intermediate programmer, an advanced programmer needs a concise technical description that is easy to scan for details, and that's how the description of CompileTool is written. If you are a beginning or intermediate programmer and you really need to use CompileTool, expect to spend a little more time reading the material carefully and experimenting than you need for a comparable number of pages in the rest of this book.

What CompileTool Does

GSoft BASIC programs can make tool calls, so they need to know what tool calls are available, what parameters to pass, and how to pass those parameters. The function of a tool interface file is to describe the tools to a language so it knows how to make those tool calls. Tool interface files serve this purpose; they are typed just like programs.

One of the most time consuming parts of compiling a program in C or Pascal that uses tool calls is processing the tool interface files. Rather than force GSoft BASIC to process these text interface files each time it starts, we use CompileTool to predigest the interface files, writing them in a form that GSoft BASIC can load very quickly. CompileTool writes a compiled tool interface file with a file type of $5E and an auxiliary file type of $8007.

For details on how GSoft BASIC searches for the tool interface files, see *The GSoft BASIC Toolbox Interface* in Chapter 19.

Command Line Interface

CompileTool runs from the ORCA shell. It might seem odd to create a utility for GSoft BASIC that can't run from the Finder or from the GSoft BASIC shell, but the reason is straight-forward. The only common use for this tool is to create interface files for user tools. You can't create a user tool from an interpreted language like GSoft BASIC; it must be done from some other language, usually assembly language. A programmer writing a user tool will probably want to create the interfaces in the same environment as the user tool itself, so CompileTool was designed to work from the ORCA shell, where the assemblers and compilers work.

The syntax for CompileTool is:

The flags can be coded in any order. They are:

flag	use
L	Writes the tool interface file to standard out as the file is compiled.
P	Writes progress information as the file is compiled.
V	Writes the version number and copyright message.

filename is the name of the text tool interface file. It can be a TXT or SRC file, and if it is an SRC file, the language stamp does not matter. By convention, the name of a tool interface file ends in .int, but this is not required.

The output file name is based on filename. If there is a dot in the file name, the last dot and everything after it is removed, then .gst (GSoft Tool) is added to the name. For example,

```
COMPILETOOL GSoftTools.int
```

will create an output file named GSoftTools.gst. The output file has a file type of $5E, which prints as DVU in catalog commands. The auxiliary file type is $8007.

CompileTool writes any error messages to the console beneath the line that caused the error, with a pointer to the offending token.

The Syntax of Tool Interface Files

This section describes the syntax for tool interface files. The first section gives an overview that describes the various parts of the tool interface file. It assumes you already know GSoft BASIC. Next is a concise summary of the syntax, presented as modified BNF syntax charts. The last section describes the differences between tool interface files and GSoft BASIC programs.

Examples are always helpful. The tool interface file GSoftTools.int, located in :GSoft.Extras:Libraries:GSoftBASIC and the corresponding folder on your hard drive if you have installed GSoft BASIC, is the tool interface file for Apple's tools, GS/OS, the ORCA shell, and Talking Tools. Refer to that file for extensive examples.

Structure of a Tool Interface File

Tool interface files are a series of constant, type and tool declarations. Comments may be added at any point in the tool interface file.

Comments

Comments start with the semicolon character and extend to the end of the current line.

```
const maxint = 32767 ; maximum positive integer
```

Identifiers and Types

Identifiers start with a letter or underscore, which may be followed by other letters, digits, or underscore characters—but unlike BASIC, you cannot include a type character. Types are always given explicitly by a type statement, not assumed from a type character appended to the identifier.

While CompileTool does not use type characters as part of the identifier, they do still have a use. Type characters can be used as a substitute for a type name anywhere the type name is allowed. For example, these declarations are completely equivalent:

```
type char as byte
type char as ~
```

The type characters and their equivalent type generally match GSoft BASIC, but there is one addition. The complete list of available types is

type	character
BYTE	~
INTEGER	%
LONG	&
SINGLE	!
DOUBLE	#
STRING	$
UNIV	?

UNIV is a new type, used to indicate a four byte typeless value. A UNIV record field or parameter is type compatible with any four byte value, including LONG; SINGLE; any pointer; any four byte record; and even a string, which is treated as a pointer to the first character of the string. INTEGER and BYTE values can also be assigned to UNIV record fields and passed as UNIV parameters; they are converted to LONG values.

Constants

Constants assign a fixed value to a name. INTEGER and LONG constants are supported; SINGLE, DOUBLE and STRING constants are not. Identifiers declared as constants can be used both in the tool interface file and in GSoft BASIC programs, and have the same affect as typing the number itself.

Hexadecimal constants are allowed, both in CONST declarations and in expressions throughout the tool compiler. As in GSoft BASIC, a hexadecimal constant with five or more digits is a LONG value, while a constant with four or fewer digits is an INTEGER.

```
const keyboard = $00C000 ; This value is 49152; it is a LONG
const keyboard2 = $C000  ; This value is -16384; it is an INTEGER
```

Named Types

There are two kinds of type declarations. The first assigns a name to a simple type or a pointer to a simple type.

Unlike GSoft BASIC, array types are allowed—but in the tool compiler, arrays are limited to a single subscript.

This series of declarations from the GSoftTools.int file show how these rules apply in actual type declarations.

```
type char as byte
type pString(256) as char
type pStringPtr as pointer to pString
```

Records

Record types generally work like they do in GSoft BASIC. You can substitute type characters for type names, and arrays are limited to a single subscript. With those exceptions, record types are compatible.

Tool Declarations

Tool declarations are essentially the first line of a SUB or FUNCTION with a tool number added in front of the declaration. Here's an example from the GSoftTools.int file:

```
Tool $04, $63 SUB    PaintArc ((Rect), %, %)
```

The tool declaration starts with the reserved word TOOL, and is followed by two values separated by a comma. The first is the tool number; QuickDraw II is tool number 4. The second number is the tool call number. When the tool is called from the GSoft BASIC program, GSoft BASIC will place any required parameters on the stack and do a JSL to $E10000, the main tool entry vector.

Three of the four differences between GSoft BASIC declarations and tool interface files are immediately apparent in this example. The first has already been mentioned: type characters can be substituted for type names, so % can be used instead of the longer INTEGER.

Names are not needed for tool parameters, since there is no body of the subroutine where the names can be put to use. As a result, parameter names are neither required nor allowed. AS is not needed to separate the parameter name from the type, so it is also omitted.

The third major difference is the way pass by value and pass by reference is handled. In BASIC, the calling expression determines whether a value is passed by value or by reference. If the types are identical and the calling expression passes an l-value, the parameter is passed by value; if the types are not identical, or the calling expression uses any form of an expression, the parameter is passed by reference. The same parameter can be passed by value on one call, and by reference on another. Toolbox calls don't work that way. Parameters passed by reference must always be passed as a pointer to the value, while parameters passed by value must always be placed on the calling stack as a value. As a result, tool declarations must have a way of describing whether a parameter is passed by reference or by value in the declaration itself. Parentheses around the parameter type indicate a parameter is passed by reference; GSoft BASIC will always place the address of a value on the stack. Parameter types that are not surrounded by parentheses are always passed by value, even if the call uses an l-value with an exactly matching type for the parameter.

There are two special cases. Records passed by value are still passed by placing a pointer to the first byte of the record on the stack, unless the record contains exactly four bytes. In that case, the record value is pushed on the calling stack. This behavior mimics the way Apple wrote the existing toolbox calls. Strings are passed as a pointer to the first character of a null terminated string. Procedures in user tools *must not* change the length of the string.

The last difference between tool interface file declarations and GSoft BASIC has to do with FUNCTION declarations. You must specify the return type of a FUNCTION explicitly, as in

```
TOOL $04, $52 FUNCTION NotEmptyRect ((Rect)) as Boolean
```

User Tool Declarations

User tool declarations follow the same rules as tool declarations, but start with USERTOOL rather than TOOL. When GSoft BASIC calls a user tool, it does a JSL to $E1008, the main entry vector for user tools.

GS/OS Declarations

GS/OS declarations also follow the same rules as tool declarations. GS/OS declarations start with the word GSOS instead of TOOL. GS/OS call numbers are two bytes long; the least significant byte is listed first, followed by the most significant byte. For example, the call number for OpenGS is $2010; the declaration looks like this:

```
GSOS $10, $20 SUB      OpenGS ((openOSDCB))
```

GSoft BASIC expects that all GS/OS calls will pass a single parameter by reference, and enforces that restriction. The parameter is pushed onto the stack and the call is made by a JSL to the GS/OS entry vector at $E100A8.

31

ORCA Shell Declarations

    The ORCA shell shares the GS/OS entry point and calling conventions. ORCA shell calls are declared as if the ORCA shell is an extension to GS/OS.

```
GSOS $49, $01 SUB          InitWildcardGS ((initWildcardDCBGS))
```

BNF For Tool Interface Files

    The complete BNF for tool interface files is shown below. The individual statements are listed in alphabetical order. The grammar starts at tool-interface.

    Comments start with the semicolon character and extend to the end of the physical line. The semicolon was used instead of the exclamation point because the exclamation point is also used as the type character for SINGLE values.

```
array-subscript ::= '(' integer ')'

as-type ::= [ POINTER TO ] type-name

const-declaration ::= CONST identifier '=' [ '-' ] integer

decimal-integer ::= [ '0'..'9' ]+

field-declaration ::= identifier [ array-subscript ] AS as-type

gsos-declaration ::= GSOS tool-number sub-or-function

hexadecimal-integer ::= '$' [ 'A'..'F' | 'a'..'f' | '0'..'9' ]+

identifier ::= letter [ letter | '0'..'9' ]*

integer ::= hexadecimal-integer | decimal-integer

letter ::= 'a'..'z' | 'A'..'Z' | '_'

record-type ::=
      TYPE identifier
      [ record-variant | field-declaration ]*
      END TYPE

parameter ::=
      '(' as-type ')'
      | as-type

parameter-list ::= '(' parameter [ ',' parameter ]* ')'

record-variant ::= CASE ( identifier | integer )
```

```
simple-type ::= TYPE identifier [ array-subscript ] AS as-type

sub-or-function ::=
    SUB identifier parameter-list
  | FUNCTION identifier parameter-list AS as-type

tool-declaration ::= TOOL tool-number sub-or-function

tool-interface ::= [
    type-declaration
  | const-declaration
  | tool-declaration
  | usertool-declaration
  | gsos-declaration ]*

tool-number ::= integer ',' integer

type-character ::= '~' | '%' | '&' | '!' | '#' | '$' | '?'

type-declaration ::= simple-type | record-type

type-name ::= BYTE | INTEGER | LONG | SINGLE | DOUBLE | STRING
  | UNIV | identifier | type-character

usertool-declaration ::= USERTOOL tool-number sub-or-function
```

Summary of Differences from GSoft BASIC

The toolbox interfaces are defined for use in GSoft BASIC, so it makes sense that toolbox interfaces look a lot like BASIC. Actually, though, the two serve different purposes, and despite obvious similarities, they are difference languages. This section outlines the differences between GSoft BASIC and tool interface files, serving as a quick reference for programmers familiar with both.

Other than the obvious difference that a tool file has declarations but no executing statements, the differences are:

• Strings and floating-point values are not allowed in tool interface files.

• Comments start with the semicolon character in tool interface files, and with REM or an exclamation point in GSoft BASIC programs. Comments can start on the same line as a declaration in a tool interface file, but must start on a fresh line or after a colon continuation character in GSoft BASIC.

• Arrays in tool interface files are limited to one subscript.

• Array types are allowed in tool interface files, but not in GSoft BASIC programs.

- Types are not assumed based on the name of a variable in tool interface files, as they are in GSoft BASIC programs. Type characters are not allowed as part of a name in tool interface files.

- Type characters can be substituted for type names in tool interface files; % has the same meaning as INTEGER.

- The UNIV type, a four byte type compatible with any other four byte entity, is allowed in tool interface files, but not in GSoft BASIC programs.

- TOOL, USERTOOL or GSOS precedes procedure declarations in a tool interface file. These are followed by the tool number and tool call number, or the GS/OS call number split into two bytes.

- Parameters are given as a type in tool interface files, and as a parameter name optionally followed by a type in GSoft BASIC programs.

- Parameters passed by reference are enclosed in parentheses in tool interface files. In GSoft BASIC programs, the way the parameter is passed by the caller determines if it is passed by reference or by value.

- Function return types are required in tool interface files; function return types are assumed from the function name if the return type is not given in GSoft BASIC programs.

ORCA Shell GSoft BASIC

The ORCA shell version of GSoft BASIC installs and runs from any ORCA 2.0 or later shell, or from environments like GNO and APW 2.0 or greater that are compatible with the ORCA shell. Once installed, GSoft BASIC works almost exactly like any other ORCA language.

The ORCA shell ships with ORCA/M, ORCA/Pascal, ORCA/C, ORCA/Modula-2 and *Toolbox Programming in GSoft BASIC*. It is documented in those books, so that documentation is not duplicated here. The remainder of this section describes the unique features of GSoft BASIC when used from the ORCA shell, what the installer does, and documents a utility that converts GSoft BASIC tokenized files to SRC files you can use from the ORCA shell.

For a discussion of the advantages and disadvantages of the ORCA shell version of GSoft BASIC as compared to the version that is described in this manual, see *The Three Worlds of GSoft BASIC* in Chapter 2.

Using GSoft BASIC from the ORCA Shell

With GSoft BASIC installed in the ORCA shell, you can enter BASIC programs with the ORCA editor just as you do for any other language. Be sure to set the language type for the file to BASIC, either by typing BASIC just before you edit a new file, or by using the ORCA shell's CHANGE command to set the language type of an existing file.

Once you have created a program, use the RUN command to execute it. For example, if you have created a file named Hello.bas, you would execute the program with the command

```
run hello.bas
```

Since GSoft BASIC is an interpreter, there are no separate compile, link and execute steps. You use the RUN command each time you want to execute the program.

While it isn't required, we recommend appending .bas to the end of all GSoft BASIC source files. This convention conforms to the naming conventions used in the ORCA shell, allowing dot prefixes for partial compiles and multi-lingual compiles if a GSoft BASIC compatible compiler is ever made available.

The DeToke Utility

Most programs written with GSoft BASIC are saved as tokenized BASIC programs. The ORCA editor can't read these files. The DeToke utility gives you an easy way to convert the tokenized files used by the GSoft BASIC shell to the source files used by the ORCA shell. It is designed so it is fairly easy to convert large numbers of files with a single command.

The DeToke utility accepts one or more input files. Each name can include wild cards, in which case all matching files are converted. Since the utility is designed to convert large numbers of files at once, accepting many source files, it forms its own output file name. The output file name is formed by stripping any suffix from the original file (a suffix starts with the last period in the name and continues to the end of the name), then appending .bas to the source file name. If the resulting name would exceed 15 characters, it is shortened.

For example, if there are two tokenized BASIC files with these names

```
prog.tok
longprogramname
```

in your folder and you convert them to source files using the command

```
detoke =prog=
```

the output files will be

```
prog.bas
longprogram.bas
```

Since DeToke is designed to convert large numbers of files in a single pass, it does not complain when one of the input files is not a tokenized GSoft BASIC program. In fact, you can safely convert all of the programs in a folder that contains GSoft BASIC programs mixed with other files using the command

```
detoke =
```

There are two command line flags, each of which must be used before the first file name. They may be used in any order, so long as they precede the file names.

The -v flag prints version and copyright information. Use this flag if you need to check the version number for the DeToke utility.

The -p flag prints progress information. Use this flag if you would like a list of the files DeToke examines. This list shows which files were converted and which were skipped because they were not GSoft BASIC tokenized source files.

For example, to see the copyright information and get a complete list of the files processed, you could use the command

```
detoke -v -p =
```

Installing GSoft BASIC in the ORCA Shell

You should normally install GSoft BASIC using the installer that comes with it. This process is described in *Installing GSoft BASIC on a Hard Disk* in Chapter 1. This detailed description of the installation process is here for those of you that want all the details, those who may be installing GSoft BASIC in an ORCA compatible shell, and those who want to customize the installation.

Assuming you already have the ORCA shell installed and working, there are several other files you will want to install or modify to use GSoft BASIC. In all cases these descriptions assume you have installed ORCA in a folder named ORCA, and give partial path names based on that folder.

:GSoft.Extras:Languages:BASIC

This is the GSoft BASIC interpreter itself. It must be installed in the languages folder. In a typical installation of ORCA, this will be in the ORCA:Languages: folder.

:GSoft:Shell:SysCmnd

The SysCmnd file is a catalog of all of the commands, languages and utilities in your installation of ORCA. You will need to add two lines to this file, which is located at ORCA.:Shell:SysCmnd. These lines should be inserted in alphabetical order. Use the same spacing you see in the existing commands in the file.

```
BASIC     *L   260   GSoft BASIC
DETOKE    *U         GSoft BASIC tokenized file converter
```

:GSoft:Shell:SysTabs

The SysTabs file contains default tab lines and default editor settings for each programming language installed in the ORCA shell. You should insert the following lines in the file, located at ORCA:Shell:SysTabs.

Proper order is important. Each language uses three lines, although line wrapping may make it look like there are more that three lines, as it does in this manual. The third line contains the tab stops, and must be a single line, unbroken by line feeds. The first line contains the language number, and it is important that the three line sets of information be in numerical order by language number.

```
260
10011001
00000010000000010000000001000000000010000000100000000110000000011000
00010000001000000000001000000000100000001000000010000001000000001000
00000100000010000002
```

You can customize these lines so the editor suits your personal preferences. See *Setting Editor Defaults* in Chapter 6 for details.

:GSoft.Extras:Libraries:GSoftDefs:

GSoft BASIC comes with several tool interface files. The exact number of files may change in the future, so the best policy is to copy all of the files you find in this folder on the GSoft BASIC disk into the ORCA:Libraries:GSoftDefs folder.

As this manual is written, the files we ship with GSoft BASIC are:

GSoftTools.gst   This file contains the interfaces for Apple's toolbox calls, GS/OS, the ORCA shell, and Talking Tools. All of these are discussed in detail in various parts of this manual; see the index for details.

User001.gst   This is the interface for the Game Paddle Library, a custom library supplied with GSoft BASIC that supports game paddles and joysticks.

User002.gst   This is the interface for the Time Library, a custom library supplied with GSoft BASIC that supplies time and date manipulation subroutines.

GSoft BASIC will work without these files, but you won't be able to use the tools and libraries they represent unless the files are installed.

:GSoft.Extras:System:Tools:

GSoft BASIC libraries are implemented as user tools. As this manual is written, there are two of them, but this number may increase. As with the tool interface files, the best policy is to copy all of the files you find in the GSoft.Extras:System:Tools folder into the corresponding file on your boot disk, *:System:Tools.

The two files that ship with GSoft BASIC are:

UserTool001   This is the executable code for the Game Paddle Library. This file must be present if your programs use any of the game paddle procedures.

UserTool002   This is the executable code for the Time Library. This file must be present if your programs use any of the Time Library procedures.

Keep in mind that these files are also needed by anyone who is running your programs. We grant a royalty free license to distribute these files with any program written in GSoft BASIC, but it is up to you to make sure they are included with your program!

:GSoft.Extras:Utilities:DeToke

The DeToke utility, described earlier in this chapter, is used to convert GSoft BASIC tokenized files to ORCA SRC files. It should be installed in ORCA:Utilities, and the changes described for the SysTabs file must also be made.

This utility is not required by GSoft BASIC, and it also doesn't depend on GSoft BASIC. Without it, though, you will have to use the GSoft BASIC shell to convert tokenized files to a format you can use from the ORCA shell.

:GSoft.Extras:Utilities:Help:DeToke

The help file for the DeToke utility should be installed in ORCA:Utilities:Help.

Chapter 5 - The Command Processor

There are three versions of GSoft BASIC, two of which can be used to write programs. One of these runs from any ORCA compatible shell, and one runs directly from the Finder. This chapter describes the commands used with the Finder version of GSoft BASIC.

In general you can think of this version of GSoft BASIC as a souped up implementation of the Applesoft BASIC programming environment. It has some powerful new commands, but the commands you learned with Applesoft BASIC generally work just like they always did.

The third version of GSoft BASIC is used to run programs directly from the Finder. You can't write programs with this version, just run them. The program used to create these GSoft BASIC programs is called MakeRuntime. It is covered in Chapter 4, *GSoft BASIC Utilities*.

The Line Editor

When you enter the Finder version of GSoft BASIC, you're in a text based shell. This shell lets you type commands like CATALOG to catalog a disk or EDIT to use the full screen editor to edit a file. It also lets you enter programs directly from the keyboard if you are willing to use line numbers.

In addition to the obvious keys used to type characters, there are several editing commands available as you type a line. Some allow you to position the cursor in the middle of the line. It doesn't matter if the cursor is at the end of the line or not—when you press RETURN or ENTER, the entire line is processed.

The editing keys are:

39

key	action
LEFT-ARROW	The cursor will move to the left. If the cursor is already in the first character position on the line, the key is ignored.
RIGHT-ARROW	The cursor will move to the right. If the cursor is already positioned just to the right of the last typed character (which may be a space), the key is ignored.
delete	Deletes the character to the left of the cursor, moving the cursor left. If the cursor starts in the first character position on the line, the key is ignored.
return or enter	Executes the typed command.
esc or clear	Deletes all characters in the line.
⌘X or ⌘x	Deletes all characters in the line. These keys are generally used for the cut command on the Apple IIGS. The closest equivalent in the line editor is deleting all of the characters.
⌘Z or ⌘z	Deletes all characters in the line. These keys are generally used for the undo command on the Apple IIGS. The closest equivalent in the line editor is deleting all of the characters.
⌘F or ⌘f	Deletes the character under the cursor, moving the remaining characters in the line left one position. The key is ignored if the cursor is just past the last typed character.
⌘Y or ⌘y	Deletes characters from the cursor to the end of the line.
⌘> or ⌘.	The cursor moves to the end of the line, just past the last typed character. New characters will appear at the end of the line.
⌘< or ⌘,	The cursor moves to the first position in the line.
⌘LEFT-ARROW	The cursor moves left one word.
⌘RIGHT-ARROW	The cursor moves right one word.
⌘E or ⌘e	Toggles the insert mode. If the cursor is in insert mode, new characters are inserted, pushing existing characters from the cursor to the end of the line right one position. In overstrike mode, typed characters replace the character under the cursor. The default is overstrike mode.

File Names

Many of the commands described later in this chapter use file names. File names in
GSoft BASIC follow standard GS/OS conventions.

File names are discussed in detail in *File Names*, found in Chapter 14.

Types of Files Used by GSoft BASIC

GSoft BASIC can read four file types and write three different types of files as BASIC programs. Both the LOAD command and EDIT command will load any of the four supported file types as the active program. There are three versions of the save command, SAVE, SSAVE and TSAVE, used to save the program in the workspace in one of the three supported file types.

Source Files

The most natural way for GSoft BASIC to deal with a file is the same way the file is stored in memory: as a tokenized file.

Tokenized files have a file type of $AF and an auxiliary file type of $0104. In some utilities, you will see the file type listed as TOK for tokenized source files.

The disadvantage to tokenized source files is that they can only be read by GSoft BASIC itself. On the other hand, they are shorter than an equivalent text file, load faster, and save faster.

The SAVE command saves a GSoft BASIC program as a TOK file. Use this command when you intend to use the Finder version of GSoft BASIC described in this chapter.

The format for tokenized source files is described in detail in Appendix F, *Implementation Details*.

GSoft BASIC Tokenized Files

The most natural way for GSoft BASIC to deal with a file is the same way the file is stored in memory: as a tokenized file.

Tokenized files have a file type of $AF and an auxiliary file type of $0104. In some utilities, you will see the file type listed as TOK, for tokenized source files.

The disadvantage to tokenized source files is that they can only be read by GSoft BASIC itself. On the other hand, they are shorter than an equivalent text file, load faster, and save faster.

The SAVE command saves a GSoft BASIC program as a TOK file. Use this command when you intend to use the Finder version of GSoft BASIC described in this chapter.

The format for tokenized source files is described in detail in Appendix F, *Implementation Details*.

Text Files and Source Files

There are two supported file types that are essentially the same inside. Both store the file as ASCII characters, with carriage return characters (CHR$(13)) at the end of each line.

The first of these file types is the generic Apple IIGS text file. Text files have a file type of $04. The auxiliary file type is not used by most programs. In most utilities, you will see this file type listed as TXT. Text files can be read and written by practically any Apple II program that edits text in any form.

The second file type is the SRC file, used in programming environments like ORCA. SRC files have the same internal format as TXT files, but use the auxiliary file type to indicate which language should be used to process a file. GSoft BASIC can read any source file, and will attempt to tokenize what it reads as a BASIC program. It writes SRC files with a file type of $B0, listed as SRC by most utilities that show a file type, and with an auxiliary file type of $0104. Programming environments like ORCA that support multiple languages will display BASIC for the auxiliary file type.

The TSAVE command saves a GSoft BASIC program as a TXT file. Use TXT files when you need to read the program with some other text processing utility, or when you want to convert a GSoft BASIC program to Applesoft BASIC.

The SSAVE command saves a GSoft BASIC program as an SRC file with an auxiliary file type of $0104. Use this command if you want to save a program in a format that can be used by

41

the shell version of GSoft BASIC. See *The Three Worlds of GSoft BASIC* in Chapter 2 for an in depth look at how the shell version of GSoft BASIC uses SRC files.

Applesoft BASIC Files

GSoft BASIC can read Applesoft BASIC programs, but cannot write them directly.

GSoft BASIC is similar to Applesoft BASIC, but it is not a true superset of Applesoft BASIC. Applesoft BASIC is tied directly to the eight bit Apple II architecture in several important ways that could not be ported to the sixteen bit environment of the Apple IIGS using GS/OS. In addition, GSoft BASIC has many commands that are not supported in Applesoft BASIC, and some, like DEF FN, that have important new features.

At the same time, a natural use for GSoft BASIC is porting older eight bit programs to the faster, more powerful world of GS/OS. That's why GSoft BASIC lets you load Applesoft BASIC programs. If features are found that don't exist under GSoft BASIC, or that are suspect, the old line is flagged with a comment that tells you to examine the line for possible changes. The actual comment varies with the command, but you can find them all by searching the file for the characters >>>. That's easy to do from the GSoft BASIC text editor. See the EDIT command, below, for details.

The same problem exists in reverse. Applesoft BASIC can't detect conflicts with GSoft BASIC if the file is saved as a tokenized Applesoft BASIC file, so you don't have that option. You can save a GSoft BASIC program as a TXT file, though, and import it to Applesoft BASIC. That gives Applesoft BASIC a chance to scan the file, making sure it understands all of the commands.

See Appendix E for more details on compatibility between Applesoft BASIC and GSoft BASIC.

Entering BASIC Programs

The Active Program

GSoft BASIC always has one program that is the active, or current, program. This program is stored in an area of memory referred to in this manual as the workspace or program buffer.

Only one program is active at a time. This is the program that is edited, executed, saved and so forth. You can delete this program, emptying the workspace, with the NEW command. You can also load a new program into the workspace using the LOAD command, by using the EDIT command with a file name, or by entering a completely new program.

Entering Programs From the Command Line

Any line that starts with a number is treated as a new GSoft BASIC program line, and is entered in the current program. The line is inserted in the program in numeric order, replacing any existing line with the same line number.

For example, the commands

```
NEW
10 FOR I = 1 TO 10
20 PRINT J
30 NEXT
20 PRINT I
```

create the program

```
10 FOR I = 1 TO 10
20 PRINT I
30 NEXT
```

While this is a comfortable way to hack out short programs, the full screen editor you invoke with the EDIT command is generally a better way to create and edit programs. It also doesn't require line numbers.

Typing a line number with nothing after it deletes an existing line with the same line number. If no matching line is found, the program doesn't change. For example, typing

```
20
```

changes the sample program to

```
10 FOR I = 1 TO 10
30 NEXT
```

There is a restriction on this feature. Line numbers are optional in GSoft BASIC. If the current program has even one line that does not have a line number, you can't enter new lines directly from the command line.

Executing BASIC Commands

If you type a line that does not start with a line number and is not one of the commands listed in the Command Reference, below, GSoft BASIC tries to execute the command as if it were a line in a BASIC program.

43

There are many uses for this feature, but one of the most powerful is debugging. After stopping a program with STOP, you can interrogate the values of variables, or even change their values. The program can be restarted with CONT.

Of course, the other common use for this feature is to experiment with commands, or even do useful work with one-liners. After all,

```
PRINT $0201
```

is a pretty convenient hexadecimal to decimal converter!

Command Reference

Bye

BYE

Exits GSoft BASIC.

If you have edited the program in the workspace since the last time the program was saved, you will get a prompt asking if you are sure you want to exit GSoft BASIC.

Catalog

CAT [pathname]

CATALOG [pathname]

Catalogs a directory.

If no path name is given, the current directory is cataloged. If given, the path name can be a full or partial path name, the name of a volume, or the name of a device.

The abbreviation CAT can be used instead of the full name of CATALOG. They do exactly the same thing.

The files are listed, along with information about the files, in a tabular format. The listing of files starts with a header labeling the columns. The names used in the header, along with the meaning for the items in that column, are:

name	meaning
Name	The file name for the file.
Type	The type of the file. This is shown as a three letter type identifier. The types that are of interest to GSoft BASIC programmers are shown later in the description of CATALOG.
Blocks	The number of blocks used by the file. This shows how much disk space is used by the file, not the number of bytes used by the file in memory. Each disk block uses 512 bytes on the disk, which is one-half of a kilobyte.
Modified	The last date and time the file was changed.
Created	The date and time the file was created.
Access	The access flags for the file. See the description of access flags later in this section for details.
Subtype	The auxiliary file type for the file. This is shown as a hexadecimal value.

File types are stored as a single byte, but it makes more sense to see them as a three letter abbreviation. The table below shows the three letter designation used by the various file types that are used in GSoft BASIC. See *File Type Notes*, a technical note written by Apple Computer and published by Byte Works, for a complete list of file types and information about the internal format of many of the file types.

Name	Hex	Use
TXT	$04	ASCII Text. See *Text Files and Source Files*, earlier in this chapter.
BIN	$06	Binary files, often used as input and output by GSoft BASIC. See *Binary Files* in Chapter 14.
DIR	$0F	Directories, which contain other files and directories. See CREATE and PREFIX, later in this chapter; and MKDIR and CHDIR in Chapter 14.
DVU	$5E	Development Utilities, which contain information used by programming tools. The format varies widely, depending on which tool created the file. GSoft BASIC tool interface files have this file type, along with an auxiliary file type of $8007.
TOK	$AF	GSoft BASIC tokenized source file.
SRC	$B0	Program source file. See *Text Files and Source Files*, earlier in this chapter.
S16	$B3	Programs that can be launched from the Finder or executed immediately upon booting the computer. See *Creating GSoft BASIC Programs that Run From the Finder* in Chapter 2.
BAS	$FC	Applesoft BASIC source file. See *Applesoft BASIC Files*, earlier in this chapter.

Access flags indicate how a file can be used. They are listed as an uppercase letter if the privilege is set, and a space if it is not.

You can enable and disable some of these flags with LOCK and UNLOCK, described later in this chapter.

The various access flags and their meanings are:

flag	meaning
D	Delete. When set, the file can be deleted. You can clear this flag with LOCK, and set it with UNLOCK.
N	Rename. When set, the file can be renamed. You can clear this flag with LOCK, and set it with UNLOCK.
B	Backup. When set, the file has not been saved by a backup utility since the last time it was modified.
W	Write. When set, the file can be changed. You can clear this flag with LOCK, and set it with UNLOCK.
R	Read. When set, the file can be read.

You can pause in the middle of a long list of files by pressing the space bar. Press the space bar again to continue. Use `esc` or ⌦ to abort the list of files.

After all of the files are listed, the CATALOG command prints a trailer that tells how many blocks on the disk are used, how many are free, and the total number of blocks on the disk.

Copy

COPY `from` `to`

Copy a file from one location to another.

You can use full or partial path names for either file. For example,

```
COPY :MYPROGRAMS:PROGRAM PROGRAM
```

copies the file PROGRAM from the directory :MYDISK:MYPROGRAM to the current directory.

A full path name can also be used as the destination.

If the source file name and destination file name are the same, as in the example above, you can omit the last file name. The command

```
COPY :MYDISK:MYPROGRAMS:PROGRAM
```

does exactly the same thing as the first example.

Create

CREATE pathname

Creates a new directory.

You can create a directory with a full or partial path name. The most common way to create a directory is in the current directory, though. The command

```
CREATE NewFolder
```

creates a new directory called NewFolder in the current directory.

DEBUG

DEBUG [linenumber | filename]

Runs a program, with the same options as the RUN command. The difference is that DEBUG enters an ORCA compatible debugger (like ORCA/Debugger or Splat!), breaking on the first line executed.

Once inside the debugger, you can step or trace through the GSoft BASIC program, examine variables, and carry out any other activity the debugger supports.

Do not use this command unless an ORCA compatible debugger is installed! ORCA compatible debuggers work by intercepting the 65816 COP instruction. There is no way for GSoft BASIC to tell if a debugger is installed or not, so it will issue the COP instruction whether or not a debugger is actually present. If there is no debugger installed, this causes the computer to crash. While this does no actual harm, the only way to recover is to reboot.

Del

DEL start [',' end]

Delete a single line or a range of lines.

The DEL command cannot be used with programs that do not use line numbers on every line.

Delete

DELETE filename

Deletes the named file. The file name can be a file name, partial path name, or full path name.

The file can be a directory. After checking to be sure the user really wants to delete the directory and its contents, all files in the directory and the directory itself are deleted.

Edit

EDIT [filename]

Enters an ORCA compatible editor, displaying the program in memory. If a file name is given, the file is loaded and edited exactly as if the commands

```
LOAD filename
EDIT
```

were used.

The editor should be installed in a directory named Shell. This directory should be in the same directory as the GSoft BASIC application. Placing GSoft BASIC in the same directory with ORCA.Sys16 in an existing ORCA/M environment will satisfy all editor requirements for both GSoft BASIC and the ORCA shell.

See Chapter 6 for a complete description of the full screen editor that comes with GSoft BASIC.

List

LIST [line-number [',' [line-number]]]

Lists the entire program, a single line, or a range of lines. See PR for a way to print the file to a printer.

To list the entire program use the LIST command with no parameters. You can use this version of the LIST command with programs that do not use line numbers.

You can pause in the middle of a long listing by pressing the space bar. Press the space bar again to continue. Use esc or ⌘. to abort the listing.

If you use a single line number, LIST lists that line from the program. If there is no line with the given line number, LIST does nothing.

If you use two line numbers separated by a comma, LIST lists all of the lines whose line number is greater than or equal to the first number, and less than or equal to the second. If there are no lines in the given range, LIST does nothing.

For example,

```
LIST 100, 400
```

lists all of the lines from line 100 to line 400.

You can omit the first line number, in which case LIST lists all of the lines whose line number is less than or equal to the second parameter. For example,

```
LIST , 400
```

lists all of the lines from the start of the program to line 400. Omitting the second number, but including the comma, lists all of the lines greater than or equal to the first line number. For example,

```
LIST 100,
```

lists all of the lines from line 100 to the end of the program.

You can use LIST to list programs that do not have line numbers on each line, but the entire program is always listed. In general, you should use the full screen editor to view programs that do not use line numbers on each line.

Load

`LOAD    filename`

Load a program from disk.

The program may be a GSoft BASIC tokenized file, a TXT file, a BASIC SRC file or an Applesoft BASIC file.

If the file is a TXT or SRC file, it is handled almost as if the NEW command was used, then each of the lines in the file was typed in turn. The single difference is that lines with no line number are accepted.

Applesoft BASIC programs are loaded and converted to GSoft BASIC programs, with comment lines added after any line that may cause a problem in GSoft BASIC or GS/OS. Search for the characters >>>, which appear at the start of all of these comments.

Lock

`LOCK    filename`

Locks a file. Locked files cannot be renamed, deleted, or written to.

Move

`MOVE    from    to`

Move a file from one location to another.

Move works exactly like a COPY command followed by a DELETE command that deletes the original file.

New

NEW

The program in the workspace is deleted.

Prefix

PREFIX [pathname]

Changes the default prefix (GS/OS prefix number 8) to the given path name.

If no prefix is given, the current value for the prefix is shown.

There is one special name. A path name consisting of two periods, as in

    PREFIX ..

moves up one directory level. For example, assuming disks and folders with the appropriate names exist, the commands

    PREFIX :MyDisk:GSoft:Samples
    PREFIX ..
    PREFIX

prints the current prefix, which is

    :MyDisk:GSoft:

You can use CATALOG after PREFIX if you aren't sure that you arrived at the proper spot.

PR

PR [line-number [',' [line-number]]]

The PR command works exactly like LIST, but the program listing is sent to the printer instead of the text screen. The .PRINTER driver described in Chapter 4 must be installed for this command to work.

See LIST for a description of the parameters.

Rename

RENAME old new

Rename a file.

You can use full or partial path names, but RENAME will not move a file, so any path names used must match.

RENAME can be used to change the case of a file name. For example, if a file is named MYPROG, but you would like to see MyProg when you catalog a disk, use the command

    RENAME MyProg MyProg

The letter case of the first name doesn't matter, since it is only used to find the file to rename, and file name matching is case insensitive. The case of the second name is used for the new file name. It is preserved, showing up when you catalog the directory containing the file.

Renumber

RENUMBER first ',' step [',' start [',' end]]

Renumber a program.

The parameters identify the lines to renumber and the way to renumber the file.

parameter	meaning
first	First line number to use.
step	Increment between new line numbers.
start	First line to renumber.
end	Last line to renumber.

RENUMBER is generally used to update the line numbers in a program that has been edited, inserting many lines in one part of the program. You could run out of available line numbers in the part of the program you are editing, or you might want to renumber the program for aesthetic reasons. For a more esoteric example, consider the short program

```
1000 DIM FLAGS%(8190)
2000 PRINT "10 iterations"
2010 FOR ITER = 1 TO 10
2020   COUNT = 0
2030   FOR I = 0 TO 8190
2040     FLAGS%(I) = 1
2050   NEXT I
2060   FOR I = 0 TO 8190
2070     IF NOT FLAGS%(I) THEN 2160
2080     PRIME = I + I + 3
2090     PRINT PRIME
2100     K = I + PRIME
2110     IF K > 8190 GOTO 2150
2120     FLAGS%(K) = 0
2130     K = K + PRIME
2140     GOTO 2110
2150     COUNT = COUNT + 1
2160   NEXT I
2165   NEXT ITER
2170 PRINT COUNT;" primes"
```

This is a classic benchmark for computers. It finds all of the prime numbers from 3 to 8190. This particular version will run from either Applesoft BASIC or GSoft BASIC. Traditionally, line 2090 is removed before timing the program.

The amount of time it takes to handle a long line number is actually longer than the time it takes to handle a short one. Renumbering the program with the command

```
RENUMBER 1, 1
```

created a compacted program whose first line number is 1, with line numbers that increment by 1. Here's the renumbered program:

```
1 DIM FLAGS%(8190)
2 PRINT "10 iterations"
3 FOR ITER = 1 TO 10
4   COUNT = 0
5   FOR I = 0 TO 8190
6     FLAGS%(I) = 1
7   NEXT I
8   FOR I = 0 TO 8190
9     IF NOT FLAGS%(I) THEN 17
10    PRIME = I + I + 3
11    K = I + PRIME
12    IF K > 8190 GOTO 16
13    FLAGS%(K) = 0
14    K = K + PRIME
15    GOTO 12
16    COUNT = COUNT + 1
17  NEXT I
```

```
18 NEXT ITER
19 PRINT COUNT;" primes"
```

This program actually runs about 2% faster than the original. That's not a very big improvement, but it does show how RENUMBER works!

You can get an even bigger savings by converting the program completely to GSoft BASIC, taking advantage of all its features. Here's the same program fully converted to GSoft BASIC.

```
DIM FLAGS%(8190)
PRINT "10 iterations"
FOR ITER% = 1 TO 10
  COUNT% = 0
  FOR I% = 0 TO 8190
    FLAGS%(I%) = 1
  NEXT I%
  FOR I% = 0 TO 8190
    IF FLAGS%(I%) THEN
      PRIME% = I% + I% + 3
      K% = I% + PRIME%
      WHILE K% <= 8190
        FLAGS%(K%) = 0
        K% = K% + PRIME%
      WEND
      COUNT% = COUNT% + 1
    END IF
  NEXT I%
NEXT ITER%
PRINT COUNT%;" primes"
```

This program runs about 14% faster than the original. The biggest saving comes from using integers throughout the program rather than floating-point numbers.

Just for comparison, the original program runs in about 2/3 the time it takes the same program to run under Applesoft BASIC. The fully converted program runs in about 57% of the time—almost twice as fast as Applesoft BASIC.

There is one possible error you can encounter with the RENUMBER command. If the last renumbered line is larger than the first subsequent line that is not renumbered, or if RENUMBER would create a line number greater than 65535, the renumber command fails. It prints an error message and refuses to renumber the program. You can generate this error from the original prime number sample with the command

```
RENUMBER 2000, 100, 2000, 2100
```

This command attempts to renumber the lines from 2000 to 2100 beginning with line number 2000 and incrementing by 100. If this command were carried out, line 2100 would be renumbered as line 3000. Since the line right after the original line 2100 is numbered 2110, this command would create a program with overlapping lines, so the command aborts rather than demolishing your program.

53

This command may be used with programs that contain lines without line numbers, but you cannot use start and end line numbers unless the entire program is numbered. In short, if the program doesn't start with all lines numbered, you have to renumber the entire program. Also, RENUMBER will not renumber line number *uses* correctly if the program has duplicate line numbers. For example, the program

```
        GOTO 100
        DATA 10
100 READ J
        FOR I = 1 TO J
        CALL P
        NEXT
        END
```

```
        SUB P
        GOTO 100
100 DATA I, II, III
        FOR I = 1 TO 3
        READ J$
        PRINT J$
        NEXT
        END
```

should not be renumbered. It is a legal GSoft BASIC program, and will work correctly as shown. When the program is renumbered, though, the two line numbers will be changed, but the line numbers used in the GOTO statements will both point to the same line. Before renumbering a program like this one, remove duplicate line numbers manually. You don't have to insure that the line numbers are in order, but you do need to insure that all line numbers are unique before you renumber a program.

Of course, in this particular example, the program would work if you left the GOTO statements and line numbers out.

Run

RUN [line-number | filename]

Run a program.

If a number is supplied as a parameter, program execution starts at that line.

If a file name is supplied as a parameter, the file is loaded and executed. The file may be a GSoft BASIC tokenized file, a TXT file, a BASIC SRC file or an Applesoft BASIC file.

The RUN command clears the variable space before the program begins execution. Any old values, types and DEF FN declarations are erased. If you would like to execute the program without erasing existing values, use the GOTO statement to jump into the program at a specific spot.

Save

SAVE   **filename**

Save a program to disk as a GSoft BASIC tokenized file.

The resulting file can be executed from the shell version of GSoft BASIC, and it can be used as input to the MakeRuntime utility.

See Appendix F for details about the file format.

SSave

SSAVE   **filename**

Save a program to disk as an ORCA SRC file.

The file has a file type of $B0 (shown as SRC by the CATALOG command) and an auxiliary file type of $0104. See *Text Files and Source Files*, earlier in this chapter, for a discussion of this file type.

TSave

TSAVE   **filename**

Save a program to disk as a generic ASCII text file.

The file has a file type of $04 (shown as TXT by the CATALOG command). See *Text Files and Source Files*, earlier in this chapter, for a discussion of this file type.

Unlock

UNLOCK   **filename**

Unlocks a file locked with the LOCK command. Locked files cannot be renamed, deleted, or written to.

Chapter 6 – The Text Editor

GSoft BASIC's EDIT command opens the current program using any ORCA compatible editor. This chapter describes the ORCA editor, which ships with GSoft BASIC. The ORCA editor allows you to write and edit source and text files.

The first section describes briefly how the ORCA editor works with GSoft BASIC interpreted files. The second section in this chapter, "Modes," describes the different modes in which the editor can operate. The third section, "Macros," describes how to create and use editor macros, which allow you to execute a string of editor commands with a single keystroke. The fourth section, "Using Editor Dialogs," gives a general overview of how the mouse and keyboard are used to manipulate dialogs. The next section, "Commands," describes each editor command and gives the key or key combination assigned to the command. The last section, "Setting Editor Defaults," describes how to set the defaults for editor modes and tab settings for each language.

How Text Editors Work With GSoft BASIC Tokenized Files

When you use GSoft BASIC's environment to write and execute programs, the program you are working on is stored in a tokenized form. Tokenizing a program is a process that converts common words, like PRINT, into single byte quantities. As a result, the program is much smaller than a standard text representation of the file, and it runs faster than would be possible if the program was not tokenized. The space savings are actually dramatic—tokenized files are generally smaller than a compiled version of the same program. GSoft BASIC's SAVE command saves the file in this tokenized form, too.

These tokens are converted back to text whenever you list or edit a program. Editing a program with any editor becomes a multi-step process. First GSoft BASIC converts the tokenized file to ASCII text. This text is actually saved in a scratch file called SysBASICTemp; this file is stored in the current default directory. It's this scratch file that the ORCA editor actually edits. You will see this name at the bottom of your edit screen. After you finish editing the file, the editor saves the results back to the scratch file, and GSoft BASIC loads the result just as if you used the LOAD command.

All of this is fairly transparent, but it explains two points you should keep in mind.

First, the file you type is converted to tokens. In the process, all identifiers are converted to uppercase letters and extra spaces are removed. When the program is converted from tokenized form to ASCII text spaces are inserted to make the program more readable. You don't have any control over this process. If you are picky about the way you format a program, this can be annoying—the only alternative is to use a standard ORCA compatible programming environment, installing GSoft BASIC as a standard language. On the other hand, since GSoft BASIC will format your program in a reasonable way, it frees you from keeping the text pretty as you type it in the editor. No matter how sloppy your spacing and indenting might be, GSoft BASIC will clean it up.

57

The second point is that you can edit other text files while in the editor. The ORCA editor doesn't know how to edit a tokenized BASIC file, but it can load any files you save using the TSAVE command. If you have BASIC subroutines that you want to use in different programs, save them with the TSAVE command so you can load them into the editor while you are editing your program.

Modes

The behavior of the ORCA editor depends on the settings of several modes, as follows:

- Insert.
- Escape.
- Auto Indent.
- Text Selection.
- Hidden Characters.

Most of these modes have two possible states; you can toggle between the states while in the editor. The default for these modes can be changed by changing flags in the SysTabs file; this is described later in this chapter, in the section *Setting Editor Defaults*. All of these modes are described in this section.

Insert

When you first start the editor, it is in over strike mode; in this mode the characters you type replace any characters the cursor is on. In insert mode, any characters you type are inserted at the left of the cursor; the character the cursor is on and any characters to the right of the cursor are moved to the right.

The maximum number of characters the ORCA editor will display on a single line is 255 characters, and this length can be reduced by appropriate settings in the tab line. If you insert enough characters to create a line longer than 255 characters, the line is wrapped and displayed as more than one line. Keep in mind that most languages limit the number of characters on a single source line to 255 characters, and may ignore any extra characters or treat them as if they were on a new line.

To enter or leave the insert mode, type ⌂E. When you are in insert mode, the cursor will be an underscore character that alternates with the character in the file. In over strike mode, the cursor is a blinking box that changes the underlying character between an inverse character (black on white) and a normal character (white on black).

Escape

When you press the ESC key, the editor enters the escape mode. For the most part, the escape mode works like the normal edit mode. The principle difference is that the number keys allow you to enter repeat counts, rather than entering numbers into the file. After entering a repeat count, a command will execute that number of times.

For example, the ⊙B command inserts a blank line in the file. If you would like to enter fifty blank lines, you would enter the escape mode, type 50⊙B, and leave the escape mode by typing the ESC key a second time.

In the normal editor mode, ⊙ followed by a number key moves to various places in the file. In escape mode, the ⊙ key modifier allows you to type numbers.

The only other difference between the two modes is the way CTRL_ works. This key is used primarily in macros. If you are in the editor mode, CTRL_ places you in escape mode. If you are in escape mode, it does nothing. In edit mode, ⊙CTRL_ does nothing; in escape mode, it returns you to edit mode. This lets you quickly get into the mode you need to be in at the start of an editor macro, regardless of the mode you are in when the macro is executed.

The remainder of this chapter describes the standard edit mode.

Auto Indent

You can set the editor so that RETURN moves the cursor to the first column of the next line, or so that it follows indentations already set in the text. If the editor is set to put the cursor on column 1 when you press RETURN then changing this mode causes the editor to put the cursor on the first non-space character in the next line; if the line is blank, then the cursor is placed under the first non-space character in the first non-blank line above the cursor. The first mode is generally best for line-oriented languages, like assembly language. The second is handy for block-structured languages like GSoft BASIC.

To change the return mode, type ⊙RETURN.

Select Text

You can use the mouse or the keyboard to select text in the ORCA editor. This section deals with the keyboard selection mechanism; see *Using the Mouse*, later in this chapter, for information about selecting text with the mouse.

The Cut, Copy, Delete and Block Shift commands require that you first select a block of text. The ORCA editor has two modes for selecting text: line-oriented and character-oriented selects. As you move the cursor in line-oriented select mode, text or code is marked a line at a time. In the character-oriented select mode, you can start and end the marked block at any character. Line-oriented select mode is the default for assembly language; for text files and most high-level languages, character-oriented select mode is the default.

While in either select mode, the following cursor-movement commands are active:

- bottom of screen
- top of screen
- cursor down
- cursor up
- start of line
- screen moves

In addition, while in character-oriented select mode, the following cursor-movement commands are active:

- cursor left
- cursor right
- end of line
- tab
- tab left
- word right
- word left

As you move the cursor, the text between the original cursor position and the final cursor position is marked (in inverse characters). Press RETURN to complete the selection of text. Press the ESC key to abort the operation, leave select mode, and return to normal editing.

To switch between character-oriented and line-oriented selection while in the editor, type Ć⌥x.

Hidden Characters

There are cases where line wrapping or tab fields may be confusing. Is there really a new line, or was the line wrapped? Do those eight blanks represent eight spaces, a tab, or some combination of spaces and tabs? To answer these questions, the editor has an alternate display mode that shows hidden characters. To enter this mode, type Ć=; you leave the mode the same way. While you are in the hidden character mode, end of line characters are displayed as the mouse text return character. Tabs are displayed as a right arrow where the tab character is located, followed by spaces until the next tab stop.

Macros

You can define up to 26 macros for the ORCA editor, one for each letter on the keyboard. A macro allows you to substitute a single keystroke for up to 128 predefined keystrokes. A macro can contain both editor commands and text, and can call other macros.

To create a macro, press ⌘ESC. The current macro definitions for A to J appear on the screen. The LEFT-ARROW and RIGHT-ARROW keys can be used to switch between the three pages of macro definitions. To replace a definition, press the key that corresponds to that macro, then type in the new macro definition. You must be able to see a macro to replace it - use the left and right arrow keys to get the correct page. Press OPTION ESC to terminate the macro definition. You can include ⌘ CTRL key combinations, OPTION key combinations, and the RETURN, ENTER, ESC, and arrow keys. The following conventions are used to display keystrokes in macros:

Key	What You See
CTRL key	The uppercase character key is shown in inverse.
⌘ key	An inverse A followed by key (for example, AK)
OPTION key	An inverse B followed by key (for example, BK)
ESC	An inverse left bracket (CTRL [).
RETURN	An inverse M (CTRL M).
ENTER	An inverse J (CTRL J).
UP-ARROW	An inverse K (CTRL K).
DOWN-ARROW	An inverse J (CTRL J).
LEFT-ARROW	An inverse H (CTRL H).
RIGHT-ARROW	An inverse U (CTRL U).
DELETE	A block

Each ⌘ key combination or OPTION key combination counts as two keystrokes in a macro definition. Although an ⌘ key combination looks (in the macro definition) like a CTRL A followed by key, and an OPTION key combination looks like a CTRL B followed by key, you cannot enter CTRL A when you want an ⌘ or CTRL B when you want an OPTION key.

If you make a mistake typing a macro definition you can back up with option-DELETE. If you wish to retype the macro definition, press OPTION ESC to terminate the definition, press the letter key for the macro you want to define, and begin over. When you are finished entering macros, press OPTION ESC to terminate the last option definition, then press OPTION to end macro entry. If you have entered any new macro definitions, a dialog will appear asking if you want to save the macros to disk; select OK to save the new macro definitions, and Cancel to return to the editor. If you select Cancel, the macros you have entered will remain in effect until you leave the editor.

Macros are saved on disk in the file SYSEMAC in the shell prefix. If you are using the GSoft BASIC environment, the shell prefix is located in the same directory as GSoft.SYS16.

To execute a macro, hold down OPTION and press the key corresponding to that macro.

Using Editor Dialogs

The text editor makes use of a number of dialogs for operations like entering search strings, selecting a file to open, and informing you of error conditions. The way you select options, enter text, and execute commands in these dialogs is the same for all of them.

Figure 6.1 shows the Search and Replace dialog, one of the most comprehensive of all of the editor's dialogs, and one that happens to illustrate many of the controls used in dialogs.

```
Search string:
┌──────────────────────────────────────────────┐
│                                              │
└──────────────────────────────────────────────┘

Replace string:
┌──────────────────────────────────────────────┐
│                                              │
└──────────────────────────────────────────────┘

◇1 White space compares equal
◇2 Case sensitive
◇3 Whole word
◇4 Replace all

[Replace]  ◇0 [Cancel]
```

Figure 6.1

The first item in this dialog is an edit line control that lets you enter a string. When the dialog first appears, the cursor is at the beginning of this line. You can use any of the line editing commands from throughout the ORCA programming environment to enter and edit a string in this edit line control; these line editing commands are summarized in the table below.

command	command name and effect
LEFT-ARROW	**cursor left** - The cursor will move to the left.
RIGHT-ARROW	**cursor right** - The cursor will move to the right.
⌘> or ⌘.	**end of line** - The cursor will move to the right-hand end of the string.
⌘< or ⌘,	**start of line** - The cursor will move to the left-hand end of the string.
⌘Y or CTRL-Y	**delete to end of line** - Deletes characters from the cursor to the end of the line.
⌘Z or CTRL-Z	**undo** - Resets the string to the starting string.
ESC or CTRL-X	**exit** - Stops string entry, leaving the dialog without changing the default string or executing the command.
⌘E or CTRL-E	**toggle insert mode** - Switches between insert and over strike mode. The dialog starts out in the same mode as the editor, but switching the mode in the dialog does not change the mode in the editor.
DELETE	**delete character left** - Deletes the character to the left of the cursor, moving the cursor left.

The Search and Replace dialog has two edit line items; you can move between them using the TAB key. You may also need to enter a tab character in a string, either to search specifically for a string that contains an imbedded tab character, or to place a tab character in a string that will replace the string once it is found. To enter a tab character in an edit line string, use ⌘TAB. While only one space will appear in the edit line control, this space does represent a tab character.

Four options appear below the edit line controls. Each of these options is preceded by an ⌘ character and a number. Pressing ⌘x, where x is the number, selects the option, and causes a check mark to appear to the left of the option. Repeating the operation deselects the option, removing the check mark. You can also select and deselect options by using the mouse to position the cursor over the item, anywhere on the line from the ⌘ character to the last character in the label, and clicking.

At the bottom of the dialog is a pair of buttons; some dialogs have more than two, while some have only one. These buttons cause some action to occur. In general, all but one of these buttons will have an ⌘ character and a number to the left of the button. You can select a button in one of several ways: by clicking on the button with the mouse, by pressing the RETURN key (for the default button, which is the one without an ⌘ character), by pressing ⌘x, or by pressing the first letter of the label on the button. (For dialogs with an edit line item, the last option is not available.)

Once an action is selected by pressing a button, the dialog will vanish and the action will be carried out.
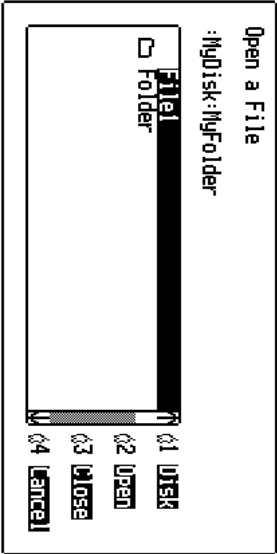
Figure 6.2 shows the Open dialog. This dialog contains a list control, used to display a list of files and directories.

You can scroll through the list by clicking on the arrows with the mouse, dragging the thumb with the mouse (the thumb is the space in the gray area between the up and down arrows), clicking in the gray area above or below the thumb, or by using the up and down arrow keys.

If there are any files in the list, one will always be selected. For commands like Open that require a file name you will be able to select any file in the list; for commands like New that present the file list so you know what file names are already in use, only directories can be selected. You can change which file is selected by clicking on another file or by using the up or down arrow keys. If you click on the selected name while a directory is selected, the directory is opened. If you click on a selected file name, the file is opened.

Figure 6.2

Using the Mouse

All of the features of the editor can be used without a mouse, but the mouse can also be used for a number of functions. If you prefer not to use a mouse, simply ignore it. You can even disconnect the mouse, and the ORCA editor will perform perfectly as a keyboard-based editor.

The most common use for the mouse is moving the cursor and selecting text. To position the cursor anywhere on the screen, move the mouse. As soon as the mouse is moved, an arrow will appear on the screen; position this arrow where you would like to position the cursor and click.

Several editor commands require you to select some text. You can select the text before using the command by clicking to start a selection, then dragging the mouse while holding down the button while you move to the other end of the selection. Unlike keyboard selection, mouse selections are always done in character select mode. You can also select words by double-clicking to start the selection, or lines by triple clicking to start the selection. Finally, if you drag the mouse off of the screen while selecting text, the editor will start to scroll one line at a time.

The mouse can also be used to select dialog buttons, change dialog options, and scroll list items in a dialog. See *Using Editor Dialogs* in this chapter for details.

64

Command Descriptions

This section describes the functions that can be performed with editor commands. The key assignments for each command are shown with the command description.

Screen-movement descriptions in this manual are based on the direction the display screen moves through the file, not the direction the lines appear to move on the screen. For example, if a command description says that the screen scrolls down one line, it means that the lines on the screen move up one line, and the next line in the file becomes the bottom line on the screen.

CTRL@   **About**

Shows the current version number and copyright for the editor. Press any key or click the mouse button to get rid of the About dialog.

CTRLG   **Beep the Speaker**

The ASCII control character BEL ($07) is sent to the output device. Normally, this causes the speaker to beep.

⌘, or ⌘<   **Beginning of Line**

The cursor is placed in column one of the current line.

⌘DOWN-ARROW   **Bottom of Screen / Page Down**

The cursor moves to the last visible line on the screen, preserving the cursor's horizontal position. If the cursor is already at the bottom of the screen, the screen scrolls down twenty-two lines.

CTRLC or ⌘C   **Copy**

When you execute the Copy command, the editor enters select mode, as discussed in the section *Select Text* in this chapter. Use cursor-movement or screen-scroll commands to mark a block of text (all other commands are ignored), then press RETURN. The selected text is written to the file SYSTEMP in the work prefix. (To cancel the Copy operation without writing the block to SYSTEMP, press ESC instead of RETURN.) Use the Paste command to place the copied material at another position in the file.

CTRLW or ⌘W   **Close**

Closes the active file. If the file has been changed since the last update, a dialog will appear, giving you a chance to abort the close, save the changes, or close the file without saving the changes. If the active file is the only open file, the editor exits after closing the file; if there are other open files, the editor selects the next file to become the active file.

DOWN-ARROW   **Cursor Down**

The cursor is moved down one line, preserving its horizontal position. If it is on the last line of the screen, the screen scrolls down one line.

LEFT-ARROW   **Cursor Left**

The cursor is moved left one column. If it is in column one, the command is ignored.

RIGHT-ARROW   **Cursor Right**

The cursor is moved right one column. If it is on the end-of-line marker (usually column 80), the command is ignored.

UP-ARROW   **Cursor Up**

The cursor is moved up one line, preserving its horizontal position. If it is on the first line of the screen, the screen scrolls up one line. If the cursor is on the first line of the file, the command is ignored.

CTRL-X or ⌂X   **Cut**

When you execute the Cut command, the editor enters select mode, as discussed in the section *Select Text* in this chapter. Use cursor-movement or screen-scroll commands to mark a block of text (all other commands are ignored), then press RETURN. The selected text is written to the file SYSTEMP in the work prefix, and deleted from the file. (To cancel the Cut operation without cutting the block from the file, press ESC instead of RETURN.) Use the Paste command to place the cut text at another location in the file.

⌂ESC   **Define Macros**

The editor enters the macro definition mode. Press OPTION ESC to terminate a definition, and OPTION to terminate macro definition mode. The macro definition process is described in the section *Macros* in this chapter.

⌂DELETE   **Delete**

When you execute the delete command, the editor enters select mode, as discussed in the section *Select Text* in this chapter. Use any of the cursor movement or screen-scroll commands to mark a block of text (all other commands are ignored), then press RETURN. The selected text is deleted from the file. (To cancel the delete operation without deleting the block from the file, press ESC instead of RETURN.)

CTRL-F or ⌂F   **Delete Character**

The character that the cursor is on is deleted and put in the Undo buffer (see the description of the Undo command). Characters to the right of the cursor are moved one column to the left to fill in the gap.

DELETE or CTRL-D   **Delete Character Left**

The character to the left of the cursor is deleted, and the character that the cursor is on, as well as the rest of the line to the right of the cursor, are moved 1 column to the left to fill in the gap. If the cursor is in column one and the over strike mode is active, no action is taken. If the cursor is in column one and the insert mode is active, then the line the cursor is on is appended to the line

above and the cursor remains on the character it was on before the delete. Deleted characters are put in the undo buffer.

⌃T or CTRLT

The line that the cursor is on is deleted, and the following lines are moved up one line to fill in the space. The deleted line is put in the Undo buffer (see the description of the Undo command).

CTRLY or ⌃Y

Delete Line

The character that the cursor is on, and all those to the right of the cursor to the end of the line, are deleted and put in the Undo buffer (see the description of the Undo command).

⌃G

Delete to EOL

When you execute the delete word command, the cursor is moved to the beginning of the word it is on, then delete character commands are executed for as long as the cursor is on a non-space character, then for as long as the cursor is on a space. This command thus deletes the word plus all spaces up to the beginning of the next word. If the cursor is on a space, that space and all following spaces are deleted, up to the start of the next word. All deleted characters, including spaces, are put in the Undo buffer (see the description of the Undo command).

⌃. or ⌃>

Delete Word

If the last column on the line is not blank, the cursor moves to the last column. If the last column is blank, then the cursor moves to the right of the last non-space character in the line. If the entire line is blank, the cursor is placed in column one.

⌃? or ⌃/

End of Line

Displays the help file, which contains a short summary of editor commands. Use ESC to return to the file being edited.

The help file is a text file called SYSHELP, found in the shell prefix. Since it is a text file, you can modify it as desired.

⌃B or CTRLB

Help

A blank line is inserted at the cursor position, and the line the cursor was on and the lines below it are scrolled down to make room. The cursor remains in the same horizontal position on the screen.

⌃SPACEBAR

Insert Line

A space is inserted at the cursor position. Characters from the cursor to the end of the line are moved right to make room. Any character in column 255 on the line is lost. The cursor remains in the same position on the screen. Note that the Insert Space command can extend a line past the end-of-line marker.

Insert Space

New

CTRL-N or ⌘N

A dialog like the one shown in Figure 6.3 appears. You need to enter a name for the new file. After entering a name, the editor will open an empty file using one of the ten available file buffers. The file's location on disk will be determined by the directory showing in the dialog's list box.

While the New command requires selecting a file name, no file is actually created until you save the file with the Save command.

New File Name
:MyDisk:MyFolder

File1
Folder

◇1 Disk
◇2 Open
◇3 Close
◇4 Cancel

File Name

◇5 Save

Figure 6.3

Open

CTRL-O or ⌘O
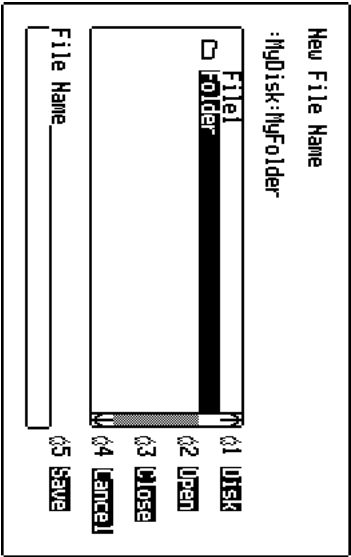
The editor can edit up to ten files at one time. When the open command is used, the editor moves to the first available file buffer, then brings up the dialog shown in Figure 6.4. If there are no empty file buffers, the editor beeps, and the command is aborted.

Open a File
:MyDisk:MyFolder

File1
Folder

◇1 Disk
◇2 Open
◇3 Close
◇4 Cancel

Figure 6.4

Selecting Disk brings up a second dialog that shows a list of the disks available. Selecting one changes the list of files to a list of the files on the selected disk.

When you use the open button, if the selected file in the file list is a TXT or SRC file, the file is opened. GSoft BASIC tokenized files saved with the SAVE command cannot be opened with the open command, but GSoft BASIC files saved with the SSAVE command are SRC files, and those saved with the TSAVE command are TXT files, and both TXT and SRC files can be opened. If a directory is selected, the directory is opened, and the file list changes to show the files inside the directory. You can also open a file by first selecting a file, then clicking on it with the mouse.

If a directory is open, the close button closes the directory, showing the list of files that contains the directory. You can also close a directory by clicking on the path name shown above the file list. If the file list was created from the root volume of a disk, the close button does nothing.

The cancel button leaves the open dialog without opening a file.

For information on how to use the various controls in the dialog, see *Using Editor Dialogs* in this chapter.

CTRL V or ⌘V

Paste

The contents of the SYSTEMP file are copied to the current cursor position. If the editor is in line-oriented select mode the line the cursor is on and all subsequent lines are moved down to make room for the new material. If the editor is in character-oriented select mode the material is copied at the cursor column. If enough characters are inserted to make the line longer than 255 characters, the excess characters are lost.

CTRL Q or ⌘Q

Quit

The quit command leaves the editor. If any file has been changed since the last time it was saved to disk, each of the files, in turn, will be made the active file, and the following dialog will appear:

Save Changes?

Yes   No   Cancel

Figure 6.5

If you select Yes, the file is saved just as if the Save command had been used. If you select No, the file is closed without saving any changes that have been made. Selecting Cancel leaves you in the editor with the active file still open, but if several files had been opened, some of them may have been closed before the Cancel operation took affect.

CTRL R or ⌘R

Remove Blanks

If the cursor is on a blank line, that line and all subsequent blank lines up to the next non-blank line are removed. If the cursor is not on a blank line, the command is ignored.

1 to 32767

Repeat Count

When in escape mode, you can enter a repeat count (any number from 1 to 32767) immediately before a command, and the command is repeated as many times as you specify (or as many times as is possible, whichever comes first). Escape mode is described in the section *Modes* in this chapter.

RETURN

Return

The RETURN key works in one of two ways, depending on the setting of the auto-indent mode toggle: 1) to move the cursor to column one of the next line; or 2) to place the cursor on the first non-space character in the next line, or, if the line is blank, beneath the first non-space character in the first non-blank line on the screen above the cursor. If the cursor is on the last line on the screen, the screen scrolls down one line.

If the editor is in insert mode, the RETURN key will also split the line at the cursor position.

CTRL-A or ⌘A

Save As

The Save As command lets you change the name of the active file, saving it to a new file name or to the same name in a new file directory. When you use this command, this dialog will appear:

```
Save as
:MyDisk:MyFolder

┌──────────────┐  ⌘1 [Disk]
│ File1        │
│□ Folder      │  ⌘2 [Open]
│              │
│              │  ⌘3 [Close]
│              │
│              │  ⌘4 [Cancel]
│▓▓░░░░░░░░░░░░│
└──────────────┘  ⌘5 [Save]

File Name
[_____]
```
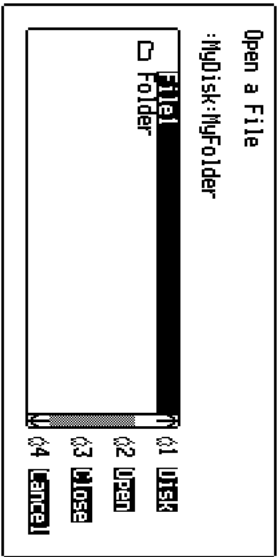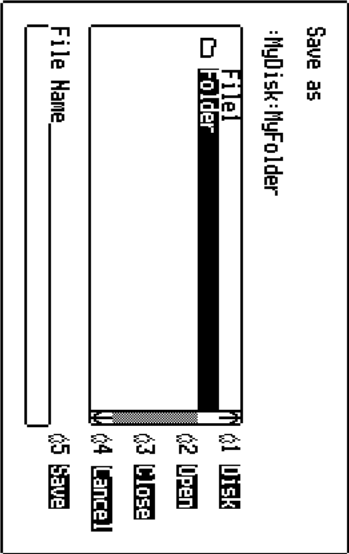
Figure 6.6

Selecting Disk brings up a second dialog that shows a list of the disks available. Selecting one changes the list of files to a list of the files on the selected disk.

When you use the Open button, the selected directory is opened. While using this command, you cannot select any files from the list; only directories can be selected.

If a directory is open, the close button closes the directory, showing the list of files that contains the directory. You can also close a directory by clicking on the path name shown above the file list. If the file list was created from the root volume of a disk, the close button does nothing.

The cancel button leaves the open dialog without opening a file.

70

The Save button saves the file, using the file name shown in the edit line item labeled "File Name." You can also save the file by pressing the RETURN key.

For information on how to use the various controls in the dialog, see *Using Editor Dialogs* in this chapter.

CTRL S or **⌂S**   **Save**

The active file (the one you can see) is saved to disk.

⌂-1 to ⌂-9   **Screen Moves**

The file is divided by the editor into 8 approximately equal sections. The screen-move commands move the file to a boundary between one of these sections. The command ⌂1 jumps to the first character in the file, and ⌂9 jumps to the last character in the file. The other seven ⌂n commands cause screen jumps to evenly spaced intermediate points in the file.

⌂}   **Scroll Down One Line**

The editor moves down one line in the file, causing all of the lines on the screen to move up one line. The cursor remains in the same position on the screen. Scrolling can continue past the last line in the file.

⌂]   **Scroll Down One Page**

The screen scrolls down twenty-two lines. Scrolling can continue past the last line in the file.

⌂{   **Scroll Up One Line**

The editor moves up one line in the file, causing all of the lines on the screen to move down one line. The cursor remains in the same position on the screen. If the first line of the file is already displayed on the screen, the command is ignored.

⌂[**Scroll Up One Page**

The screen scrolls up twenty-two lines. If the top line on the screen is less than one screen's height from the beginning of the file, the screen scrolls to the beginning of the file.

⌂L   **Search Down**

This command allows you to search through a file for a character or string of characters. When you execute this command, this dialog appears:
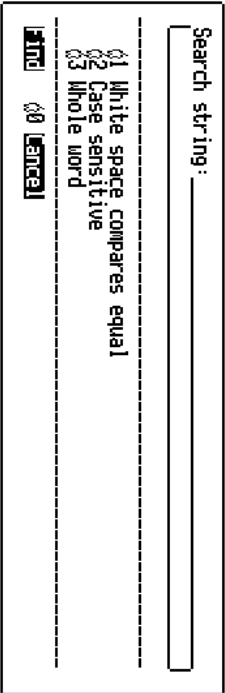
Figure 6.7

If you have previously entered a search string, the previous string appears after the prompt as a default. Type in the string for which you wish to search, and press RETURN. The cursor will be moved to the first character of the first occurrence of the search string after the old cursor position. If there are no occurrences of the search string between the old cursor position and the end of the file, an alert will show up stating that the string was not found; pressing any key will get rid of the alert.

By default, string searches are case insensitive, must be an exact match in terms of blanks and tabs, and will match any target string in the file, even if it is a subset of a larger word. All of these defaults can be changed, so we will look at what they mean in terms of how changing the defaults affects the way string searches work.

When you look at a line like

```
100    PRINT "Hello, world."
```

without using the hidden characters mode, it is impossible to tell if the spaces between the various fields are caused by a series of space characters, two tabs, or perhaps even a space character or two followed by a tab. This is an important distinction, since searching for 100<space><space><space>PRINT won't find the line if the 100 and PRINT are actually separated by a tab character, and searching for 100<tab>PRINT won't find the line if the fields are separated by three spaces. If you select the "white space compares equal" option, though, the editor will find any string where 100 and PRINT are separated by any combination of spaces and tabs, whether you use spaces, tabs, or some combination in the search string you type.

By default, if you search for print, the editor will also find PRINT, since string searches are case insensitive. Selecting the "case sensitive" option makes the string search case sensitive, so that the capitalization becomes significant. With this option turned on, searching for PRINT would not find print.

Sometimes when you search for a string, you want to find any occurrence of the string, even if it is imbedded in some larger word. For example, if you are scanning your program for places where it handles spaces, you might enter a string like "space". You would want the editor to find the word whitespace, though, and normally it would. If you are trying to scan through a source file looking for all of the places where you used the variable i, though, you don't want the editor to stop four times on the word Mississippi. In that case, you can select the "whole word" option, and the editor will only stop if it finds the letter i, and there is no other letter, number, or underscore

character on either side of the letter. These rules match the way languages deal with identifiers, so you can use this option to search for specific variable names – even a short, common one like i.

This command searches from the cursor position towards the end of the file. For a similar command that searches back towards the start of the file, see the Search Up command.

For a complete description of how to use the mouse or keyboard to set options and move through the dialog, see the section *Using Editor Dialogs* in this chapter.

Once a search string has been entered, you may want to search for another occurrence of the same string. ORCA ships with two built-in editor macros that can do this with a single keystroke, without bringing up the dialog. To search forward, use the option-L macro; to search back, use the option-K macro.

⌘K

Search Up

This command operates exactly like Search Down, except that the editor looks for the search string starting at the cursor and proceeding toward the beginning of the file. The search stops at the beginning of the file; to search between the current cursor location and the end of the file, use the Search Down command.

⌘J

Search and Replace Down

This command allows you to search through a file for a character or string of characters, and to replace the search string with a replacement string. When you execute this command, the following dialog will appear on the screen:

```
┌─────────────────────────────────┐
│ Search string:                  │
│ ┌─────────────────────────────┐ │
│ └─────────────────────────────┘ │
│                                 │
│ Replace string:                 │
│ ┌─────────────────────────────┐ │
│ └─────────────────────────────┘ │
│                                 │
│ ○1 White space compares equal   │
│ ○2 Case sensitive               │
│ ○3 Whole word                   │
│ ○4 Replace all                  │
│                                 │
│ ⟦Replace⟧  ⌘⊙ ⟦Cancel⟧          │
└─────────────────────────────────┘
```

Figure 6.8

The search string, the first three options, and the buttons work just as they do for string searches; for a description of these, see the Search Down command. The replace string is the target string that will replace the search string each time it is found. By default, when you use this command, each time the search string is found in the file you will see this dialog:

73

Figure 6.9

```
Replace with "target"?
⬧Replace  ⌘1 Skip  ⌘2 Cancel
```

If you select the Replace option, the search string is replaced by the replace string, and the editor scans forward for the next occurrence of the search string. Choosing Skip causes the editor to skip ahead to the next occurrence of the search string without replacing the occurrence that is displayed. Cancel stops the search and replace process.

If you use the "replace all" option, the editor starts at the top of the file and replaces each and every occurrence of the search string with the target string. On large files, this can take quite a bit of time. To stop the process, press ⌘. (open-apple period). While the search and replace is going on, you can see a spinner at the bottom right corner of the screen, showing you that the editor is still alive and well.

⌘H

This command operates exactly like Search and Replace Down, except that the editor looks for the search string starting at the cursor and proceeding toward the beginning of the file. The search stops at the beginning of the file; to search between the current cursor location and the end of the file, use the Search and Replace Down command. If you use the "replace all" option, this command works exactly the same way the Search and Replace Down command does when it uses the same option.

Search and Replace Up

option–~

The editor can edit up to ten files at one time. When you use this command, a dialog appears showing the names of the ten files in memory. You can then move to one of the files by pressing option–n, where n is one of the file numbers. You can exit the dialog without switching files by pressing ESC or RETURN.

See also the Switch Files command.

Select File

option–TAB

If there is a tab stop in the same column as the cursor, it is cleared; if there is no tab stop in the cursor column, one is set.

Set and Clear Tabs

option–[

If this command is issued when no text is selected, you enter the text selection mode. Pressing RETURN leaves text selection mode.

At any time while text is selected, using the command shifts all of the selected text left one character. This is done by scanning the text, one line at a time, and removing a space right before the first character on each line that is not a space or tab. If the character to be removed is a tab character, it is first replaced by an equivalent number of spaces. If there are no spaces or tabs at the start of the line, the line is skipped.

Shift Left

If a large amount of text is selected, this command may take a lot of time. While the editor is working, you will see a spinner at the bottom right of the screen; this lets you know the editor is still processing text. You can stop the operation by pressing ⌘., but this will leave the selected text partially shifted.

option-]

Shift   Right

RETURN leaves text selection mode.

If this command is issued when no text is selected, you enter the text selection mode. Pressing

At any time while text is selected, using the command shifts all of the selected text right one character. This is done by scanning the text, one line at a time, and adding a space right before the first character on each line that is not a space or tab. If this leaves the non-space character on a tab stop, the spaces are collected and replaced with a tab character. If a blank line is encountered, no action is taken.

If a large amount of text is selected, this command may take a lot of time. While the editor is working, you will see a spinner at the bottom right of the screen; this lets you know the editor is still processing text. You can stop the operation by pressing ⌘., but this will leave the selected text partially shifted.

option-n

Switch   Files

The editor can edit up to ten files at one time. Each of these files is numbered, starting from 0 and proceeding to 9. The numbers are assigned as the files are opened from the command line. To move from one file to the next, press option-n, where n is a numeric key.

When you switch files, the original file is not changed in any way. When you return to the file, the cursor and display will be in the same place, the undo buffer will still be active, and so forth. The only actions that are not particular to a specific file buffer are those involving the clipboard – Cut, Copy and Paste all use the same clipboard, so you can move chunks of text from one file to another.

See also the Select File command.

TAB

Tab

In insert mode, or when in over strike mode and the next tab stop is past the last character in the line, this command inserts a tab character in the source file and moves to the end of the tab field. If you are in the over strike mode and the next tab stop is not past the last character on the line, the Tab command works like a cursor movement command, moving the cursor forward to the next tab stop.

Some languages and utilities do not work well (or at all) with tab stops. If you are using one of these languages, you can tell the editor to insert spaces instead of tab characters; see the section *Setting Editor Defaults*, later in this chapter, to find out how this is done.

⌘TAB

Tab   Left

The cursor is moved to the previous tab stop, or to the beginning of the line if there are no more tab stops to the left of the cursor. This command does not enter any characters in the file.

⌘RETURN

Toggle Auto Indent Mode

If the editor is set to put the cursor on column one when you press RETURN, it is changed to put the cursor on the first non-space character; if set to the first non-space character, it is changed to put the cursor on column one. Auto-indent mode is described in the section *Modes* in this chapter.

ESC

Toggle Escape Mode

If the editor is in the edit mode, it is put in escape mode; if it is in escape mode, it is put in edit mode. When you are in escape mode, pressing any character not specifically assigned to an escape-mode command returns you to edit mode. Escape and edit modes are described in the section *Modes* in this chapter.

When in escape mode, ⌘CTRL_ will return you to edit mode. In edit mode the command has no effect. From edit mode, CTRL_ will place you in escape mode, but the command has no effect in escape mode. These commands are most useful in an editor macro, where you do not know what mode you are in on entry.

CTRLE or ⌘E

Toggle Insert Mode

If insert mode is active, the editor is changed to over strike mode. If over strike mode is active, the editor is changed to insert mode. Insert and over strike modes are described in the section *Modes* in this chapter.

CTRL⌘X

Toggle Select Mode

If the editor is set to select text for the Cut, Copy, and Delete commands in units of one line, it is changed to use individual characters instead; if it is set to character-oriented selects, it is toggled to use whole lines. See the section *Modes* in this chapter for more information on select mode.

⌘UP-ARROW

Top of Screen / Page Up

The cursor moves to the first visible line on the screen, preserving the cursor's horizontal position. If the cursor is already at the top of the screen, the screen scrolls up twenty-two lines. If the cursor is at the top of the screen and less than twenty-two lines from the beginning of the file, then the screen scrolls to the beginning of the file.

CTRLZ or ⌘Z

Undo Delete

The last operation that changed the text in the current edit file is reversed, leaving the edit file in the previous state. Saving the file empties the undo buffer, so you cannot undo changes made before the last time the file was saved.

The undo operation acts like a stack, so once the last operation is undone, you can undo the one before that, and so on, right back to the point where the file was loaded or the point where the file was saved the last time.

Word Left

CLEFT-ARROW

The cursor is moved to the beginning of the next non-blank sequence of characters to the left of its current position. If there are no more words on the line, the cursor is moved to the last word in the previous line or, if it is blank, to the last word in the first non-blank line preceding the cursor.

Word Right

CRIGHT-ARROW

The cursor is moved to the start of the next non-blank sequence of characters to the right of its current position. If there are no more words on the line, the cursor is moved to the first word in the next non-blank line.

Setting Editor Defaults

When you start the ORCA editor, it reads the file named SysTabs (located in the shell prefix), which contains the default settings for tab stops, return mode, insert mode, tab mode, and select mode. The SysTabs file is an ASCII text file that you can edit with the ORCA editor.

Each language recognized by ORCA is assigned a language number. The SysTabs file has three lines associated with each language:

1. The language number.
2. The default settings for the various modes.
3. The default tab and end-of-line-mark settings.

The first line of each set of lines in the SysTabs file specifies the language that the next two lines apply to. ORCA languages can have numbers from 0 to 32767 (decimal). The language number must start in the first column; leading zeros are permitted and are not significant, but leading spaces are not allowed.

The second line of each set of lines in the SysTabs file sets the defaults for various editor modes, as follows:

1. If the first column contains a zero, pressing RETURN in the editor causes the cursor to go to column one in the next line; if it's a one, pressing RETURN sends the cursor to the first non-space character in the next line (or, if the line is blank, beneath the first non-space character in the first non-blank line on the screen above the cursor).

2. If the second character is zero, the editor is set to line-oriented selects; if one, it is set to character-oriented selects.

3. This flag is not used by the current version of the ORCA editor. It should be set to 0.

4. The fourth character is used by the ORCA/Desktop editor, and is used to set the default cursor mode. A zero will cause the editor to start in over strike mode; a one causes the editor to start in insert mode.

5. If the fifth character is a 1, the editor inserts a tab character in the source file when the Tab command is used to tab to a tab stop. If the character is a 0, the editor inserts an appropriate number of spaces, instead.

6. If the sixth character is a 0, the editor will start in over strike mode; if it is a 1, the editor starts in insert mode. Using a separate flag for the text based editor (this one) and the desktop editor (see the fourth flag) lets you enter one mode in the desktop editor, and a different mode in the text based editor.

7. The seventh character is used by the ORCA/Desktop editor to indicate if a file uses the old or new style of imbedded debug character. GSoft BASIC doesn't use either one.

8. If the eighth character is a 0, the editor saves the cursor position, tab stops and these flags.

For example, the following lines define the defaults for GSoft BASIC:

```
260
10011001
0000001000000010000000010000000100000001000000010000000100000001000
0010000000100000001000000010000000100000001000000010000000010000000
0001000000010000000100000001000000010000000100000001000000010000000
000001000000010000000002
```

The third line of each set of lines in the SysTabs file sets default tab stops. There are 255 zeros and ones, representing the 255 character positions available on the edit line. The ones indicate the positions of the tab stops. A two in any column of this line sets the end of the line; if the characters extend past this marker, the line is wrapped. The column containing the two then replaces the default end-of-line column (the default right margin) when the editor is set to that language.

The last line continues on for a total of 255 characters, so it is too long to show on one line in this manual.

If no defaults are specified for a language (that is, there are no lines in the SysTabs file for that language), then the editor assumes the following defaults:

• RETURN sends the cursor to column one.
• Line-oriented selects.
• Word wrapping starts in column 80.
• There is a tab stop every eighth column.
• The editor starts in over strike mode.
• Tab characters are inserted to create tabbed text.

Note that you can change tabs and editing modes while in the editor.

Chapter 7 – Program Symbols

BASIC programs are made up of a series of program symbols called tokens. Tokens are the words used to write a program. They consist of identifiers, symbols, and constants. These tokens form lines, which are also a fundamental part of the BASIC language—some commands, like SUB, must appear at the start of a line; most commands are restricted to appearing on a single line; and some commands cannot have anything following them on a line.

Identifiers

Identifiers in BASIC start with an alphabetic character or underscore and are followed by zero or more alphabetic characters, numeric characters, or underscores. The last character in the identifier is an optional type character. BASIC is a case insensitive language, which means that the identifiers matrix and Matrix are the same identifier.

Because GSoft BASIC is interpreted, the program is converted to a more efficient internal storage format than plain ASCII text. In the process, all identifiers are converted to uppercase letters. There is nothing preventing you from maintaining the program as a text file, especially if you use the version of GSoft BASIC that runs from the ORCA shell, but there is no way to stop this conversion to uppercase letters.

GSoft BASIC imposes a limit of 255 characters on the length of any single identifier.

Type characters indicate the default data type for the identifier. If no type character is used, the default type is single-precision real. You can override the default type using the DIM statement.

The type character, if it appears at all, becomes a part of the identifier. For example, R and R! both default to a data type of single-precision real, but they are different identifiers, and cannot be used interchangeably.

The type characters and their equivalent BASIC data types are shown in the table blow. The internal formats for the data types are described in detail in the next chapter.

Character	Type
~	BYTE
%	INTEGER
&	LONG
!	SINGLE
#	DOUBLE
$	STRING

identifier.

Some examples of legal BASIC identifiers are shown below. They each represent a different identifier.

MAIN	ARRAY	my_var	S$	I%	B~
_subroutine	X1	_	D#	L&	R!

Reserved Words

Reserved words are identifiers that have special meaning in BASIC. A reserved word can only be used for the meaning that BASIC assigns to it, except that reserved words can appear in comments or string constants. The reserved words in BASIC are shown below.

ABS	ALLOCATE	AND	APPEND	AS	ASC	
AT	ATN	BINARY	BREAK	BYTE	CALL	CASE
CDBL	CHDIR	CHR$	CINT	CLEAR	CLNG	CLOSE
CONT	COS	CSRLIN	CSNG	CURDIR$	DATA	DEF
DIM	DIR$	DISPOSE	DO	DOUBLE	ELSE	END
EOF	ERL	ERR	ERROR	EXP	FN	
FOR	FRE	FUNCTION	GET	GOSUB	GOTO	GSOS
HCOLOR=	HGR	HOME	HPLOT	HTAB	IF	INPUT
INT	INTEGER	INVERSE	KILL	LEFT$	LEN	
LET	LINE	LOOP	MID$	LOC	LOF	
LOG	LONG	LOADLIBRARY	NORMAL	MKDIR	MOUSETEXT	
NAME	NEXT	NIL	OUTPUT	NOT	ON	
ONERR	OPEN	OR	PRINT	PEEK	POINTER	
POKE	POP	POS	RESUME	PUT	RANDOM	
READ	REM	RESTORE	SELECT	RETURN	RIGHT$	
RMDIR	RND	SEEK	SPC	SETMEM	SGN	
SIN	SINGLE	SIZEOF	STRING	SPEED=	SQR	
STEP	STOP	STR$	THEN	SUB	TAB	
TAN	TCP	TEXT	WAIT	TO	TOOL	LIBRARY
TOOLERROR	TYPE	UNLOADLIBRARY		UNTIL	WHILE	
USING	VAL	VTAB		WEND		

Reserved Symbols

Reserved symbols are the punctuation of the BASIC language. Reserved symbols are used as mathematical operators, for forming array subscripts and parameter lists, for separating statements, and so forth. With some restrictions, reserved symbols can also be used in comments and string constants. See the sections below for details.

The reserved symbols in BASIC are:

! : + – * /

^ < = @ ( )

> ,

Constants

Constants are used to place numbers and strings into the source code of the program. Each kind of constant has its own unique format, so they are discussed separately.

Decimal Integers

Decimal integers come in two sizes, referred to as integer and long integer.

Integers consist of one to five digits. The number represented must range from 0 to 32767. You may use a leading - character to form a negative number, although the - character and the number are technically two separate tokens. In practice, this technical distinction is not important.

If the number exceeds 32767, the number becomes a long integer. Long integer constants can range from 32768 to 2147483647.

The table below shows some examples of legal decimal constants.

0	58	32767	32768	400000

Hexadecimal Integers

Hexadecimal numbers are integers represented in base sixteen, rather than the more familiar base ten. Hexadecimal numbers are made up of the digits 0 to 9 and the letters a to f or A to F. In BASIC, hexadecimal numbers are distinguished from decimal numbers by a leading $ character.

As with decimal integers, hexadecimal integers can be long or short. Any hexadecimal constant with 5 or more digits—even if those digits are zero—is a long integer constant. Hexadecimal constants with four or less digits are integer constants.

Integers and long integers are stored in two's complement notation, so hexadecimal constants can be negative. Any value with the most significant bit set is a negative number.

If you are not familiar with hexadecimal notation and two's complement notation you can find out more in most assembly language books and many general computer science books.

The table below shows some legal hexadecimal constants, along with their decimal equivalent and the kind of integer created.

hexadecimal	decimal	size
$0	0	integer
$00000	0	long
$000A	10	integer
$0010	16	integer
$8000	-32768	integer
$08000	32768	long
$7FFF	32767	integer
$7FFFFFFF	2147483647	long
$FFFF	-1	integer

Real Numbers

Floating-point constants are used to represent numbers that do not have an integral value, or that cannot be represented using an integer because they are too large or too small. The general format is a sequence of digits, followed by a decimal point, followed by another sequence of digits, and an exponent, as in

3.14159e-14

The exponent can start with either an uppercase E, or a lowercase e, as shown.

The format for floating-point constants can vary quite a bit from this general form. You can leave out the digit sequence before or after the decimal point, as in 1.e10 or .1e10. In fact, you can leave off the exponent, too, as in 1. or .1. You must have either an exponent or a decimal point, but if you specify an exponent, you can omit the decimal point, as in 12e40.

String Constants

String constants consist of any sequence of characters except the end of line or the quote character, enclosed in quote characters.

Internally, strings are represented as a sequence of bytes, one for each character in the string, followed by a terminating null byte. (A null byte has a value of zero.) The value of each byte is the ordinal value for the character, as specified by the ASCII character set, described in Appendix C.

Here are some legal string constants:

```
"Hello, world."
""
" "
```

White Space

White space characters consist of the characters created by the space bar and tab key.

White space characters can appear between any two tokens, but may not be used between the characters of a token. The interpreter strips whitespace characters from your program as it converts it to internal, tokenized form, and prints white space characters between tokens when you list or edit the program. You have no control over where these whitespace characters are placed, other than maintaining a copy of your program as an ASCII file.

Comments

!   any-ascii-characters

REM   any-ascii-characters

Technically, comments in BASIC are a statement that is executed at run time. The statement skips to the start of the next line, ignoring any characters that appear after the comment command. GSoft BASIC allows absolutely any character to appear between the comment character and the end of the line, although the editor effectively limits the allowed characters to those available on the keyboard.

There are two distinct comment commands in GSoft BASIC. REM is an almost universal comment command in implementations of BASIC. The ! character is less common, but not unusual. Both of these work the same way, and are equivalent in all respects.

Chapter 8 – Types of Data

This chapter describes those BASIC data types which are built into the language. The next chapter covers derived and user-defined data types.

Some of the information in this chapter deals with the way information is stored internally in the memory of the computer. This information is provided for very advanced programmers who need to write assembly language subroutines that will deal with BASIC data, or who need to do strange and dangerous tricks with the data to work with the machine at the hardware level. You do not need to understand this information to use GSoft BASIC for normal BASIC programming. If it does not make sense to you, or if you will not be using the information, simply ignore it.

Integers

GSoft BASIC supports three different types of integers, BYTE, INTEGER and LONG.

Integers defined as BYTE are the smallest. BYTE values require one byte of storage. The allowed values range from 0 to 255. By default, identifiers ending with the character ~ are defined as BYTE integers.

INTEGER variables require two bytes of storage. INTEGER values can range from -32768 to 32767. By default, identifiers ending with the character % are defined as INTEGER variables.

Integers defined as LONG require four bytes of storage. LONG values range from -2147483648 to 2147483647. By default, identifiers ending with the character & are defined as LONG variables.

Internally, all integers are represented as binary values. Negative integers are represented as two's complement numbers. All of the integers that occupy more than one byte of storage are stored with the least significant byte first, proceeding to the most significant byte. This is the natural byte order for the 65816 processor used in the Apple IIGS.

It is likely that there will be versions of GSoft BASIC for other platforms, and many, including the Macintosh, use a different byte order for integer values. If you write programs that make assumptions about the byte order for integers, your program will not execute properly on some other computers.

Reals

GSoft BASIC supports two storage formats for real numbers.

SINGLE numbers are stored in the IEEE floating point number format, with the least significant byte first. They require four bytes of storage. SINGLE values are accurate to seven decimal digits, and allow exponents with a range of about 1e-38 to 1e38. By default, identifiers ending with the character ! or without a type character are defined as SINGLE variables.

85

DOUBLE numbers require eight bytes of storage each. They are stored least significant byte first, using the IEEE floating point format. They are accurate to fifteen decimal digits, and allow exponent ranges from 1e-308 to 1e308. By default, identifiers ending with the character # are defined as DOUBLE variables.

Infinity

The IEEE number format supports infinity. This is a specially coded value that indicates the number is too large to represent. It is signed, so you can have either positive infinity or negative infinity. The value prints as inf or -inf, depending on the sign.

All numeric operations in GSoft BASIC support this number, and do reasonable things with it based on the mathematical concept of infinity. For example, adding any number to infinity is still infinity, while dividing a number by infinity gives zero.

In some cases, your calculations may actually work fine, even if an intermediate result is infinity. In other cases, detecting a value of infinity is simply an indication that you need to tune your algorithms or switch from SINGLE to DOUBLE.

If your program treats a numeric overflow as an error, you can check for the overflow by comparing the result with infinity. The code snippet shows one way to create a value to compare with, and handles both negative and positive infinity. This particular example generates a run time error when the number X is infinity. Unless intercepted by an ONERR GOTO error handler, ERROR 25 will generate an error message that prints "Illegal quantity error."

Snippet

```
INF = 1E50
IF ABS(X) = INF THEN ERROR 25
```

NaN

The IEEE number format supports an error number that prints as NaN; this stands for "not a number." It is generated whenever a numeric result is simply not valid. An example is taking the square root of a negative number.

All numeric operations in GSoft BASIC support this number. In general, it propagates through the calculation, so that all of the values that are based on NaN are also NaN. For example, adding any number to NaN results in NaN.

In most cases, NaN indicates an error condition, but the fact that it doesn't trigger an immediate halt to the program or force you to write a tricky ONERR GOTO handler gives you more options for handling the error gracefully in your program.

For those cases where you want to stop the program when NaN is detected, the code snippet shows how to test for the condition and flag a run time error. Unless intercepted by an ONERR GOTO error handler, ERROR 19 will generate an error message that prints "Illegal quantity error."

Snippet

```
NAN = SQR (-1)
IF X = NAN THEN ERROR 25
```

Strings

Strings are stored internally as a sequence of ASCII characters terminated by a null character. The null character is a character with an ordinal value of 0.

Unlike all other variable types, strings are stored in special buffer to allow their length to change without using large amounts of memory. A string value changes, the old value is abandoned and a new string is allocated at the bottom of the string heap. When a string value changes, the old value is actually a four byte pointer into a dynamic string heap. When a string value changes, the old value is abandoned and a new string is allocated at the bottom of the string heap. If the heap grows to fill all available memory, garbage collection is performed, squashing all strings together to recover unused space.

A side effect of this storage method is that you cannot change the length of a string from a toolbox subroutine. You can return a string; GSoft BASIC allocates space for the value you return in the string buffer in the normal way. You can also safely change the characters in a string as long as the length of the string does not change.

By default, identifiers ending with the character $ are defined as STRING variables.

See SETMEM for a way to change the size of the variable buffer used by strings. See FRE for a way to determine the amount of free space available in this buffer, as well as for a way to force garbage collection so it does not occur at inopportune times.

Pointers

Pointers are represented internally as four-byte unsigned numbers, with the least significant byte stored first. A value of zero is used to represent a null pointer. Using type casting, pointers can be treated as long integers in mathematical equations with no loss of precision.

The memory of the Apple IIGS is divided into units of 8 bits called bytes. Each of these bytes has a unique address, represented as a number between 0 and 16777215. If you convert any valid pointer to an integer using CLNG, the result will be a number from 1 to 16777215. Zero is reserved as a special pointer called NULL, which is defined as not pointing to anything.

These bytes are grouped into larger chunks to represent the familiar data types used by BASIC. An INTEGER value occupies two adjacent bytes, LONG, pointer and SINGLE values each use four adjacent bytes, and DOUBLE values use eight bytes. The number of bytes used by a string varies; strings use one byte per character in the string, plus a single byte to mark the end of the string.

When you create a variable, the proper number of bytes are set aside for its use. Setting the variable changes the value of the bytes set aside for that variable value. Using the address operator, @, you can get a pointer to the variable. Using CLNG, you can convert this pointer to the integer value representing the address of the first byte of storage used for the value. You can use type casting to convert from an integer to a pointer.

While there is no memory location beyond 16777215 ($00FFFFFF) on an Apple IIGS, pointer values can technically go up to 4294967295, so long as they are reduced to a legitimate memory address before being used. In fact, most Apple IIGS computers don't have 16777216 ($01000000) bytes of memory, and even if they do, it's not all RAM. The memory space of the

87

Language Reference Manual

Apple IIGS is actually divided into three distinct kinds. Random Access Memory (RAM) is the kind you'll generally be using with pointers. RAM starts at byte 0 and extends sequentially. For an Apple IIGS with 4 megabytes of RAM, the valid addresses would be 0 to 4194303 ($003FFFFF). Read Only Memory (ROM) starts as byte 16777215 ($00FFFFFF) and extends down; how far depends on the model of your computer. Sprinkled here and there are some exceptions to this general scheme, mostly in the form of memory mapped I/O. Memory mapped I/O are bytes that appear to a program to be normal memory, but are in fact connected to hardware in a special way. A classic example is the Apple IIGS keyboard, which causes values for the characters typed to appear at memory location 49152 ($0000C000).

For more information about the memory organization of the Apple IIGS , see *Apple IIGS Firmware Reference* and *Apple IIGS Hardware Reference, Second Edition*. Both of these books are available as reprints from the Byte Works.

Chapter 9 – BASIC Programs

The Anatomy of a BASIC Program

BASIC programs are made up of a series of lines, each of which can have zero or more BASIC statements. Blank lines are allowed, and multiple statements can appear on the same line if the statements are separated by a colon character, with a few exceptions noted in the descriptions of individual statements.

The program begins execution with the first line, continuing sequentially unless the normal flow of execution is changed by one of the statements that is encountered.

Here is a simple BASIC program that prints a string to the screen.

```
PRINT "Hello, world."
```

Subroutines

BASIC programs can contain subroutines and functions, declared with the SUB and FUNCTION statement. Subroutines and functions are referred to as procedures in situations where either one is allowed. These procedures appear after the main program, one after the other. While it is possible to type lines between these subroutines, they are never executed.

The main program always appears first, and execution always starts there. The main program can call subroutines, which can in turn call other subroutines.

Line Numbers

Lines start with an optional line number. The line number is an integer in the range 1 to 65535 appearing at the beginning of the line.

There are two fundamentally different ways to use line numbers, and in some important ways they are incompatible.

Traditional BASIC programs require a line number on each program line. If you are importing an Applesoft BASIC program, or if you are using the line editor in the GSoft BASIC shell, this may still be a convenient way to organize your program. Using this model, a line number must appear at the start of every line. The line numbers must be unique, and they are forced to be sequential. If you load an ASCII program with line numbers that are out of order, they are sorted as they are loaded. If a duplicate line number appears, the newer line replaces the original line with the same number.

Like most BASICs implemented after the early 1980's, GSoft BASIC does not require line numbers. If any line does not use a line number, the second model kicks in. Line numbers are still

89

available, but they are usually not used unless you need a destination for a branching statement, such as GOTO or ONERR. Line numbers do not have to be sequential, and in fact, they don't even have to be unique. It's common to see the same line number used in various subroutines within a program. Since there is no requirement that the line numbers be unique, you can copy a subroutine from one program and paste it into another without worrying about conflicts between the line numbers.

Line numbers appearing in the same subroutine or in the same part of the program should be unique within that part of the program, although the interpreter does not enforce this restriction. For example, the subroutine

```
    SUB Duplicate
10  PRINT "Start"
    GOTO 10
10  PRINT "Done"
    END
```

will not generate an error, although it will perform an infinite loop. When duplicate line numbers appear in the same part of a program, the first is always found and subsequent numbers are invisible to the interpreter.

Multiple Statements on One Line

Normally, each BASIC statement appears on a separate line in the program. In most cases, it is technically possible to place more than one statement on a line if you separate the statements with the : character. For example, the line

```
IF A < B THEN C = A : A = B : B = C
```

uses this feature to execute three separate statements instead of one when A is less than B. In GSoft BASIC, it would be more common to use a block IF statement to do the same thing, like this:

```
IF A < B THEN
   C = A
   A = B
   B = C
END IF
```

This may look a bit peculiar to the Applesoft BASIC programmer, but most people find programs written this way to be easier to read after they get used to the format.

Chapter 10 – Declaring Variables and Types

What Is a Type?

A Short History of Types in BASIC

BASIC was originally intended as a simpler version of FORTRAN, and was targeted at scientists and engineers who needed to write short programs. The original implementations of BASIC supported a wide variety of mathematical operations, including built-in matrix mathematics operations. Other than simple numbers and strings, about the only data type was the array, which doubled as a matrix.

Matrix operations were dropped as BASIC made the move from the science and engineering community to personal computers in the late 70's and early 80's. By that time, computer scientists were embracing languages like ALGOL, which supported records and pointers. Microsoft eventually installed records in its BASIC interpreters, which pretty much made them a standard, but until the advent of GSoft BASIC, no commercially available BASICs we are aware of supported both pointers and records in the same fluid way that they are implemented in Pascal and C.

The Kinds of Types

Simple Types

At the simplest level, a type is a kind of value that can be stored in a variable. GSoft BASIC supports six simple types. The name of each type is itself a type, and can be used as a type in DIM and TYPE statements. The six simple types are:

BYTE	A single integer byte with the range 0 to 255.
INTEGER	A two byte signed integer with a range of -32768 to 32767.
LONG	A four byte signed integer with a range of -2147483648 to 2147483647.
SINGLE	A four byte single-precision floating-point number with exponents of approximately 1E-38 to 1E38.
DOUBLE	An eight byte double-precision floating-point number with exponents of approximately 1E-308 to 1E308.
STRING	Strings are sequences of up to 32767 characters. Each string requires one byte of storage per character, plus an overhead of five bytes. One of the five bytes of

overhead is used to mark the end of the string; it is a zero value that appears after the last character. The other four bytes are a pointer to the first character of the string; this is the value actually stored in the string variable.

Arrays

Arrays are sequences of the same type of variable. A particular variable is selected using a subscript, which is a numeric value that specifies which of the various variables should be used. The first value in each array has a subscript of 0; subsequent values are numbered sequentially.

BASIC supports multiply subscripted arrays, too. Multiple subscripts are separated by commas. Here is an example that shows the creation and initialization of a unit matrix with 5 columns and 5 rows.

```
DIM A(4, 4)
FOR I% = 0 TO 4
FOR J% = 0 TO 4
    A(I%, J%) = 0.0
  NEXT
  A(I%, I%) = 1.0
NEXT
```

BASIC followed the lead of FORTRAN when it picked the order that values are stored in memory, and unfortunately, some BASIC programs take advantage of this order. Naturally enough, in an array with a single subscript, the 0th element comes first, followed by the 1st element, the 2nd element, and so on. In an array with multiple subscripts, the leftmost index increases the fastest. For the 5 by 5 matrix shown in the code snippet, the memory order for the array elements is

A(0,0) A(1,0) A(2,0) A(3,0) A(4,0) A(0,1) A(1,1) A(2,1) …

In the vast majority of cases, the base type for an array is one of the number types, but GSoft BASIC also allows arrays of strings, arrays of records, and arrays of pointers.

GSoft BASIC imposes two practical limitations on arrays. The first is a limit on the range of a subscript; subscripts cannot be larger than 32767, and as was mentioned before, the smallest subscript is 0. There is also a limit on the total size of each array. The maximum size of all of the data in an array cannot exceed 65536 bytes. This is a limit on the size of any single array, not on the total size of all arrays. If there is enough memory in the computer, and you have used SETMEM to make the variable space large enough, you can easily create several arrays, each of which is 65536 bytes long.

You can calculate the number of bytes used by an array by multiplying the size of one element of the array in bytes by the number of elements in each subscript. For example, the array in the code snippet uses 100 bytes; 4 bytes for each SINGLE value multiplied by 5 rows multiplied by 5 columns. Arrays of strings use 4 bytes per string entry, so an array of strings cannot hold more than 16384 strings. The remaining memory used by the string, which includes

one byte per character plus an extra byte to mark the end of the string, comes from another location in memory, and doesn't count against the total size of the array.

There is no limit on the number of subscripts other than the obvious limit imposed by restricting arrays to 65536 bytes.

Records

Records are collections of variables that are generally not the same type. Each value in the record is called a field; fields can be simple types, arrays, pointers, or other records.

No single record can exceed 65536 bytes. As with arrays, this is a limit on the size of a single record, not the total memory used by all records. If there is enough free memory, and SETMEM has been used to reserve enough of it for use by variables, you can create several records whose total size is far larger than 65536 bytes.

You can calculate the number of bytes used by a record by adding the sizes of each field. One ADDRESS record like the one in the code snippet requires 20 bytes; four for each of the four strings and four more for the LONG zip code. The MAILLIST array uses 10,000 bytes, so it is still well within the 65536 byte limit for a single array.

<u>Snippet</u>

```
TYPE ADDRESS
  NAME$
  STREET$
  CITY$
  STATE$
  ZIP&
END TYPE
DIM MAILLIST AS ADDRESS(500)
MAILLIST(0).NAME$ = "Albert Einstein"
MAILLIST(0).STREET$ = "1 Light Way"
MAILLIST(0).CITY$ = "Forever"
MAILLIST(0).STATE$ = "Relative"
MAILLIST(0).ZIP& = 30000000
```

Pointers

A pointer is not a type by itself. A pointer always points to some specific type of value; the complete type is POINTER TO followed by whatever type it points to. For example, to specify a type that is a pointer to an integer value, you use the type POINTER TO INTEGER.

Each pointer requires four bytes of storage. Since one or more pointers can point to the same thing, and they can point either to variables in the traditional variable space or to areas outside of the memory normally used by GSoft BASIC, they don't generally require any more storage than the four bytes for the pointer itself—the memory for the value the pointer points to is already accounted for by the variable it points to, or was allocated from outside of GSoft BASIC's memory space using ALLOCATE.

A pointer can point to a simple type, a record, or another pointer. A pointer cannot point to an array, but it can point to an element of an array. In most practical cases, pointing to the first value in an array amounts to the same thing as pointing to the array itself.

The snippet shows a simplified version of a typical use for pointers. Several records are created by allocating the memory directly with ALLOCATE. These records are chained together by creating a pointer within each record that points to the next record in a sequence. In a way, this has the same effect as creating an array, but unlike an array, the number of things in a linked list like the one in the example is not fixed. The list can grow to fill all of available memory if need be, but if less memory is needed, it only occupies the actual amount of memory that is required to hold all of the entries in the list.

In the snippet, the values are printed and disposed of, freeing the memory for other uses. That's another important difference between linked lists and arrays: If the program is finished with the list, the memory can be reused without stopping the program or erasing all of the other variables, too.

<u>Snippet</u>

```
TYPE NUMBER
    AFTER AS POINTER TO NUMBER
    VALUE AS INTEGER
END TYPE
DIM NUMBERS AS POINTER TO NUMBER
DIM TEMP AS POINTER TO NUMBER
FOR I% = 1 TO 10
    ALLOCATE (TEMP)
    IF TEMP <> NIL THEN
        TEMP^.AFTER = NUMBERS
        TEMP^.VALUE = I%
        NUMBERS = TEMP
    END IF
NEXT
WHILE NUMBERS <> NIL
    TEMP = NUMBERS
    NUMBERS = TEMP^.AFTER
    PRINT TEMP^.VALUE
    DISPOSE (TEMP)
WEND
```

Named Types

The last kind of type is a named type. Named types are types you create using the TYPE statement. Each record is a named type, and you can create other named types by giving the name and the expanded form of the type, like this:

```
TYPE IPTR AS POINTER TO INTEGER
```

The name IPTR is now a type, just like SINGLE. It can be used to declare variables the same way SINGLE is used.

94

There are two typical reasons to create a name for a type this way. The first is essentially a typing shortcut and an aid to understanding the program. By using the type name, rather than the expanded form of the type, your program is shorter and sometimes easier to follow.

The second reason to create a name for a type is type casting. The name of any pointer type can be used as a function. The argument to the function is any pointer type, and the value returned by the function is a pointer to the same byte of memory, but with the named type.

See *Type Casting* in Chapter 11 for a more in depth description of type casting.

See the next section, *Type Compatibility*, for a more complete discussion of when and why type casting is necessary.

See the description of the TYPE statement, later in this chapter, for a detailed discussion of the mechanics of defining a type.

Type Compatibility

With the plethora of new types available in GSoft BASIC, it's important to understand when two types are compatible. If two types are compatible, their values can be used interchangeably: a value can be assigned to a variable or passed as a parameter if their types are compatible, and two values can be compared if their types are compatible.

There are actually two shades of type compatibility. Two values are type compatible if they are completely interchangeable. Two values are assignment compatible if one value can be assigned to a variable of the other type, or when the value can be passed as a parameter. Since BASIC automatically converts numbers of one type to another, assignment compatibility is not as restrictive as type compatibility. This distinction between assignment compatibility and type compatibility is only important for numbers and pointers to numbers.

Numeric Type Compatibility

All of the numeric types are assignment compatible with each other: you can freely mix numbers of different types. For two numbers to be type compatible, though, the numbers must be exactly the same kind.

There are four situations where the difference between type compatibility and assignment compatibility is important.

First, some of the automatic type conversions can lead to a loss of precision. If you assign 1.9 to an integer variable, the value that is stored is 1. The same is true when you use the SINGLE value 1.9 as an array subscript: the number is first converted to an integer value 1, and the integer result is used to determine the array element to select. This can lead to strange results if roundoff error causes a value to be slightly less than the expected integer. For example, try this:

```
FOR I = 0 TO 10
  A(I) = I
NEXT
I = 4 / 3
```

```
I = I * 3
PRINT A(I)
```

Obviously, the program should print 4. Actually, it prints 3. The reason is that 4 / 3 isn't stored as exactly one and one-third, it's stored as approximately 1.333333. When multiplied by 3, the result is approximately 3.999999, and when truncated, the index used to access the array is 3, not 4.

In general, your program will be both faster and less prone to bugs of this kind if you always use integer or long integer values when calculating an array subscript. If you must use floating-point subscript values, add 0.5 to the subscript to eliminate the possibility of this kind of roundoff error.

The second place the difference between type compatibility and assignment compatibility is important is when the numeric values involved cannot be converted. For example, assigning the SINGLE value 3E5 to an INTEGER variable causes a run-time error, since an INTEGER cannot hold values larger than 32767.

The third place this difference is important is when you are passing parameters to subroutines. Whenever you pass a named variable as a subroutine parameter, the variable is passed by reference. "Passed by reference" means the subroutine can alter the variable's value directly. For example,

```
I = 4
CALL CHANGE (I)
PRINT I
END

SUB CHANGE (X)
X = X * 2
END SUB
```

prints the value 8: the subroutine changed the value of the original variable that was passed as a parameter. This only happens when a variable is passed, though. If you pass the result of any expression, even an expression as simple as preceding the variable by a + operator or enclosing it in parentheses, the parameter is passed by value. When a parameter is passed by value, the subroutine cannot change the original variable. The subtle change of adding parentheses, like this:

```
I = 4
CALL CHANGE ((I))
PRINT I
END

SUB CHANGE (X)
X = X * 2
END SUB
```

causes the program to print 4, rather than 8.

The reason this change is important for type compatibility is that variables that are passed by reference must match the type of the parameter exactly—no conversion of any kind is done. If a subroutine expects an INTEGER parameter, you cannot pass a SINGLE variable by reference. You can pass a SINGLE constant, or you can turn the SINGLE value into an expression that is passed by value by enclosing the variable name in parentheses, but the variable name itself cannot be passed.

The last situation where the difference between type compatibility and assignment compatibility is important is for pointers to numbers. Two pointers to numeric types are compatible only if the underlying numeric type is exactly the same.

Strings

Strings are always compatible with each other. Strings are not compatible with any other type.

Strings can be converted to numbers, and numbers can be converted to strings, using the STR$ and VAL functions.

Records

Record values are compatible with each other if both record variables are defined from the same base type. Different names can be involved, so long as the underlying type is the same, but if the record types are different, the values are not compatible, even if the fields of each record are the same.

To explore how this works, we'll use these type and variable declarations.

```
TYPE POINT
    X
    Y
    Z
END TYPE
TYPE VECTOR
    X
    Y
    Z
END TYPE
TYPE XYZ AS POINT
DIM P AS POINT
DIM POINTS(5) AS POINT
DIM PP AS POINTER TO POINT
DIM V AS VECTOR
DIM ALPHA AS XYZ
```

With these types in place, the value P is compatible with POINTS(3) and PP^, since all three refer to the same base type, POINT. P is also type compatible with ALPHA; even though

97

Pointers

ALPHA is defined as a record whose type is XYZ. XYZ is itself defined as a POINT. P is not type compatible with V, though. Even though POINT and VECTOR have the same number of fields, and the fields are the same type—indeed, they even have the same names in this example—POINT and VECTOR are distinct record types.

Two pointers are type compatible if they point to values that are compatible. In the case of pointers to numbers, the underlying number must be exactly the same type; a pointer to an INTEGER is not type compatible with a pointer to a BYTE.

You can use type casting to convert pointers from one type to another, as well as to convert a number to a pointer. See *Type Casting* in Chapter 11 for details.

Pointers can be converted to numbers using the function CLNG.

Default Types

There are two ways to create a named variable in BASIC. The most direct way is with the DIM statement. The DIM statement is traditionally used to dimension arrays, but it can also be used to create a variable with any type you like. For example, the DIM statement

```
DIM I AS INTEGER
```

creates a new variable called I, and makes this variable an INTEGER.

The most common way to create a new variable, though, is to simply use it. BASIC guarantees that the value will be created and initialized to 0 or an empty string, as appropriate. This doesn't really matter in most cases, since a variable is almost always assigned an initial value the first time it is used.

There needs to be some way to assign a type to a new variable, though. BASIC uses a special set of characters that can appear at the end of any identifier. If the variable is declared by using it in an expression, this trailing character determines the variable's type.

For example, % is used for integers. The program

```
FOR I% = 1 TO 10
    PRINT I%
NEXT
```

creates the variable I% when the FOR loop starts. Since the last character in the variable's name is %, this variable is an INTEGER.

Here's a complete list of the type characters in GSoft BASIC.

Character	Type
~	BYTE
%	INTEGER
&	LONG
!	SINGLE
#	DOUBLE
$	STRING

Variables that don't have a type character are declared as SINGLE.

If the type character is used, it becomes a part of the variable name. Using the variable without the type character will create a completely different variable. This short program creates two distinct variables, as the PRINT statement proves when you run the program.

```
DIM I AS INTEGER
I% = 4
I = 5
PRINT I%, I
```

While a type character at the end of an identifier becomes a part of the variable name, type characters cannot be used anywhere else in the variable name. Only one is allowed, and that one type character, if it is used at all, must be the last character in the variable's name.

If a variable is created by the DIM statement, the type you give in the DIM statement overrides any type character, or the absence of a type character. In the example just shown, the variable I would have been a SINGLE variable if the DIM statement was not in the program, since variables without a type character default to SINGLE. In this case, though, the variable I is an INTEGER variable. It's just as possible to create a variable called I% that holds a SINGLE value using the same idea, but of course anyone who does this for anything but a prank deserves to have their fingers smacked with a ruler. Any BASIC programmer will assume that a variable with a trailing % character is an INTEGER; making it something else could cause confusion.

Arrays can also be defined by using the array, rather than with the DIM statement. The number of subscripts matches the number used in the expression where the array first appears. The maximum subscript is always 10. For example, encountering the statement

```
A(1, 1) = 11
```

without first encountering a DIM statement creates an array of SINGLE values. The array has two subscripts, and each can range from 0 to 10, so there are 121 SINGLE values in the array.

Arrays and non-arrays are distinct, so you can have an array and a non-array variable with the same name. This isn't generally a good idea, but it is possible. Arrays are not distinct, though. For example, you can't create two arrays with the same name, even if they have a different number of subscripts.

Declaring Types and Variables

```
DIM identifier [ subscript ] [ As type ] [ ',' 
identifier [ subscript ] [ As type ] ]*
```

The DIM statement is used to create a variable. Variables can be created by simply using them in a BASIC expression, but there are two situations where you need more control over how the variable is created than you get when you simply use a variable. In addition, many programmers find that dimensioning each and every variable is a good way to document what variables are used in a program and how they are used—a comment just before or after the DIM statement is very handy for remembering how a program's data structures are used.

Dimensioning Arrays

The first situation where you need control over how a variable is created is dimensioning an array. This is the most common use for DIM, and it's also the traditional use that gives the statement its name. To dimension an array, give the name of the array with the maximum value for each subscript. For example,

```
DIM V(5), A(5, 5)
```

dimensions two arrays. The first is an array with six SINGLE values, subscripted from 0 to 5. The second array has two subscripts, each ranging from 0 to 5. The full array contains 36 SINGLE values.

As with any variable, you can use type characters to specify the type of the elements in the array. For example,

```
DIM A#(7, 7)
```

creates an array of 64 DOUBLE values. See *Default Types*, earlier in this chapter, for a complete discussion of type characters. You can also specify a specific type using AS; this is covered in *Assigning a Type With AS*, right after this section.

In most situations it makes more sense to use an INTEGER value for array subscripts, but it is possible to use any numeric value. Values are always converted to INTEGER before being used as a subscript. For floating-point values, the value is truncated, so a subscript of 3.999 is treated as the INTEGER value 3. The potential problems of this sort of round-off error are the main reason floating-point values should not normally be used for array subscripts. Calculating the array subscript also takes much longer using floating-point arithmetic; in some programs the speed difference can be dramatic.

In most cases DIM statements appear right at the start of a program or subroutine, and the subscripts are constant values. These normal use rules come about because it makes sense to organize programs with the DIM statements at the beginning, and in most cases the size of an

array is fixed. There are situations where it makes sense to use an expression for the size of an array, though, and occasionally it even makes sense to imbed the DIM statement in the program.

For example, here's an array that uses the value stored in I% to determine the size of the array. This value might be read from a disk file just before reading the numbers for the array, or it might be entered by the person using the program.

```
DIM SPEED(I%)
```

Here's an example that uses one of two sizes for several arrays, depending on how much memory is available.

```
IF FRE(0) > 64*1024 THEN
   DIM NAME$(BIG%)
   DIM ADDRESS$(BIG%)
   DIM CITY$(BIG%)
   DIM STATE$(BIG%)
   DIM ZIP&(BIG%)
ELSE
   DIM NAME$(SMALL%)
   DIM ADDRESS$(SMALL%)
   DIM CITY$(SMALL%)
   DIM STATE$(SMALL%)
   DIM ZIP&(SMALL%)
END IF
```

The lowest allowed subscript in any array is 0, and the largest allowed subscript is 32767. The maximum size for a single array is 65536 bytes. See *Arrays*, earlier in this chapter, for a more complete discussion of these limits.

Assigning a Type With AS

The AS clause is used to assign a type to a variable. AS is followed by the name of a type. This can be something as simple as the name of a default type or as complex as the name of a record.

For example,

```
DIM I AS INTEGER, J AS INTEGER, K AS INTEGER
```

creates three INTEGER variables that don't need % as the last character of the variable name. You can use this idea with any of the built-in types. The built-in types are BYTE, INTEGER, LONG, SINGLE, DOUBLE and STRING. See *Default Types*, earlier in this chapter, for a complete discussion of these types.

While it rarely if ever makes sense to do so, you can use the AS clause to override the type of a variable. For example,

```
DIM COST$ AS SINGLE
```

creates a variable named COST$; this would normally be a string, but because of the AS clause, the variable is SINGLE.

A less mundane use of the AS clause is to create a record variable. Record types are created with the TYPE statement. For example,

```
TYPE CUBE
    TOP
    BOTTOM
    LEFT
    RIGHT
    FRONT
    BACK
END TYPE
```

creates a record type whose type name is CUBE. Each CUBE record contains six SINGLE fields. The DIM statement

```
DIM C AS CUBE, CUBES(10) AS CUBE
```

creates two variables. C is a CUBE record variable, while CUBES is an array of 11 CUBE records.

Type names aren't always so simple. You can use POINTER TO before any type name to create a pointer to a value. For example,

```
DIM CP AS POINTER TO CUBE
```

creates a pointer variable named CP. This pointer points to a CUBE record. You can allocate a cube record using ALLOCATE, or point CP to an existing CUBE record with the address operator, like this:

```
CP = @CUBES(4)
```

Using Default Types With DIM

The last use of the DIM statement is to create a variable using its normal default type. This is entirely optional, since BASIC will create the variable for you and initialize it to zero the first time it is used in the subroutine or program, but this is a convenient way to create a dictionary of your variables and describe how they are used in the program.

The statements

```
DIM I%: REM Loop/index variable
DIM INTEREST: REM Annual interest rate
```

create two variables, the INTEGER variable I% and the SINGLE variable INTEREST, and give a clue as to how these variables are used in the program.

```
TYPE  identifier
[  (  field-name  [  As  type  ]  )
|  (  CASE  [  expression  ]  )  ]+
END  TYPE
```

The TYPE statement has two major forms. The form described in this section is used to create record types. The other form, described right after this one, is used to assign a name to a simple type or pointer.

Declaring Record Types

A record contains one or more values, just like an array. Unlike an array, the values in a record do not have to be the same type. Each of these values is called a field, and each has its own name and its own type, just like a variable that is not a field in a record.

The field declarations appear between the TYPE and END TYPE statements. Each field declaration looks exactly like a variable defined in a DIM statement, but without the DIM. Just as with a DIM statement, you can declare fields using an AS clause, using default types, or using array subscripts.

Here's a classic example of a record. Each record contains an address.

```
TYPE  ADDRESS
  NAME$
  STREET$
  CITY$ : STATE$ : ZIP&
END TYPE
```

This particular record has five fields, four strings and a LONG zip code. It's customary to put each field on a separate line, but this example shows how to put multiple fields on one line.

How Records Are Stored In Memory

Knowing how records are stored in memory is important for some kinds of programming, especially toolbox programming. In toolbox programming and other situations where a GSoft BASIC program is communicating with another program or device, it's often necessary to lay out a record that will occupy bytes in memory in a very specific way. Knowing how records are stored is also key to understanding how values can be overlaid in memory using variant records.

To explore how records are stored in memory, we'll use these contrived record types, which show an example of each kind of variable that can appear in a record.

Language Reference Manual

```
TYPE POINT
  H AS INTEGER
  V AS INTEGER
END TYPE
TYPE ALLKINDS
  B AS BYTE
  I AS INTEGER
  L AS LONG
  S AS SINGLE
  D AS DOUBLE
  STR AS STRING
  A(3) AS INTEGER
  P AS POINT
  PTR AS POINTER TO INTEGER
END TYPE

DIM R AS ALLKINDS
```

Each field in a record occupies one or more bytes in memory. The amount of memory and exact storage method is discussed more completely in the various sections that describe the data types. For our purpose in this section, it is enough to know that a BYTE value requires one byte, an INTEGER uses two bytes, LONG and SINGLE values and pointers to any value require four bytes, and DOUBLE values require eight bytes. Strings are a special case. The string value is stored in a special memory pool; the string variable, which is what is stored in the record, requires four bytes of memory, so for this discussion we treat all strings as if they require four bytes of memory.

Fields within a record are stored sequentially. The record POINT consists of two INTEGER values, so each record variable of type POINT needs four bytes of memory. The field H will occupy the first two bytes of the record, and the field V will occupy the last two bytes in the record variable. The usual way to describe the position of the fields is to say that H is zero bytes past the start of the record, or that it has a displacement of zero, while V has a displacement of two bytes.

The variable R is a record of type ALLKINDS. It uses 39 bytes of memory, laid out like this:

Displacement	Size	Field Name
0	1	B
1	2	I
3	4	L
7	4	S
11	8	D
19	4	STR
23	8	A
31	4	P
35	4	PTR

Variant Records

A variant record is used to treat memory in different ways, setting up variables that occupy the same memory. A simple example is the toolbox rectangle record, RECT. Rectangles require four integer values, one each for the top, bottom, left and right edges of the rectangle. Due to some schizophrenic naming, there are three different naming schemes for rectangle records on the Apple IIGS—the names used on the Macintosh, a method that treats a rectangle as two corner points, and the one most people use, which names the edges H1, H2, V1 and V2. Here's how this record is declared:

```
TYPE RECT
   CASE NORMAL
      V1 AS INTEGER
      H1 AS INTEGER
      V2 AS INTEGER
      H2 AS INTEGER
   CASE MAC
      TOP AS INTEGER
      LEFT AS INTEGER
      BOTTOM AS INTEGER
      RIGHT AS INTEGER
   CASE POINTS
      TOPLEFT AS POINT
      BOTRIGHT AS POINT
END TYPE
```

The CASE clause causes the displacement counter to start over. The expression that appears after CASE is for your benefit, not BASIC's. It serves as a comment, telling what the variant part is used for. You can leave it off, or make it the same as a variable you use to keep track of the variant parts within a record.

In this simple example the result is a record with three different names for each of the integers that are stored in the record. For example,

```
DIM R AS RECT
R.V1 = 4
PRINT R.TOP
```

prints 4, since R.V1 and R.TOP are different names for the same value. Here's a table that shows the names for each of the offsets in a RECT record.

Displacement	Size	Field Names		
0	2	V1	TOP	TOPLEFT.V
2	2	H1	LEFT	TOPLEFT.H
4	2	V2	BOTTOM	BOTRIGHT.V
6	2	H2	RIGHT	BOTRIGHT.H

The CASE clause is a marker that divides the various parts of the record from each other. Each of the parts is called a variant part. All variables from the first CASE to the next, or until the END TYPE for the last CASE clause, form a variant part, which is overlaid on all of the other variant parts. They don't have to be the same length; the total length of the record is determined by the longest CASE clause. For example, in the record

```
TYPE SIZE
    CASE ONE
        I AS INTEGER
    CASE TWO
        S AS SINGLE
END TYPE
```

the record will be four bytes long. The first variant part is a single two byte INTEGER, while the second is a four byte SINGLE. Each variable will need four bytes so it can hold the longest possible variant part.

Another use of variant records is to treat the same value two different ways. For example, let's assume you need to split a four byte LONG value into two INTEGER values. That's a real problem in a surprising number of situations. Here's a variant record that can be used to do it.

```
TYPE CONVERT
    CASE LONGWORD
        L&
    CASE WORD
        I1%
        I2%
END TYPE
DIM C AS CONVERT
;
L& = $00020001
PRINT C.I1%, C.I2%
```

The first integer in the record overlays the first two bytes of the long integer value, which is the least significant integer on an Apple IIGS; the second integer overlays the most significant integer. The program demonstrates this concept by stuffing a hexadecimal value into the long integer value. The hexadecimal constant makes it easy to see that the least significant two bytes of the long value should be 1, while the most significant two bytes should be 2. The program shows this is true by printing the values.

You can also place variables before the first CASE clause. Variables that appear before the first CASE are not overlaid at all; they appear in all versions of the record. A great example is a control record, used to describe the characteristics of a control in a window. All controls have a field for the rectangle that surrounds the control, so this field needs to be available for all records. Only a scroll bar has a thumb rectangle, though, and it doesn't need a key equivalent like some of the other controls. Rather than waste space by reserving memory for a key equivalent in a scroll

bar or for a thumb rectangle in a button. Apple's engineers used a variant record to lay out controls. Leaving out the dozens of other fields, the three we discussed look like this in the type declaration for a control record.

```
TYPE CTLREC
    CTLRECT AS RECT
    CASE BTNORCHECK
        KEYEQUIV AS KEYEQUIVREC
    CASE SCROLL
        THUMBRECT AS RECT
END TYPE
```

In this record, CTLRECT has a displacement of zero. Rectangles use 8 bytes of memory, so both KEYEQUIV and THUMBRECT have offsets of 8.

Using the Record Type In The Record (Linked Lists)

A type must be declared before it can be used, but the type name is already declared as the fields are being created. This means that the name of the record type can be used when creating a field within the same record—but only when you are declaring a pointer to the type. We'll discuss the reason for this restriction after looking at a concrete example of a record type name used inside the same record.

The short program below shows a linked list, which is the classic reason to use a record's name when declaring a field in the record. A linked list is a record with at least one field that is a pointer to another record of the same kind. Linked lists are frequently used in database applications, since they can adapt to a widely varying amount of memory. Linked lists are also used extensively in the Apple IIGS toolbox. For example, window records have a pointer to another window record. When you open a new window, the Window Manager creates a new window record. Each window has a pointer to the next window. The Window Manager has a single pointer to the first open window, and scans the list to deal with all of the windows that are open. This is a powerful combination: The Window Manager can handle any number of windows, so long as you have enough memory, but it never uses more memory than it needs for the actual number of windows that are open. An array doesn't have either of these advantages.

In the example below, NUMBER is a record type that holds an INTEGER value. AFTER is a field within NUMBER that points to the next record in the linked list. The program builds the linked list by allocating memory for each record using ALLOCATE, then filling in the values. NUMBERS is a pointer to the first record in the linked list; the program can find all of the other records by tracing through the pointers to subsequent records.

```
TYPE NUMBER
    AFTER AS POINTER TO NUMBER
    VALUE AS INTEGER
END TYPE
DIM NUMBERS AS POINTER TO NUMBER
DIM TEMP AS POINTER TO NUMBER
;
```

```
FOR I% = 1 TO 10
  ALLOCATE (TEMP)
  IF TEMP <> NIL THEN
    TEMP^.AFTER = NUMBERS
    TEMP^.VALUE = I%
    NUMBERS = TEMP
  END IF
NEXT
WHILE NUMBERS <> NIL
  TEMP = NUMBERS
  NUMBERS = TEMP^.AFTER
  PRINT TEMP^.VALUE
  DISPOSE (TEMP)
WEND
```

If you've never dealt with pointers and linked lists, this will seem rather strange, but if you plan to do anything resembling database programming or toolbox programming, you'll eventually get very comfortable with linked lists. For an introduction to linked lists, see *Learn to Program in GSoft BASIC*, a companion course that teaches general programming techniques using GSoft BASIC.

Thinking about how records are stored in memory should make it easy to see why a record's name can only be used as a pointer inside that same record. Something like this:

```
TYPE NUMBER
  N AS NUMBER
  I AS INTEGER
END TYPE
```

just wouldn't make sense. For one thing, BASIC has no idea how many bytes to reserve for the record variable N, since the record hasn't been completed yet. For another, it's not really clear even after you look at the entire record just how it should look in memory. After all, the field N has a NUMBER record inside it, and that NUMBER record has another field N, which has another NUMBER record, and so on.

A pointer to the record doesn't share those problems. A pointer always uses four bytes of memory, so BASIC knows exactly how many bytes to set aside for the field.

TYPE identifier AS type

The TYPE statement has two major forms. The form described in this section is used to assign a name to a simple type or pointer. The other form, described right before this one, is used to create record types.

This form of the TYPE statement creates a new named type. There are two common reasons to do this: organization and naming pointer types.

It's common to declare a record and use pointers to the record. You'll see this throughout the Apple IIGS toolbox, where windows, controls, menus, menu bars, points and rectangles are

108

frequently dealt with via a pointer to the value. When a pointer to a particular kind of value is used in many places, it makes sense to create a type for the pointer itself. For example, the toolbox defines a rectangle pointer like this:

```
TYPE RECTPTR AS POINTER TO RECT
```

With this type in place, you can create a pointer to a rectangle with the DIM statement using this type name.

```
DIM RP AS RECTPTR
```

Strictly speaking, this doesn't create a new type, it just attaches a name to a frequently used type. For example, if we create a completely separate pointer to a rectangle, like

```
TYPE RPTR AS POINTER TO RECT
```

and another variable using this second type,

```
DIM RP2 AS RPTR
```

or even a variable that doesn't use either type, as in

```
DIM RP3 AS POINTER TO RECT
```

all of the variables are compatible. You can assign one to the other, or pass any of them as a parameter to a subroutine that expects a pointer to a rectangle.

The second common reason to create a named type is for organization. If you're creating a program that deals with the physical quantities speed and mass, you could define all of your variables as SINGLE.

```
DIM SPEED1, SPEED2, MASS1, MASS2
```

That would work well, and it's what you will find in most programs that deal with speed and mass. An alternative is to create new types for speed and mass, like this:

```
TYPE SPEEDTYPE AS SINGLE
TYPE MASSTYPE AS SINGLE
DIM SPEED1 AS SPEEDTYPE, SPEED2 AS SPEEDTYPE
DIM MASS1 AS MASSTYPE, MASS2 AS MASSTYPE
```

At first it probably seems brain-dead to add all that typing to the program. Let's assume, though, that the entire program has several thousand lines of code, and declares dozens of values

that are speeds or masses in various subroutines throughout the program. If you suddenly discover that you need DOUBLE values for masses because your program needs more than seven significant digits, you're stuck searching through the entire program line by line for all the locations that need to be changed. If you organized the program with types, though, you only need to change one line:

```
TYPE MASSTYPE AS DOUBLE
```

The method you choose often depends on the size of programs you write and how many times the program is likely to be changed, but organizing a program with types is a powerful way to make it easy to change.

Chapter 11 – Expressions and Assignments

Expressions

BASIC works on values, manipulating, storing, and retrieving information stored in the bytes of your computer's RAM and on various external storage devices. Almost all of the statements in BASIC that accept a value as a parameter allow you to use an expression. An expression can be something as simple as the number 1, or as complicated as a complex mathematical formula taking information from a disk, pointers to records, and arrays. This chapter describes how expressions work.

Kinds of Expressions

There are four different kinds of expressions in BASIC. They can be intermixed, and putting them together follows the same underlying rules, so all four are described here as a group. The difference between them lies simply in the result they produce.

Mathematical Expressions

Most BASIC commands and statements work on numbers. A mathematical expression is any expression that results in a number, whether that number is an INTEGER, LONG, SINGLE or DOUBLE value.

When you see the term **expression** in the model for a BASIC statement, it generally means that the expression is a mathematical expression. In a few cases, like the LET statement, it can also mean that the expression can be a mathematical expression, pointer expression or a string expression. Those cases are usually obvious. After all, you need to be able to assign values to string variables, just as you assign values to numerical variables, so LET must support string expressions as well as mathematical expressions. Just as obviously, you can't take the square root of a string, so the expression accepted by SQR must be a mathematical expression. In any case, the description of the command will point out what kinds of expressions are valid by telling you what the operation is and by explicitly stating when strings or pointers are allowed.

Logical Expressions

Some commands test to see if a condition is true or false. The IF statement is the classic example. The condition is called a logical expression.

At one level, logical expressions are exactly the same as mathematical expressions. Both are calculated the same way. Both result in a number. The difference is not how they are created, but how they are used.

While the result of a logical expression is a number, what is needed is a logical value—either true or false. To make this jump, BASIC treats any number whose value is zero as false, and any other value as true.

Operations that return a logical value, like the the OR operation, always return 0 for false and 1 for true. While it is not technically required, most implementations of BASIC seem to follow this rule. In GSoft BASIC, the number is always returned as an INTEGER. In most situations this doesn't make much difference, but operations on integers are a bit faster than operations on other kinds of numbers.

There is one place where conversion of numbers can lead to some unexpected results. Be careful of floating-point values used as logical values. The value 0.01, for example, is not a zero, so by itself it has a logical value of true. If you save the value in an integer, though, it converts to zero, changing the result to false. This causes the following program to print TRUE the first time, and FALSE the second, even though strict logic requires both values to be the same.

```
L = 0.1
L% = L
PRINT_LOGICAL(L OR 0)
PRINT_LOGICAL(L% OR 0)
END

SUB PRINT_LOGICAL (L)
IF L THEN
    PRINT "TRUE"
ELSE
    PRINT "FALSE"
END IF
END SUB
```

Pointer Expressions

GSoft BASIC is rare. Unlike older implementations of BASIC, GSoft BASIC handles pointers smoothly, just like C and Pascal.

Pointers are values that point to the location where another value is stored. Pointer expressions return a pointer to some other value.

Variables have a type. For example, I% is a variable whose type is INTEGER. Pointers have a type, too, but the type can vary. A pointer points to something, and the type of the pointer is POINTER TO whatever the other type happens to be. Two pointers are interchangeable if they point to the same kind of value, but cannot be assigned or compared if they point to different kinds of values. You can, however, change the type of a pointer using type casting, discussed later in this chapter.

String Expressions

String expressions return strings, as the name implies. Generally the string is the result of a function, like LEFT$, but the math operation + also works on strings, combining them to form a longer string.

112

Evaluating Expressions

To help us analyze how expressions are constructed, we'll divide the discussion into two categories. This section will discuss the various operations that accept two numbers or strings and return a single number or string. These are technically known as binary operators. In the next section, we'll discuss terms, which are numbers, variables, and operations that work on a single value and return a single result. As we'll see, you can always think of a term as a single value, and these values can be combined with the operations in this section two at a time.

Operator Precedence

When you write a mathematical formula, you expect that some operations are performed before others. For example, if you see

1 + 2 x 3

you expect the result to be 7, because multiplication is always done before addition. The technical term for this choice of order is operator precedence. Operations with a higher precedence are always done before those with a lower precedence.

The following table shows operator precedence for all of the operations in BASIC, both the binary operations (those that take two arguments, like addition and division) described in this section and the unary operations (those that take a single argument, like NOT) described with terms. The operations at the top of the table have a higher precedence, and are always performed first.

When operations have the same precedence, the operation is always done in left-to-right order. Normally this doesn't matter, but in some numerically sensitive equations it can make a difference.

Operations By Precedence

.^{1}	^^{2}	.^{3}	()^4
@			
+^{5}	-^{5}	NOT	
*	/		
^^{6}			
+	-		
=	<	>	<= >= <>
AND			
OR			

Notes for the operator precedence table:

1   Used to access fields in a record.

2   Used to dereference a pointer, returning the value pointed to rather than the value of the pointer.

3   Used to access the field of a record that is pointed to by a pointer.

4   In this table, the parentheses indicate accessing an array.

5   These are the unary versions of the operations. For example, -X uses the unary subtraction operation.

6   This is the exponentiation operation. 2^3 is 2*2*2, or 8.

You can use parentheses to change the order of operations. For example,

```
PRINT 1 + 2 * 3
```

prints 7, but

```
PRINT (1 + 2) * 3
```

prints 9.

Binary Conversions

GSoft BASIC supports five different kinds of numbers: BYTE, INTEGER, LONG, SINGLE and DOUBLE. Binary conversions are the rules used to perform operations on different kinds of numbers. These rules tell you both what the result will be, and in some cases how the calculation is performed.

Most of the time these differences are not important. BASIC converts numbers back and forth as needed without causing much trouble. There are a few situations where the difference is

important, though. Understanding them could save you hours of staring at a program that *ought* to work, but just doesn't seem to give you the answer you expect.

Unless otherwise noted, when two numbers of the same kind are used with a binary operation, the result is a number of the same kind, too. For example, if you add two integers, as in I% + J%, the result is also an integer.

The one universal exception is BYTE. GSoft BASIC treats BYTE variables as a special case of INTEGER that uses less storage. Operations involving BYTE values always work as if the values were INTEGER.

When you mix two different kinds of number in the same operation, the numbers are *first* converted to the same number type, then the operation is performed. For example, R * I%, where R is a SINGLE number, is carried out by converting I% to a SINGLE value, then doing the multiplication. The result is SINGLE.

The following table shows how binary conversions are carried out. Since BYTE numbers are always converted to INTEGER, they are not shown in the table. The rightmost column shows both the number format the values are converted to before the calculation is performed and the kind of number the operation returns.

If one value is…	and the other is…	the calculation is…
DOUBLE	SINGLE	DOUBLE
DOUBLE	LONG	DOUBLE
DOUBLE	INTEGER	DOUBLE
SINGLE	LONG	SINGLE
SINGLE	INTEGER	SINGLE
LONG	INTEGER	LONG

Unary Conversions

There are many places in GSoft BASIC where a number is converted from one type to another. For example, this happens during binary conversions, described above. Unary conversions are also made when you assign a number that is one type to a variable of another type using a LET statement, or when you pass a value as a parameter. You can also force a unary conversion using the functions CINT, CLNG, CSNG and CDBL. In all of these cases, the conversion from one number type to another is done in exactly the same way. This section describes how these conversions are performed.

Converting DOUBLE to SINGLE

Converting a DOUBLE value to a SINGLE value results in a loss of precision in the number, which drops from about 16 decimal digits to about 7 decimal digits. In some cases, this loss is completely transparent. Both formats can represent the number 4.5 with complete accuracy, so converting the DOUBLE value 4.5 to the SINGLE value 4.5 doesn't loose any accuracy at all. On the other hand, converting the DOUBLE value 4.50000001 to SINGLE will result in the number 4.5.

DOUBLE values also support a larger exponent range than SINGLE values. The exponent range for SINGLE is about 1E-38 to 1E38. If the DOUBLE number is too close to zero to represent with the smallest available SINGLE exponent, the result is 0.0. If the DOUBLE value is too large to represent with the largest available SINGLE exponent, the result is infinity or negative infinity, which prints as inf and -inf, respectively.

Converting DOUBLE to LONG

The value is first converted to an integer by rounding down to the largest integer that is less than or equal to the original value. Some typical values are:

DOUBLE	LONG
-100.6	-101
-99.2	-100
-0.1	-1
0.1	0
3.3	3
3.99	3

The maximum range for LONG values is -2147483648 to 2147483647. After truncating, if the double value is outside this range, the program stops with a run time error.

Converting DOUBLE to INTEGER

The rules for converting DOUBLE to INTEGER are essentially the same as for converting DOUBLE to LONG. The only difference is that INTEGER values have a smaller range than LONG values, so overflows that result in a run time error can occur with numbers that are valid for LONG. The valid range for INTEGER values is -32768 to 32767; if the truncated DOUBLE value is outside this range, a run time error stops the program.

Converting SINGLE to DOUBLE

Converting SINGLE to DOUBLE always works, and there is no loss of precision.

Converting SINGLE to LONG

Converting a SINGLE value to a LONG value follows the same rules as converting a DOUBLE value to a LONG value.

Converting SINGLE to INTEGER

Converting a SINGLE value to an INTEGER value follows the same rules as converting a DOUBLE value to an INTEGER value.

Converting LONG to DOUBLE

All LONG values can be represented with no loss of precision by a DOUBLE value. The conversion always works, with no possibility of an error or loss of precision.

Converting LONG to SINGLE

Converting a LONG value to a SINGLE value always works, with no possibility of an error, but there can be a loss of precision. The mantissa of a SINGLE value uses 24 bits, which gives about 7 significant decimal digits. LONG values larger than 16777216 or smaller than -16777216 cannot be stored without loss of precision in a SINGLE value. The least significant bits are lost.

If you decide to verify this range, be sure to convert the SINGLE value back to LONG before printing it. Only the first seven significant digits of SINGLE values are normally printed, and you need to see eight digits to verify there was no loss of precision.

Converting LONG to INTEGER

Converting a LONG value to an INTEGER value works for any value in the range -32768 to 32767. A LONG value outside this range triggers a run time error.

Converting INTEGER to DOUBLE

All INTEGER values can be represented with no loss of precision by a DOUBLE value. The conversion always works, with no possible error or loss of precision.

Converting INTEGER to SINGLE

All INTEGER values can be represented with no loss of precision by a SINGLE value. The conversion always works, with no possible error or loss of precision.

Converting INTEGER to LONG

All INTEGER values can be represented with no loss of precision by a LONG value. The conversion always works, with no possible error or loss of precision.

Converting BYTE to Any Other Type

BYTE values are always converted to INTEGER values before any operation is performed. The result is always an integer in the range 0 to 255.

Converting Any Other Type to BYTE

Conversion of any value to a BYTE always starts by converting the value to an INTEGER. If the original value is outside the range -32768 to 32767, the conversion triggers a run time error.

117

Once the value has been reduced to an INTEGER, the least significant 8 bits of the two's complement integer value are used. If the value is in the range 0 to 255, the result is exact. If the value is outside that range, the result seems strange unless you are familiar with the way integers are stored. If you would like to explore how integers are stored, refer to any assembly language programming book, or any general computer science text that discusses two's complement notation.

For positive numbers the value that results is the same as you would get from the expression

```
I% - 256 * INT (I% / 256)
```

For negative numbers the value is the same as the result of this expression:

```
65536 + I% - 256 * INT ((65536 + I%) / 256)
```

Addition

The symbol for addition is +.

Addition works on both numbers and strings, and can be used in one way with pointers. String and pointer addition are discussed later in this section.

When you add two numbers, the addition operation returns the sum of the two numbers. For example, 1 + 1 returns 2.

If you are adding integers, and the result is larger than 32767 or smaller than -32768, the result cannot be an integer value. GSoft BASIC quietly detects the overflow and converts the result to a SINGLE number. For example, 30 + 30 returns the integer 60, but 30000 + 30000 returns the floating-point number 60000.0. The same thing happens with LONG values, although the range is somewhat larger. LONG values can range from -2147483648 to 2147483647. If the result of an addition of long integers falls outside this range, the values are quietly converted to DOUBLE.

SINGLE and DOUBLE values can overflow, too. If the result of an addition is too close to zero to represent, the value returned will be 0.0. If the value is too large or too small to represent, the result will be infinity or negative infinity. Infinity prints as inf. All of the various math operations in GSoft BASIC know how to handle infinity in a reasonable way. Adding infinity to any other value except negative infinity gives infinity, and adding negative infinity to any value except positive infinity gives negative infinity. Adding infinity to negative infinity results in a value called "not a number," which prints as NaN. This is handled reasonably, too. NaN added to any other value still returns NaN.

Addition also works on strings. When you add two strings, the second is tacked onto the end of the first. For example,

```
PRINT "Hello, " + "world."
```

prints the string "Hello, world."

Addition works on pointers, but in a limited way. You can add an integer value to a pointer, but you cannot add two pointers, nor can you add a pointer to an integer. Floating-point values can be used instead of integers; they will be converted to an integer by truncating before the addition takes place.

Adding an integer to a pointer causes the pointer to point a specific number of items further in memory. For example, let's assume we are adding 1 to the integer pointer IP, and that IP originally points to the memory location 100000. Adding 1 to IP causes IP to point to the integer after the original integer. Since each integer requires two bytes of memory, IP + 1 will be an integer pointer that points to memory location 100002. Adding 4 to a SINGLE pointer that originally points to 100000 gives a SINGLE pointer that points to memory location 100016, four SINGLE values later in memory.

Adding a negative value to a pointer is supported, too. It is treated like a pointer subtraction. See *Subtraction*, below, for a complete discussion.

A classic example uses pointers to access the values in an array. This short program fills an array using pointer addition, then prints the values using standard array notation.

```
DIM IP AS POINTER TO INTEGER
DIM A(10) AS INTEGER
IP = @A(0)
FOR I% = 0 TO 10
    IP^ = I%
    IP = IP + 1
NEXT
FOR I% = 0 TO 10
    PRINT A(I%)
NEXT
```

Subtraction

The symbol for subtraction is -.

Subtraction works with numbers, and in a limited sense, with pointers. Pointers are discussed at the end of this section.

When you subtract two numbers, the result is the number on the left minus the number on the right. For example, 4 - 10 returns -6.

Just as with addition, overflows do not cause errors. If subtracting one INTEGER from another results in a value that cannot be an INTEGER, the original numbers are converted to SINGLE and the operation is performed again. If two LONG values result in an overflow, the operation is performed again after the two numbers are converted to DOUBLE.

Just as with addition, subtraction of SINGLE or DOUBLE values that result in a number too close to zero to represent returns 0.0, while results too large or too small to represent return infinity or negative infinity.

The table below shows how infinity and NaN behave for subtraction. NaN stands for "not a number," and indicates that the result of an operation is not a valid real number. "Any value" refers to any number, including infinity or NaN, that is not listed explicitly in the table.

This…	minus this…	gives this…
NaN	any value	NaN
any value	NaN	NaN
inf	inf	NaN
-inf	inf	-inf
inf	-inf	inf
-inf	-inf	NaN
inf	any value	inf
-inf	any value	-inf
any value	inf	-inf
any value	-inf	inf

Subtraction works on pointers, but in a limited way. You can subtract an integer value from a pointer, but you cannot subtract two pointers, nor can you subtract a pointer from an integer. Floating-point values can be used instead of integers; they will be converted to an integer by truncating before the subtraction takes place.

Subtracting an integer from a pointer causes the pointer to point to a specific number of items earlier in memory. For example, let's assume we are subtracting 1 from the integer pointer IP, and that IP originally points to the memory location 100000. Subtracting 1 from IP causes IP to point to the integer before the original integer. Since each integer requires two bytes of memory, IP - 1 will be an integer pointer that points to memory location 99998. Subtracting 4 from a SINGLE pointer that originally points to 100000 gives a SINGLE pointer that points to memory location 99984, four SINGLE values earlier in memory.

Subtracting a negative value from a pointer is supported, too. It is treated like a pointer addition. See *Addition*, above, for a complete discussion.

Multiplication

The symbol for multiplication is *.

Multiplying two numbers returns their product. For example, 4 * 5 returns 20.

If the product of two INTEGER numbers is outside the range -32768 to 32767, the values are converted to SINGLE before the multiplication is performed, and the result is SINGLE. If the product of two LONG numbers is outside the range -2147483648 to 2147483647, the values are converted to DOUBLE before the multiplication is performed, and the result is DOUBLE.

If the product of two SINGLE or two DOUBLE values is too close to zero to represent, the result is 0.0. If the values are too large or too small to represent, the result is infinity or negative infinity. These print as inf and -inf.

The table below shows how infinity and NaN behave for multiplication. NaN stands for "not a number," and indicates that the result of an operation is not a valid real number. "Any value" refers to any number, including infinity or NaN, that is not listed explicitly in the table.

This…	times this…	gives this…
NaN	any value	NaN
any value	NaN	NaN
inf	any positive value	inf
inf	any negative value	-inf
inf	0	NaN
any positive value	inf	inf
any negative value	inf	-inf
0	inf	NaN
-inf	any positive value	-inf
-inf	any negative value	inf
-inf	0	NaN
any positive value	-inf	-inf
any negative value	-inf	inf
0	-inf	NaN

Division

The symbol for division is /.

Division divides the number to the left of the operator by the number to the right. For example, 4.8 / 1.5 returns 3.2.

Division always returns a SINGLE or DOUBLE value. After binary conversions, if the operands are INTEGER or LONG, the values are converted to SINGLE and the result is SINGLE.

If the result is too close to zero to represent, the result is 0.0. If the values are too large or too small to represent, the result is infinity or negative infinity. These print as inf and -inf.

The table below shows how infinity and NaN behave for division. NaN stands for "not a number," and indicates that the result of an operation is not a valid real number. "Any value" refers to any number, including infinity or NaN, that is not listed explicitly in the table.

This…	divided by this…	gives this…
NaN	any value	NaN
any value	NaN	NaN
any positive value	0	inf
any negative value	0	-inf
inf	any positive value	inf
inf	any negative value	-inf
-inf	any positive value	-inf
-inf	any negative value	inf
any value	inf	0.0
any value	-inf	0.0
inf (+ or -)	inf (+ or -)	NaN

Exponentiation

The symbol for exponentiation is the carrot character, ^.

Exponentiation raises the number to the left of the operator to the power of the number to the right. For example, 3 ^ 4 is 3 * 3 * 3 * 3, or 81.

Exponentiation always returns a SINGLE or DOUBLE value. After binary conversions, if the operands are INTEGER or LONG, the values are converted to SINGLE and the result is SINGLE.

If the result is too close to zero to represent, the result is 0.0. If the values are too large or too small to represent, the result is infinity or negative infinity. These print as inf and -inf.

While it is mathematically valid for integer powers, the exponentiation operator in GSoft BASIC will not raise a negative number to any power. Raising a negative number to any power always results in NaN (not a number), which indicates that the result of an operation is not a valid real number. The reason for this restriction has to do with the way exponentiation is implemented for floating-point numbers. The exponentiation operation a ^ b is equivalent to EXP (B * LOG (A)).

The table below shows how infinity and NaN behave for exponentiation. "Any value" refers to any number, including infinity or NaN, that is not listed explicitly in the table.

This…	raised to the power…	gives this…
NaN	any value	NaN
any value	NaN	NaN
any negative value	any value	NaN
inf (+ or -)	any value	NaN
any value	inf	inf
any value	-inf	0.0

AND

The AND operator logically combines two arguments, returning false (an INTEGER 0) if either of the arguments is 0, and true (an INTEGER 1) if both of the arguments are not zero. Both infinity and NaN (not a number, indicating a result that is not a valid number) are treated as true.

AND is generally used with other logical arguments. For example, to check to see if the value A lies between LOW and HIGH, you could use the test

```
IF LOW < A AND A < HIGH THEN CALL PROCESS(A)
```

This condition first tests to see if LOW is less than A, then checks to see if A is less than HIGH. If both conditions are true, AND returns true and the subroutine PROCESS is called. If either condition is not true, AND returns false, and subroutine PROCESS is not called.

OR

The OR operator logically combines two arguments, returning false (an INTEGER 0) if both of the arguments are 0, and true (an INTEGER 1) if either of the arguments are not zero. Both infinity and NaN (not a number, indicating a result that is not a valid number) are treated as true.

OR is generally used with other logical arguments. For example, to check to see if a character is either an uppercase or lowercase alphabetic character, you could use this test:

```
A$ = LEFT$(LINE$, 1, 1)
IF A$ >= "A" AND A$ <= "Z" OR A$ >= "a" AND A$ <= "z" THEN
    GETWORD(LINE$)
END IF
```

This condition first tests to see if A$ is an uppercase character (A$ >= "A" AND A$ <= "Z"), then checks to see if A$ is a lowercase letter (A$ >= "a" AND A$ <= "z"). If either of these tests is true, the result of the OR operation is true, and GETWORD is executed. If both conditions are false, the first letter of LINE$ is not an alphabetic character, and GETWORD is not called.

Comparison Operators

Comparison operators are used to compare two values. They return true (an INTEGER 1) if the comparison is true, and false (an INTEGER 0) if the comparison is not true. You can compare numbers, pointers or strings.

For example, 4.9 < 5 is true, so the result is an integer 1. "Fred" >= "Sam" is not true, so the result is an integer 0.

There are six comparison operators. The symbol, what the operation does, some examples and the result of the compare are shown in the table below.

Symbol	Operation	Example	Result
<	less than	-3 < 6	1
		6.1 < 6.1	0
		9 < 6	0
<=	less than or equal	-4 <= 4	1
		7 <= 7	1
		43 <= 16	0
>	greater than	2 > 7	0
		-10 > -10	0
		-10 > -20	1
>=	greater than or equal	2 >= 8	0
		3.14 >= 3.14	1
		6.1 >= 6.0	1
=	equal	9 = 9	1
		9 = -9	0
<>	not equal	3 <> 3	0
		4 <> 3	1

Some BASICs allow the multi-character comparison operators with the characters in any order. For example, >< is allowed instead of <>. Spaces are also allowed between the characters. GSoft BASIC supports these conventions, but you should generally use the standard form for the operations shown in the table.

Numbers are compared using normal rules for arithmetic. Strings, on the other hand, are compared more or less by alphabetical order. For example, "Fred" is less than "Sam". There are some surprises, though, because the characters are compared using their ASCII character orders. Uppercase letters are always less than lowercase letters, so "fred" is greater than "Sam". The ASCII character chart also includes characters other than alphabetical characters. The ASCII character chart is shown in Appendix C.

If the first letter of a string matches, the compare continues with the next letter. For example, "Sam" is less than "Susan". If all of the characters match, but the strings have different lengths, the longer string is greater than the shorter string. This means that "Fred" is less than "Frederick".

Of course, if all of the characters match and the strings are the same length, the strings are equal.

Pointers are compared by comparing their relative locations in memory. An accurate way of thinking of pointer compares is to think of comparing P1 and P2 as a two step process: first, the pointers are converted to long integers, then the resulting integers are compared. For example,

```
IF P1 < P2 THEN
```

behaves exactly like

Terms

The first part of this chapter dealt with expressions from the standpoint of traditional mathematical operations like addition, comparisons, and parentheses, just as these operations are generally used in algebra. In algebra, these operations use numbers or variables as arguments. For example,

```
IF CLNG(P1) < CLNG(P2) THEN
```

```
4 + Y
```

is a perfectly reasonable statement in algebra, and it's perfectly acceptable in BASIC, too.

In BASIC, the numbers that are used by the mathematical operations covered earlier in this chapter are called terms, and they can be many things besides just numbers or variables. In each case, though, the term is fully evaluated, giving a resultant number or string, before any of the operations discussed earlier is performed.

Constants

Constants include numbers and strings. Numbers can be integers, long integers, single-precision floating-point or double-precision floating-point. Integers and long integers can be written in either standard decimal form or using hexadecimal notation.

Chapter 7 describes the format and limitations for numeric and string constants.

Unary Math Operations

You can use a - operation before any term. This operation doesn't require a number to the left. In affect, the BASIC term

```
-V
```

works as if you typed

```
0 - V
```

When it is used this way, the - operation is technically referred to as unary subtraction to distinguish it from the similar subtraction operator.

You'll generally use this operation to indicate a negative constant or to change the sign of a variable, as in

```
X = -4
Y = -X
```

but it's perfectly legal to use the operation in the middle of an expression. For example, it is legal to write

```
z = x - -4
```

The effect is exactly the same as

```
z = x + 4
```

Eliminating the extra operation by using + rather than two - operators saves a small amount of space and time. Still, there are rare cases where the program makes more sense if the natural operations are left in place, and if clarity is more important than a byte of space and a little speed, it might make sense to use the unary subtraction operator in the middle of an expression.

There is also a unary operation for +. It rarely makes sense to use it in a program, since it doesn't actually do anything but occupy space and time.

NOT

NOT is the unary negation operation for logical values. BASIC uses numbers for logical values, assigning false the value of 0 and treating any number other than 0 as true. The NOT operation returns true if its operand is false, and false if the operand is true. From a strictly mathematical standpoint, NOT returns the INTEGER 0 if the operand is nonzero, and it returns the INTEGER 1 if the operand is zero.

In practice, NOT is usually used with logical operations or variables used to store logical values. A common example uses a variable DONE% to indicate if a loop has completed its work. The loop continues until DONE% is true.

```
DONE% = 0
WHILE NOT DONE%
    HANDLEEVENT
WEND
```

Array Subscripts

Elements of an array are selected by enclosing the array subscripts in parentheses after the array name. The subscripts are expressions, and can contain functions and other array elements. In all cases, the subscripts are evaluated first, in left to right order, then the corresponding element of the array is extracted.

If an array has more than one subscript, the subscripts are separated by commas.

Here is an example of arrays in an expression that computes the length of a multidimensional vector.

```
LENGTH = 0.0;
FOR I% = 1 TO DIMENSIONS
    LENGTH = LENGTH + VECTOR(I%)*VECTOR(I%)
NEXT
LENGTH = SQR(LENGTH)
```

Using BASIC Functions

BASIC has many built in functions. These functions return either a number or string. Most of the functions are described later in this chapter, broken down into sections based on whether they deal primarily with strings or numbers. The functions ERR and ERL, used in error handling, are described in Chapter 12.

Most functions require one or more arguments, called parameters. For example, the SQR function returns the square root of another number. Taking the square root of 4 is written like this:

```
SQR (4)
```

The parameter is 4.

This entire sequence is a single term. The parameter is an expression, and it can include anything you see in any other expression, including calls to other BASIC functions. The parameters are evaluated first, then the function is called, and finally the value returned by the function is used in the expression.

In some rare cases, the actual order in which the parameters are evaluated can affect the way the program operates. It's generally best to rethink the program so this doesn't happen. For the rare cases where it matters, GSoft BASIC evaluates parameters in left to right order.

Using FUNCTION Functions

Like most modern BASICs, GSoft BASIC supports multi-line functions. Chapter 17 tells how to create these functions, and gives examples of how they are used.

Functions defined with the FUNCTION statement behave just like built in BASIC functions in an expression.

Using DEF FN Functions

DEF FN functions are a simple way to create a function that only requires one line. Chapter 17 describes creation of DEF FN functions, and gives examples of how they are used.

In an expression, a DEF FN function looks almost like a built in BASIC function. The big difference is that the name is preceded by FN. For example, if you create a DEF FN function called LENGTH to return the length of a line, an expression that uses the function might look like this:

```
L = FN LENGTH (X, Y)
```

127

As with built-in BASIC functions, DEF FN functions are treated as a single term. The parameters are evaluated first in left to right order, then the function is evaluated, and finally the returned value is used in the expression.

The Address Operator

The symbol for the address operator is @ .

The address operator returns a pointer to a storage location. This pointer is generally used to set the value of a pointer. For example, if you need to use a pointer to index into an array, you could set a pointer to point to the first entry of the array using the address operator like this:

```
DIM IP AS POINTER TO INTEGER
DIM A(20) AS INTEGER
IP = @A(0)
FOR I% = 0 TO 20
    IP^ = I%
    IP = IP + 1
NEXT
FOR I% = 20 TO 0 STEP -1
    PRINT A(I%)
NEXT
```

The variable storage location can be any l-value. Loosely, an l-value is any term that you could assign a value to. L-values are described in detail later in this chapter.

The type for the pointer is "POINTER TO type," where type is the type of the value whose address is taken. In the example above, the type of the pointer returned by @A(0) is POINTER TO INTEGER. This value could be assigned to any pointer to an integer, but not to pointers to other types. For example, the following code is not legal.

```
DIM IP AS POINTER TO INTEGER
DIM A(3) AS SINGLE
IP = @A(3) : REM This is an illegal operation.
```

Type Casting

There are some situations, particularly when dealing with pointers to toolbox records, where the type of the pointer returned by the address operator or stored in some variable doesn't match the type of the pointer needed, yet the value is still the one needed. Type casting is used in this case. A type cast looks like a function call, but the name that appears where the function name would be is the name of a type, and the function argument is a pointer value. The value returned is a pointer to the same memory location, but with the new type. Putting this to work with the example from the address operator, you could turn the illegal assignment into a legal one like this:

```
TYPE IPTR AS POINTER TO INTEGER
DIM IP AS IPTR
DIM A(3) AS SINGLE
IP = IPTR(@A(0))
```

Type casting can also be used to convert numbers to pointers. Floating-point values are truncated to yield a long integer result, and integer values are extended to long integer values. The result is treated as a pointer. A classic example is accessing the keyboard. Rather than using PEEK and POKE statements, you can read the Apple IIGS keyboard using pointers like this:

```
PRINT KEY
END

FUNCTION KEY AS STRING
TYPE BYTEPTR AS POINTER TO BYTE
DIM KEYBOARD AS BYTEPTR
DIM STROBE AS BYTEPTR

KEYBOARD = BYTEPTR($00C000)
STROBE = BYTEPTR($00C010)
WHILE KEYBOARD^ < 128
  ! Wait for a keypress
WEND
KEY = CHR$ (KEYBOARD^ - 128)
STROBE^ = 0
END FUNCTION
```

Type casting is only supported for converting numeric values to pointers and for converting one pointer type to another pointer type. See CLNG for a way to convert a pointer to a number.

Dereferencing Pointers

In most cases, when a pointer is used in an expression, it's not the pointer value that's needed, but the value the pointer points to. To get the value of the pointer, use the name of the pointer. To get the value the pointer points to, use the name of the pointer followed by the ^ character.

For example, to assign the address of a value to a pointer, you want to change the pointer itself. In this case, you use the pointer name without the ^ character. To change the value the pointer points to, the ^ character is added, as this short example shows.

```
DIM IP AS POINTER TO INTEGER
IP = @I%
IP^ = 4
PRINT I%
```

Accessing Record Fields

The . operator is used to extract a field from a record. It appears after the name of the record, and before the name of the field.

The field can be an array, pointer, or another record. In that case, the field is extracted from the record first, then the array subscript, pointer dereference, or subsequent field dereference is handled. The record itself can also be a pointer or an array. In that case, the array access or pointer dereference appears immediately after the record name, just before the . operator.

Here are some examples of legal field accessing. These have no particular use; they are just intended to show how the pointer dereference operator is used in conjunction with array and pointer operators.

Snippet

```
TYPE POINT3D
    X
    Y
    Z
END TYPE
TYPE CUBE
    CORNER1 AS POINT3D
    CORNER2 AS POINT3D
END TYPE
TYPE PPTR AS POINTER TO POINT3D
DIM P AS POINT3D
DIM PP AS PPTR
DIM POINTS(5) AS POINT3D
DIM C AS CUBE

P.X = 1.2
PPTR = @P
PPTR^.Y = PPTR^.X * 2.0
P.Z = P.X * P.Y
POINTS(0) = P
POINTS(1).Y = POINTS(0).Y
C.CORNER1.X = 3.5
```

L-Values

In a few places in this manual, you will see a reference to something called an l-value. This is a rather descriptive term borrowed from the C language. It means any expression that can appear on the left side of an equal sign in a LET statement. In practice, it's any expression that gives the location of a value in memory.

The remainder of this section gives a very technical description of just what is and is not an l-value. Whether you wade through this description to get a full understanding of l-values or not, keep in mind that the concept is simpler than the description. An l-value is any expression that gives a place where a value is stored in memory.

L-Values are required for the location to store a value with a LET statement, for some kinds of parameters passed to subroutines and procedures, and as arguments for the address operator (@).

The simplest l-value is the name of a variable. Constants, such as 4.5, are not l-values. Think of it this way: you can store 7.1 in the variable X, but you can't store 7.1 in the number 4.5.

Arrays are a series of l-values, and an element of an array is an l-value. For example, A(X) is an l-value.

Fields in a record are l-values. For the record P in

```
TYPE POINT
    X AS SINGLE
    Y AS SINGLE
END TYPE
DIM P AS POINT
```

P.X is an l-value. So is P itself—you can store one record into another, as long as they have the same record type.

Both a pointer and the value it points to are l-values. Building on the point record, PPTR in

```
DIM PPTR AS POINTER TO POINT
```

is an l-value; you can store a pointer value in a pointer.

You can also build l-values from combinations of these operators. PPTR^.X is also an l-value, since both pointers and fields from a record are l-values.

All other expression operations are not l-values. Even something as innocent as enclosing a value in parentheses, using a type cast, or putting a + operation in front of an l-value yields an expression that is not an l-value. For example,

```
LET +X% = 4 : ! Illegal!
```

is not a legal BASIC statement, since the expression to the left of the = operator in a LET statement must be an l-value.

The Assignment Statement

[`LET`] `l-value` `'='` `expression`

The expression to the right of the = operation is evaluated and stored in the location given by the l-value.

LET is optional, and is almost always omitted from BASIC programs. It has been part of the language since the original implementation, though, and has been kept by virtually every implementation so old programs will still work.

If the expression yields a number of any kind, and the l-value is a different kind of number, the number is converted to the proper type before it is stored. If the expression is a floating-point value and the l-value is an INTEGER or LONG value, the number is truncated to the largest integer that is less than or equal to the value of the expression. For example, after

```
I% = 3.9
PRINT I%
```

prints 3.

If you assign a value to an INTEGER or LONG that is too long for the variable, a math error is generated. For example,

```
R = 40000.0
I% = R
```

generates an error.

If you assign a DOUBLE to a SINGLE, and the DOUBLE value is too large to be represented as a SINGLE, the result is infinity. The lines

```
D# = 1D40
D = D#
PRINT D#, D
```

prints

```
1.000000E40    inf
```

In addition to simple numeric values, you can also assign strings, pointers and records. The types must match exactly, though. You can't assign a numeric value to any of these values, nor can you assign one to another.

For example, while you can assign a string to another string, you cannot assign the number 4.5 to a string (or vice versa).

For pointers and records, the type must match, too. To assign one pointer to another, both pointers must point to the same thing (or you must use a type cast). To assign one record to another, both records must be the same kind of record.

The snippet shows a few examples that you can experiment with to see how this works.

Snippet

```
DIM P AS POINT, R AS POINT
P.H = 3
P.V = 6
R = P
PRINT R.H, R.V
```

132

```
S$ = "Hello, world."
H$ = LEFT$(S$, 5)
PRINT H$
DIM SP AS POINTER TO STRING, SP2 AS POINTER TO STRING
SP = @S$
SP2 = SP
PRINT SP2^
```

Mathematical Functions

ABS '(' expression ')'

Returns the absolute value of the argument.

The argument must be a numeric type. The type of the result is the same as the type of the argument. For example, if the argument is SINGLE, the result is SINGLE; if the argument is INTEGER, so is the result.

The absolute value is the same as the argument if the argument is zero or positive, and the negative of the argument if the argument is negative. For example, ABS(4) is 4, and so is ABS(-4).

The absolute value of negative infinity is infinity. The absolute value of NaN is NaN.

ATN '(' expression ')'

Returns the arc tangent of the argument. The angle is returned in radians.

If the argument is DOUBLE, the result is also DOUBLE. For SINGLE, LONG and INTEGER arguments, the result is SINGLE.

The arc tangent is the number whose tangent is the same as the argument. For any particular argument, there are actually an infinite number of answers. For example, the tangent of π/4 (roughly 0.785398) is 1, so the arc tangent of 1.0 is approximately 0.785398. As with any angle, though, adding 2π to the angle gives an equivalent angle.

The result of the ATN function is the angle between the X axis and a line from the origin and a point in the X-Y plane. The value of the argument is the Y coordinate of the point divided by the X coordinate.

θ = ATN(Y/X)

The ATN function always returns a result between 0.0 and π/2 for positive arguments, and 0.0 and -π/2 for negative values. It can't tell if a positive number should be a point where both X and Y are positive, or a reflected angle where both X and Y are negative. Both coordinates are needed to return a value from 0 to 2π; the code snippet does double duty by showing a function that returns the arc tangent over the entire range of angles from 0 to 2π, as well as showing the ATN function in action.
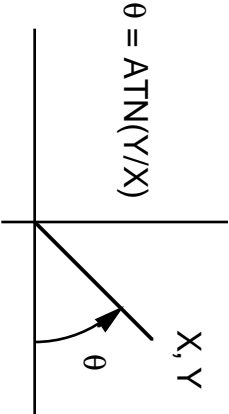
Snippet

```
FUNCTION ATN2(X, Y) AS SINGLE
PI = 3.1415927

IF X = 0 THEN
    IF Y >= 0 THEN
        ATN2 = PI / 2.0
    ELSE
        ATN2 = - PI / 2.0
    END IF
ELSE
    A = ATN ( ABS (Y / X))
    IF X >= 0 THEN
        IF Y >= 0 THEN
            ATN2 = A
        ELSE
            ATN2 = 2.0 * PI - A
        END IF
    ELSE
        IF Y >= 0 THEN
            ATN2 = PI - A
        ELSE
            ATN2 = PI + A
        END IF
    END IF
END IF
END FUNCTION
```

CDBL '(' expression ')'

Converts any numeric argument or pointer argument to a DOUBLE value.

CDBL is generally used in an expression to force the calculation to be performed using double-precision floating-point operations. For example, if you are about to multiply two real values, and would like to maintain as many significant digits as possible, you could use CDBL to force one of the arguments to DOUBLE before doing the multiply, like this:

```
PRODUCT# = CDBL(X) * Y
```

This gives a different result from the statement

```
PRODUCT# = X * Y
```

Without CDBL, the calculation is performed using a single-precision multiply, truncating the result to approximately 7 significant decimal digits. This result is extended to a double-precision value. Using CDBL, X is extended to double-precision immediately. This also forces Y to double-precision—see *Binary Conversions*, earlier in this chapter, for the complete explanation of why. The multiplication produces a result that has about 14 significant decimal digits.

At first glance, it might seem like these extra digits have no meaning. In fact, there are many numerically sensitive algorithms that depend on exactly this kind of extra precision at just the right point in the calculation. And the extra digits are real in at least one sense—computer based multiplication always doubles the number of significant digits, but these are generally discarded before you see the result.

You don't need this function when assigning a value to a DOUBLE variable. Conversion between numeric types is generally automatic. CDBL is only needed for extraordinary situations like the one described, where the precision of a number must be changed within a calculation.

CDBL can also be used to convert a pointer to an equivalent numeric value, but it generally isn't used this way. A round-off error converting the floating-point value back to a pointer could easily end up in a pointer value that is off by one byte from the expected value. In most cases, it makes more sense to use CLNG to convert a pointer to a number, and to use integer expressions to manipulate pointer values.

CINT '(' expression ')'

Converts any numeric argument or pointer argument to an INTEGER value.

CINT is frequently used in calculations that will end up producing an integer, but use intermediate floating-point values. For example, to calculate a new position for a graph, you might use an equation like

    H% = H1% + D * SIN(THETA)

Using CINT, you can force the SINGLE result to an INTEGER value immediately, like this:

    H% = H1% + CINT(D * SIN(THETA))

The result is the same, but integer addition is much faster than floating-point addition. Forcing the SINGLE value to an integer allows the use of the faster integer addition.

When the argument is a floating-point number, CINT truncates the value to convert to an integer. The result is the largest integer that is less than or equal to the floating-point value. For example, CINT(4.6) gives 4, while CINT(-4.6) gives -5.

Converting a number that is too large to an integer gives an error. The valid range for integers is -32768 to 32767.

CINT can also be used to convert a pointer to an equivalent integer value, but it generally isn't used this way. Pointers can, and usually do, have values much larger than 32767, which is the largest value an integer can hold. In most cases, it makes more sense to use CLNG to convert a pointer to a number.

CLNG '(' expression ')'

Converts any numeric argument or pointer argument to a LONG value.

CLNG is sometimes used to convert INTEGER values to LONG, extending the precision of a calculation. You might do this to avoid the automatic conversion to SINGLE that occurs when an integer operation overflows. For example, in the expression

```
I = 30000
L& = I + I
```

you know the integer value will overflow. In this case, BASIC will try the integer operation first. When it overflows, both arguments will be converted to SINGLE, an operation that takes a great deal more computer time than a LONG addition, then a SINGLE addition is performed. The SINGLE addition takes still more time. Finally, the result is converted to a LONG, which takes even more time.

It is vastly more efficient in terms of execution time to force the calculation to be performed as a LONG addition by converting both arguments to LONG right away. You can do this with CLNG:

```
I = 30000
L& = CLNG(I) + I
```

CLNG can also be used to convert floating-point arguments to LONG for faster calculations that involve both LONG and floating-point values. See CINT for an example based on this idea.

When the argument is a floating-point number, CLNG truncates the value to convert to an integer. The result is the largest integer that is less than or equal to the floating-point value. For example, CLNG(4.6) gives 4, while CLNG(-4.6) gives -5.

Converting a number that is too large to fit into a long integer gives an error. The valid range for long integers is -2147483648 to 2147483647.

CLNG is also used to convert pointers to numbers. A long value is large enough to hold any legal pointer value, so you can safely convert the pointer to a number, manipulate the number, and perhaps convert back to a pointer using a type cast.

COS '(' expression ')'

Returns the cosine of the argument. The argument is expressed in radians.

If the argument is DOUBLE, the result is also DOUBLE. For SINGLE, LONG and INTEGER arguments, the result is SINGLE.

The COS function is not accurate for very large angles. By very large angles, we mean angles larger than about 1300 radians for SINGLE arguments, and 2E8 radians for DOUBLE arguments, although the accuracy drops off gradually as the angle increases. For this reason, it is best to keep angles between -2π and 2π whenever possible. For very large arguments, COS always returns 0.0.

Refer to any book that covers trigonometry for a discussion of the cosine.

<u>Snippet</u>

```
X = LENGTH * COS(THETA)
```

CSNG '(' expression ')'

Converts any numeric argument or pointer argument to a SINGLE value.

See CDBL and CINT for some thoughts on when this function might be useful. Keep in mind that SINGLE math operations take significantly less time than their DOUBLE counterparts, so reducing a DOUBLE value to SINGLE at an appropriate place in an equation can often speed a program up significantly, in some cases with little or no loss of precision.

CSNG can also be used to convert a pointer to an equivalent numeric value, but it generally isn't used this way. A round-off error converting the floating-point value back to a pointer could easily end up in a pointer value that is off by one byte from the expected value. In most cases, it makes more sense to use CLNG to convert a pointer to a number, and to use integer expressions to manipulate pointer values.

EXP '(' expression ')'

Returns the natural exponent of the argument.

If the argument is DOUBLE, the result is also DOUBLE. For SINGLE, LONG and INTEGER arguments, the result is SINGLE.

The natural exponent is the result of raising e to a power. The number known as e is approximately 2.71828. The exponent is also the inverse of the natural logarithm, LOG. For values that are valid for both functions, EXP(LOG(X)) always returns X.

The EXP function is frequently used to manipulate powers, such as interest rates. For example, if you earn 4% per year on a passbook savings account for 10 years, the value of your initial investment M is given by

$$V = M * EXP(10.0 * LOG(1.04))$$

Snippet

```
FUNCTION POWER10 (X)
! Returns 10 raised to a power.
POWER10 = EXP(X * LOG(10.0))
END FUNCTION
```

INT '(' expression ')'

Returns the largest integer value that is less than or equal to the argument.

For INTEGER and LONG arguments, INT returns the argument. The value returned is still an INTEGER for INTEGER arguments, and LONG for LONG arguments.

For SINGLE and DOUBLE arguments, the value returned is still SINGLE or DOUBLE, but any fraction part is lost. The value returned is the largest integer that is less than or equal to the argument.

The table shows some results for various floating-point arguments.

LOG '(' expression ')'

Returns the natural logarithm of the argument.

If the argument is DOUBLE, the result is also DOUBLE. For SINGLE, LONG and INTEGER arguments, the result is SINGLE.

The natural logarithm is not defined for zero or negative arguments. If the argument is less than or equal to zero, LOG returns NaN.

expression	result
INT(5.4)	5.0
INT(3.99)	3.0
INT(-0.1)	-1.0
INT(-10.9)	-11.0

Snippet

```
FUNCTION LOG10 (X)
! Returns the base 10 logarithm of X
LOG10 = LOG (X) / LOG (10.0)
END FUNCTION
```

RND '(' expression ')'

RND returns a pseudo-random number greater than or equal to zero and less than 1.0. The value returned is always SINGLE.

While the rest of this discussion refers to the values RND returns as random numbers, they really aren't random. Pseudo-random numbers is the technical term that refers to functions like RND, which return sequences of numbers with no apparent pattern. Of course, there is a pattern—but it's a pattern that can't be detected by a series of tests for randomness. The result is a series of numbers that can be used for tasks like shuffling a deck of cards, and that will produce results as good as shuffling by hand.

Each time RND returns a value, the value is computed using a formula, and is based on the last value returned. The original value determines the sequence of numbers you get. This original value is called the seed. There is a way to specify the seed for RND, which we'll look at in a moment. In most cases, though, you should let RND pick its own seed. It bases the seed on the current date and time.

There are three ways to call RND. If the argument is a positive value, RND returns a random number. Subsequent calls return other seemingly unrelated random numbers.

If you call RND with an argument of zero, it returns the same value it returned on the previous call. This is a useful shortcut when you need to use the same random value in several places in an equation.

If you call RND with a negative argument, the argument is used as a new seed for the random number generator. After producing a series of numbers, calling RND with the same negative argument will cause RND to regenerate the same sequence of numbers. This is a very useful feature when you are debugging a program that uses RND: By temporarily placing a line like

at the start of the program, it will always generate the same series of numbers, making bugs easier to reproduce.

Snippet

```
R = RND(-1.0)
```

! Print 10 random numbers
```
FOR I = 1 TO 10
    PRINT RND(1.0)
NEXT
```

! Print 10 different random numbers based on our seed.
```
PRINT RND(-1.0)
FOR I = 1 TO 9
    PRINT RND(1.0)
NEXT
```

! Print the same 10 random numbers again.
```
PRINT RND(-1.0)
FOR I = 1 TO 9
    PRINT RND(1.0)
NEXT
```

SGN '(' expression ')'

Returns -1, 0 or 1, depending on the argument. If the argument is zero, SGN returns 0. If the argument is less than zero, SGN returns -1. If the argument is greater than zero, SGN returns 1.

The type of the result is the same as the type of the argument. For example, if the argument is SINGLE, the result is SINGLE; if the argument is INTEGER, so is the result.

Snippet

! Jump to various spots based on the sign of the number
```
ON 2 + SGN(X) GOTO 10, 20, 30
```

SIN '(' expression ')'

Returns the sine of the argument. The argument is expressed in radians.

If the argument is DOUBLE, the result is also DOUBLE. For SINGLE, LONG and INTEGER arguments, the result is SINGLE.

The SIN function is not accurate for very large angles. By very large angles, we mean angles larger than about 1300 radians for SINGLE arguments, and 2E8 radians for DOUBLE arguments, although the accuracy drops off gradually as the angle increases. For this reason, it is best to keep angles between -2π and 2π whenever possible. For very large arguments, SIN always returns 0.0.

Refer to any book that covers trigonometry for a discussion of the sine.

Snippet

```
Y = LENGTH * SIN(THETA)
```

SQR '(' expression ')'

Returns the square root of the argument.

If the argument is DOUBLE, the result is also DOUBLE. For SINGLE, LONG and INTEGER arguments, the result is SINGLE.

The square root of a number is the number that, multiplied by itself, gives the argument. For example, the square root of 4 is 2, since 2 * 2 is 4.

The square root function is not defined for negative numbers, and returns NaN if the argument is negative.

Snippet

```
HYPOTENUSE = SQR(X * X + Y * Y)
```

TAN '(' expression ')'

Returns the tangent of an angle. The argument is expressed in radians.

If the argument is DOUBLE, the result is also DOUBLE. For SINGLE, LONG and INTEGER arguments, the result is SINGLE.

The TAN function is not accurate for very large angles. By very large angles, we mean angles larger than about 1300 radians for SINGLE arguments, and 2E8 radians for DOUBLE arguments, although the accuracy drops off gradually as the angle increases. For this reason, it is best to keep angles between -2π and 2π whenever possible. For very large arguments, TAN always returns NaN.

The tangent tends toward infinity as the argument approaches π/2. If the argument gets too close to π/2, TAN returns inf. If the argument gets too close to -π/2, the result will be -inf.

Refer to any book that covers trigonometry for a discussion of the tangent.

Snippet

```
ALTITUDE = BASE_LINE * TAN(THETA)
```

String Functions

ASC '(' string-expression ')'

Returns the ASCII value for the first character in the string. If there are no characters in the string, ASC returns 0.

Characters are represented internally as a number. For example, the character A is stored as the number 65. The ASC function returns this number.

See Appendix C for a complete list of the ASCII character set, as well as the extended Apple character set.

See also CHR$, which returns the ASCII character corresponding to a given number.

Snippet

140

```
FUNCTION TOUPPER (S$) AS STRING
! Return the string as uppercase letters
S2$ = ""
WHILE LEN(S$) > 0
    C$ = LEFT$(S$, 1)
    S$ = RIGHT$(S$, LEN(S$) - 1)
    IF (C$ >= "a") AND (C$ <= "z") THEN
        C$ = CHR$ ( ASC ("A") + ASC (C$) - ASC ("a"))
    END IF
    S2$ = S2$ + C$
WEND
TOUPPER = S2$
END FUNCTION
```

CHR$ '(' expression ')'

Returns the ASCII character for a given number. The character is returned as a string of length 1.

The expression value is evaluated and converted to an INTEGER by truncation. See *Unary Conversions*, earlier in this chapter, for a detailed description of this process.

Characters are represented internally as a number. For example, the character A is stored as the number 65. Given the ASCII number, CHR$ returns the character.

The character value for 0 is used internally to mark the end of a string. CHR$(0) returns a string of length 0.

See Appendix C for a complete list of the ASCII character set, as well as the extended Apple character set.

See also ASC, which returns the ASCII number for the first character of a string. There is also a code sample showing CHR$ and ASC used together.

FRE '(' expression ')'

Forces string garbage collection, then returns the number of bytes of free space available for strings, variables and local variable space for subroutines. The number of bytes of free space are returned as a long integer.

The expression value should be zero to allow for possible expansion, but is actually ignored.

FRE can be used to determine the amount of memory available in a program, but is more often used to force garbage collection.

See Appendix F for a complete discussion of GSoft BASIC's memory use and for a description of garbage collection.

See also SETMEM. SETMEM is used to change the amount of space available for variables.

<u>Snippet</u>

```
RAM& = FRE(0)
```

LEFT$ '(' string-expression ',' expression ')'

Returns the leftmost characters in a string.

The second parameter is evaluated and converted to an INTEGER by truncation. See *Unary Conversions*, earlier in this chapter, for a detailed description of this process. If this value is negative or larger than the number of characters in the string that is passed as the first parameter, LEFT$ returns the string. If this value is positive and less than the number of characters in the string, the specified number of characters is returned, beginning with the first character of the string.

One use of LEFT$ is to peel characters from a string, processing them one by one. See the code snippet for ASC for an example.

<u>Snippet</u>

```
! Remove the first word from S$.
I% = 1
WHILE (I% < LEN(S$)) AND (MID$(S$, I%, 1) <> " ")
    I% = I% + 1
WEND
W$ = LEFT$(S$, I% - 1)
```

LEN '(' string-expression ')'

Returns the number of characters in a string. The terminating null character that marks the end of the string is not a part of the string, and is not counted by LEN. The number of characters is returned as an INTEGER.

GSoft BASIC uses the character CHR$(0) to mark the end of a string. If you intentionally imbed this character in a string, LEN will return the number of characters appearing before this character.

<u>Snippet</u>

```
I% = LEN(S$)
```

MID$ '(' string-expression ',' expression ',' expression ')'

Returns characters from any position in a string.

The last two expressions give the index of the first character to return, counting from one, and the number of characters to return, respectively. Both of these values are converted to INTEGER by truncating before being used. See *Unary Conversions*, earlier in this chapter, for a detailed description of this process.

For example,

```
MID$ ("Hello, world.", 8, 5)
```

returns the string "world".

If the index is larger than the number of characters in the string, MID$ returns the empty string. For example,

```
MID$ ("Hello, world.", 14, 5)
```

returns the string `""`, a string with no characters.

If the number of characters from the index to the end of the string is smaller than the number of characters specified by the last parameter, all available characters from the index character to the end of the string are returned. For example,

```
MID$ ("Hello, world.", 8, 20)
```

returns the string "world.".

RIGHT$ '(' string-expression ',' expression ')'

Returns the rightmost characters in a string.

The second parameter is evaluated and converted to an INTEGER by truncation. See *Unary Conversions*, earlier in this chapter, for a detailed description of this process. If this value is negative or larger than the number of characters in the string that is passed as the first parameter, RIGHT$ returns the string. If this value is less than the number of characters in the string, the specified number of characters is returned from the end of the string.

For example,

```
RIGHT$ ("Testing, 1, 2, 3", 7)
```

returns the string "1, 2, 3", while

```
RIGHT$ ("Testing, 1, 2, 3", 50)
```

returns the entire input string.

One use of RIGHT$ is to return what's left of a string after one character or a sequence of characters has been peeled off of the start of the string. See the code snippet for ASC for an example.

STR$ '(' expression ')'

STR$ accepts any number and returns the number as a string. The format matches the format the PRINT statement uses to print strings.

The actual value returned depends to some extent on the type of the number. For example, the SINGLE value returned by STR$(100000000.0) will be returned as the string "1.000000E9", while the same value returned as a LONG, as in STR$(CLNG(100000000.0)) returns the string "100000000".

See PRINT for a description of the formatting rules.

VAL '(' string-expression ')'

VAL accepts any string and returns the equivalent number.

Language Reference Manual

VAL examines the string beginning with the first character. It forms the largest possible string that is a valid number, then converts this value to a DOUBLE value. Leading signs, decimal points and exponents are allowed, but no leading or imbedded spaces can be used. The exponent character can be E, e, D or d. Allowing D or d for the exponent character is consistent with the syntax for double-precision constants in the program, although it makes no difference in the VAL function, since all results are double-precision.

If there are no characters in the string, or if the first character cannot be a part of a number, VAL returns 0.0.

This function...	...returns this number.
VAL("4")	4.0
VAL("-1e45")	-1E45
VAL("1d2")	100.0
VAL(" 4")	0.0
VAL("7th")	7.0

Chapter 12 – Control Statements

This chapter describes the statements that control the order statements are executed, and in some cases, whether a statement is executed or not.

Looping

```
DO   [   WHILE expression   |   UNTIL expression  ]
  [  statement ]*
LOOP [   WHILE expression   |   UNTIL expression  ]
```

DO-LOOP loops are a powerful looping construct that you can use any time you need to perform an action multiple times, but the number of times you need to loop can't be calculated in advance. It is typically used to loop while some condition is true.

The DO-LOOP structure is a very flexible loop statement that replaces several simpler statements in other programming languages, adding capability at the same time. In fact, the WHILE-WEND loop is a special case of the DO-LOOP loop. This flexibly comes at a price, though: It's tough to grasp how the statement works until you think it through carefully.

To understand this loop, we'll examine its four simplest parts first, then see how they can be combined.

The first form of the DO-LOOP is equivalent to the WHILE-WEND. Using the same example from the description of WHILE-WEND, the DO version looks like this:

```
DO  WHILE NOT EOF (1)
    INPUT #1, A$
    PRINT A$
LOOP
```

This loop starts by checking a condition to see if it is true; in this case, we're checking to see if the end of a file has been reached. If the condition is true—if the end of file has not been reached—the statements between DO and LOOP are executed, then the condition is tested again. This process repeats until the end of file is reached, at which time the loop is finished and the statement after LOOP is executed.

In this example, we looped while we were not at the end of the file. It's a bit more natural to think of this loop as looping until we get to the end of the file, and that's how the second version of this loop works. In BASIC, this condition looks like this:

145

```
DO UNTIL EOF (1)
    INPUT #1, A$
    PRINT A$
LOOP
```

The main advantage of this form of the DO statement is the natural way the condition is expressed. Leaving out the NOT makes the statement easier to read and easier to think about. There is also a very slight speed improvement, and the program is shorter by one byte, but these considerations are minor compared to making the statement easier to read and understand: Programs that are easy to read and understand are easier to write, easier to debug, and easier to modify than programs that are written in an unnatural way.

In this situation, we needed to test the condition before the loop started; after all, it is possible that the file might be empty, so we start off at the end of the file, and never want to execute the contents of the loop. There are also situations where you know the loop must be executed at least once. Many times these situations arise when the value to be tested doesn't even exist until the loop executes at least one time. In these situations, the test needs to be performed after the statements in the body of the loop have executed once. That's exactly what the next two versions of this loop do.

A classic example is the event loop in a desktop program. Desktop programs start by clearing the event queue. They then loop, waiting until something interesting happens, handling these events as they occur. The event loop terminates when the program is complete—signaled by some subroutine setting an exit variable to true.

A desktop program event loop might look something like this:

```
DO
    GETEVENT
    HANDLEEVENT
LOOP UNTIL DONE
```

At some point, the person using the program will do something like pulling down the File menu and selecting Quit. The subroutine that handles this event cleans up any open windows, then sets DONE to some nonzero value and returns. The loop finishes, and the program quits.

As with the DO statement with the condition at the top, you can use WHILE to reverse the sense of the loop. The DO statement would do exactly the same thing if you coded it as

```
DONE = 0
DO
    GETEVENT
    HANDLEEVENT
LOOP WHILE NOT DONE
```

It is possible to use a condition both at the top and bottom of a loop. For example, you could write a loop to cheer Karen up like this:

146

```
DO UNTIL KAREN.FEELING$ = "HAPPY"
   THROWPARTY
LOOP UNTIL BROKE
```

This loop starts off checking to see if Karen is happy. If so, nothing is done. If Karen is not happy, the body of the loop executes. Once it executes, we check to see if there is any money left. If not, the loop is finished. If there is money left, control returns to the top of the loop, where we check to see how Karen is doing. The process repeats until Karen is happy or we go broke.

While this is legal, it's usually easier to understand a loop that keeps all of the conditions together. This loop does pretty much the same thing as the first one.

```
DO UNTIL (KAREN.FEELING$ = "HAPPY") OR BROKE
   THROWPARTY
LOOP
```

The difference is that we check to see if we're broke first—probably a good idea in this case. If you really have two conditions, and one must be tested before the loop starts, but the other can't be, splitting them up can make a program shorter and more efficient. If the conditions can be put together, though, it's a good idea to do so.

It is very poor form, but technically legal, to jump into or out of a DO loop. It is not legal, and will cause an error, to jump into a DO loop in such a way that the DO statement is not encountered before the LOOP statement. It is also illegal to jump out of a DO loop without completing it; this leaves incomplete DO statements in an internal stack, and will generate an error when you exit the program. It can also cause the stack to overflow, generating a different message.

You can nest DO statements and other loop statements up to 10 levels deep. This is not a limit on the total number of looping statements, just on how many can be nested inside other loop statements. See the description of the FOR loop for an example of nested loops.

FOR identifier '=' expression TO expression [STEP
expression]
   [statement]*
NEXT [identifier] [',' identifier]*

The FOR-NEXT statement is used to loop over a series of statements a fixed number of times. If you need to loop over a series of statements, but can't calculate in advance how many times you will need to loop, use the DO-LOOP or WHILE-WEND statement.

A typical FOR-NEXT statement would print all of the numbers from 1 to 10, like this:

```
FOR I = 1 TO 10
   PRINT I
NEXT
```

Language Reference Manual

The statement begins by assigning 1 to the loop control variable I. All of the statements up to the NEXT statement are then executed. While this example only shows one statement, there are typically many statements between the FOR and NEXT statements.

As you can see, the loop control variable can be used inside the loop. While it is legal to change the value of the loop control statement inside the FOR loop, it's generally not a good idea to do so. Changing the loop control variable can create subtle and difficult to locate bugs.

When the NEXT statement is reached, the loop control value is incremented by 1. If the new value is less than or equal to the second expression, 10 in our example, the statements between FOR and NEXT are executed again. This process repeats until the loop control variable's value is larger than 10; at that point, execution continues with the statement immediately after the NEXT statement.

While the FOR loop requires the stop and start values to be known in advance, they do not have to be fixed constants like those in the first example. For example, assume you need to change all of the characters in a string to uppercase letters. With an appropriately defined function called TOUPPER somewhere else in your program, you could use a FOR loop to change the string, like this:

```
R$ = ""
FOR I% = 1 TO LEN(S$)
    R$ = R$ + TOUPPER (MID$ (S$, I%, 1))
NEXT
S$ = R$
```

A subtle point about expressions in the start and stop value is that they are calculated once, as the loop starts, and the values saved until the FOR loop completes. This has two practical implications. The first is that the program will not be faster if you calculate the values before you use them in the FOR statement. The LEN function is a perfect example: compared to other instructions, this function takes a fair amount of time. Since the LEN function is only called once, and the result stored, there is no speed gain from calculating the value in advance.

The second implication is just as important. If you change the length of the string in the loop, LEN is not recalculated. A loop that changes the value as the loop progresses is a perfect example of a loop whose stop value cannot be calculated in advance. The FOR loop would not work properly, but a loop using DO-LOOP would handle the varying string length correctly.

In the examples so far, the loop control variable is incremented by 1 each time through the loop. It is possible to tell the FOR loop to step by some other increment using the optional STEP clause. A classic example is drawing a circle using trigonometric functions. This loop steps around the circle using radians, dividing the circle into 360 one degree lines.

```
SUB CIRCLE(RADIUS, XCENTER, YCENTER)
PI = 3.141592653 5
X% = XCENTER + RADIUS
Y% = YCENTER
MOVETO (X%, Y%)
```

```
FOR ANGLE = 0 TO 2 * PI STEP PI / 180
   X% = XCENTER + RADIUS * COS(ANGLE)
   Y% = YCENTER + RADIUS * SIN(ANGLE)
   LINETO (X%, Y%)
NEXT
END SUB
```

STEP values can be calculated, like the start and stop values. STEP values can also be negative, which reverses the direction of the loop. For example, this loop will count down from 10, rather than up.

```
FOR I = 10 TO 1 STEP -1
   PRINT I
NEXT
```

You can optionally include the loop control value on the NEXT statement. This provides a quick check; if the loop control variables don't match, the program stops with an error. You can also put more than one loop control variable on the next statement, separating them with commas. In fact, you can simply use the comma to indicate that the NEXT statement applies to more than one loop control variable.

The examples below are all legal, and show the various ways you can code the NEXT statement.

```
! Matrix addition: C = A + B
FOR I% = 1 TO 10
   FOR J% = 1 TO 10
      C(I%, J%) = A(I%, J%) + B(I%, J%)
   NEXT
NEXT

FOR I% = 1 TO 10
   FOR J% = 1 TO 10
      C(I%, J%) = A(I%, J%) + B(I%, J%)
   NEXT J%
NEXT I%

FOR I% = 1 TO 10
   FOR J% = 1 TO 10
      C(I%, J%) = A(I%, J%) + B(I%, J%)
   NEXT J%, I%

FOR I% = 1 TO 10
   FOR J% = 1 TO 10
      C(I%, J%) = A(I%, J%) + B(I%, J%)
NEXT ,
```

The FOR loop control variable can be any numeric type, including BYTE, INTEGER, LONG, SINGLE or DOUBLE. It must be a single value, though, not an element of an array.

When you exit a FOR loop, the value of the FOR loop control variable is guaranteed to be the first value that would fail the loop termination test. For example, in the array addition above, both I% and J% will be 11 when the loops finish.

It is very poor form, but technically legal, to jump into or out of a FOR loop with a GOTO statement. It is not legal, and will cause an error, to jump into a FOR loop in such a way that the FOR statement is not encountered before the NEXT statement. It is also illegal to jump out of a FOR loop without completing it; this leaves incomplete FOR statements in an internal stack, and will generate an error when you exit the program. It can also cause the stack to overflow, generating a different message.

You can nest FOR statements and other loop statements up to 10 levels deep. This is not a limit on the total number of looping statements, just on how many can be nested inside other loop statements. The array addition example showed a FOR loop inside another LOOP; these were nested 2 levels deep.

WHILE
[statement]*
WEND

The WHILE-WEND loop is a simpler, more efficient version of a DO-LOOP that loops until a condition is met. It is typically used when you need to loop over a series of statements, but don't know in advance how many times you will need to loop. When the number of times you need to loop is known, use the FOR-NEXT statement.

The expression must result in a numeric value. If the result is not zero, execution continues with the first statement past the WHILE statement. Once WEND is reached, the process repeats. If the expression result is zero, the loop is skipped. In this case, execution continues with the statement just past WEND.

It is very poor form, but technically legal, to jump into or out of an executing WHILE loop. It is not legal, and will cause an error, to jump into a WHILE loop in such a way that the WHILE statement is not encountered before the WEND statement. It is also illegal to jump out of a WHILE loop without completing it; this leaves incomplete WHILE statements in an internal stack, and will generate an error when you exit the program. It can also cause the stack to overflow, generating a different message.

You can nest WHILE statements and other loop statements up to 10 levels deep. This is not a limit on the total number of looping statements, just on how many can be nested inside other loop statements. See the description of the FOR loop for an example of nested loops.

Snippet

```
+--------------------------------------------------+
|                                                  |
|  PrintFile - Print a text file                   |
|                                                  |
|  Parameters:                                     |
|                                                  |
|    name - name of the file to print              |
|                                                  |
+--------------------------------------------------+
SUB PRINTFILE (NAME$)
OPEN NAME$ FOR INPUT AS #1
WHILE NOT EOF (1)
    INPUT #1, A$
    PRINT A$
WEND
CLOSE #1
END SUB
```

Making Decisions

```
IF   expression   THEN   statement

IF   expression   GOTO   line-number

IF   expression   THEN
[   ELSE IF   expression ]*
[   ELSE ]
END IF
```

The IF statement is used to execute other statements when a specific condition exists. One form of the IF statement also allows you to check a series of conditions, selecting the appropriate alternative.

BASIC has evolved over the years, leaving us with two rather different variations of the IF statement. The classic BASIC IF statement is contained completely on a single line. The expression right after IF is evaluated. It must result in a number. If this number is zero, execution continues with the line after the IF statement. If the result of the expression is anything except zero, the statement after THEN is executed.

Here's an example of the classic IF statement in action:

```
LIFE_FORCE = LIFE_FORCE - SWORD_HIT
IF LIFE_FORCE <= 0 THEN CALL PLAYER_DIED
```

One of the interesting features of the classic IF statement is that it executes everything from THEN to the end of the line, even if there are multiple statements. You could put this to use to implement the classic shell sort.

```
DO
   SWAP = 0
   FOR I% = 1 TO ARRAY_SIZE - 1
      IF A(I%) < A(I% + 1) THEN T = A(I%) : A(I%) = A(I% + 1) : A(I% + 1) = T
:    SWAP = 1
   NEXT
LOOP WHILE SWAP
```

Before structured programming statements like the DO-LOOP and IF-THEN-ELSE statements were added to BASIC, it was very common to see statements like

```
10 IF A < B THEN GOTO 40
20 C = B
30 GOTO 50
40 C = A
50 REM
```

It was so common, in fact, that a shorthand was invented. When you have an IF statement with THEN GOTO, you can omit the THEN, as in

```
10 IF A < B GOTO 40
```

This form of the if statement isn't used much anymore, though. Its been replaced by the block IF statement, which can span multiple lines. Recoding this example with the block IF statement gets rid of the line numbers and makes the program considerably easier to follow.

```
IF A < B THEN
   C = B
ELSE
   C = A
END IF
```

This form of the IF statement allows multiple statements between the IF and ELSE, and again between the ELSE and END IF. In fact, you can even imbed other IF statements.

The ELSE clause is optional. As in this example, it is used when you need to do one thing if the condition is true, and another if the condition is not true. If you don't need to do anything when the condition is false, you can leave the ELSE out entirely.

The last important option is the ELSE IF statement, used when the IF statement must choose between several alternatives. Here's an example that changes the direction of a ball when it hits the

side of the screen. X represents the ball's position, and VX the speed. The same statements would appear right after these to handle the vertical direction.

```
X = X + VX
IF X <= 0 THEN
    VX = -VX
    IF X < 0 THEN X = -X
ELSE IF X >= 320 THEN
    VX = -VX
    IF X > 320 THEN X = 640 - X
END IF
```

You can use more than one ELSE IF clause, stringing them out to handle as many different variations on a possibility as you like. You can also mix ELSE IF with ELSE, but if you do, the ELSE clause must appear after all ELSE IF causes. When multiple ELSE IF clauses are used, the program tests them in order. When one matches, the statements between it and the next ELSE IF, ELSE or END IF are executed, and all remaining ELSE IF and ELSE clauses are skipped.

It's possible to use the ELSE IF clause to do different things based on the value of a variable, such as this color complement example.

```
IF COLOR = RED THEN
    COLOR = GREEN
ELSE IF COLOR = ORANGE THEN
    COLOR = BLUE
ELSE IF COLOR = YELLOW THEN
    COLOR = VIOLET
ELSE IF COLOR = GREEN THEN
    COLOR = RED
ELSE IF COLOR = BLUE THEN
    COLOR = ORANGE
ELSE
    COLOR = VIOLET
END IF
```

When you see a series of ELSE IF statements like this that compare the same value over and over to various possibilities, though, it's time to consider the SELECT CASE statement.

```
SELECT  CASE  expression
[  CASE  case-range  [ ','  case-range  ]*  ]*
[  CASE  ELSE  ]
END  SELECT
```

The SELECT CASE statement is used when you want to do one of several different things, making the choice based on a single value. For example, to change a color to its complementary color, you could use

```
SELECT CASE COLOR
    CASE RED
        COLOR = GREEN
    CASE ORANGE
        COLOR = BLUE
    CASE YELLOW
        COLOR = VIOLET
    CASE GREEN
        COLOR = RED
    CASE BLUE
        COLOR = ORANGE
    CASE VIOLET
        COLOR = YELLOW
END SELECT
```

The expression after SELECT CASE is evaluated one time, then compared to a succession of values. As soon as a match is found, the statements following the matching CASE are executed. There can be more than one statement after each CASE statement, even though our example only shows a single statement after each CASE. Once the statements are executed, control passes to the statement after END SELECT.

If no matching statements are found, control skips to the statement after END SELECT without executing any of the imbedded statements. If you need a catch-all case, use the CASE ELSE clause, like this:

```
FOR I = I TO 10
    PRINT I
    SELECT CASE I
        CASE 1: PRINT "st"
        CASE 2: PRINT "nd"
        CASE 3: PRINT "rd"
        CASE ELSE: PRINT "th"
    END SELECT
NEXT
```

This example also shows a compact and easy to read way to code a SELECT CASE when all of the actions fit on a single line. It is much easier to scan this SELECT CASE statement for a particular situation to see how it is handled than it is to scan the more verbose form of the first example. When multiple statements appear after each CASE, though, it is usually easier to read the program if the statements following the CASE appear on separate lines.

If you need to execute the same series of statements for several alternate values, separate the values with commas, like this:

```
SELECT CASE MID$(A$, 1, 1)
CASE "A", "E", "I", "O", "U"
  PRINT "vowel"
CASE "W", "Y"
  PRINT "sometimes vowel"
CASE ELSE
  PRINT "consonant"
END SELECT
```

You can also compare a value to a range of values. This is particularly useful with strings and floating-point numbers.

```
SELECT CASE WAVELENGTH
CASE 0.0 TO 1E-11: PRINT "Gamma rays"
CASE 1E-11 TO 1E-9: PRINT "X-rays"
CASE 1E-9 TO 4E-7: PRINT "Ultraviolet"
CASE 4E-7 TO 7E-7: PRINT "Light"
CASE 7E-7 TO 1E-5: PRINT "Infrared"
CASE 1E-5 TO 100: PRINT "Short wave radio"
CASE 100 TO 1E4: PRINT "Radio"
CASE 1E4 TO 1E38: PRINT "Long wave radio"
END SELECT
```

If you are used to languages like C and Pascal, that last example deserves a second look. Unlike most languages, BASIC can handle floating-point and strings, and due to the fact that it supports ranges, it handles them quite well. As with single values, you can code several ranges, or even ranges mixed with discrete values, on a single CASE statement. Multiple values are separated by commas.

It is possible, and with ranges even likely, that more than one CASE clause will match the value from the SELECT CASE statement. The CASE clauses are examined in the order they appear in the program. The statements after the first matching CASE clause are executed; statements after subsequent CASE clauses are not executed, and in fact, the conditions are not even tested. Once a matching CASE clause is found and the statements executed, control jumps immediately to the statement after END SELECT without examining any other possibilities.

Because of this, it is important that the CASE ELSE clause appear after all CASE clauses, just before the END SELECT statement.

Jumping Around

GOTO `line-number`

The GOTO statement is used to jump immediately to another line in the program.

While there is nothing fundamentally wrong with the GOTO statement, its overuse can lead to programs that are almost impossible to read or debug. As structured programming became popular,

the zealous attacks of the structured programmers led to the nickname of GOTO-less programming for structured programming. This isn't entirely accurate; structured programs occasionally use the GOTO statement for error handling and other abort situations. Still, the GOTO statement should be avoided unless it results in a dramatic increase in performance.

```
        GOTO 40
        DATA 1.65235, 30.656, 5.6665, 3.1556
        ...
40      DATA 1.45564, 28.667, 4.4453, 3.1327
40      !
```

ON expression GOTO line-number [',' line-number]*

The ON-GOTO statement uses an index to jump to one of several locations in a program. In modern BASICs, it has largely been replaced by the SELECT CASE statement, which is considerably easier to read and debug.

The expression is evaluated, then truncated to an integer. Counting from one, one of the line numbers is selected from the list of line numbers immediately after GOTO, and the program jumps to that line. If there are no matching line numbers, execution continues with the line after the ON-GOTO statement.

Snippet

```
        ON ERROR_NUMBER GOTO 10, 20, 30
        PRINT "Unknown error"
        GOTO 40
10      PRINT "I don't know how to "; VERB$
        GOTO 40
20      PRINT "You can't go "; DIRECTION$; " from here."
        GOTO 40
30      PRINT "You are too weak to move"
40      REM
```

Handling Errors

ERROR expression

The ERROR statement is used to trigger a run time error. The parameter is the number of the error to trigger. See Appendix A for a complete list of error numbers and the corresponding messages.

If there is an ONERR GOTO error handler active when ERROR is used, control passes to the ONERR GOTO error handler. From there, the ERR statement can be used to read the error number.

The principal use for the ERROR statement is in ONERR GOTO error handlers. It allows you to pass any error your error handler is not designed to handle back to BASIC. See the

description of ONERR GOTO for an example that shows how to use ERROR effectively in an ONERR GOTO error handler. Pay special attention to the fact that

```
ONERR GOTO 0
```

should be used before ERROR statements that appear inside an ONERR GOTO error handler. If you don't turn ONERR GOTO handling off before using ERROR, it will jump right back to the ONERR GOTO handler, generally causing an infinite loop.

ONERR GOTO line-number

When BASIC encounters any condition this manual calls an error, the normal reaction is to stop the program and print an error message. ONERR GOTO gives you a way to intercept these errors, handle them, and continue with the program.

ONERR GOTO doesn't do much when it is executed. In fact, all that happens is that the line number is recorded. If an error never occurs, nothing is ever done with the line number. If an error occurs, though, execution immediately jumps to the line you specified. From there, you can detect what error occurred using the ERR statement, and where it occurred using ERL. If it's something you want to handle, you can deal with the error, then pop back to the line where the error occurred using RESUME.

The ONERR GOTO statement can appear anywhere in the program, but the line number where the error is handled must appear in the main program, not in a SUB or FUNCTION.

You can use more than one ONERR GOTO statement in the program. If an error occurs, execution continues with the line number specified by the most recently executed ONERR GOTO statement.

Using a line number of 0 turns off ONERR GOTO error handling.

The snippet shows a short program that deliberately generates an error by assigning a value that is too large to be an integer. This error is trapped and corrected by the error handler at line 99. The next error is one the error handler is not designed to handle, though, so it uses the ERROR statement to flag the error in the normal way.

Pay special attention to the line:

```
ONERR GOTO 0
```

in the error handler. Using a line number of 0 turns ONERR GOTO error handling off. This must be done before using ERROR to flag the error. If the original error handler is still in effect when ERROR is encountered, the program will jump right back to the start of the error handler, causing an infinite loop.

157

Snippet

```
ONERR GOTO 99
X = 40000
I% = X
PRINT I%
NEXT I%
END
```

```
99  IF ERR = 19 THEN
      X = 32767
      RESUME
    END IF
    ONERR GOTO 0
    ERROR ERR
```

ERL

The ERL function is used in ONERR GOTO error handlers. When BASIC triggers an error, or when your program uses the ERROR statement to force an error, BASIC records the line number where the error occurred. ERL returns that line number.

If the statement where the error occurred did not have a line number, ERL returns 0.

The value returned by ERL is not defined before an error has been triggered.

ERR

The ERR function is used in ONERR GOTO error handlers. When BASIC triggers an error, or when your program uses the ERROR statement to force an error, the error number is recorded. ERR returns that error number.

In the case of the ERROR statement, the error number returned by ERR is the same as the parameter for the ERROR statement.

Appendix A lists the error numbers that can be triggered directly by BASIC.

The value returned by ERR is not defined before an error has been triggered.

See ONERR GOTO for an example of an ONERR GOTO error handler that uses the ERR function to determine if the error that occurred is one the error handler can deal with.

RESUME

The RESUME statement is used in an ONERR GOTO error handler to pass control back to the statement where the error occurred. Execution begins at the start of the line where the error occurred, not at the statement imbedded in that line. For example, in the line

```
IF I > 0 THEN J% = I
```

if I has the value 40000, the statement generates an error because I is larger than 32767, which is the largest value J% can hold. RESUME does not start with the assignment J% = I; the program resumes with the IF statement.

If the error occurs in a subroutine or function, RESUME restores control at the call in the main program, not in the subroutine itself.

See ONERR GOTO for an example of an error handler that uses RESUME to recover from an error.

Stopping and Starting a Program

Using CTRL-C and Command-. To Stop a Program

You can stop a program at any time by holding down the control key and pressing C, or by holding down the Command key (they key with the open apple) and pressing the period key. This has exactly the same effect as executing a STOP statement—the program can be restarted with CONT, and you can examine and change the values of variables.

While you can use these keys to stop a program during an INPUT statement, the program won't stop until after you press the return key to complete the input. All of the characters you type except the CTRL-C or Command-. are processed in the normal way. The INPUT statement processes the text you type, assigns the values to variables, and then the program stops.

Using CTRL-S To Pause a Program

You can pause a program at any time by holding down the control key and pressing S. The program freezes until you press any other key.

BREAK

Use the BREAK statement to trigger any ORCA compatible debugger, such as PRIZM, ORCA/Debugger or Splat!

When this statement is encountered, a COP instruction is executed. This can do one of two things. If you have an ORCA compatible debugger installed, this will trigger the debugger. It will display your GSoft BASIC program, starting with the line containing the BREAK. Depending on the capabilities of the debugger, you will be able to step, trace, watch or change variables, examine RAM, and so forth. When you tell the debugger to return control to the executing program, your GSoft BASIC program will continue to execute normally.

Do not use this command unless an ORCA compatible debugger is installed! ORCA compatible debuggers work by intercepting the 65816 COP instruction. There is no way for GSoft BASIC to tell if a debugger is installed or not, so it will issue the COP instruction whether or not a debugger is actually present. If there is no debugger installed, this causes the computer to crash. While this does no actual harm, the only way to recover is to reboot.

159

Snippet

```
! Try this program to see how your debugger reacts to GSoft BASIC
BREAK
SUM = 0
FOR I = 1 to 10
  SUM = SUM + I
NEXT
PRINT SUM
```

END

END is used to stop the program. The most common use is just before the first SUB or FUNCTION statement.

END can actually be used anywhere in the program. It stops execution, leaving intact all of the variables and even the various internal stacks that track FOR loops, subroutines and the like. This makes it possible to use the END statement as a quiet version of the STOP statement. By comparison, the END statement simply stops execution, while the STOP statement prints a message telling you the STOP statement was issued, and if there was a line number, which line the STOP statement appeared on.

END is also the first token in several special token pairs. The snippet shows two examples, END IF and END FUNCTION. See IF, SELECT CASE, FUNCTION, SUB and TYPE for descriptions of END used with another token.

Snippet

```
! Print a table of vector lengths
PRINT "", 1, 2, 3, 4
FOR Y = 1 TO 4
  PRINT Y, ;
  FOR X = 1 TO 4
    IF X = 4 THEN
      PRINT VECTOR_LENGTH(X, Y)
    ELSE
      PRINT VECTOR_LENGTH(X, Y), ;
    END IF
  NEXT X
NEXT Y
END

FUNCTION VECTOR_LENGTH(X, Y)
  VECTOR_LENGTH = SQR (X * X + Y * Y)
END FUNCTION
```

CONT

CONT tells GSoft BASIC to continue execution after a STOP or END. It is used from the GSoft BASIC command line, not inside a program.

After stopping a program with STOP or END, and perhaps after examining or even changing a few variables, you can use CONT to restart the program. So long as no lines have been changed, CONT picks up execution right after the STOP or END, continuing on as if nothing happened.

STOP

STOP is used as a simple debugging command. STOP stops the program, printing a message that includes the line number where the STOP statement is located—assuming, of course, that the line has a line number. You can examine variables, change the values of variables, and list lines. Once you have finished examining the state of the program, use CONT to continue execution.

WAIT expression ',' expression

WAIT is used to test bits in the computer's memory. It waits until one of several bits are set, then continues execution. In practice, WAIT is generally used to interact with hardware, watching various special memory locations that appear to the computer to be normal RAM, but are in fact set by hardware ports.

The remainder of this discussion assumes you know how bits are stored in bytes as two's complement numbers. If you do not understand how this is done, refer to a book on assembly language programming or basic computer technology for a quick course in binary mathematics.

One example that exists on every Apple IIGS is the keyboard. When you press a key, the most significant bit of the memory location $00C000 is the keyboard. When you press a key, the keyboard character pressed is placed in the other 7 bits of the same byte. After noticing that there is a key available and retrieving the value, you clear the keyboard value for the next character by storing any value to the location $00C010. The snippet shows a subroutine that takes advantage of these facts, reading a key without allowing GSoft BASIC's standard input commands to get in the way and interpret any information.

The first expression is the memory location to examine. The second expression is a bit mask. This value is logically anded with the 8 bit byte at the memory location given by the first expression. If the result is not zero—that is, if any of the memory bits were also set—the WAIT statement proceeds to the next line. If none of the bits are set, WAIT cycles again, waiting indefinitely until one of the bits turns to 1.

Memory locations start at 0 and are numbered sequentially to $00FFFFFF (16,777,215 decimal). Your computer's RAM occupies sequential memory addresses starting at zero and continuing until the RAM is exhausted, while the built-in ROM occupies the memory from $00FFFFFF down. Some of these memory locations have special uses; for example, $00C000 to $00CFFF is used for memory mapped input and output to devices like the keyboard and the cards plugged into various slots in your computer, and $E12000 to $E19FFF is the graphics display. For the Apple IIGS computer, the actual memory locations and how they are used is documented in *Apple IIGS Firmware Reference* and *Apple IIGS Hardware Reference, Second Edition*. Both of these books are available as reprints from the Byte Works. For hardware devices that are not built in, if the documentation exists at all, it probably came with the hardware itself.

Snippet

```
! ---------------------------------------------------------------
!
! GetKey - read a key directly from the keyboard
!
! Returns: Key read
!
! Notes: This subroutine does not work if the Event Manager is
! active.
!
! ---------------------------------------------------------------

FUNCTION GETKEY AS STRING
WAIT $00C000, $80
GETKEY = CHR$ ( PEEK ($00C000) - $80)
POKE $00C010, 0
END FUNCTION
```

Chapter 13 – Input and Output

There are many kinds of input and output on a modern computer. BASIC has built-in commands to deal with two of these: disk input and output and input from the keyboard with output to the computer's monitor. BASIC also has a set of commands to read data imbedded in the program itself. While you could argue that reading data from the program isn't really input, the commands look and work a lot like the other input commands, and are lumped in with them here.

There is a lot of overlap between commands that read the keyboard and write to the console, and commands that read and write disk files. At the same time, these are very different operations. This chapter covers commands that are generally used with the keyboard and monitor, as well as the commands that read data imbedded in the program. The next chapter covers commands that are generally used to read and write disk files.

Printing Text

```
PRINT [ '#' expression ]

      [ expression

      | SPC '(' expression ')'

      | TAB '(' expression ')'

      | ';'

      | ',' ]*
```

The PRINT statement is used to write text to the monitor and to text disk files. It has two major forms. The simplest to use is covered in this section; it's easy to understand, but doesn't give a lot of control over how numbers are formatted. PRINT USING gives a great deal of control over how numbers are formatted, but it is a bit harder to understand and use. PRINT USING is covered in the next section.

The PRINT statement writes a line of text to the screen. It normally moves the position for the next character, called the cursor position, to a new line after it prints the values in the expressions. The short program

```
PRINT "Hello, world."
PRINT 1990 + 8
```

163

writes these two lines to the text screen:

```
Hello, world.
1998
```

Using ? As a Typing Shortcut

PRINT is a frequently used command, especially in the immediate execution mode, where you can use BASIC as a calculator. Rather than typing the entire word, you can use a shortcut and type ?. For example,

```
?4+5
```

prints 9. It is completely equivalent to

```
PRINT 4 + 5
```

and in fact that's what you will see if you use the line in a program and then list or edit the program.

Printing Strings

String output is relatively simple. All of the characters in the string are printed, one after the other, to the text screen or disk file.

Keep in mind that string expressions work, too. The line

```
PRINT LEFT$("Hello, world.", 5)
```

prints

```
Hello
```

Printing BYTE, INTEGER and LONG Values

Any expression that evaluates to one of these integer number formats prints the result as decimal digits with no leading zeros or spaces. If the number is less that zero, it is printed with a leading - character.

The table shows several examples. The PRINT statement in the first column prints the line shown in the second column.

PRINT Statement	Output Line
PRINT 320	320
PRINT 0	0
PRINT 4000 - 5000	-1000
PRINT CINT(45.67)	45
PRINT CLNG(40)	40
PRINT $E12000	147753792
PRINT -2147483647 - 1	-2147483648

Printing SINGLE and DOUBLE Values

Floating-point numbers print three different ways, depending on the value of the number.

The most common representation for a floating-point number displays a whole number, a decimal point, and the decimal fraction. Negative numbers are preceded by the - character. For example, π to seven significant digits is written 3.141593. This is the format used for SINGLE and DOUBLE values whose absolute value is greater than or equal to 0.01 and smaller than 1E8 for SINGLE values, or 1D13 for DOUBLE values. Numbers whose absolute value is smaller than 0.01, and SINGLE numbers whose absolute value is greater than or equal to 1E8, or DOUBLE numbers whose absolute value is greater than or equal to 1D13, print in scientific notation.

SINGLE values are precise to slightly more than seven significant decimal digits, so that is the number of digits that print. If there are more than seven significant digits the number is rounded to seven significant digits before printing. Any trailing zeros appearing after the decimal point are removed; if all of the digits appearing after the decimal point are zero, the decimal point is also removed. Numbers smaller than 1.0 always print with a leading zero. After following these rules, whatever is left of the number is printed.

DOUBLE values are precise to slightly more than 13 decimal digits, so they print thirteen significant digits. The same rules are used for removing trailing zeros after the decimal point, and for removing the decimal point itself if there are no non-zero digits after the decimal point.

Scientific notation is used for numbers that are very close to zero and very far away from zero. In scientific notation, the number is multiplied or divided by 10 until the absolute value is greater than or equal to 1.0 and less than 10.0. This portion of the number is called the mantissa. For SINGLE numbers, seven significant digits are printed; for DOUBLE numbers, eight significant digits are printed. Unlike numbers in standard form, the decimal point and all trailing zeros are printed, even if all of the digits to the right of the decimal point are zero. The mantissa is followed by the letter E and the exponent for the number. Raising 10 to the power of the exponent and multiplying the result by the mantissa gives the actual number. For example, 1000000000 prints as 1.000000E9, while 0.001 prints as 1.000000E-3.

The table shows several examples. The PRINT statement in the first column prints the line shown in the second column.

PRINT Statement	Output Line
PRINT .00123456789	1.234568E-3
PRINT .0123456789	0.01234568
PRINT .123456789	0.1234568
PRINT 1.23456789	1.234568
PRINT 12.3456789	12.34568
PRINT 123.456789	123.4568
PRINT 1234.56789	1234.568
PRINT 12345.6789	12345.68
PRINT 123456.789	123456.8
PRINT 1234567.89	1234568
PRINT 12345678.9	12345680
PRINT 123456789.0	123456E8
PRINT 1.23456789123456789	1.234567891234
PRINT 1.23456789123456789D-3	1.23456789E-3
PRINT 1.23456789123456789D-2	0.01234567891234
PRINT 1.23456789123456789D-1	0.1234567891234
PRINT 1.23456789123456789D0	1.234567891234
PRINT 1.23456789123456789D12	1234567891234
PRINT 1.23456789123456789D13	12345678912.34
PRINT 1.23456789123456789D13	1.234567E13
PRINT 1.001	1.001
PRINT 1.000001	1.000001
PRINT 1.0000001	1
PRINT 1.0000007	1.000001
PRINT -10000 * 10000	-1.0000000E8
PRINT EXP(1)	2.718282

Pointers and Records

The PRINT statement can print numbers or strings, but not pointers or records. Of course, you can print the value the pointer points to if it is a number or string, and you can also print fields from a record.

Printing Multiple Expressions With Commas and Semicolons

You can print more than one number or string using a single PRINT statement by separating the expressions with commas or semicolons.

Semicolons simply separate the expressions. The expressions are crammed together with no intervening spaces. A good example of semicolons in action is

```
PRINT "The circumference of a circle whose diameter is "; R; " is "; PI *
R; "."
```

If R is 3 and PI is 3.1415926, this statement prints

The circumference of a circle whose diameter is 3 is 9.424778.

Commas tab the next item to the next tab position. Tabs appear in each column divisible by 16. The leftmost column is numbered 1. Commas give you a simple way to quickly create tables, like this table that prints the sine for each angle from 1 to 20 degrees.

```
DTR = 3.1415926 / 180.0
FOR A = 1 TO 20
   PRINT A, SIN (A * DTR)
NEXT
```

These tables are created by printing the correct number of spaces to the screen, not by using tab characters. This means the PRINT statement will work correctly with any output device that uses a monospaced font, including printers.

The 16 character columns formed by comma tabbing on the default text screen divide the 80 column screen into five equal columns. If you print a line that extends past column 64, then use a comma, the next column starts in column 1 of the next line. To keep screen and printer output as similar as possible, this tabbing method is also used in disk files and any other printed output.

You can use more than one comma in a row to skip multiple columns. For that matter, you can use multiple semicolons, too, or even mix commas and semicolons.

Controlling Spaces Using SPC and TAB

SPC and TAB are special functions you can use inside a PRINT statement. They are only valid in a PRINT statement. Each returns a string with a varying number of spaces.

SPC takes a single parameter, which it evaluates and converts to INTEGER. If the result is negative, it is replaced by 0. SPC returns a string with the resulting number of spaces. This short example shows how SPC can be used to create a table of numbers that are right justified in any desired field size. WIDTH% is the width of the columns.

```
WIDTH% = 12
DTR = 3.1415926 / 180.0 : ! Converts degrees to radians.
H1$ = "Angle"
H2$ = "Cosine"
PRINT SPC (WIDTH% - LEN (H1$)); SPC (WIDTH% - LEN (H2$)); H2$
H1$ = "-----"
H2$ = "------"
PRINT SPC (WIDTH% - LEN (H1$)); H1$; SPC (WIDTH% - LEN (H2$)); H2$
FOR A = 0 TO 10
   V1$ = STR$ (A)
   V2$ = STR$ ( COS (A * DTR))
   PRINT SPC (WIDTH% - LEN (V1$)); V1$; SPC (WIDTH% - LEN (V2$)); V2$
NEXT
```

TAB also returns a string with some number of spaces. Tab evaluates the parameter, then subtracts the column number where the next character will be printed, counting columns from 1. If the result is negative, it is replaced with zero. TAB returns a string with the specified number of

Language Reference Manual

spaces. The effect is that TAB inserts spaces so the next character you print will appear in the column you specify.

Here's the same program you just saw, but this time the second column is left justified using TAB.

```
WIDTH% = 12
DTR = 3.1415926 / 180.0 : ! Converts degrees to radians.
H1$ = "Angle"
H2$ = "Cosine"
PRINT SPC (WIDTH% - LEN (H1$)); H1$; TAB (WIDTH% + 4); H2$
H1$ = "-----"
H2$ = "--------"
PRINT SPC (WIDTH% - LEN (H1$)); H1$; TAB (WIDTH% + 4); H2$
FOR A = 0 TO 10
  V1$ = STR$ (A)
  V2$ = STR$ ( COS (A * DTR))
  PRINT SPC (WIDTH% - LEN (V1$)); V1$; TAB (WIDTH% + 4); V2$
NEXT
```

Controlling New Lines With Semicolons

In all of the examples shown so far, PRINT always starts at the beginning of a fresh line, and always finishes by moving to the start of a new line. You can prevent this behavior by ending the PRINT statement with a semicolon. This leaves the cursor at the end of the line you were printing, rather than skipping to a fresh line. The next PRINT statement, or any other statement that sends characters to the output device, picks up where the PRINT statement left off.

<u>Snippet</u>

```
DIM A$(5)
A$(0) = "red"
A$(1) = "orange"
A$(2) = "yellow"
A$(3) = "green"
A$(4) = "blue"
A$(5) = "violet"
FOR I% = 1 TO 6
  PRINT "The "; I%;
  SELECT CASE I%
    CASE 1
      PRINT "st";
    CASE 2
      PRINT "nd";
    CASE 3
      PRINT "rd";
    CASE ELSE
      PRINT "th";
  END SELECT
```

```
PRINT " color in the rainbow is ";A$(I% - 1);"."
NEXT
```

Printing Blank Lines

A PRINT statement with no parameters at all skips to the start of a new, fresh output line. If the cursor starts at the beginning of a line, this prints a blank line. If the cursor is on a line that already contains characters, perhaps because a semicolon appeared at the end of the last PRINT statement, the PRINT statement with no parameters finishes the current line, setting the cursor so the next printed character will appear at the start of the next line.

Printing To Disk Files

If the first thing after the PRINT statement is a # character, the printed text will go to a disk file rather than the text screen. The # character is followed by an INTEGER expression; this value must match one of the files currently open for output. See OPEN in the next chapter to see how to open a file for output.

Each character that would have been written to the text screen is written to the disk file instead. Whenever the PRINT statement would have skipped to the start of a fresh line, the character CHR$(13) is written to the disk file. This is the standard end of line character used for all text files and program source files on the Apple IIGS.

PRINT ['#' expression] USING format-string ';'
expression [(',' | ';') expression]* (',' | ';')

There are two forms of the PRINT statement. The PRINT USING statement gives accurate control over exactly how values will be displayed on the screen. The normal text is printed just as it appears. The standard PRINT statement, described above, doesn't give much control over the output, but it is much easier to use than PRINT USING.

Every PRINT USING statement has a format string right after USING. This format string contains normal text plus format models, which are sequences of special characters that describe how a value should be printed. The normal text is printed just as it appears; the format models are replaced by one of the values that follow the semicolon that appears after the format string. If there is more than one expression, you can separate the expressions with either commas or semicolons; unlike PRINT. With the PRINT statement, the comma and semicolon serve different purposes, but with PRINT USING, both punctuation marks simply separate expressions.

You can use a semicolon or comma after all of the expressions to indicate that a carriage return should not be printed. As with the semicolon used with the simple PRINT statement, this causes the next characters printed by a PRINT statement or an INPUT prompt to appear on the same line, right after the last character printed by the PRINT USING statement.

There is a wide variety of format models available. The sections that follow start with the most basic format model for a number, #, and gradually add the other characters that can be mixed with the # character to control the way numbers are formatted. After covering the format models used with numbers, the string format models are explained. Each section ends with a list of examples that show a format model as a format string, a value, and the characters that are printed.

To keep things simple, these examples only show one format model per line, and don't use expressions, but multiple format models can be used in a single format string, and other characters can be used, too. The examples all use a vertical bar just before and just after the format model. These bars are not required, or even common, in real format strings; they are included here so you can clearly see spaces and what happens when a value is too large for the format model.

Examples of real format strings that mix text and multiple format models appear at the end of the description of the PRINT USING statement.

Formatting Numbers

One or more # characters set up a format model for a number. The number of # characters used determines the minimum width of the output field. If the number doesn't need all of the output field, spaces are printed before the number; if there isn't enough room for the number, the entire value is still written.

If the value is a floating-point number, the number is rounded to the closest integer value. This doesn't convert the value to an INTEGER, it simply rounds the floating-point number to the closest whole number.

Format Model	Value	Prints
" |#|"	1	|1|
" |##|"	12	|12|
" |##|"	123	|123|
" |###|"	-3	|-3|
" |###|"	45.67	|46|
" |###|"	0.25	|0|

The Decimal Point

Replacing one of the # characters in a numeric format model with a period turns the output from an integer to a fixed point output. The classic use for this format model is to print a dollar amount with exactly two digits to the right of the decimal point.

If a number is too large to represent in the available space, it prints as # characters.

SINGLE numbers are precise to slightly more than seven significant digits, and DOUBLE numbers are significant to more than 13 significant digits. If you provide space for more significant digits than are available, you will get something, but the extra digits are artifacts of the number conversion, not valid values. This is shown in the last example in the table, which gets the eighth digit right—more by accident than design—but the ninth significant digit is clearly more than the available precision for a SINGLE value.

Format Model	Value	Prints
"##.##"	1	1.00|
"##.##"	12	12.00|
"##.##"	123	##.##|
"##.#"	-3.4	-3.4|
"##.##|"	-23.4	##.#|
"##.##|"	45.678	45.68|
"##.##|"	9.999	10.00|
"##.##|"	0.0049	0.00|
"###.#####|"	123.456789	123.456780|

Adding Commas

Replacing one or more of the # characters (except the first one¹) with a comma causes the number to be printed with a comma between every three digits of the whole number part, counting left from the decimal place. While it isn't required, it makes sense to put the commas in the same positions they will print. This makes the format model easier to read.

Substituting a comma for a # character does not extend the number of characters available to print values, so be sure you leave enough room for the number's significant digits and for the comma characters. If you need to print eight significant digits, as shown in the first example of the table, you will need a total of ten characters in the format model—eight for the numeric digits, and two for commas. As the second example shows, the number of commas is not the issue. The issue is how many commas the PRINT USING statement will need to insert to represent the value.

Format Model	Value	Prints
"##,###,###|"	1e6	1,000,000|
"#,#######|"	1e6	1,000,000|
"##,###,###.##|"	1.23456789123D7	12,345,678.91|
"##,###.####|"	1234.5678D0	1,234.5678|

Controlling Positive and Negative Signs

By default, if a number is positive no sign is printed, and if a number is negative a - character is printed before the first digit. You can change this behavior two ways.

First, you can indicate that both + and - characters should be used for the sign by substituting a + character for the first # character. The number will always be preceded by a sign.

The second option is to replace the last # character with a + character or - character. If you use a - character, the last character will be a - for negative numbers and a space for positive numbers. If you use a + character, the last character will still be - for negative numbers, but it will be + for positive numbers.

Leading - signs are not used in format models. A - character appearing before a format model is treated like any other character that is not used in a format model: It is simply printed.

Format Model	Value	Prints
" |####.##| "	-1.23	| -1.23|
" |+###.##| "	-1.23	| -1.23|
" |+###.##| "	1.23	| +1.23|
" |###.##-| "	-1.23	| 1.23-|
" |###.##-| "	1.23	| 1.23 |
" |###.##+| "	-1.23	| 1.23-|
" |###.##+| "	1.23	| 1.23+|

Dollar Signs

Replacing the first # character with a $ character causes a $ to be printed before the number and all leading spaces. Replacing the first two # characters with two $ characters causes a single $ character to be printed right before the first digit.

If you are using both a leading + sign and a leading $ or $$, you can put them in any order, so long as they all come before the first # character.

Format Model	Value	Prints
" |$###.##| "	-1.23	|$ -1.23|
" |$###.##| "	1.23	|$ 1.23|
" |$+##.##| "	-1.23	|$ -1.23|
" |+$##.##| "	1.23	|$ +1.23|
" |$$##.##| "	1.23	| $1.23|
" |$$##.##| "	-1.23	| -$1.23|
" |$$+#.##| "	1.23	| +$1.23|

Filling Spaces in Numbers

If the format model leaves more space to the left of the decimal point than is needed, the extra space is filled with blanks. In some applications, such as writing checks, it's a good idea to fill in any spaces so someone else doesn't fill them in for you later!

Replacing the first # character with an * prints an * in the first space. Replacing the first two # characters with ** prints an * in all leading spaces.

You can mix * characters, $ characters and + or - characters in any order, so long as they all appear before the first # character, and so long as pairs of * or $ characters appear as a pair. In all cases, dollar signs, positive signs and negative signs are placed in the available space just as they always were, and * characters fill any remaining spaces.

Format Model	Value	Prints
"|*###.##|"	-1.23	|* -1.23|
"|*###.##|"	1.23	|* 1.23|
"|*###.##|"	-1.23	|**-1.23|
"|**##.##|"	1.23	|***1.23|
"|**$#.##|"	-1.23	|$*-1.23|
"|$**#.##|"	1.23	|$**1.23|
"|$+##.##|"	-1.23	|$*-1.23|
"|+$**.##|"	1.23	|$*+1.23|
"|$$**.##|"	-1.23	|*-$1.23|
"|$$**.##|"	1.23	|**$1.23|
"|$$+**.##|"	1.23	|**+$1.23|
"|$$-**.##|"	-1.23	|-**$1.23|

Formatting Numbers In Scientific Notation

Following any number format with ^ prints the number in scientific notation. The exponent prints as the letter e, a + or - sign, and the numeric exponent. Use four ^ characters to hold any SINGLE exponent, and five to hold any DOUBLE exponent. The format model ##.#####^^^^ will print all significant digits of any SINGLE number, along with the sign and exponent. The format model ##.#############^^^^^ does the same for DOUBLE values.

A minimum of three characters are needed to print a one digit exponent; one character for the "e", one for the sign, and one for the digit. Because of this minimum size, you must use at least three ^ characters to get scientific notation.

Format Model	Value	Prints
"|##.###^^^^|"	1		1.000e+0|
"|##.###^^^^|"	-0.1234	|-1.234e-1|
"|+#.#####^^^^|"	123.45678	|+1.234568e+02|

Formatting Strings

There are three format models for strings. The & character prints an entire string, printing all characters, regardless of the size of the string. The ! character prints the first letter of a string. Two backslash characters with any number of intervening spaces prints as many of the characters as will fit. The backslashes count, so a backslash, two spaces and a backslash prints the first four characters from a string. If the format model is wider than the string, the string is printed, then blanks are printed to fill the available space.

Format Model	Value	Prints
"&"	"testing"	testing|
"!"	"testing"	t|
"\\ \\"	"testing"	te|
"\\ | \\"	"testing"	tes|
"\\ | | \\"	"testing"	test|
"\\ | | | \\"	"testing"	testing|
"\\ | | | | \\"	"testing"	testing |
"\\ | | | | | \\"	"testing"	testing |

Mixing Text and Format Models

All of the examples so far show mixing of text and format models, but only to show where the value being printed started and stopped, making it clear where spaces were printed. In actual programs it's common to see a format string with more than one format model, and to see significant text mixed in with the format models.

For example, the program

```
PV = 100
Y = 7
I = 10
FV = PV * EXP (Y * LOG (1 + I / 100))
PRINT USING "$$##.## compounded for # years at #% interest returns
$$##.##."; PV, Y, I, FV
```

prints

```
$100.00 compounded for 7 years at 10% interest returns $194.87.
```

Printing Format Characters as Text

You may need to print one of the special format characters from the format string. To prevent PRINT USING from using a character as a format character, precede it with the underscore character. To print an underscore, place two underscore characters in the format string.

For example,

```
PRINT USING "_\#_\"; 45
```

prints

```
\45\
```

even though a backslash character is usually used as a fixed length string format model.

Too Many and Too Few Format Models

If there are fewer values than format models, printing stops when the first extra format model is found. For example,

```
PRINT USING "|#|#|#|"; 1, 2
```

prints

```
|1|2|
```

Too many values for the available format models reuses the format model. This is actually a useful feature, allowing you to create multi-column tables with a single format model. The line

```
PRINT USING "|#|"; 1, 2, 3
```

prints

```
|1||2||3|
```

The only problem with a format model that will cause the program to stop with an error is providing a string to a number specifier or providing a number to a string format model.

Printing To Disk Files

Using # followed by a number between PRINT and USING redirects the text that would normally be printed to a disk file. The value that follows # must match one of the files currently open for output. See OPEN in the next chapter to see how to open a file for output.

Each character that would have been written to the text screen is written to the disk file instead. The character CHR$(13) is written to the disk file at the end of each line. This is the standard end of line character used for all text files and program source files on the Apple IIGS.

SPEED expression

SPEED causes a slight pause right after each character is written to the screen. The expression tells how long this pause should be.

A speed of 255 writes characters as rapidly as possible; this is the default. A value of 0 introduces a long delay after each character is written. Intermediate values cause progressively longer or shorter delays. Values outside of the range 0 to 255 will cause an "Illegal Quantity" error.

Keep in mind that the speed applies to every character printed to the text screen, regardless of the source, not just to PRINT and PRINT USING statements.

Snippet

```
SPEED 255
PRINT "Wow! This is slow!"
SPEED 0
```

Choosing Character Types

The ASCII character set you normally use to display characters uses 96 printable characters, numbered 32 to 127. Printing characters with an ordinal value from 0 to 31 sometimes causes something to happen, depending on the console driver, but it never results in a character being displayed on the screen. (See Appendix B, *Console Control Codes*, for a description of what actually happens for these characters.)

The display hardware that paints characters on your monitor has 256 character images. That leaves room for 160 characters other than the ASCII character set. When Apple's engineers designed the Apple IIGS ROMs, they made use of those extra characters to give you a complete inverse ASCII character set, printing black on white rather than the standard white on black, as well as for 32 extra images called mousetext characters that don't look like any normal printing character. That's where their creativity ran out, though. There are still 32 available characters, but rather than creating 32 new images, Apple's engineers repeated the uppercase letters and a few special characters that appear near them in the ASCII character chart.

The commands described in this section give you access to all of these characters.

Here's a short program that displays all of the characters in your computer's character generator, along with the output from a standard Apple IIGS.

```
! Clear the screen
HOME
FOR I = 0 TO 16
   PRINT
NEXT
! Set up the line addresses
DIM L(15)
L(0) = 1024
L(8) = 1064
FOR I = 1 TO 7
   L(I) = L(0) + I * 128
   L(I + 8) = L(8) + I * 128
NEXT
! Display the characters
FOR R = 0 TO 15
   FOR C = 0 TO 15
      POKE L(R) + C, R + C * 16
   NEXT
NEXT
```

The Apple IIGS Printing Characters

INVERSE

After using the INVERSE statement, all of the normal printable characters print as black on white rather than white on black.

To change back to standard characters, use NORMAL.

Snippet

```
INVERSE
FOR I = 32 TO 127
  PRINT CHR$(I);
NEXT
NORMAL
```

MOUSETEXT

After using the MOUSETEXT statement, all of the ASCII characters from "@" to "_", which includes the uppercase alphabetic characters, print as a series of special display characters known as the mousetext character set. These characters have ordinal values ranging from 64 for "@" to 95 for "_". All other characters print as inverse characters.

For example, the lines

```
MOUSETEXT
PRINT "A";
NORMAL
```

will print the open apple mousetext character on your screen.

The table shows the mousetext characters next to the printing character you use to print the given mousetext character. The snippet displays the entire ASCII character set with MOUSETEXT

enabled, which shows all of these characters on your screen, plus the inverse characters you will see if you print other characters with MOUSETEXT enabled.

Snippet

```
MOUSETEXT
FOR I = 32 TO 127
  PRINT CHR$(I);
NEXT
NORMAL
```

Mousetext Characters

NORMAL

Switches to standard ASCII character output. NORMAL is used after MOUSETEXT or INVERSE to switch back to the default character set.

Reading Text

INPUT ['#' expression ','] [string-expression ';']
l-value [',' l-value]*

INPUT is used to read values from the keyboard or a disk file. These values can be numbers or strings. Multiple inputs on the same typed line or line from a disk file are separated with commas.

Reading from Disk Files

INPUT normally reads from the keyboard. To read from a disk file, follow INPUT with the # character and a value that matches the file number for an open file. For example, the statement

```
INPUT #4, A$
```

reads a string from a disk file.

See OPEN for details on file numbers.

Prompts

INPUT will print a prompt on the screen to indicate it expects input. If you don't do anything special this prompt character is a question mark. For example, if you use the statement

```
INPUT NAME$
```

INPUT will print a ? character on the screen, then wait for a typed response. You can supply your own prompt string, followed by a semicolon, like this:

```
INPUT "What is your name? "; NAME$
```

You can use string expressions for the prompt, not just string constants. If the first string is followed by a semicolon, it is treated as a prompt and printed. If it is followed by a comma, it is treated like an input variable. In that case, you get the ? prompt and the INPUT statement expects a string value on the input line.

Here's a couple of examples to illustrate this subtle point. This first INPUT statement reads two comma separated strings from the keyboard:

```
INPUT A$, NAME$
```

This INPUT statement uses A$ as a prompt, reading one string from the keyboard:

```
A$ = "What is your name?"
INPUT A$; NAME$
```

Of course, you don't want ? characters showing up all over the screen while reading a disk file. If you've specified a file number using the # parameter INPUT won't print a default prompt. You can still add your own prompt, perhaps as a debugging aid, and that will still get printed to the text screen.

If you don't want a prompt and you're reading input from the keyboard, code an empty string as the prompt, like this:

```
INPUT ""; NAME$
```

Multiple Inputs

You can read several values with a single INPUT statement. For each value you are reading, the INPUT statement scans the text typed from the keyboard, forming a chunk of characters. This chunk of characters starts with the first unread character and extends until a comma or end of line mark is found. All of the characters that are read are converted into a string or a number, depending on the type of the parameter. The resulting value is saved.

The rules used to convert the text to numbers are the same as the rules used to convert program symbols to numbers. You can put spaces before the first character of the number and after the last, but not between the characters of the number. You can use leading plus or minus signs, decimal points, and exponents.

Putting this all together, here's an example that shows how to read three values from a line.

```
INPUT "Enter a point in 3 dimensions: "; X, Y, Z
```

This INPUT statement prints the prompt, then looks for three numbers. The numbers are separated by commas. One acceptable response is

```
45, .098, 2.99E4
```

Several things can go wrong with the input process. The worst is typing more inputs than the program expects. For example, if you use the INPUT statement

```
INPUT "Please enter your name: "; A$
```

and the response is

```
Fred Pennymaker, Jr.
```

the INPUT statement sees two values, one before the comma and one after. This causes the program to stop with a run-time error. You can intercept it with an ONERR-GOTO, but that's a lot of work for such a simple error.

Almost as bad is trying to read a number, but getting input that isn't a number. For example, if the INPUT statement

```
INPUT A
```

gets the response

```
4ever
```

it will choke, beeping the speaker and printing the error message "Number expected: Reenter". The program won't move on until it gets a valid number.

If the INPUT statement expects more inputs than it gets, it waits for more input. If the statement

```
INPUT "Please enter your city and state: "; CITY$, STATE$
```

gets the response

```
Indianapolis
```

the program waits for more input. If the person using the program realizes what's happening, and types

```
Indiana
```

the program continues along with no problems. The end of line marker is a perfectly acceptable substitute for a comma. But a blank screen and an apparently frozen program can be fairly confusing for an unsuspecting user of the program.

All of these problems mean that INPUT is a great command for quickly hacking out a solution to a problem when you are the only person who would use the program, or, at worst, you will be available to help and train the people who will use the program. For programs that will be used more than a few times, though, it's worth the extra work to use LINE INPUT and parse the input yourself.

LINE INPUT [**'#'** **expression** **';'**] **l-value** [**';'** **l-value**]* [**string-expression** **';'**] **l-value** [**','** **l-value**]*

LINE INPUT is almost the same command as INPUT. The only difference is that it doesn't use the comma to mark multiple inputs. It can still use multiple inputs, but each input must be typed on a separate line.

INPUT is a simple, effective way to get typed responses into a program. It works well if the person using the program understands its limitations. But there are many limitations to the INPUT statement, both in terms of error handling and because you can never use a comma in an input string.

LINE INPUT solves these problems. For reading lines of text, it can't be beat. The problems arise when you need to read values from the line—in other words, your application would work better with INPUT, but you need to create a program that will handle bogus typed responses better than INPUT allows.

In this situation, you can use LINE INPUT to read lines of text, then parse it yourself. This takes a fair amount of work, and the techniques involved vary greatly from application to application. In general, though, keep in mind that the VAL command can convert strings to numbers.

181

Positioning the Cursor

Simple programs that don't use the desktop environment of the Apple IIGS display information on the text screen. For many programs, it's a good choice, offering ease of programming in exchange for a less capable user interface. There is an intermediate between the simple printer style input and output that always places new text at the bottom of the screen and the full desktop interface, though. It relies on accurately positioning text information on the screen at specified positions. In fact, by combining the ability to place text anywhere on the screen with the special mousetext character set, you can recreate a good substitute for the desktop environment using the text screen.

As used in this section, the cursor is the position on the text display where the next printed character will appear. When a program expects input, this location is traditionally marked with a special character of some sort. GSoft BASIC uses an inverted box; a flashing character and an underscore character are also common. When this section uses the term cursor, though, it is not implying that the cursor position is marked in any special way, it is simply referring to a screen position.

The commands in this section deal with detecting the position of the cursor and changing it to a new location. They all treat the text display more or less like a graphics display that uses characters instead of pixels. The top left position on the display is column 1, row 1; the lower right corner is column 80, row 24.

There are actually three ways to deal with the text screen. The methods described here are very general across platforms that use the BASIC language. While the specific commands often vary from one platform to another, the capabilities you see here are available in almost all implementations of BASIC.

Another method relies on the console driver used to place characters on the screen. Almost all displays have a driver that places characters on the screen, scrolls text, and allows some other cursor controls. The Apple IIGS is no exception: Appendix B, *Console Control Codes*, describes the special characters you can use to control the screen in GSoft BASIC.

Finally, you can resort to placing characters directly on the text display, bypassing all of the console drivers. You can do this with BASIC's POKE statement, as shown in the example for *Choosing Character Types*, earlier in this chapter. You can find the addresses for the screen itself in several places, including *The Apple IIGS Hardware Reference*.

CSRLIN

Returns the vertical position, or line number, of the cursor. Lines are numbered starting from 1 at the top of the text display to 24 at the bottom.

<u>Snippet</u>

```
SUB MOVEUP
! Move up one line, stopping at the top of the screen
IF CSRLIN > 1 THEN
VTAB CSRLIN - 1
END IF
END SUB
```

HOME

Clears the screen, displaying spaces in all character positions, and sets the cursor position to the top left corner of the display.

HTAB expression

Sets the horizontal cursor position to the given value. This changes the cursor's column.

Columns are numbered starting from 1 at the left side of the screen to 80 at the right side. Values outside this range are allowed, as long as they can be converted to an INTEGER. Values less than 1 are treated as 1, while values greater than 80 are treated as 80.

<u>Snippet</u>

```
SUB MOVERIGHT
! Move right one column, stopping at the right edge of the screen
IF POS < 80 THEN
HTAB POS + 1
END IF
END SUB
```

POS ['(' expression ')']

Returns the horizontal position, or column number, of the cursor. Columns are numbered from 1 at the left edge of the screen to 80 at the right edge.

Some implementations of BASIC require a dummy parameter for POS to function. It doesn't do anything, but it must be there. GSoft BASIC doesn't require a parameter to POS, but in the interest of making programs easier to move between implementations of BASIC, it allows a parameter. In GSoft BASIC, the expression can evaluate to anything at all, and is not used. The value must be an INTEGER in some implementations of BASIC, though, so if you use it at all, use an INTEGER constant.

<u>Snippet</u>

```
SUB MOVELEFT
! Move left one column, stopping at the left edge of the screen
IF POS > 1 THEN
HTAB POS - 1
END IF
END SUB
```

VTAB `expression`

Sets the vertical cursor position to the given value. This changes the cursor's line.

Lines are numbered from 1 at the top of the screen to 24 at the bottom. Values outside this range are allowed, as long as they can be converted to an INTEGER. Values less than 1 are treated as 1, while values greater than 24 are treated as 24.

<u>Snippet</u>

```
SUB MOVEDOWN
! Move down one line, stopping at the bottom of the screen
IF CSRLIN < 24 THEN
    VTAB CSRLIN + 1
END IF
END SUB
```

Imbedding Data In The Program

The statements in this section are used to read data that is imbedded in a program. For example, a program that calculates the positions of planets might encode the orbital parameters for the planets in DATA statements, reading them into arrays using READ.

DATA `any-ascii-characters`

DATA statements provide input for READ statements. The information is stored exactly as you type it, more or less like a comment. This text is scanned by the READ statement using exactly the same rules as for the INPUT statement.

In general, you will place one or more values in a DATA statement, separating the values with commas.

DATA statements can hold either strings or numbers. The strings can be enclosed in quote marks, but this isn't required. Using quote marks preserves leading space characters and the case of characters; without them, leading spaces are removed and lowercase letters are converted to uppercase.

These sample DATA statements review the rules for coding numbers and strings, which are the same for DATA statements as for lines in BASIC or for INPUT. Any of the DATA statements can be read as a string. The first three lines, which contain two strings each, contain DATA statements that can only be read into a string; attempting to read a non-number into a numeric value will cause an error.

```
DATA Albany, New York
DATA Seattle, Washington
DATA "Santa Fe", "New Mexico"
DATA 1, $400, -32767
DATA 1000000, -123456
DATA 3.1415, .5, 1e9, -3.4D-6
```

READ l-value [',' l-value]*

READ reads one or more values from DATA statements.

If READ is in a SUB or FUNCTION, the DATA statement must be in the same SUB or FUNCTION, and if the DATA statement is in the main program, the READ must be there, too. DATA statements in other parts of the program are only visible to a READ in the same part of the program.

The first READ statement reads the first value from the first DATA statement, placing the value from the DATA statement into the first variable in the READ statement. This process continues, with the READ statement reading data sequentially until all of the parameters have been filled in. The next READ statement picks up where the first left off, reading the next available piece of data from a DATA statement. The number of parameters in each READ and DATA statements don't have to match. If a READ statement only reads, say, two of the available four pieces of data in a DATA statement, the next READ starts with the third value from the DATA statement. Conversely, if a READ statement reads two values, but only one is left in the current DATA statement, it skips ahead to read another value from the next available DATA statement.

There are two possible errors, either of which will stop the program with a run time error. You cannot read more data than is available, although you can use RESTORE to start over. You also can't read a number if the data supplied is not a valid number after removing leading and trailing spaces.

<u>Snippet</u>

```
! Wavelength data
DATA "red", "orange", "yellow", "green", "blue", "violet"
DATA 760, 647, 585, 575, 491, 424, 380
DIM COLORS$(6)
DIM WAVELENGTH(6)
! Read the data
FOR I = 1 TO 6
    READ COLORS$(I)
NEXT
FOR I = 0 TO 6
    READ WAVELENGTH(I)
NEXT
! Print the table
PRINT "Color       Wavelength (nm)"
PRINT "-----       ---------------"
FOR I = 1 TO 6
    PRINT USING "\   \ ### to ###";COLORS$(I), WAVELENGTH(I - 1), WAVELENGTH(I)
NEXT
```

RESTORE

RESTORE resets the pointer used to track which element of a DATA statement will be read next. After RESTORE, the next READ statement reads the first piece of data from the first available DATA statement.

185

RESTORE only resents the DATA statement pointer for the procedure in which is appears. For example, if you have DATA statements and READ statements in the main program and use RESTORE in a subroutine, the READ statement in the main program is not affected.

Snippet

```
DATA 1, 2, 3, 4
FOR I = 1 TO 4
    RESTORE
    FOR J = 1 TO I
        READ N
        PRINT N,;
    NEXT
PRINT
NEXT
```

Chapter 14 – Disk Files

This chapter covers disk access, including commands used to read from and write to disk, commands used to manipulate files on a disk, and commands used to manipulate the directory structure of a disk.

Many of the commands normally used to read information from the keyboard and write information to the computer's display can also be used to read and write text to disk files. See the previous chapter for information about these commands. The commands described in the last chapter that can also read or write disk files are INPUT, LINE INPUT, PRINT and PRINT USING.

File Names

ProDOS and HFS Names

Files, disks and directories are identified by name. The naming convention is tied to the way information is stored on disk, called the file system. The Apple IIGS supports several file systems, each with different requirements for file names. The two most common file systems used with the Apple IIGs are the ProDOS file system and the HFS file system.

ProDOS is the older of the two file systems. It is used on both the Apple]I computers using the ProDOS disk operating system and on the Apple IIGS using the ProDOS 16 and the GS/OS disk operating systems. The boot disk must use the ProDOS file system, and it's still popular for most other uses because accessing ProDOS disks takes less time than accessing HFS disks.

ProDOS file names, directory names and volume names are limited to 15 characters. The first character must be an alphabetic character. The remaining characters can be any combination of alphabetic characters, numeric digits and periods. Letter case is preserved on the disk, so a file named "Bark" will look different than a file named "Bark", but the case of the letters is not significant. Whether you use the name "Bark" or "bark", you will access the same file.

HFS is the dominant file system for the Macintosh computer. HFS has two distinct advantages over the ProDOS file system. First, ProDOS disks are limited to 32 megabytes, while standard HFS disks can handle up to 2 gigabytes, which is 2048 megabytes. Variants of HFS now available on the Macintosh can handle even larger volumes. The second advantage of HFS is the file names, which are longer and allow more characters than ProDOS. This allows more natural file names.

HFS file names and directory names are limited to 31 characters; volume names are limited to 27 characters. Any typable ASCII character except the colon is allowed anywhere in the name. (Technically, it is legal to use non-printing characters in an HFS name, but Apple recommends against it.) As with ProDOS, letter case is preserved when the file is created on disk, but it isn't significant.

187

Other File Systems

While ProDOS and HFS format disks are by far the most common on the Apple IIGS, the GS/OS operating system actually supports several others to some extent, including Microsoft's DOS, Apple Pascal, and Apple DOS. GSoft BASIC uses GS/OS for all file manipulation, and will support any disk format allowed by GS/OS to the extent GS/OS itself supports the file system.

Devices

Each device that can be accessed by the GS/OS disk operating system has at least two, and sometimes three, names. Not all devices are disks, either. It's possible to use disk input and output commands to drive the screen and keyboard, although that would interfere with the way BASIC handles input and output. Printers are handled using file commands; this is described in more detail in the section *Printing*, later in this chapter. Serial ports and networks are also handled as if they were files. In fact, by writing your own GS/OS file driver, you can use disk input and output commands to handle input and output to absolutely any kind of device.

Each device has one name based on the device number assigned when GS/OS starts. This number is dependent on the exact order the device drivers are loaded, and in some cases on which devices are started, whether they have disks inserted, and which disk is inserted. This device name consists of a period, the letter D, and the device number. For example, device 14 is named ".D14". Since these numbers can change, Apple recommends against using device names based on the device number in most situations.

The second device name is assigned by the device driver. Disks generally have names like ".DEV14". A critical device name for GSoft BASIC programmers is ".PRINTER", which allows you to print to a printer.

Finally, devices like disk drives generally have one or more volumes. Each of these volumes has another name; this is the name you normally think of for the disk itself. In cases where a disk has a single partition, using the device name based on the number, the device name assigned by the device driver, and the volume name are all equivalent in all of the GSoft BASIC commands.

Path Names

Files on disk are organized by directories, also called folders. A directory has a name, just like the disk and file. Each directory can hold files and other directories.

The full path name is a combination of the volume name, any directory names that apply, and the file name. You need the full path name of a file to uniquely identify the file. The full path name starts with a colon, and the various file names, volume names and directory names are separated by colons. You can also use a slash character instead of the colon, but all of the GSoft BASIC commands return path names with colon characters separating the parts.

As an example of path names, we'll assume we have a disk with information about various cities. The information for each city is stored in a file whose name matches the name of the city.

Of course, there are situations where two cities have the same name, although they will be in different countries or different states within the same country.

Consider the case of Manhattan, Kansas. It's located in the United States, on the continent of North America. We'll choose the name Earth for the disk itself. On the disk Earth is a directory named NorthAmerica; this in turn contains a directory named UnitedStates, which has a directory named Kansas. The directory Kansas has a file named Manhattan that holds information about that city. The full path name is

`:Earth:NorthAmerica:UnitedStates:Kansas:Manhattan`

Another file containing information about another city by the same name might have the path name

`:Earth:NorthAmerica:UnitedStates:Illinois:Manhattan`

The directory UnitedStates contains at least two directories, one named Kansas and the other named Illinois. Each of these directories contains a file named Manhattan. The fact that the file name is the same shows why the full path name is often needed to uniquely identify a file.

The Default Prefix

In practice, we don't generally use the full path name. There is a prefix called the default prefix that identifies a specific directory. In the example above, we might identify a file by first setting the default prefix to

`:Earth:NorthAmerica:UnitedStates:Kansas:`

and then using the file name

`Manhattan`

This makes a lot of sense if we're going to access several other files from the same directory. For example, if we're doing a comparison of cities in Kansas, we might also be accessing files named Topeka, KansasCity and Wichita. All of these would be in the same directory as Manhattan. By setting the default prefix to Kansas first, we can use the short file names for the various cities.

Another way to use the default prefix is to use partial path names. If you are comparing Manhattan, Kansas to Manhattan, Illinois, you can set the default prefix to

`:Earth:NorthAmerica:UnitedStates:`

and use the partial path names

`Kansas:Manhattan`

and

```
Illinois:Manhattan
```

to access information about the two cities. The difference between the partial path name and a file named Manhattan on a disk named Kansas is that the partial path name does not start with a colon.

Printing

GSoft BASIC comes with a GS/OS device driver called ".PRINTER". When this driver is properly installed in your System directory by the installer, you can open .PRINTER as a file and write text information to the file. In a nutshell, that's how you print.

The .PRINTER driver is not a sophisticated graphics printer like the ones typically used with desktop programs. That happens to be its main advantage. Since it only handles text, and doesn't deal with fonts, the .PRINTER driver is simple to install and use. It will work with almost any Apple IIGS printer, too. Any printer that can be used from AppleWorks will work using the .PRINTER driver.

For the most part, printing works just like printing text to a text file. The only special feature you need to know about is ejecting pages. Printing the character CHR$(12) ejects the current page, starting the next line at the top of a fresh page. Manually ejecting a page from some printers will confuse the printer driver, though, so you should write your programs so they eject pages by printing CHR$(12).

Here's a short program that shows just how easy it is to print using disk commands.

```
OPEN ".PRINTER" FOR OUTPUT AS #1
PRINT #1 "Hello, printer."
PRINT #1 CHR$(12)
CLOSE #1
```

Various printers respond in different ways to the non-printing characters. Most don't do anything with tab characters, but GSoft BASIC handles text output in such a way that you don't generally need tab characters. Check with your printer to see exactly what special characters it respects, then use CHR$ to send them to the printer.

The GS/OS Option

There are two different ways to deal with files from GSoft BASIC. The commands you see described in this manual are fairly common in modern implementations of BASIC. They are relatively easy to use and fairly powerful. You can bypass them entirely, though, using direct calls to the GS/OS disk operating system. Programs that access GS/OS directly tend to be larger and

more complicated than programs that use the GSoft BASIC commands, but there are some things you just can't do from GSoft BASIC. For some kinds of programs, disk access is also significantly faster if you use GS/OS commands.

In general, we recommend using the built-in commands unless you have a compelling reason for calling GS/OS directly. If you need to make GS/OS calls, refer to *Apple IIGS GS/OS Reference.*

File Numbers

Most of the commands that deal with files refer to the file by a number. This number usually appears right after a # character.

The file number is a number you pick when you open the file with the OPEN statement. It must be an integer value from 1 to 32767. While almost all of the examples in this section will use the constant 1, this is not a requirement. You can use an expression to calculate the file number, perhaps using a loop like

```
FOR I% = 1 TO 5
    CLOSE #I%
NEXT
```

to close several open files.

File Input and Output Examples

Rather than inserting short, generally meaningless examples of file input and output throughout the chapter, most of the examples of file input and output are collected here. While the examples tend to be short, they still illustrate the basic techniques used to create, write and read files in GSoft BASIC.

Line Oriented Text Files

There are two very simple programs below. The first creates a new file and writes a few lines of text. It uses the simple PRINT statement to create the information, but you could also use PRINT USING or PUT to create the file.

The second program opens the file created by the first and writes the lines to the screen. LINE INPUT isn't stopped by commas, so it handles all of the lines gracefully. This shows the basic techniques for dealing with any file organized as a series of variable length lines of text.

EOF is used to test for the end of the file. The program can read any number of lines of text without knowing in advance how many lines are in the file.

You can open the file from most editors, including the editor you use to edit GSoft BASIC programs.

Program That Writes the File

```
! Create a new file and write some test lines.
OPEN "temp" FOR OUTPUT AS #1
PRINT #1, "This is a test."
PRINT #1, "This is only a test."
PRINT #1, "If this had been a real program, we would have written something
useful."
CLOSE #1
```

Program That Reads the File

```
! Read a text file and print its contents.
OPEN "temp" FOR INPUT AS #1
WHILE NOT EOF (1)
    LINE INPUT #1, A$
    PRINT A$
WEND
CLOSE #1
```

Binary Files

There is a fundamental difference between binary files and text files. In a text file, everything is converted to ASCII characters and saved in text representation. The INTEGER value 100, which normally uses two bytes of memory, is expanded to three one byte characters. The conversion process takes time, generally requires more memory, and can lead to loss of precision if you are reading and writing floating-point values.

Binary files save information in the same raw internal format used to save the information in the computer's memory. An INTEGER takes two bytes, regardless of the value. SINGLE values always use four bytes, and there is no loss of precision as the number is written to disk and read back in. Since the number does not need to be converted to and from text, reading and writing the value is much, much faster than reading and writing text files, and since each value has the same length, you have the option of using random access files.

This program creates a new binary file and writes three SINGLE values to the file.

```
OPEN "temp" FOR BINARY AS #1
DIM A(2)
A(0) = 1.2
A(1) = 3.4
A(2) = 5.6
FOR I% = 0 TO 2
    PUT #1, , A(I%)
NEXT
CLOSE #1
```

This program reads the file, writing the values to the text screen.

```
OPEN "temp" FOR BINARY AS #1
WHILE NOT EOF (1)
    GET #1, , A
    PRINT A
WEND
CLOSE #1
```

Most of the programs in this section are rather short, but here's one very practical program. It reads any file and prints the contents. The contents of the file are printed both as hexadecimal values and, when possible, as ASCII values. You can use this program to examine the various files created by these samples to see exactly how they are stored on disk.

The sample is also on your GSoft BASIC disk in the directory :GSoft:Samples:Text.Samples. If you have installed GSoft BASIC on a hard drive, the file is also in the Samples folder there.

File Dump Example

```
! ------------------------------------------------------------
!
! Dump a file
!
! ------------------------------------------------------------
!
! This program prints the contents of any file in
! both hexadecimal and ASCII form.
!
! ------------------------------------------------------------
!
! Set up the variables
!
DIM FILENAME AS STRING
DIM COUNT AS LONG
DIM BYTES(15) AS BYTE
DIM LINECOUNT AS INTEGER
!
! Get the name of the file to dump
!
INPUT "File to dump: ";FILENAME
!
! Open and dump the file
!
OPEN FILENAME FOR INPUT AS #1
LINECOUNT = 0
```

```
    WHILE NOT EOF (1)
        GET #1, , BYTES(LINECOUNT)
        LINECOUNT = LINECOUNT + 1
        IF LINECOUNT = 16 THEN
            CALL PRINTLINE(COUNT, BYTES(), LINECOUNT)
            COUNT = COUNT + 16
            LINECOUNT = 0
        END IF
    WEND
    IF LINECOUNT <> 0 THEN
        CALL PRINTLINE(COUNT, BYTES(), LINECOUNT)
    END IF
    CLOSE #1
END

!---------------------------------------------------
!
! PRINTLINE - Print one line from the file
!
! Print the file displacement
!
! Parameters:
!
!    count - number of bytes before this line
!    bytes - line of bytes
!    lineCount - number of bytes in this line
!
!---------------------------------------------------

SUB PRINTLINE(COUNT AS LONG , BYTES() AS BYTE , LINECOUNT AS INTEGER )

! Print the file displacement

    CALL PRINTBYTE(COUNT / 256)
    CALL PRINTBYTE(COUNT)
    PRINT ":";

! Print the hexadecimal bytes

    FOR GROUP = 0 TO 3
        PRINT " ";
        FOR OFFSET = 0 TO 3
            IF GROUP * 4 + OFFSET < LINECOUNT THEN
                CALL PRINTBYTE(BYTES(GROUP * 4 + OFFSET))
            ELSE
                PRINT "  ";
            END IF
        NEXT
    NEXT
```

```
!
! Print the line as ASCII text
!
PRINT " ";
FOR OFFSET = 0 TO 15
    IF OFFSET < LINECOUNT THEN
        IF (BYTES(OFFSET) >= 32) AND (BYTES(OFFSET) < 127) THEN
            PRINT CHR$ (BYTES(OFFSET));
        ELSE
            PRINT " ";
        END IF
    ELSE
        PRINT " ";
    END IF
NEXT
PRINT " "
END SUB
!
!------------------------------------------------
!
! PRINTBYTE - Print one byte
!
! Parameters:
!   b - byte to print
!
!------------------------------------------------
SUB PRINTBYTE(B AS INTEGER )
DIM B1 AS INTEGER
!
B = B - 256 * CINT (B / 256)
B1 = B / 16
B = B - B1 * 16
IF B1 > 9 THEN
    PRINT CHR$ ( ASC ("A") + B1 - 10);
ELSE
    PRINT CHR$ ( ASC ("0") + B1);
END IF
IF B > 9 THEN
    PRINT CHR$ ( ASC ("A") + B - 10);
ELSE
    PRINT CHR$ ( ASC ("0") + B);
END IF
END SUB
```

Backtracking in Files

Many file formats hold a varying amount of information and use a length at the start of the information to tell the program reading the file what to expect. An example is the JPEG graphics format, which uses an integer value to indicate how much graphic information follows.

Language Reference Manual

Unfortunately, due to the fact the graphics files are frequently compressed as they are written to disk, you may not know how much graphic data there is until it is written!

A common way to handle this problem is to record your position in a file, write a dummy length, then write the data. Once the data has been written, you can determine how many bytes were written, then move back in the file and write the length. This common technique illustrates the use of the LOC function to determine where you are in the file, the position parameter of the PUT function to write the length to a specific location in the file, and EOF and SEEK to move to the end of the file to continue writing a new block of information.

Assuming you have opened file 1 as an output file, your subroutine to write the graphics data can start with the statements

```
P1& = LOC (1)
L% = 0
PUT #1, , L%
```

This records the location in the file where the length of the data should be written, then writes a value of 0 to occupy the correct number of bytes for the length. The program will fill this value in later, when it is known.

Your program would continue with the code that actually writes the information, possibly calling subroutines to write some of the data. There is no need to keep track of how many bytes are written. Once all of the information is written, the program records the new file position, backs up and fills in the length of the data, then resets the file position to the position after all of the data, getting ready for any additional output. The code looks like this:

```
P2& = LOC (1)
L% = P2& - P1&
PUT #1, P1& + 1, L%
SEEK #1, LOF (1) + 1
```

The two programs that follow put this idea to work in a simple example. The first program reads numbers you type until you enter 0. These numbers are written to a file that starts with the number of INTEGER values in the file. It then repeats the process, so you end up with a file containing two variable length lists of numbers. The second program reads the file.

Program That Writes the File

```
!---------------------------------------------------------------
! Read two variable length lists of integers from the
! keyboard and write them to a file.
!
! The user indicates the end of a list by typing 0.
!
! In the file, each list of integers is preceded by
! the number of integers in the list.
!
!---------------------------------------------------------------

OPEN "temp" FOR BINARY AS #1
PRINT "Enter any number of integers; enter 0 to end the list."
CALL ENTERNUMBERS(1)
PRINT "Enter another list of integers, again using 0 to end the list."
CALL ENTERNUMBERS(1)
CLOSE #1
END

!---------------------------------------------------------------
!
! EnterNumbers - enter a variable length list of numbers
!
! Parameters:
!   file - file number to write the list to
!
!---------------------------------------------------------------

SUB ENTERNUMBERS(FILE AS INTEGER )
DIM OFFSET AS LONG
DIM ENDOFFSET AS LONG
DIM COUNT AS INTEGER
DIM VALUE AS INTEGER
!
! Record the file position & reserve space for the length
!
OFFSET = LOC (FILE)
COUNT = 0
PUT #FILE, , COUNT
!
! Get the list of integers and write them to the file
!
DO
    INPUT "";VALUE
    IF VALUE <> 0 THEN PUT #FILE, , VALUE
LOOP UNTIL VALUE = 0
```

```
:
! Go back and write the length
:
ENDOFFSET = LOC (FILE)
COUNT = (ENDOFFSET - OFFSET) / SIZEOF ( INTEGER ) - 1
PUT #FILE, OFFSET + 1, COUNT
SEEK #FILE, LOF (FILE) + 1
END SUB
```

Program That Reads the File

```
DIM COUNT AS INTEGER , I AS INTEGER , VALUE AS INTEGER

OPEN "temp" FOR BINARY AS #1
WHILE NOT EOF (1)
    GET #1, , COUNT
    PRINT "List of ";COUNT;" numbers:"
    FOR I = 1 TO COUNT
        GET #1, , VALUE
        PRINT USING "######";VALUE
    NEXT
WEND
CLOSE #1
```

Reading An Entire File

Reading a file in small pieces is convenient, but generally slow, especially if you need to manipulate various pieces of a file in a more or less random order. With today's computers and their large amounts of memory, it's often practical to read an entire file into memory at once, manipulate the file, and write it back to disk. The LOF command makes this easy by reporting how many bytes or records are in a file.

The programs below put this idea to work. The first program writes a random number of INTEGER values to a file. The number of integers is between 50 and 99. The second program reads this file into an array whose size is set after the size of the file is known. It then sorts the list of random numbers and writes them back to the file and to the text screen.

Compare this example to the next one, which uses RANDOM files to do the same thing.

Program That Writes the File

```
OPEN "temp" FOR BINARY AS #1
FOR I% = 1 TO 50 * (1.0 + RND (1))
    V% = CINT (10000 * RND (1))
    PUT #1, , V%
NEXT
CLOSE #1
```

Program That Reads the File

```
DIM COUNT AS INTEGER
DIM I AS INTEGER , J AS INTEGER
DIM V AS INTEGER
! Read the file
OPEN "temp" FOR BINARY AS #1
COUNT = LOF (1) / 2
DIM A(COUNT - 1) AS INTEGER
FOR I = 0 TO COUNT - 1
   GET #1, , A(I)
NEXT
! Sort the numbers
FOR I = 0 TO COUNT - 2
   FOR J = I + 1 TO COUNT - 1
      IF A(J) < A(I) THEN
         V = A(J)
         A(I) = A(J)
         A(J) = V
      END IF
   NEXT
NEXT
! Write the values
SEEK #1, 1
FOR I = 0 TO COUNT - 1
   PUT #1, , A(I)
   PRINT A(I), ;
NEXT
CLOSE #1
```

Random Access Files

Random access files use a fixed record size so a piece of information can be quickly located in the file. A common application is a database, such as a mailing list or recipe file. Our example will use a simple file of integers, performing a disk based sort on the file. Compare this with the previous sample, which does essentially the same thing, but reads the file into memory, sorts it in memory, then writes it back to disk.

There are advantages and disadvantages to both methods. Reading the entire file into memory is definitely faster, and by using ALLOCATE to grab memory you can handle virtually any file smaller than the available memory on your computer. Some files are bigger than available memory, though, and in some cases you may not want to use all of the available memory even if the file is small enough. For example, a desk accessory shouldn't grab all of the available memory, since the main application may need it. Your application may have need for other large chunks of memory, too. Some databases are so large that the time required to read the entire file and write it back to disk is also a serious problem. In all situations where the impact of loading the file into memory is inappropriate, random access files work very well.

The program below works on the same file of random integers produced by the previous example. It reads values from the file using random access, sorting the numbers and writing them back to the file.

Random Access Program

```
DIM COUNT AS INTEGER
DIM I AS INTEGER , J AS INTEGER
DIM V1 AS INTEGER , V2 AS INTEGER
:
OPEN "temp" FOR RANDOM AS #1 LEN SIZEOF ( INTEGER )
COUNT = LOF ( 1 )
FOR I = 1 TO COUNT - 1
    GET #1, I, V1
    FOR J = I + 1 TO COUNT
        GET #1, J, V2
        IF V2 < V1 THEN
            PUT #1, I, V2
            PUT #1, J, V1
            V1 = V2
        END IF
    NEXT
    PRINT V1, ;
NEXT
CLOSE #1
```

Opening and Closing Files

CLOSE ['#' expression]

Closes a file previously opened with OPEN.

If a file number is used, CLOSE closes the specific file specified by the expression. If no file number is used, CLOSE closes all files that have been opened by OPEN. CLOSE with no file number is a quick, easy way to close all open files, especially in a program that may have exited with an error.

See *File Input and Output Examples*, earlier in this chapter, for an example.

OPEN filename FOR io-kind AS '#' expression [LEN expression]

Files must be opened before you can use most disk operations. The OPEN statement opens the file, assigning a file number to the file in the process. From the time the file is opened until you are finished with the file, all file commands will use the number you assign to identify the file. Once you are finished with a file, use CLOSE to close the file.

Files are also opened in one of five specific ways. You can read from a file opened for input, but you can't write to it, for example. If you open a file for input and need to write to it, you must close the file and open it again.

`filename` is the name of the file to open. See *File Names,* earlier in this chapter, for information about legal file names.

The file may be opened in any of the following ways by substituting the token shown for the `io-kind` field.

token	use
OUTPUT	The file is opened for output. If the file already exists, any old contents are lost.
INPUT	The file is opened for input. The file must already exist, but the file type does not matter. Input starts from the beginning of the file.
APPEND	The file is opened for output. If the file already exists, the old contents are not lost. New information is written after all of the old information.
RANDOM	The file is opened for random access. The LEN field is required; each record written to or read from the file will use that number of bytes.
BINARY	The file is opened for input and output. Old information is not lost. New information written immediately after the file is open will overwrite the information at the start of the file.

The value following # is used in subsequent file commands to identify the opened file. This value can range from 1 to 32767. No two open files may use the same file number, but once the file is closed, the number is available for use by another OPEN statement. While there are many file numbers available, only 8 files can be open at one time.

If used, the LEN expression gives the internal buffer size used to cache input and output. This field is required for random access files, and matches the length of one random access record. For all other file types, larger values use more RAM but generally result in faster disk input and output, while lower values save RAM but result in slower input and output.

Opening a file for INPUT or APPEND implies that the file already exists. For all of the other file kinds, the file may exist or may not exist. If it already exists, the old file type is not changed.

If a file is opened for OUTPUT, and it doesn't already exist, it will be created as a new file with nothing in the file. The type of the file will be TXT, the generic text file type for the Apple IIGS. Normally you'll use commands like PRINT to fill this kind of file with text. Pretty much any program that reads text will be able to read the resulting file.

Opening a file that doesn't exist for RANDOM or BINARY creates an empty file whose type is BIN. This is the generic Apple IIGS file type for files that contain something other than text. Very few programs will be able to read the resulting file, but you can read and process the file from other GSoft BASIC programs.

See *File Input and Output Examples,* earlier in this chapter, for an example.

Reading and Writing Files

EOF '(' expression ')'

Returns 0 if there is unread information in a file, and -1 if there is not.

EOF is used to see if all of the information in a file has been read. You generally test for the end of the file, and if the end of the file has not yet been reached, you read and process more information.

See *Line Oriented Text Files*, earlier in this chapter, for an example.

GET ['#' expression ',' [expression] ','] 1-value

Reads a single value from the keyboard or a disk file.

The first expression is the file number, assigned when the file is opened with OPEN.

The next expression is the location in the file to write the value. For random access files, this is the record number; for all other files, this is a byte number. In both cases, the first value in the file is numbered 1.

If no file is specified, the variable must be a string. A single character is read from the keyboard, converted to a string, and saved in the variable. If no characters have been typed, GET waits for a key before returning.

If a file is given, GET reads binary information from the file. While strings are still treated as single characters, any other data type can be read, including integers, real numbers, records or pointers.

The ability of GET to read records as well as the simple data types makes it a very powerful choice for binary file input. Coupled with PUT, you can quickly write and read records in their internal, binary format. While you are restricted to fixed length record in random access files, there is no such restriction with other file types, so you can read any data written by other programs, too. If all else fails, GET can read the file byte by byte.

GET is used in several of the examples in *File Input and Output Examples*, earlier in this chapter.

LOC '(' expression ')'

Random access files use fixed length records. In a random access file, LOC returns the number of the record most recently read or written. Contrast this with all other kinds of files, where LOC returns the number of bytes that have been read or written.

While LOC can be used for many different purposes, the classic purpose is to record the current position in a file. Combined with SEEK, this lets you write subroutines that can remember a file location and return to it at a later point.

See *Backtracking in Files*, earlier in this chapter, for an example.

LOF '(' expression ')'

Returns the number of records in a random access file, or bytes in any other kind of file.

See *Reading An Entire File*, earlier in this chapter, for an example.

PUT '#' expression ',' [expression] ',' l-value

PUT writes values to files. It is usually used for binary or random access files, although technically it can be used with any file type.

The first expression is the file number, assigned when the file is opened with OPEN.

The next expression is the location in the file to write the value. For random access files, this is the record number; for all other files, this is a byte number. In both cases, the first value in the file is numbered 1.

l-value is the value to write to the file.

PUT is used in several of the examples in *File Input and Output Examples*, earlier in this chapter.

SEEK '#' expression ',' expression

Sets the file so the next read or write occurs at the position indicated by the second expression.

For random access files, the file is divided into chunks based on the length specified when the file is opened. For all other file types, the file is divided into bytes. In each case, the first chunk is numbered 1, with the remaining chunks numbered sequentially.

SEEK is used in several of the examples in *File Input and Output Examples*, earlier in this chapter.

Dealing With Directories and Files

CHDIR pathname

Change the default prefix to pathname.

Use this command to set the default prefix inside a program. After using this command, you can use partial path names or file names to open or manipulate files.

See *File Names*, earlier in this chapter, for an explanation of file names, path names and the default prefix.

The snippet shows a subroutine that moves up one level to the directory containing the current directory.

<u>Snippet</u>

```
SUB PARENT
D$ = CURDIR$
SEPARATOR$ = LEFT$ (D$, 1)
LAST% = 1
FOR I% = 1 TO LEN (D$) - 1
    IF MID$ (D$, I%, 1) = SEPARATOR$ THEN LAST% = I%
NEXT
CHDIR LEFT$ (D$, LAST%)
END SUB
```

CURDIR$

Returns the name of the current directory.

See *File Names*, earlier in this chapter, for an explanation of file names, path names and the current directory.

See CHDIR for a short sample that shows CURDIR$ in action.

DIR$ ['(' file-name ')']

Returns file names from a directory.

The first call should specify a parameter. This can be the name of a specific file or the wildcard character "*". Full or partial path names may be used. DIR$ will return the name of the file if there is a file by the given name, or the name of the first file in the directory if the wildcard character is used.

If the wildcard character is used, subsequent calls may be made without a parameter. These calls return the names of the remaining files in the directory. When all files have been returned, DIR$ returns an empty string.

This sample program uses DIR$ to identify all of the files in a directory. It changes the names of all of the files to lowercase using the NAME statement. To understand how this works, consider a file named "FINANCES". You can use the statements

```
NAME$ = "Finances"
NAME NAME$ AS NAME$
```

to change the file from all uppercase letters to an uppercase letter followed by lowercase letters. This works because file names are case insensitive when you look for an existing file, so the name "Finances" works perfectly well to open or identify the file "FINANCES", but letter case is preserved when a file is created or renamed, so the command does change the case you see when you catalog the disk.

Snippet

```
NAME$ = DIR$ ("*")
WHILE NAME$ <> ""
  NAME2$ = ""
  WHILE NAME$ <> ""
    CH$ = LEFT$ (NAME$, 1)
    NAME$ = RIGHT$ (NAME$, LEN (NAME$) - 1)
    IF (CH$ >= "A") AND (CH$ <= "Z") THEN
      CH$ = CHR$ ( ASC (CH$) - ASC ("A") + ASC ("a"))
    END IF
    NAME2$ = NAME2$ + CH$
  WEND
  NAME NAME2$ AS NAME2$
  NAME$ = DIR$
WEND
```

KILL   filename
RMDIR   filename

Deletes the file filename.

In some implementations of BASIC, RMDIR is used to delete directories and KILL is used to delete files. There is no distinction between these operations under GS/OS: The same command can delete a directory or a file. GSoft BASIC supports both command names, but RMDIR is simply an alias for KILL. Either command can delete a directory or a file.

The samples in *File Input and Output Examples*, earlier in this chapter, create a file called TEMP. You could delete this file from inside your BASIC program with the command

```
KILL "temp"
```

MKDIR   pathname

Creates a new directory with the name pathname.

Snippet

```
MKDIR "MyDirectory"
```

NAME   filename   As   filename

Renames the file, directory or disk. The first file name is the original file name, and the second is the new file name.

As an example, consider the problem of safely writing an important database file. Even if you make the outrageous assumption that your program has no errors, and the equally outrageous assumption that GSoft BASIC, the system software, and the various other programs running can never fail, and that the computer itself will never have an error, and that floppy disks can't fail, there's always the off chance of a power outage just as you're beginning to save an all-important database that you've spent months creating and hours modifying since the last backup. Murphy's Law pretty much assures you that the power outage will occur at the worst possible moment,

leaving the entire file unusable. One way to solve this problem is to never overwrite the original file until the modified one is safely saved to disk.

Assuming the original file's name is in the variable NAME$, and that WRITE writes the data to a new file, returning TRUE if there were no errors writing the file, a safe save looks like this:

```
IF WRITE ("temp") THEN
    KILL NAME$
    NAME "temp" AS NAME$
END IF
```

The disadvantage of this kind of save is that the disk must have enough room for both the old and new versions of the file, but the distinct advantage is that a file error of almost any kind while writing the file leaves the original version untouched.

See DIR$ for an example of this command used to change the letter case of file names.

Chapter 15 – Graphics

Applesoft BASIC Graphics

GSoft BASIC supports a few of the old Applesoft BASIC graphics commands. Unlike Applesoft BASIC, these commands don't draw on the old Apple II High Resolution Graphics Screen; instead, they draw on the Apple IIGS 320 mode graphics screen. The resolution of the Apple IIGS screen, at 320 pixels wide and 200 pixels high, is very close to the older version, which is 280 by 192, so the rare old program that uses just these commands will port with little or no problem.

Missing are low resolution graphics and shape tables. After polling various people, we decided to leave these old commands out. Leaving them out makes GSoft BASIC slightly smaller and faster, and it makes your programs smaller and faster, too, since the old program tokens were available for use by new commands.

These graphics commands are closely related to the drawing commands in QuickDraw II, the graphics package for the Apple IIGS toolbox. You can safely mix these commands with QuickDraw II commands. See *Apple IIGS Toolbox Reference, Volume 2* for a complete description of QuickDraw II, or *Programming the Apple IIGS Toolbox in GSoft BASIC* for a tutorial introduction to toolbox programming that discusses QuickDraw II, among many other tools. Both are available from the Byte Works, Inc.

Graphics Commands

HCOLOR= expression

Changes the color used to draw lines with HPLOT. You must use HGR at least once before using HCOLOR=.

Applesoft BASIC supported six colors, more or less. The colors were actually connected, so you could not always use, say, green next to orange.

GSoft BASIC matches these colors as close as practical. It also adds ten new colors, giving you easy access to all sixteen Apple IIGS colors.

The table below shows the sixteen color numbers you can use with HCOLOR=, along with the colors you would get in Applesoft BASIC. The table assumes you are using the default 320 mode color palette, which is what you will be using if you don't deliberately change the color palette using QuickDraw II commands. The snippet shows a short program that displays the actual colors you can use. Press the return key after running the snippet to exit the program.

Number	Applesoft Color	GSoft BASIC Color
0	black	black
1	green	green
2	violet	purple
3	white	white
4	black	dark gray
5	orange	orange
6	blue	blue
7	white	white
8		red
9		beige
10		yellow
11		brown
12		light blue
13		lilac
14		Periwinkle blue
15		light gray
		dark green

Snippet

```
HGR
FOR COLOR = 0 TO 15
  HCOLOR= COLOR
  FOR H = COLOR * 20 TO COLOR * 20 + 20
    HPLOT H, 0 TO H, 200
  NEXT
NEXT
INPUT A$
TEXT
```

HGR

HGR turns on the graphics mode and clears the graphics screen to black. You should use HGR before using HPLOT or HCOLOR=.

HGR also starts QuickDraw II, the Apple IIGS toolbox used for drawing. After using HGR you can safely use any QuickDraw II drawing commands. Technically, you could also start QuickDraw II using toolbox calls and then use HPLOT or HCOLOR=.

If you use HGR to start QuickDraw II, you don't need to shut the tool down. GSoft BASIC will shut down QuickDraw II automatically when the program stops. You also don't have to use TEXT to switch back to the text screen, since GSoft BASIC also switches to the text screen after any program stops.

See HCOLOR= for a sample program that uses HGR.

```
HPLOT [ expression ',' expression ] [ TO expression ','
expression ]*
```

HPLOT draws a line from one location to another. You must use HGR before drawing lines with HPLOT.

HPLOT draws lines to connect one or more points on the graphics screen. Each of the points is given as a horizontal coordinate followed by a vertical coordinate. Horizontal coordinates start at the left side of the screen with column 0, and continue to column 319 at the right of the screen. The vertical coordinate starts at the top of the screen with 0, and continues to the bottom of the screen, at 199. There is nothing wrong with specifying a point that lies off of the screen, so long as the value lies in the range -32768 to 32767, but only the points that are on the screen will be visible.

The simplest command draws a single point. The command

```
HPLOT 319, 0
```

draws a point at the top right of the screen. Adding TO followed by a second point draws a line from the first point to the second. The command

```
HPLOT 0, 0 TO 319, 199
```

draws a diagonal line from the top left of the screen. Adding TO followed by a second point draws a line points as you like to a single HPLOT command. The command

```
HPLOT 10, 10 TO 20, 10 TO 20, 20 TO 10, 20 TO 10, 10
```

draws a square near the top left of the screen.

The first coordinate can be omitted, as long as at least one TO clause is used. In this case, a line is drawn from the last HPLOT location. For example, the commands

```
HPLOT 10, 10 TO 10, 10
HPLOT TO 20, 20
```

are allowed. They are equivalent to

```
HPLOT 10, 10 TO 10, 20 TO 20, 20
```

Use HCOLOR= to set the color before drawing your lines.

The snippet shows an interference pattern formed by digital pixels overlapping as the lines are drawn. Press the return key after you've seen enough of the picture.

Snippet

```
HGR
COLOR = 5
FOR V = 0 TO 199
HCOLOR= COLOR
HPLOT 0, 0 TO 319, V
IF COLOR = 5 THEN
    COLOR = 6
ELSE
    COLOR = 5
END IF
NEXT
INPUT A$
TEXT
```

TEXT

Switches from the graphics display brought up by the HGR statement to the text display normally used by GSoft BASIC.

See HCOLOR= or HPLOT for sample programs that use this command.

Chapter 16 – Utility Statements

Memory Handling

ALLOCATE **'('** **l-value** **[** **','** **expression** **]** **')'**

Allocates memory from the computer's memory. l-value is set to a pointer to the allocated memory. expression is the number of bytes of memory to reserve. If expression is not used, enough memory is reserved for one value of the type l-value.

Memory allocated by ALLOCATE comes from unallocated memory in the Apple IIGS, not from the memory already set aside for variables and strings. This allows you to allocate large chunks of memory that don't interfere with the variable space or string pool. It also gives your program access to all of the memory in the Apple IIGS, which allows you to set the memory size for variables and strings fairly low, yet gives you access to more memory if the program needs it—say, to load a large database.

Once memory is allocated, it stays allocated until you use DISPOSE to free the memory or until you exit GSoft BASIC. Restarting the program with RUN or using CLEAR will erase all variables and strings, but memory allocated with ALLOCATE is still reserved until you actually leave GSoft BASIC.

In most cases ALLOCATE is used to reserve a chunk of memory for a record. The classic example is allocating an element in a linked list. For the type and variable declarations

```
TYPE NUMBER
    AFTER AS POINTER TO NUMBER
    VALUE AS INTEGER
END TYPE
DIM TEMP AS POINTER TO NUMBER
```

the ALLOCATE statement to allocate one record for the linked list is

```
ALLOCATE (TEMP)
```

For a complete sample program that shows linked lists in action, see *Using the Record Type In The Record (Linked Lists)* in Chapter 10.

The second, optional parameter to ALLOCATE is generally used for allocating memory for variant records or for allocating large chunks of memory that are manipulated with pointers.

For a simple example of how the parameter is used with variant records, consider the declarations

211

```
TYPE COLLECTABLE
   COST
   DESCRIPTION$
   CASE COIN
      YEAR AS LONG
      DENOMINATION AS INTEGER
   CASE BEANIE_BABY
      CONDITION AS INTEGER
END

DIM THING AS POINTER TO COLLECTABLE
```

A record holding information about a coin needs 14 bytes, while a record holding information about a Beanie Baby needs 10 bytes. In cases where saving memory is of the utmost importance, you can use this fact to allocate exactly the amount of memory you need for a particular record. For example, to allocate a new record for a Beanie Baby, you would use the statement

```
ALLOCATE (THING, 10)
```

There are two potential pitfalls to avoid. First, using a size that is too small can quickly lead to disaster. If you only allocated 8 bytes for the Beanie Baby, then filled in the CONDITION field, you would write over memory that does not belong to this record. The resulting bug may not show up during routine testing, and becomes very hard to track down later. The best insurance against this kind of bug is to collect all of the sizes you use in one location, storing them in variables, and always use the variable when allocating memory. This gives you a single point in the program to change if the record ever changes.

Using this idea, the ALLOCATE call looks like this. At some point in your program, you set up the size as

```
DIM BBSIZE AS INTEGER
BBSIZE = 10
```

Later, the ALLOCATE uses this size:

```
ALLOCATE (THING, BBSIZE)
```

The second pitfall is changing the contents of a record. With a statement like

```
ALLOCATE (THING)
```

ALLOCATE allocates enough memory to hold the largest variant part, in this case 14 bytes. If you override this value, like we did in the example above, but later change the value in the record and fill in information for a coin, you will again write into memory that has not been reserved for the record.

DISPOSE '(' l-value ')'

Disposes of memory previously allocated with ALLOCATE.

It is an error to dispose of memory using a pointer that was not assigned by ALLOCATE or to dispose of the same memory twice. BASIC cannot catch this error. An error of this type may eventually lead to corrupted memory or a crash.

For a complete sample program that shows DISPOSE used to get rid of a linked list, see *Using the Record Type In The Record (Linked Lists)* in Chapter 10. The snippet shows a recursive subroutine that disposes of a binary tree.

Snippet

```
TYPE TREE
    LEFT AS POINTER TO TREE
    RIGHT AS POINTER TO TREE
    NAME$
END TYPE

SUB DUMP (T AS TREE)
IF T^.RIGHT <> NIL THEN CALL DUMP (T^.RIGHT)
IF T^.LEFT <> NIL THEN CALL DUMP (T^.LEFT)
DISPOSE (T)
END SUB
```

NIL

Returns a pointer value that is type compatible with all pointers, and that indicates a pointer which is not pointing to any memory location.

All pointers are initially set to NIL.

The ordinal value for NIL is 0.

See the snippet for DISPOSE for an example that uses NIL. NIL is universally used to indicate that a pointer doesn't point to anything. The recursive subroutine that is disposing of the binary tree tests for NIL, so it can stop whenever it gets to a pointer that doesn't point to another record.

SETMEM '(' expression ',' expression ')'

Sets the size of a memory buffer. The first expression is the memory buffer to set; this is 0 for the variable buffer and 1 for the program buffer. The second expression is the new size for the buffer in bytes.

It is not an error to use something other than 0 or 1 for the first parameter, but if you do the command is ignored. This allows future versions of GSoft BASIC to add other buffers whose size can be set with SETMEM, yet the programs will still work with this version of GSoft BASIC.

SETMEM must be used in the main part of a program, not in subroutines or functions. In general, SETMEM should be the first command in the program.

Memory buffer 0 is used for variables, types, strings, and some record keeping involved in subroutine calls. The default size for this buffer is 64K. If you get out of memory errors when you

are not using ALLOCATE, use SETMEM to change the size of the variable buffer. Using SETMEM also deletes any existing variables; in this respect it works exactly like CLEAR.

Memory buffer 1 is used for the program itself. Like the variable buffer, this buffer defaults to 64K bytes. If your program is larger than about 32K, consider setting the buffer to about twice the size of your program. You generally do this from the command line. One way to get an idea of the size of your program is to look at the program on disk with the CATALOG command. CATALOG shows the number of blocks used by the program. On a ProDOS format disk, the memory used by the program will be a little smaller than the number of blocks times 512.

If you are using the version of GSoft BASIC that runs from ORCA, or if you create a stand-alone program with the MakeRuntime utility, the program buffer is set to the actual size of the program. You should not need SETMEM to change the size of the program buffer in these cases, but it is available.

<u>Snippet</u>

```
! Double the size of the variable buffer, setting it to 128K.
SETMEM (0, CLNG (128) * 1024)
```

SIZEOF '(' (type | identifier) ')'

Returns the size required to store one value of a given type, or the size used by the variable identifier. The size is given in bytes.

SIZEOF is generally used as the LEN parameter to an OPEN statement or as the second parameter to ALLOCATE. For example, to open a file for random access that will hold a record called PERSON, you would use the command

```
OPEN "MailingList" FOR INPUT AS #1 LEN SIZEOF (PERSON)
```

If you want to reserve memory to read this entire file into memory at once, you could follow this up with

```
ALLOCATE (PERSON_POINTER, LOF (1) * SIZEOF (PERSON))
```

You can use any type name, including the built-in types. For example, SIZEOF (INTEGER) returns 2.

Peeks and Pokes

The commands in this section are used to read and write information directly to specific memory locations. This is generally done for one of two reasons: Either the program needs to read or write to a memory location that is mapped to an external device, or the program has been ported from Applesoft BASIC, which relied heavily on PEEKs and POKEs for operations that are commands in GSoft BASIC.

See Appendix E, *Converting Applesoft BASIC Programs to GSoft BASIC*, for some common hardware locations on the Apple IIGS, as well as for a table showing some common Applesoft BASIC PEEKs and POKEs and their equivalent GSoft BASIC commands.

For a complete list of the various documented memory locations that you might want to use with PEEK or POKE, see *Apple IIGS Hardware Reference* and *Apple IIGS Firmware Reference*. Reprints of both books are available from the Byte Works. You might also want to use PEEK and POKE to control some hardware cards; see the documentation that comes with the card itself for a list of the appropriate memory locations.

PEEK `'('` `expression` `')'`

Returns the value of the byte located at the address `expression`.

The expression can address any memory location, which, in general, requires a `LONG` value. Memory locations on the Apple IIGS range from 0 to 16777215 ($00FFFFFF hexadecimal). It is best to avoid floating-point values for the expression, since a round-off error could easily cause you to read the wrong value from memory.

The snippet shows a subroutine that looks to see if a keypress is available. This subroutine should not be used if the Event Manager is active; it looks directly at the keyboard strobe, which is something the Event Manager also does.

<u>Snippet</u>

```
FUNCTION KEYPRESS AS INTEGER
   KEYPRESS = PEEK ($00C000) > 127
END FUNCTION
```

POKE `expression` `','` `expression`

The least significant 8 bits of the value in the second expression are stored in the memory location specified by the first expression.

POKE can address any memory location, which, in general, requires a `LONG` value. Memory locations on the Apple IIGS range from 0 to 16777215 ($00FFFFFF hexadecimal). It is best to avoid floating-point values for the expression, since a round-off error could easily cause you to write the value to the wrong location in memory.

The value poked into memory should be in the range 0 to 255, which is all a byte of memory can hold. If the value lies outside this range, it is converted to the range 0 to 255 by converting the second expression to an INTEGER, then using the least significant 8 bits of the result. If you understand the two's complement representation used to represent integers in memory, it's easy enough to figure out what the resulting value will be—if you don't, it's best to make sure the value being poked is in the expected range.

The snippet reads a character from the keyboard. This method should not be used if the Event Manager is being used. GSoft BASIC does not normally use the Event Manager, but you can turn it on from your GSoft BASIC program.

```
FUNCTION READKEY AS STRING
WHILE PEEK ($00C0000) < 128
WEND
READKEY = CHR$ (PEEK ($00C0000) - 128)
POKE $00C0010, 0 : ! Clear the key
END FUNCTION
```

Clearing the Workspace

CLEAR

Erases all types, variables and strings. Variables are removed whether they were created with the DIM statement or by being used without encountering a DIM statement.

This statement is generally used to completely reset a program. This is occasionally handy during debugging or when writing an ONERR GOTO error handler.

GSoft Version Number

VERSION

Returns the GSoft BASIC version number encoded as a long integer. This long integer can be easily compared to see if the version of GSoft BASIC contains some critical new feature. When properly formatted and printed, this version number will match the version number printed when the GSoft BASIC shell starts, as well as the version number shown by the Finder for GSoft.Sys16.

The GSoft BASIC version number consists of five parts. These are:

major version
This is the major version release number. This number doesn't change very often, and when it does, it indicates major changes. This probably includes a new manual, and may include changes that may some old programs fail under the new version of the language. This number starts at 1.

minor version
This version number changes whenever a new feature is added to GSoft BASIC or a change to an existing feature is made that affects the documentation. These changes are generally minor changes. They are typically documented in a release notes file, and do not require a new manual, although an updated manual may be available. This number starts at 0 each time the major version number changes.

bug version
This version number is changed each time GSoft BASIC changes, no matter how minor the change might be. A change in the bug version usually indicates a release that fixes errors or expands limitations in an earlier

release. Nothing in the documentation changes, but the changes to the program will be listed in the release notes file that accompanies each release. This number starts at 0 each time the minor version number changes.

release type   The release type is one of four values.

0   Indicates a commercial release of GSoft BASIC. This is the most common value.

1   Development release. This designation is used for versions that are in the early stages of development. All planned features are not implemented and testing is incomplete. Programming errors are expected in a development release.

2   Alpha release. An alpha release contains all of the features initially planned for the commercial release, although some of the features may not be in their final form. Most testing is complete, and major bugs are either eliminated or clearly identified.

3   Beta release. A beta release is a nearly complete commercial release. All planned features are implemented, and no changes to these features is anticipated by the programmers or testers, although subsequent beta testing may identify areas that still need to be changed. All known bugs are eliminated. Formal testing is complete.

release version   This indicates the development, alpha or beta release level. It starts at 1 each time the release designation changes.

The initial release of GSoft BASIC, and the one this manual describes, is 1.0.0. VERSION returns the value 1000000.

A typical use of the VERSION command is to check to see if a particular feature is available. For example, let's assume you are writing a program using a later version of GSoft BASIC, say version 1.1.0, and your program uses a feature that was not implemented in this release. You could start your program by calling this subroutine to make sure the version of GSoft BASIC is appropriate.

```
FUNCTION CHECK_VERSION AS INTEGER
IF VERSION >= 10100000
   CHECK_VERSION = 1
ELSE
   PRINT "This program requires GSoft BASIC 1.1.0 or later."
   INPUT "Press the RETURN key to continue."; A$
   CHECK_VERSION = 0
END IF
END FUNCTION
```

Here is a short program that changes the version number into a string.

217

```
PRINT VERSION$

FUNCTION VERSION$ AS STRING
DIM V AS LONG
V$ = STR$ ( CLNG ( VERSION / 100000000))
V = VERSION - CLNG ( VERSION / 100000000) * 100000000
V$ = V$ + "." + STR$ ( CLNG (V / 100000))
V = V - CLNG (V / 100000) * 100000
V$ = V$ + "." + STR$ ( CLNG (V / 1000))
V = V - CLNG (V / 1000) * 1000
IF V <> 0 THEN
    SELECT CASE CLNG (V / 100)
        CASE 1:V$ = V$ + " D"
        CASE 2:V$ = V$ + " A"
        CASE 3:V$ = V$ + " B"
    END SELECT
    V$ = V$ + STR$ (V - CLNG (V / 100) * 100)
END IF
VERSION$ = V$
END FUNCTION
```

Chapter 17 – Subroutines

GOSUB Subroutines

Subroutines based on GOSUB are simple to implement and easy to understand. They are also a part of virtually every implementation of BASIC. GOSUB statements do require the use of line numbers, the line number doesn't tell you as much as a SUB name about what a call does, and in long programs the time required to find the line number can significantly slow down a program that calls lots of subroutines. For these reasons, SUB is generally a better way to handle subroutines in GSoft BASIC.

GOSUB line-number

Control jumps to the first line whose line number matches `line-number`. `line-number` must be an integer constant. When a RETURN statement is encountered, control jumps to the statement after GOSUB. The following program illustrates this by printing 1, 2 and 3 using subroutine calls.

```
       I = 1
       GOSUB 10
       I = I + 1
       GOSUB 10
       I = I + 1
       GOSUB 10
       END

10 PRINT I
   END
```

Subroutines can be nested up to 24 levels deep. Recursion is allowed so long as this limit is not exceeded. Here's a simple example of a recursive subroutine that calculates a positive integer exponent. The maximum exponent that can be used is 23, as shown, since adding 1 to the exponent would cause a 25th subroutine call.

```
       X = 3
       R = 1
       E = 23
       GOSUB 10
       PRINT R
       END
```

219

```
10 IF E = 0 THEN RETURN
   E = E - 1
   R = R * X
   GOSUB 10
   RETURN
```

The variables used in the subroutine are identical to the variables used in the rest of the program, so in

```
     I = 4
     GOSUB 10
     PRINT I
     END
10   I = 5
     RETURN
```

the value printed is 5, not 4.

If GOSUB is used in a subroutine or function, the destination line must be in the same procedure. If GOSUB is used in the main program, the destination line must also be in the main program.

ON expression GOSUB line-number [',' line-number]*

The ON-GOSUB statement is similar to the ON-GOTO statement. It uses an index to jump to one of several locations in a program.

The expression is evaluated, then truncated to an integer. Counting from one, one of the line numbers is selected from the list of line numbers immediately after GOSUB, and the program does a GOSUB call to that line. If there are no matching line numbers, execution continues with the line after the ON-GOSUB statement.

Just as with the GOSUB statement, RETURN is used to return from the subroutine call. Control continues with the statement immediately after the ON-GOSUB statement.

Snippet

```
     FOR I = 1 TO 3
        ON I GOSUB 10, 20, 30
     NEXT
     END
10   PRINT "one"
     RETURN
20   PRINT "two"
     RETURN
30   PRINT "three"
     RETURN
```

POP

Removes one GOSUB return address from the stack. In effect, this turns the most recent GOSUB into a GOTO.

RETURN

Returns from the most recent GOSUB or ON-GOSUB, transferring control to the statement following the GOSUB statement.

See GOSUB for examples of RETURN.

DEF FN Functions

DEF FN identifier '(' identifier [',' identifier]* ')' '=' expression

Creates a local function. The function is called using FN followed by the function name and a parameter list.

The function definition must be encountered before the first time it is used, so

```
DEF FN SQUARE(X) = X * X
PRINT FN SQUARE(2)
```

works fine, printing 4, but

```
PRINT FN SQUARE(2)
DEF FN SQUARE(X) = X * X
```

fails.

Parameters and the value returned by the function can be any numeric or string type. Types are assigned using trailing type characters, as in A$ for a string. For example,

```
DEF FN PATH$(PREFIX$, FILE$) = PREFIX$ + ":" + FILE$
PRINT FN PATH$("mydisk:myfolder", "myfile")
```

prints the full path name

```
:mydisk:myfolder:myfile
```

When the function is called using a FN term in an expression, each parameter in the call is evaluated and assigned to the corresponding parameter variable. The expression is then evaluated. The expression must result in a value that is type compatible with the function name. The

expression can use constants, parameter variables, other variables that do not have the same name as a parameter, and other functions—but recursive calls are not allowed.

If the function uses variables that are not parameters, the variables are shared with the program or procedure containing the function. For example

```
DEF FN F(X) = X * Y
Y = 4
X = 5
PRINT FN F(2), X
```

prints 8, then 5. The value of Y used by the function comes from the main program, and is set to 4. The value of the parameter X is 2, set when the function is called by the PRINT statement; the parameter X is completely different from the variable X in the program.

Functions created with DEF FN are local to the main program or procedure in which they are created.

You can redefine a DEF FN function; the latest definition is the one used. Thus

```
DEF FN F(X) = X * X
DEF FN F(X) = X * X * X
PRINT FN F(2)
```

is legal, and prints 8. The first function is replaced by the second. A more common use for this feature is to define the function based on some input parameter. For example,

```
IF SQUARE THEN
    DEF FN F(X) = X * X
ELSE
    DEF FN F(X) = X * X * X
END IF
```

While this short example is a bit contrived, practical examples are not difficult to come by. For example, you might create several functions for calculating interest, then choose the appropriate function using a condition like the one shown. All of the calculations in the rest of the program would be the same, and there would be no time consuming testing in the program itself to choose the correct interest calculation.

DEF FN functions are limited to a single line with no control statements. Essentially, the function must be something that could be handled with a LET statement. This disadvantage is offset somewhat by two advantages: DEF FN functions are generally faster than an equivalent function defined with the FUNCTION statement; and you can have more than one function with the same name, choosing the correct one as the program runs or even replacing a function that is in use with a new one by the same name.

Subroutines and Functions

SUB and FUNCTION Parameter Lists

Both SUB subroutines and FUNCTION functions support parameter lists. The rules for parameter lists are the same for both. In this section, subroutines and functions will be referred to as procedures, a name that encompasses both subroutines and functions.

Parameter lists follow the procedure name, enclosed in parentheses. A parameter list consists of one or more parameter declarations separated by commas. Each parameter declaration is a variable, optionally followed by AS and a type. If no type is given explicitly, the type is derived from the name of the variable.

For example, this function returns the hyperbolic sine of a value. Like all of the procedures in this section, it's shown with a simple test program, which shows how a parameter is coded when the procedure is called. The examples form very short programs, but they are complete and will run as shown, so you can type them in and try variations to explore how procedures work.

```
PRINT SINH (2)
END

FUNCTION SINH (X)
SINH = 0.5 * ( EXP (X) - EXP ( - X))
END FUNCTION
```

The parameter doesn't have an AS type clause, so the type is assumed from the variable name. Just as with a variable anywhere else in the program, a name with no trailing type character is SINGLE. For that matter, the function itself returns a SINGLE value for the same reason.

There are two ways to create a similar function that takes a DOUBLE argument and returns a DOUBLE result. The first uses type characters, like this:

```
PRINT SINH# (2)
END

FUNCTION SINH# (X#)
SINH# = 0.5D0 * ( EXP (X#) - EXP ( - X#))
END FUNCTION
```

The other method requires a bit more typing, but you don't have to type # after the name when you use the function. It looks like this:

```
PRINT SINH (2)
END
```

```
FUNCTION SINH (X AS DOUBLE) AS DOUBLE
SINH = 0.5D0 * ( EXP (X) - EXP ( - X) )
END FUNCTION
```

Arrays, records, pointers, strings and all numeric types are allowed as parameters. All of the numeric types and strings work exactly like the example of SINGLE and DOUBLE in the SINH function, above.

You can declare a pointer parameter two ways. The first is to create a type name in the main part of the program, then use the type name in the parameter list, like this:

```
TYPE IPTR AS POINTER TO INTEGER
DIM I AS INTEGER
DIM IP AS IPTR
IP = @I
I = 4
CALL TEST(IP)
END

SUB TEST(P AS IPTR)
PRINT P^
END SUB
```

The second way to declare a pointer parameter is more direct. It uses the POINTER TO type qualifier, like this:

```
DIM I AS INTEGER
DIM IP AS POINTER TO INTEGER
IP = @I
I = 4
CALL TEST(IP)
END

SUB TEST(P AS POINTER TO INTEGER)
PRINT P^
END SUB
```

Passing records is just as straight forward. In the case of a record, the type must be predefined. Types defined in the main part of the program can be used in the parameter list—or in the procedure itself, for that matter. The issue of global and local variables is discussed more completely in the section *Local Variables and Types*, below. Here's a short sample that demonstrates how records are passed.

```
TYPE POINT3D
    X:Y:Z
END TYPE
DIM P AS POINT3D
P.X = 1.2
P.Y = 3.4
P.Z = 5.6
CALL PRINTPOINT(P)
END

SUB PRINTPOINT(P AS POINT3D)
PRINT USING "(#.#  #.#  #.#)";P.X, P.Y, P.Z
END SUB
```

The only parameter type that that isn't completely straight forward is an array. There are two problems that contribute to the rather odd way arrays are passed, and one unexpected benefit. The first problem is that BASIC traditionally doesn't support types, and even GSoft BASIC doesn't support types that are an array of something. Because of this historical limitation, the designers of early implementations of BASIC had to come up with a way of passing arrays that did not depend on types. In addition, you can have an array and a non-array with the same name;

```
DIM A, A(4)
```

is perfectly legal. It creates two variables, both named A, but one is a SINGLE variable and the other is an array of five SINGLE values.

To solve these twin problems, BASIC uses parentheses immediately after the name of the array, both when the procedure is declared and when it is called. Nothing goes inside the parentheses in either case, though. Using A(4) as a parameter when you call a procedure passes the specific SINGLE value at that index, just as printing A(4) prints a specific SINGLE value from the array. Specifying the maximum length of the array in the procedure declaration would have worked, but the designers of BASIC chose not to.

All of this leads to the rather odd looking empty subscripts you see in the following example of array parameter passing, but it also leads to a powerful benefit. Since there is no maximum subscript in the procedure declaration, we don't have to limit procedure calls to arrays of a specific length, as the example shows. We can use the same procedure to handle arrays of several sizes. This example uses a single procedure to calculate the length of a vector, but the subroutine can handle vectors in two or three dimensions—or more, for that matter.

```
DIM V2(1), V3(2)
V2(0) = 3
V2(1) = 4
V3(0) = 2
V3(1) = 2
V3(2) = 2
PRINT LENGTH(V2(), 2), LENGTH(V3(), 3)
END
```

```
FUNCTION LENGTH(V(), DIMENSIONS AS INTEGER )
SQUARES = 0.0
FOR I% = 0 TO DIMENSIONS - 1
  SQUARES = SQUARES + V(I%) * V(I%)
NEXT
LENGTH = SQR (SQUARES)
END FUNCTION
```

Surprisingly, this flexibility is not accompanied by the danger of an array overflow, as it is in C. If you try to access an illegal array subscript, perhaps because you passed the wrong dimension in the example above, BASIC detects the error and stops the program.

Multiply subscripted arrays are handled exactly the same way, as the example below shows. This example computes the determinant of a matrix with two or more rows and columns using cofactor reduction. The important point here isn't whether you know what cofactor reduction is, or even what the determinant of a matrix is. The important point is that the sample shows clearly how a multi-dimensional array is passed as a procedure parameter. It also shows a clever use of BASIC's ability to handle variable dimensioned arrays, since the procedure calls itself with successively smaller arrays until the 2 by 2 case is reached. This is something that C can't do. Most Pascal implementations can't do this, either.

```
DIM I AS INTEGER , J AS INTEGER
DIM DIMENSIONS AS INTEGER
:
DIMENSIONS = 3
DIM A(DIMENSIONS - 1, DIMENSIONS - 1)
:
FOR I = 0 TO DIMENSIONS - 1
  FOR J = 0 TO DIMENSIONS - 1
    A(I, J) = 10 * I + J
  NEXT
NEXT
PRINT DETERMINANT(A(), DIMENSIONS)
END

FUNCTION DETERMINANT(A(), DIMENSIONS AS INTEGER )
IF DIMENSIONS = 2 THEN
  DETERMINANT = A(0, 0) * A(1, 1) - A(0, 1) * A(1, 0)
ELSE
  DIM B(DIMENSIONS - 1, DIMENSIONS - 1)
  DIM I AS INTEGER , J AS INTEGER
  DIM R AS INTEGER , C AS INTEGER
  :
```

```
SIGN = 1.0
SUM = 0.0
FOR I = 0 TO DIMENSIONS - 1
    R = 0
    FOR J = 0 TO DIMENSIONS - 1
        IF J <> I THEN
            FOR C = 1 TO DIMENSIONS - 1
                B(R, C - 1) = A(J, C)
            NEXT
            R = R + 1
        END IF
    NEXT
    SUM = SUM + SIGN * A(I, 0) * DETERMINANT(B(), DIMENSIONS - 1)
    SIGN =- SIGN
NEXT
DETERMINANT = SUM
END IF
END FUNCTION
```

Passing Parameters by Reference and Value

There are two fundamentally different ways to pass a parameter to a subroutine, and GSoft BASIC allows them both.

The first is called pass by reference. When you pass a parameter by reference, changes made inside the subroutine affect the original variable, too. For example, the program

```
I = 4
J = 5
CALL DOUBLE(I)
CALL DOUBLE(J)
PRINT I, J
END

SUB DOUBLE(X)
X = X + X
END SUB
```

prints 8 and 10; changes made to the variable X inside the subroutine also change the original variable. Any time you pass the name of a variable as a parameter, you are passing the parameter by reference.

The second way to pass a parameter is by value. When you pass a parameter by value, changes made inside the subroutine have no effect on the original value passed. In BASIC, all expressions, no matter how simple, are passed by value. It's traditional to enclose a variable in parentheses to pass it by value. Recoding the sample,

```
I = 4
J = 5
CALL DOUBLE((I))
CALL DOUBLE((J))
PRINT I, J
END

SUB DOUBLE(X)
X = X + X
END SUB
```

prints 4 and 5.

There is one subtle point about numeric variables passed as parameters. Forcing BASIC to convert from one type to another is an expression. If you pass, say, an integer variable to a procedure that expects a SINGLE variable, the value is converted. When this happens, the variable is passed by value, never by reference, so

```
I% = 4
J# = 5
CALL DOUBLE(I%)
CALL DOUBLE(J#)
PRINT I%, J#
END

SUB DOUBLE(X)
X = X + X
END SUB
```

prints 4 and 5.

Arrays and records must be passed by reference, since they cannot be used in an expression. Variables of all other types can be passed either way.

Using Parameters

Inside a procedure, a parameter works just like any other variable. You can use the parameters in expressions, change the value of a parameter, or pass the parameter as a parameter to yet another procedure.

Space used by parameters vanishes as soon as the procedure returns.

Local Variables and Types

Variables declared inside the procedure survive until the procedure returns, but no longer. If the procedure is called again, an entirely new set of variables is allocated. This prevents you from storing values inside a subroutine for later use. For example, the program

```
FOR I = 1 TO 10
  CALL TEST
NEXT
END

SUB TEST
  J = J + 1
  PRINT J
END SUB
```

looks, at first glance, like it might print the numbers 1 to 10. In fact, it prints 1 ten times. Every time the subroutine TEST returns to the main program, the value J vanishes; each time TEST is called again, a new variable named J is created and initialized to zero.

Variables from outside the procedure cannot be accessed from inside, although passing parameters by reference does give you a way to change values outside the procedure. The program

```
X = 5
CALL TEST
PRINT X
END

SUB TEST
  PRINT X
END SUB
```

prints 0 and 5, not 5 and 5. The variable X from the main program is not available inside the subroutine. Using X in the PRINT statement creates a new variable called X inside the subroutine and initializes it to 0. The PRINT statement prints 0. Upon return to the main program, the variable X that was created inside the subroutine vanishes, so the PRINT statement in the main program prints the original variable called X whose value is 5.

Types defined in the main program are, however, available in procedures as well as the program, as are types declared in tool interface files. This allows you to use a type declared in the main program as the type of a parameter, and to declare variables that are type compatible with the parameter inside the procedure.

Recursion with SUB and FUNCTION

Recursion is a process where a procedure calls itself. It is usually used to break a problem down into smaller pieces. BASIC supports recursion, as you can see from the many examples throughout the book, including the determinant example from *SUB and FUNCTION Parameter Lists*, earlier in this chapter.

The only limit on recursion depth is the memory available in the variables buffer. Each time you call a procedure, some memory is used to store various values, like the location of the line to return to after the procedure completes. Space is also used for parameters and local variables. If you

try to call a procedure and there isn't enough memory left to store these values, your program will stop with an out of memory error.

You can use FRE to check the available memory in a recursive subroutine, switching to a non-recursive substitute or handling the error in some other way if memory runs low.

CALL identifier [parameter-list]

Calls a subroutine defined by a SUB statement. See SUB for details, or *SUB and FUNCTION Parameter Lists* for several examples.

FUNCTION identifier [parameter-definition-list] [As type]
[statement]*
END FUNCTION

Defines a function. The FUNCTION definition appears after the BASIC program, mixed with any SUB definition and other FUNCTION definition in any order. The program must not execute a FUNCTION statement, so the program itself should end with an END statement. For example, the program

```
PRINT SQUARE (2)
END

FUNCTION SQUARE (X)
SQUARE = X * X
END FUNCTION
```

works just fine, but leaving the END statement out would cause an error. The program would run, and it would still print 4, but right after the PRINT statement the program would try to execute the FUNCTION declaration, and that would cause an error.

The identifier is the name of the function. This is followed by the parameter list, if any, and the type returned by the function. The statements that appear between the FUNCTION statement and the END FUNCTION statement are executed as if they were a program, then the last value set for the function is returned to the caller.

The parameter list appears after the function name in parentheses. Parameter lists for FUNCTION and SUB procedures follow the same rules. These rules are discussed in *SUB and FUNCTION Parameter Lists*, earlier in this chapter.

Last is the type of the function, coded as AS followed by a type name. A function can return any simple type, such as LONG or STRING; and it can return a pointer. Functions cannot return records or arrays, although they can return pointers to either a record or an array. If no type is given, the type is assumed from the function name, just as it is for a variable name. Following these rules,

```
FUNCTION F (X)
```

is a function that returns a SINGLE value.

```
FUNCTION F% (X)
```

and

```
FUNCTION F (X) AS INTEGER
```

both return an INTEGER value.

The value returned by the function is set by assigning a value to the function name. This can be done more than one time; the last value set is the one returned. If no value is set, 0 is returned for numeric functions, a null string for strings, and a null pointer for pointers.

A function returns to the main program when the END FUNCTION statement executes.

See *Local Variables and Types* for a discussion of local variables, and *Recursion with SUB and FUNCTION* for a discussion of recursion. These sections and *SUB and FUNCTION Parameter Lists*, appear earlier in this chapter; all three have extensive examples of functions.

SUB identifier [parameter-definition-list]
[statement]*
END SUB

Defines a subroutine.

The SUB definition appears after the BASIC program, mixed with any FUNCTION definitions and other SUB definitions in any order. The program must not execute a SUB statement, so the program itself should end with an END statement. For example, the program

```
CALL HELLO
END

SUB HELLO
PRINT "Hello, world."
END SUB
```

works just fine, but leaving the END statement out would cause an error. The program would run, and it would still print "Hello, world.", but right after the CALL statement the program would try to execute the SUB declaration, and that would cause an error.

The identifier is the name of the subroutine, used when it is called. This is followed by the parameter list, if any. The statements that appear between the SUB statement and the END SUB statement are executed as if they were a program.

The parameter list appears after the subroutine name in parentheses. Parameter lists for FUNCTION and SUB procedures follow the same rules. These rules are discussed in *SUB and FUNCTION Parameter Lists*, earlier in this chapter.

Subroutines are called with the CALL statement. This is followed by the name of the subroutine and any parameters.

A subroutine returns to the main program when the END SUB statement executes.

See *Local Variables and Types* for a discussion of local variables, and *Recursion with SUB and FUNCTION* for a discussion of recursion. These sections and *SUB and FUNCTION Parameter Lists*, appear earlier in this chapter; all three have extensive examples of subroutines.

Chapter 18 – Standard Libraries

When GSoft BASIC is properly installed, the commands you see in this chapter work just like the commands built into the GSoft BASIC language, yet these commands are not a part of GSoft BASIC. They are part of the standard libraries package that we expect to implement on all platforms that have the hardware necessary to support these features.

These libraries are installed for you by the installer that ships with GSoft BASIC. See *Installing GSoft BASIC on a Hard Disk* in Chapter 2 for more information about the installer. More in-depth information about how libraries are constructed and installed can be found in Appendix D, *Writing User Tools for GSoft BASIC*. For the most part, the only detail you need to be aware of is that the libraries require user tools that are not built into the GSoft BASIC language, so if your programs will be used by people who do not have GSoft BASIC, you need to remember to include the user tools with your program. The Byte Works, Inc. grants a royalty free license to include these tools with any program written in GSoft BASIC. See *Including Libraries with GSoft BASIC Programs* in Chapter 4 for details.

The Game Paddle Library

Libraries in GSoft BASIC are numbered. The Game Paddle Library is library number 1.

One capability that is built into Applesoft BASIC but not GSoft BASIC is a command to read joysticks and game paddles connected to the game paddle port. One of the problems with the game paddle port is that subroutines have to be fine tuned for the processor speed you are actually using. For that reason, commands to read the game paddle port are not built into the language itself.

The Game Paddle Library does more than read the paddle values like Applesoft BASIC's PDL command, though. It also adds commands to read the four TTL switches on the game paddle port, as well as the ability to set and clear the four TTL annunciators. You can also do this with PEEK and POKE commands, but the Game Paddle Library gives you a cleaner interface.

Refer to the *Apple IIGS Hardware Reference* for information about the game paddle port itself, including pinout diagrams that you can use to build your own TTL and resistor based devices to connect to the game paddle port.

SUB GTBootInit

This call is made at boot time. It doesn't do anything in this library.

SUB GTStartup

You should make this call immediately after loading the library with LOADLIBRARY. GTStartup does any required initialization. It doesn't do anything in the Game Paddle Library, but it's a good idea to make the startup call for any library you load.

233

SUB GTShutDown

You should make this call just before you use UNLOADLIBRARY to unload the Game Paddle Library. GTShutDown does any required clean up. This call doesn't do anything in the Game Paddle Library, but it's a good idea to make this call when you are finished with any library.

FUNCTION GTVersion As INTEGER

Returns the version number for the library. The most significant byte is the major version, and the least significant byte is the minor version. You can print the version number in the common form of major.minor this way:

```
VERSION% = GTVERSION
MAJOR% = VERSION% / 256
MINOR% = VERSION% - MAJOR% * 256
PRINT USING "The Game Paddle Library version number is #.#."; MAJOR%, MINOR%
```

FUNCTION GTStatus As INTEGER

Returns true (a value of 1) if the Game Paddle Library has been started with GTStartup and has not yet been shut down, or false (a value of 0) if the Game Paddle Library is not started.

FUNCTION GTGetSwitch (SWITCH As INTEGER) As INTEGER

Returns the current setting of one of the four TTL input switches. The switches are numbered 0 to 3.

If SWITCH is outside the range 0 to 3, GTGetSwitch sets TOOLERROR to $0101.

SUB GTClearAnnunciator (ANNUNCIATOR As INTEGER)

Clears (turns off) one of the four TTL output annunciators. The annunciators are numbered 0 to 3.

If ANNUNCIATOR is outside the range 0 to 3, GTClearAnnunciator sets TOOLERROR to $0101.

SUB GTSetAnnunciator (ANNUNCIATOR As INTEGER)

Sets (turns on) one of the four TTL output annunciators. The annunciators are numbered 0 to 3.

If ANNUNCIATOR is outside the range 0 to 3, GTSetAnnunciator sets TOOLERROR to $0101.

FUNCTION GTGetPaddle (PADDLE As INTEGER) As INTEGER

Reads one of the four resistor inputs to the game paddle port. The resistor inputs are numbered 0 to 3. A single joystick uses paddle 0 for the X axis and paddle 1 for the Y axis; a system that

supports two joysticks will use paddles 2 and 3 for the X and Y axis, respectively. Game paddles are generally numbered, and also use inputs 0 and 1 for the standard configuration of two paddles.

GTGetPaddle returns a value from 0 to 255, depending on the resistance across the port. Game paddles generally return lower values when turned counterclockwise and higher values when turned clockwise; joysticks generally return lower values when pushed up or left, and higher values when pushed right or down.

The routines used to read the game paddle port are sensitive to the CPU speed of the computer. The Game Paddle Library is timed for an unaccelerated Apple IIGS running at fast speed (about 2.7 MHz). If you are using an accelerator card, the range of values reported by GTGetPaddle may be too low. Check your accelerator card—some have a configurable switch that slows down momentarily when the game paddle ports are read, allowing timed routines to work correctly. Make sure this setting is enabled.

If PADDLE is outside the range 0 to 3, GTGetPaddle sets TOOLERROR to $0101.

Using the Game Paddle Library

Here's a short program that shows how to use the Game Paddle Library to read the positions on a joystick or a pair of game paddles. It shows the proper way to initialize and shut down the library.

```
LOADLIBRARY 1
GTSTARTUP
FOR I = 1 TO 300
    PRINT GTGETPADDLE (0), GTGETPADDLE (1)
NEXT
GTSHUTDOWN
UNLOADLIBRARY 1
```

The Time Library

Libraries in GSoft BASIC are numbered. The Time Library is library number 2. The Time Library provides commands that tell you the current time and date.

SUB `TTBootInit`

This call is made at boot time. It doesn't do anything in this library.

SUB `TTStartup`

You should make this call immediately after loading the library with LOADLIBRARY. TTStartup does any required initialization. It doesn't do anything in the Time Library, but it's a good idea to make the startup call for any library you load.

235

SUB `TTShutDown`

You should make this call just before you use UNLOADLIBRARY to unload the Time Library. TTShutDown does any required clean up. This call doesn't do anything in the Time Library, but it's a good idea to make this call when you are finished with any library.

FUNCTION `TTVersion` AS `INTEGER`

Returns the version number for the library. The most significant byte is the major version, and the least significant byte is the minor version. You can print the version number in the common form of major.minor this way:

```
VERSION% = TTVERSION
MAJOR% = VERSION% / 256
MINOR% = VERSION% - MAJOR% * 256
PRINT USING "The Time Library version number is #.#."; MAJOR%, MINOR%
```

FUNCTION `TTStatus` AS `INTEGER`

Returns true (a value of 1) if the Time Library has been started with TTStartup and has not yet been shut down, or false (a value of 0) if the Time Library is not started.

FUNCTION `Datestring` AS `STRING`

Returns the current date as a string.

The string contains the full month name, the date, a comma, and the four digit year. For example, the command

```
PRINT DATESTRING
```

would print

```
July 22, 1998
```

if the date was, in fact, set to July 22, 1998 on the computer's clock.

There has been a lot of attention in the press about the Year 2000 bug (also known as the Y2K bug), when the computer world is apparently scheduled to end. In general, this bug does not exist on the Apple IIGS, although certain applications may contain the bug. While the Apple IIGS clock is limited to a two digit year, the operating system starts the cycle at 1940, returning dates from January 1, 1940 to December 30, 2039. Of course, this means there is a Year 2040 bug on the Apple IIGS, but at this point we can only hope we're around to care!

FUNCTION `Timestring` AS `STRING`

Returns the current time as a string.

The time is returned as a two-digit hour, a colon, a two digit minute, a colon, and a two digit
second. For example, at half past noon, TIMESTRING would return "12:30:00".

The hour is always formatted with two digits, even if they are zero. For example, 9:00 AM is
"09:00:00", and 15 minutes past midnight is "00:15:00". The hour is also returned in 24 hour
format, so 1 PM is "13:00:00".

SUB   **`Time   ((timeRecord))`**

Returns the current date and time in a time record. The record, declared in TimeTool.gst, looks
like this:

```
type timeRecord
    year as integer         ; -32768 to 32767; always 1940 to 2039
    month as integer        ; 1 to 12
    day as integer          ; 1 to 31
    hour as integer         ; 0 to 23
    minute as integer       ; 0 to 59
    second as integer       ; 0 to 59
    millisecond as integer  ; 0 to 999; always 0
end type
```

When you call TIME, you pass a variable declared as TIMERECORD as the parameter. The
record is filled in with the current date and time, as set on the computer's built-in clock.

For the most part, these fields are self-explanatory. Only two deserve special comment.

The year can range over the entire valid range of integers, but this command will never return
a value outside of the range 1940 to 2039. As discussed in the description of the DATESTRING
function, this means there is no Year 2000 bug. The wide range of years is intended for future
expansion, where other time commands that manipulate dates might return values the computer's
clock cannot.

The millisecond field is not used on the Apple IIGS, since the Apple IIGS clock doesn't
provide the time to that level of accuracy. The field will always be set to 0. It is included for use
by more accurate clocks and for use on other computers that GSoft BASIC might be ported to.

Using the Time Library

Here's a short program that shows how to use the Time Library to read the date and time. It shows the proper way to initialize and shut down the library.

```
LOADLIBRARY 2
DIM T AS TIMERECORD
PRINT "The current time is "; TIMESTRING
PRINT "The date is "; DATESTRING
TIME (T)
PRINT
PRINT "The year is        ";T.YEAR
PRINT "The month number is ";T.MONTH
PRINT "The date is        ";T.DAY
PRINT "The hour is        ";T.HOUR
PRINT "The minute is      ";T.MINUTE
PRINT "The second is      ";T.SECOND
PRINT "The millisecond is  ";T.MILLISECOND
UNLOADLIBRARY 2
```

Chapter 19 – Tool Interface

Thanks to its support for records and pointers, GSoft BASIC is the first BASIC to offer support for the toolbox that is as natural and complete as the support in other languages, like C, Pascal and Modula-2.

This chapter describes how the toolbox interface works and how to interpret Apple's toolbox documentation, written for C and assembly language, for use with GSoft BASIC. It also covers calls to GS/OS, the disk operating system of the Apple IIGS.

This chapter does not document the toolbox or GS/OS, of course. Apple's documentation of the Apple IIGS toolbox and GS/OS is in five volumes, totaling over 3500 pages. These books are available as reprints from the Byte Works. The complete set includes these books:

Apple IIGS Toolbox Reference Volume 1
Apple IIGS Toolbox Reference Volume 2
Apple IIGS Toolbox Reference Volume 3
Programmer's Reference for System 6.0.1
Apple IIGS GS/OS Reference

This chapter also doesn't teach toolbox programming, and neither do the reference books mentioned above. We offer a self-study course in toolbox and GS/OS programming:

Toolbox Programming in GSoft BASIC

This introduction to toolbox programming contains an appendix with an abridged version of the toolbox and GS/OS reference manuals. You do not need to buy the five reference books to use this course.

The Toolbox Interface

Using the Toolbox

GSoft BASIC makes use of one or more compiled toolbox interface files to handle tool calls. It comes with a tool interface file that handles the Apple IIGS toolbox, GS/OS, the ORCA Shell and Talking Tools. From your standpoint of writing a program, all of these interfaces work as if they were built right into the GSoft BASIC language.

As a quick example to show how this works, and give a basis for the rest of the chapter, here's a short program that makes use of QuickDraw II calls. Most toolbox programs create

239

programs that use the Apple desktop interface, with menu bars, windows, and so forth. In the interest of brevity, this example doesn't. It makes use of the fact that GSoft BASIC's HGR statement starts QuickDraw II to create a very short program.

```
HGR
DIM R AS RECT
FOR I = 1 TO 1000
  R.H1 = RND (1) * 320
  R.H2 = RND (1) * 320
  IF R.H2 < R.H1 THEN
    T% = R.H1
    R.H1 = R.H2
    R.H2 = T%
  END IF
  R.V1 = RND (1) * 200
  R.V2 = RND (1) * 200
  IF R.V2 < R.V1 THEN
    T% = R.V1
    R.V1 = R.V2
    R.V2 = T%
  END IF
  SET640COLOR ( RND (1) * 16)
  PAINTOVAL (R)
NEXT
GET A$
```

There are three items in this program that come from the toolbox interface files rather than from GSoft BASIC. The first is the RECT type. Rectangles are used extensively throughout the toolbox for everything from window sizes to ovals—which is how they are used here.

Near the end of the program you'll see two calls to QuickDraw II. The first is SET640COLOR; it sets the pen color to one of the 16 available colors. In this program it sets them randomly. The next is PAINTOVAL, which paints an oval that's inside the RECT parameter.

As a whole, the program draws 1000 random size ovals at random locations using random colors. When it finishes, it waits for you to press the return key, then exits.

We'll refer back to this program later in this chapter.

The GSoft BASIC Toolbox Interface

There is no need to include the tool interface files, as in other languages. Instead, GSoft BASIC loads the tool interface file automatically as it starts.

Of course, to do that, it needs to know where the tool interface file is located. GSoft BASIC searches for tool interface files in three locations. The names of these prefixes, in the order they are searched, are:

```
13:GSoftDefs:
9:
8:
```

The first prefix is the ORCA library prefix. This is the natural place to put the tool interface files if you are using the shell version of GSoft BASIC, running it as a language from the ORCA shell.

Prefix 9: is set to the prefix of the executing application, in this case either the ORCA shell for the shell version of GSoft BASIC, or GSoft.Sys16 for the Finder version. This is the natural place to put the tool interface files for the Finder version of GSoft BASIC.

The last prefix searched is 8:, which is the default prefix when the GSoft BASIC program starts to execute. You can put compiled tool interface files here for user tools used only with a specific program you are developing.

For each directory, GSoft BASIC searches all of the files looking for tool interface files. Tool interface files are created with CompileTool, documented in Chapter 4. Each tool interface file has a file type of $5E and an auxiliary file type of $8007. The files are loaded in the same order shown by the CATALOG command.

An important point about this process is that duplicate tool names and records can exist. When they do, the most recently loaded definition is used. This means you can replace existing calls or records with new versions by putting the declarations in your own tool file, compiling the tool file, and placing the compiled tool file in the same directory as your program. The newer declaration overrides the original declaration, but the original tool interface file is left unchanged, so it still works with other programs that expect the unmodified tool interface file.

The tool interface file is compiled into a form that takes little space and loads quickly, but this form can't be read. The text version of the standard tool interface file is called GSoftTools.int. If you are using the Finder version of GSoft BASIC, you can find this file in the same folder as GSoft.Sys16; for the shell version, look in the ORCA Libraries folder.

Using Apple's Documentation

There were only two Apple IIGS programming languages available when Apple's toolbox reference manuals were written: The APW Assembler (also released under the name ORCA/M) and APW C. Several programming languages were eventually released for the Apple IIGS, but a toolbox reference manual was not released for each language. That's because, with a little practice, you can read Apple's documentation and use it from any language.

There are two sources of information about how to make a particular tool call. The first is the toolbox reference manuals themselves. Looking at the documentation for PaintOval in *Apple IIGS Toolbox Reference Volume* 2, page 16-189, you see that this call has a single parameter. You can see this either from the stack diagram used by assembly language programmers, or from the example for C programmers at the bottom of the page.

The assembly language documentation is generally the easiest to read, since it is presented as a diagram. Still looking at PaintOval, you see that the parameter is labeled "POINTER to RECT defining enclosing rectangle." This tells you the call expects a pointer to a RECT for its

parameter. This illustrates the one big difference between pretty much every language around and the two languages used in the toolbox reference manual, assembly language and C. In most other languages, including BASIC, it makes more sense to pass a RECT as a reference parameter than as a pointer. (Value and reference parameters are covered in *Passing Parameters by Reference and Value* in Chapter 17.) As far as the toolbox is concerned, there's no difference: it gets a pointer to a RECT either way, since BASIC and other languages implement reference parameters by passing a pointer; the difference is entirely in how you write the program.

The second major source of information about making tool calls is the GSoft BASIC tool interface file, GSoftTools.int. Looking in that file, you see this declaration for PaintOval:

```
TOOL $04, $59 SUB    PaintOval((Rect))
```

The parentheses around the parameter type tell you the parameter is passed by reference, and parameters passed to the toolbox by reference are always passed as a pointer. You can ignore the information before SUB; that's the tool number and tool call number, needed by GSoft BASIC but not by you when you are writing BASIC programs.

Putting these facts together, you can see why the call to PaintOval in the sample program at the start of this chapter looks like

```
PAINTOVAL (R)
```

R is a RECT record variable. Since the tool interface file tells us the parameter is passed by reference, we don't need to pass it as a pointer.

With a little experience, you'll be able to predict whether a pointer parameter is passed by reference, like this example, or by value, in which case you need to pass a pointer. As a general rule, parameters are passed by reference when the value is a record you fill in, and they are passed by value when the value is likely to be a pointer passed to you by the toolbox itself. For example, window pointers are passed by value, so you actually pass a pointer. The original window pointer was created by the Window Manager and passed to you as a pointer—it's not a record you should, or even can, create and fill in yourself.

Many tool calls use more than one parameter. PaintArc is one such example; it's on the page right before PaintOval in the toolbox reference manual. The three parameters are listed top to bottom; that's the order you code them in the parameter list. The toolbox documentation tells you the parameters are a pointer to a RECT and two integers. The GSoftTools.int file shows the call like this:

```
TOOL $04, $63 SUB    PaintArc((Rect), %, %)
```

The first parameter is a RECT, just like it was for PaintOval. The next two parameters are INTEGER values. A typical call would look like this:

```
PAINTARC (R, 45, 90)
```

Some tool calls return a value. One of the many examples is TextWidth, a QuickDraw II call that returns the width of a text string in pixels. This width takes the current graphical font into account. Looking on page 16-270 of *Apple IIGS Toolbox Reference Volume 2*, you see three values shown on the stack before the call, and one after. You will only see a value on the stack after the tool call if the tool is a FUNCTION rather than a SUB; the value left on the stack is the value the tool call returns. This value is also on the stack before the call, but it is not a parameter. You can see this easily by looking at the declaration in GSoftTools.int:

```
TOOL $04, $AB FUNCTION TextWidth(Univ, %)
```

which clearly shows that TextWidth is a FUNCTION that takes two parameters.

The first parameter is a special case. Univ parameters are used when the toolbox call can accept several different types of values, or when there is no clear correlation between the value the toolbox expects and the types available in GSoft BASIC. In this case, the toolbox expects a pointer to a sequence of characters. It turns out you can pass a pointer to a string for this parameter, since GSoft BASIC strings are also a sequence of characters. The second parameter is the number of characters in the string. A typical call looks like

```
S$ = "Hello, toolbox."
WIDTH% = TEXTWIDTH (@S$, LEN(S$))
```

Some tool calls expect C strings or P strings, named after C and Pascal. C strings are sequences of characters ending with a null character, which is the character CHR$(0). This is the kind of string GSoft BASIC uses. P strings start with one byte that tells how long the string is, followed by the characters in the string. Since a byte can only hold values from 0 to 255, this limits the length of a P string to 255 characters. Fortunately, it's pretty easy to add the length byte to the front of a GSoft BASIC string to fool the toolbox. The line

```
S$ = CHR$ (LEN (S$)) + S$
```

adds the required length byte to a string.

One dirty little secret of the toolbox is that some calls were not designed for use from a high-level language. These calls return multiple values or return values that are not 2 or 4 bytes long. The vast majority of these calls are concentrated in the Integer Math Tool Set, SANE, and the Loader. Fortunately, integer math is built right into GSoft BASIC; and SANE, Apple's floating-point libraries, aren't used at all. GSoft BASIC's floating-point routines are much faster than Apple's. If you need to make any of these calls, though, you'll have to do it using a user tool that is written in assembly language and converts the toolbox parameters into something meaningful in GSoft BASIC.

GS/OS and the ORCA Shell Calls

GS/OS and ORCA Shell calls are handled the same way as tool calls. True, they are not tools, but the interfaces are created the same way and the calls are made the same way. As a GSoft BASIC programmer, Apple's system tools, your user tools, GS/OS and the ORCA Shell calls all work exactly the same way.

The Role of User Tools

GSoft BASIC is interpreted, so it can't actually call or be called from other Apple IIGS languages. On the other hand, it supports user written tools in a way no other Apple IIGS language does, providing simple calls to load and unload user tools. If you need to write a few subroutines in assembly language, you can package them in a user tool and call them from GSoft BASIC. Even better, the same subroutines can be used from any other Apple IIGS language that supports user tools. All of the ORCA languages do.

See *Loading and Unloading Libraries*, later in this chapter, for the commands used to set up user tools. Appendix D, *Writing User Tools for GSoft BASIC*, shows how the tool is actually created. Once created and loaded, you use a user tool just like a tool created by Apple.

Tool and GS/OS Errors

TOOLERROR

Returns the error code from the most recent tool, user tool or GS/OS call. A value of zero indicates there was no error.

Apple's tools, user tools, GS/OS and ORCA Shell calls all return an error code. The error code is returned by each and every call. From the time the call completes until the next call is made you can use TOOLERROR to find out what the error code was.

Zero is universally used to indicate that no error occurred. If an error does occur, the error code tells you two things. First, dividing the error by 256 and converting the result to an integer by truncation tells you the tool number for the tool that flagged the error. For example,

```
TOOLNUMBER% = TOOLERROR / 256
```

sets the INTEGER variable TOOLNUMBER% to the number of the tool that flagged the error. The tool that flagged the error might seem obvious, since TOOLERROR is used right after a tool call, but tool errors don't always originate with the tool you call. For example, it's quite possible to get error 513 ($0201), which is a Memory Manager error indicating insufficient memory, from a NEWWINDOW call, which is a Window Manager tool call.

GS/OS errors are reported just like tool errors. The tool number for a GS/OS error will be 0.

User tool errors are also reported the same way as Apple's toolbox errors. The tool number will match the user tool number, which could duplicate an Apple tool number.

The error numbers themselves are documented with the tools. You can find a summary of all of the error numbers used by Apple's toolbox and GS/OS calls in Appendix E of *Programmer's Reference for System 6.0.*

Loading and Unloading Libraries

`LOADLIBRARY` expression

Loads a user tool from disk.

The command name refers to a user tool as a library for compatibility with other versions of GSoft BASIC that may someday exist on platforms other than the Apple IIGS. The generic term "library" makes sense across all platforms, although libraries are always implemented as user tools in the Apple IIGS version of GSoft BASIC.

expression is the tool number to load. User tools are numbered 0 to 255; this number is coded as part of the file name and is used in the interface file.

GSoft BASIC looks for a file with the name UserToolXXX, where XXX is the tool number. It looks first in the local directory, which defaults to the location of the GSoft BASIC interpreter. The local directory can be changed before using LOADLIBRARY using the CHDIR command. If the tool is not found in the local directory, GSoft BASIC looks in the System directory using the path *:System:Tools:UserToolXXX.

See Appendix D for more information about writing user tools.

See also UNLOADLIBRARY.

`UNLOADLIBRARY` expression

Unloads the specified user tool, freeing the RAM used by the tool.

The command name refers to a user tool as a library for compatibility with other versions of GSoft BASIC that may someday exist on platforms other than the Apple IIGS. The generic term "library" makes sense across all platforms, although libraries are always implemented as user tools in the Apple IIGS version of GSoft BASIC.

See also LOADLIBRARY.

TOOL and GSOS Tokens

`GSOS`

When you use a GS/OS call in a GSoft BASIC program, the GS/OS call name is converted into a three byte sequence. These bytes are the GSOS token, which marks the start of the call

sequence; and a two byte call number, coded as two individual bytes, least significant byte first. These three bytes are followed by the parameter list, if any.

Except for the substitution of the word GSOS for TOOL, these tokens are used in exactly the same way as TOOL tokens, described below.

TOOL

When you use a tool call in a GSoft BASIC program, the tool name is converted into a three byte sequence. These bytes are the TOOL token, which marks the start of the tool call sequence; the tool number; and the tool call number. These three bytes are followed by the parameter list, if any.

If the tool interface file is available when you edit or list the program, GSoft BASIC automatically looks the name up based on the tool and tool call numbers, displaying the name of the tool call. If the tool call file is missing for some reason, the tool call is listed like this:

```
TOOL<$04,$DB>
```

The first number is the tool number, while the second is the tool call number. This particular token is for tool 4, QuickDraw II; and tool call $DB, Set640Color.

You can type tool calls using this format, too. This feature actually exists so you can edit a file without the tool interface file, something that might happen if you edit a program that uses a user tool you don't have installed. GSoft BASIC needs to be able to read a token in the same form so it is possible to convert the program back to its tokenized form. TOOL tokens must be typed exactly as shown, though. No spaces can be inserted, and two digit hexadecimal numbers must be used for the tool number and tool call number.

Fortunately, this isn't a big issue, since you'll normally have tool interface files for any tools you use, and with the tool interface file installed, you can use the tool call name rather than the numbers.

LIBRARY

When you use a tool call in a GSoft BASIC program, the user tool name is converted into a three byte sequence. These bytes are the LIBRARY token, which marks the start of the tool call sequence; the user tool number; and the user tool call number. These three bytes are followed by the parameter list, if any.

Except for the substitution of the word LIBRARY for TOOL, these tokens are used in exactly the same way as TOOL tokens, described above.

Appendix A – Error Messages

This appendix shows all of the errors that GSoft BASIC can generate. In starts with a table that shows the error number and error message. The error number shown is the error number returned by the ERR function.

Next is an expanded description of the short text error message printed by GSoft BASIC, along with common causes for the error.

Error	Message
0	Undefined statement
1 *	Unimplemented command
2	Syntax error
3	The line is too long
4	Line numbers must range from 1 to 65535
5	The program is too long
6	Expression too complex
7	Out of variable table space
8	Type mismatch
9	Not enough subscripts
10	Too many subscripts
11	Invalid subscript
12	Redimensioned variable error
13	The size of a value exceeded 32767 bytes
14	More than 10 nested control statements
15	NEXT without FOR
16	Too many nested subroutine calls
17	RETURN without GOSUB
18	Expected an integer
19	Integer overflow
20	GOSUB without RETURN
21	FOR without NEXT
22	POP without GOSUB
23	Invalid function parameter
24	A string value exceeded 32767 characters
25	Illegal quantity error
26 *	Math result is inexact
27	RENUMBER overlaps old and new lines
28	Expected a string
29	INPUT or DATA is too long

247

30 INPUT or DATA contains an empty or misformed number
31 Out of data
32 Could not start QuickDraw
33 WHILE without WEND
34 WEND without WHILE
35 ELSE without IF
36 Block IF without END IF
37 END IF without IF
38 Multiple ELSE clause
39 DO without LOOP
40 LOOP without DO
41 Missing CASE
42 SELECT CASE without END SELECT
43 END SELECT without SELECT CASE
44 CASE without SELECT CASE
45 PRINT USING format string has no format models
46 PRINT USING format string has \ without closing \
47 PRINT USING format string has more than 24 characters in a number model
48 Undefined function
49 Incorrect number of parameters
50 ASCII file could not be written
51 ASCII file could not be read
52 File numbers must be in the range 1 to 32767
53 Unopened file referenced
54 Too many open files
55 The file is already open
56 File I/O error
57 Out of memory
58 Output attempted to a file opened for input
59 Input attempted to a file opened for output
60 Illegal file name
61 Undefined subroutine
62 Inappropriate END SUB or END FUNCTION
63 Parameter type mismatch
64 SETMEM and CLEAR cannot be used in a procedure
65 Duplicate field
66 Redefined type
67 Expected type
68 Illegal use of a nil pointer
69 Undefined field
70 Record used inside itself
71 Functions cannot return records

72   Unknown tool
73   Could not load the library
74   Hexadecimal numbers cannot exceed $FFFFFFFF
75   Misuse of a constant
76   Misformed tool or GS/OS token
77   HGR must be used before graphics commands
78   LEN is required for RANDOM files

*   Commands marked with an asterisk can be generated by the ERROR command, but will not be generated by GSoft BASIC. These errors are used in specialized, non-commercial versions of the interpreter.

A string value exceeded 32767 characters

Strings are limited to 32767 characters; an operation was attempted that would have created a longer string.

Change the program so all strings are smaller than 32767 characters.

ASCII file could not be read

After using the EDIT command and an editor to make changes to your BASIC program, the editor writes the edited program as an ASCII file. When the editor returns control to GSoft BASIC, GSoft BASIC tries to read that file. This error indicates that the file could not be read.

The most likely reason for this error is a damaged disk, although lack of memory could cause the error. If you see this error, use a program that checks disk for bad blocks. If you have a program that will do it, check the structure of the disk, too.

ASCII file could not be written

GSoft BASIC converts the tokenized BASIC program to an ASCII file when you use a full screen editor via the EDIT command or when you debug a program using the DEBUG command. This error indicates that the file could not be written.

There are a variety of reasons why the file might not be written. The most common are a lack of space on the disk where the program is located, a disk error, or a lack of free memory.

Block IF without END IF

An IF-THEN statement was started, but no END IF was found.
Remember to include an END IF for all block IF statements. For example,

```
IF FOUND THEN CALL PROCESS
```

is a perfectly legal BASIC statement, contained entirely on one line, but

```
IF FOUND THEN
CALL PROCESS
```

needs an END IF.

CASE without SELECT CASE

A CASE statement was found outside of a SELECT CASE.

Make sure all CASE statements appear between a SELECT CASE and an END SELECT. Keep in mind that other mismatched statements might cause this error even if the CASE statement appears between a SELECT CASE and END SELECT. For example,

```
SELECT CASE COLOR
  CASE RED
    IF SHORT THEN
      PRINT "R";
    ELSE
      PRINT "RED";
  CASE GREEN
    IF SHORT THEN
      PRINT "G";
    ELSE
      PRINT "GREEN";
END SELECT
```

could cause this error, even though the real error is that the IF statement has no matching END IF.

Could not load the user tool

The LOADLIBRARY command was unable to load the user tool.

Make sure the user tool file is in the Tools folder, inside the System folder. Check to insure there is enough memory to load the tool. Check the system disk for bad blocks.

Could not start QuickDraw

The HGR statement encountered an error starting QuickDraw II. This is generally caused by lack of memory.

If you are using a program that allows more than one program to run, shut down the other programs. Try shift-booting to prevent desk accessories and inits from using memory—possibly memory critical to QuickDraw II.

DO without LOOP

A DO statement was started, but the program or procedure finished without finding a matching LOOP.

Make sure there is exactly one LOOP for each DO. Keep in mind that other mismatched statements might cause this error even if there is a matching LOOP for the DO. For example,

```
DO
    GET #1, CH
    IF CH = 13 THEN
        PRINT
LOOP WHILE NOT EOF(1)
```

would cause this error if the first value read is not 13, even though the real error is that the IF statement has no matching END IF.

Duplicate field

A record contains two fields with the same name.

Make all field names in a given record unique.

ELSE without IF

An ELSE statement was encountered when no IF statement was active.

Make sure ELSE statements are only used when an IF statement is active.

END IF without IF

An END IF was found with no matching IF-THEN statement.

Make sure IF-THEN statements have exactly one matching END IF statement. Check for other incomplete statements that might cause this error, such as

```
IF NOT DONE THEN
    WHILE P <> NIL
END IF
```

END SELECT without SELECT CASE

An END SELECT statement was found without a matching SELECT CASE statement.

Make sure there is exactly one END SELECT for each SELECT CASE. Keep in mind that other mismatched statements might cause this error even if there is a matching END SELECT for the SELECT CASE. For example,

```
SELECT CASE COLOR
CASE RED
   IF SHORT THEN
     PRINT "R";
   ELSE
     PRINT "RED";
CASE GREEN
   IF SHORT THEN
     PRINT "G";
   ELSE
     PRINT "GREEN";
END SELECT
```

could cause this error, even though the real error is that the IF statement has no matching END IF.

Expected an integer

An INTEGER value was expected, but a value that could not be converted to an INTEGER was encountered. For example, this error would occur if a string were used as an array subscript.

Change the expression so an INTEGER or a number that can be converted to an INTEGER is created. For example, you could convert a string representing a number to a number using VAL, as in

```
GET A$
PRINT A(VAL(A$))
```

Expected a string

A string value was expected, but some other type of value was encountered. For example, this error would occur if a number was used as the file name in an OPEN statement.

Change the expression so a string appears in the required position.

Expected type

A statement expected a type, such as INTEGER or the name of a record, but some other token was found.

Check for spelling errors or other typographical errors. Check the documentation for the statement involved to make sure it is entered correctly. If the statement is using a type you declared, make sure the statement that declares the type is executed before the statement that uses the type.

Expression too complex

The expression stack overflowed during evaluation of the expression. Make the expression shorter by using temporary variables.

File I/O error

GS/OS reported an error while reading or writing a file.

This can be caused by any number of reasons. The most common occur when writing a file. Problems to look for include:

- Writing to a full disk.
- Creating a new file in the root directory of a ProDOS format disk that already has 51 entries.
- A disk error. Check for bad blocks or a bad directory structure.
- A lack of free memory to open or manipulate the file.

File numbers must be in the range 1 to 32767

The programs used a file number outside the allowed range. Change the file number to a number in the allowed range.

FOR without NEXT

The program ended after starting a FOR loop, but without finishing with a NEXT.

An obvious way to make this error is forgetting a NEXT, but it can also occur if you jump out of an unfinished FOR loop. For example,

```
10 FOR I = 1 TO 10
20   GOTO 40
30 NEXT
40 END
```

is not legal, and will cause this error.

Functions cannot return records

A FUNCTION declaration has a function return type that is a record.

Functions can return pointers to records, but not records. Change the return type so it is something other than a record, or so it is a pointer to a record.

GOSUB without RETURN

The program ended after making a GOSUB call, but without a matching RETURN.

The most likely cause of this error is forgetting the RETURN at the end of a subroutine. In any case, make sure each subroutine called with GOSUB returns with a RETURN or removes the call with POP.

Hexadecimal numbers cannot exceed $FFFFFFFF

A hexadecimal constant exceeds the maximum allowed value. Change the hexadecimal value to an allowed value.

`HGR must be used before graphics commands`

A graphics command, such as HPLOT, was used before the HGR command.

Graphics commands use Apple's QuickDraw II tool set, which must be initialized before any of the graphics commands can be used. The easiest way to initialize QuickDraw II is with the HGR command, although starting the tool manually will also work, and will avoid this error.

`Illegal file name`

A file name was formed improperly.

Check the file name to make sure it conforms to the requirements for the file system in use. Keep in mind that GS/OS can use many file systems, and each has unique requirements. If you are using DIR$, check to make sure the wildcard characters used match the requirements for DIR$.

`Illegal quantity error`

A valid number that is not allowed for a particular purpose has been passed to a built-in command. For example, using a negative number for a parameter to MID$ would cause this error.

Use a correct number for the parameter.

`Illegal use of a nil pointer`

The program attempted to use the value pointed to by a pointer, but the pointer itself was set to NIL.

For example, the program

```
DIM P AS POINTER TO INTEGER
PRINT P^
```

would generate this error. The pointer has not been set to point to anything.

Give the pointer a value.

`Inappropriate END SUB or END FUNCTION`

An END SUB was found at the end of a FUNCTION or in the main program, or an END FUNCTION was found at the end of a SUB or in the main program.

Make sure exactly one END SUB appears at the end of every subroutine defined with SUB, and one END FUNCTION appears at the end of every function defined with FUNCTION. Make sure neither is used in any other way.

`Incorrect number of parameters`

A procedure call does not have the same number of parameters as the corresponding procedure declaration.

Make sure both the call and declaration use the same number of parameters.

Input attempted to a file opened for output

A command that reads from a file was used with a file number for a file opened for output only.

Make sure the file number is correct. If you need to read and write a file, open it as BINARY or RANDOM.

INPUT or DATA contains an empty or misformed number

An INPUT statement or DATA statement had a numeric variable, so it tried to read a number, but found non-numeric information.

Either the input data is incorrect or the program is incorrect. Change one or the other so INPUT and DATA get the kind of values they need.

If you need to read a value that might be a number, but you can't tell beforehand, read the value as a string. Scan what you get to see if it is a valid number. If so, you can use VAL to convert the string to a number.

INPUT or DATA is too long

A single piece of information is longer than 255 characters. Generally that will be a string that is too long in an input file, but it could be a number with lots of unnecessary leading zeros, or a number that is longer than need be because of unneeded fraction digits.

In files containing strings, consider reading the file character by character and assembling the string inside the program.

For numbers, try using scientific notation. Keep in mind that even DOUBLE numbers are only accurate to about 15 decimal digits, so any more than 16 significant digits won't change the final DOUBLE value.

Integer overflow

This error occurs due to a numeric overflow when converting to an INTEGER or LONG value, such as

```
I% = 38000.0
```

You can reduce the value or use a different kind of value. Keep in mind that GSoft BASIC does handle infinity, so you can't overflow a SINGLE or DOUBLE value.

Invalid function parameter

A type mismatch occurred in a built-in BASIC function; for example, a string may have been passed to a function that only accepts numbers.

Change the parameter so the types match. If you need to use a value that is not the correct type, consider type casting or converting the value to the correct type with functions like CINT or STR$.

Invalid subscript

A subscript is less than zero or greater than the maximum subscript for the array. For example,

```
DIM A(5)
A(6) = 6
```

would cause this error.

Change the size of the array or the subscript used.

Line numbers must range from 1 to 65535

A line number was found that lay outside the allowed range.

Change the line number so it is in the allowed range.

LEN is required for RANDOM files

A file is being opened for RANDOM input and output with the OPEN statement. RANDOM files must have a record length, specified with the LEN parameter, but no LEN parameter is present.

Add a LEN parameter to the open statement. The length should be the size of one record in the random access file.

LOOP without DO

A LOOP statement was encountered without a matching DO.

Make sure there is exactly one LOOP for each DO. Keep in mind that other mismatched statements might cause this error even if there is a DO for the LOOP statement. For example,

```
DO
  GET #1, CH
  IF CH = 13 THEN
  PRINT
LOOP WHILE NOT EOF(1)
```

would cause this error if the first value read is 13, even though the real error is that the IF statement has no matching END IF.

Math result is inexact

A valid number has been passed to a math function, but the result is not valid.

The commercial version of GSoft BASIC does not use this error. It is reserved for specialized versions that report math errors in different ways than the commercial version.

Misformed tool or Gs/Os token

A tool token in a source file does not follow the rules for writing a token. Review the rules for coding tokens in the description of TOOL in Chapter 19.

Missing CASE

SELECT was used, but there was no CASE immediately after. The correct syntax for the SELECT statement is

`SELECT CASE expression`

Misuse of a constant

An attempt was made to assign a value to a constant, extract the address of a constant, or find the size of a constant.

These operations are either forbidden on constants or make no sense when applied to a constant. If the error occurs when a constant is passed as a parameter, make sure the constant is passed by value, not by reference. (A simple way to do this is to enclose the constant in parentheses.)

More than 10 nested control statements

Control statements were nested too deeply.

Use subroutines to handle inner nested statements or redesign the logic so the nested statements are not so deep.

Multiple ELSE clause

Two ELSE statements were found for a single IF statement.

Make sure there is only one ELSE clause for each IF-THEN statement. You can use multiple ELSE IF statements after IF-THEN and before ELSE, but only one ELSE.

NEXT without FOR

A NEXT statement was encountered without a matching FOR statement.

This error can occur due to an unfinished statement. If you think the FOR-NEXT statements are matched, look for misformed statements inside the loop, like

```
FOR I = 1 TO 10
  IF A(I) < 0.0 THEN
    A(I) = 0.0
NEXT I
```

In this case, the IF statement is not complete. The NEXT appears when BASIC expects an END IF, but the actual error is a NEXT with no matching FOR.

`Not enough subscripts`

More subscripts are expected for the array than are supplied. For example,

```
DIM A(5, 5)
A(5) = 6
```

would cause this error.

Supply the required number of subscripts. Make sure you do not have more than one array with the same name.

`Out of data`

A READ statement tried to read information from a DATA statement, but there were no unused DATA statements.

Make sure the DATA statements are in the same procedure as the READ statements, or that both are in the main program. If you need to reuse data, use RESTORE.

`Out of memory`

SETMEM or a disk I/O command needed free memory, but could not find a large enough continuous piece of free memory to satisfy the need.

This error does not occur for ALLOCATE. While ALLOCATE gets memory from the same memory pool as SETMEM and the disk commands, it does not flag an error if there is not enough memory. Instead, ALLOCATE returns the NIL pointer, allowing you to check for the error and handle it inside your program.

Some possible solutions are:

- Reduce the amount of memory used for the program or variable space.
- Reduce the memory used by ALLOCATE statements.
- If you are using a program that allows you to run multiple programs, quit the other programs before running GSoft BASIC.
- Shift-boot to reduce the memory used by desk accessories and inits.

`Out of variable table space`

The variables, variable values, strings and local variable space for procedures exceeded the memory set aside for variables.

Increase the memory used for variables using the SETMEM command.

`Output attempted to a file opened for input`

A command that writes to a file was used with a file number for a file opened for input only.

Make sure the file number is correct. If you need to read and write a file, open it as BINARY or RANDOM.

Parameter type mismatch

The type of a parameter to a subroutine or function was not the same as the type of the parameter passed to the procedure, and the types could not be converted using default type conversion rules.

Make sure all parameters are present in the procedure call, and that no extra parameters have been inserted. Make sure the values passed can be converted to the type of the declared parameter. *Check SUB and FUNCTION Parameter Lists* in Chapter 17 for the rules that apply to parameter types.

POP without GOSUB

A POP statement was encountered when no GOSUB was active.

Change the program so POP is only used after a GOSUB, and multiple POP statements are not used for a single GOSUB.

**PRINT USING format string has \ without closing **

A PRINT USING format string contained a \ character, starting the format model for a fixed length string field, but no matching \ character was found to end the format model.

Make sure only space characters appear between the \ characters in a fixed length string format model. If you are trying to print a \ character from the format string, precede the \ character with a _ character, as in

```
PRINT USING "_\#"; 45
```

PRINT USING format string has more than 24 characters in a number model

A numeric format model has more than 24 format characters.

Format models for numbers are limited to 24 characters. Reduce the number of characters in the format model.

PRINT USING format string has no format models

A PRINT USING statement has a format string, but there are no format models in the string.

If you don't need to format any numbers or strings, use a PRINT statement to print a string constant, not PRINT USING. If you intended to use a format model, check the format string—no format models were found in the format string.

Record used inside itself

A record declaration appears inside the declaration of the same record, as in

```
TYPE POINTS
    SUCC AS POINTS
    H AS INTEGER
    V AS INTEGER
END TYPE
```

Records can contain fields that are records, but not if the field is the same as the record it appears in. If you want to create multiple records, use an array of records or a linked list of records, as in

```
DIM POINTS(30) AS POINT
TYPE POINTLIST
    SUCC AS POINTER TO POINTLIST
    H AS INTEGER
    V AS INTEGER
END TYPE
```

See *Using the Record Type In The Record (Linked Lists)* in Chapter 10 for an explanation that tells why the declaration of POINTLIST is valid, but the declaration of POINTS is not.

Redefined type

A type was defined using a type name already in use.

Make sure all type names are unique, and that types are defined only once.

Redimensioned variable error

An array or record was declared twice, or the same name was used for two different arrays or records.

In general, all arrays and records should appear in DIM statements at the top of a program or procedure. If they are all collected in one location, it's easy to avoid, or at least track down, this sort of error.

RENUMBER overlaps old and new lines

The RENUMBER command was used, but a renumbered line had a new line number larger than the first succeeding non-renumbered line.

Renumber the program a different way, or eliminate line numbers completely.

RETURN without GOSUB

A RETURN statement was encountered when no GOSUB was active.

Be careful of a program that drops into a subroutine, like this:

```
10 FOR I = 1 TO 10
20    GOSUB 50
30 NEXT
50 PRINT I
60 RETURN
```

will create a working program.

Adding the line

```
40 END
```

SELECT CASE without END SELECT

A SELECT CASE statement was started, but the program or procedure finished without finding a matching END SELECT.

Make sure there is exactly one END SELECT for each SELECT CASE. Keep in mind that other mismatched statements might cause this error even if there is a matching END SELECT for the SELECT CASE. For example,

```
SELECT CASE COLOR
  CASE RED
    IF SHORT THEN
      PRINT "R";
    ELSE
      PRINT "RED";
  CASE GREEN
    IF SHORT THEN
      PRINT "G";
    ELSE
      PRINT "GREEN";
END SELECT
```

could cause this error, even though the real error is that the IF statement has no matching END IF.

SETMEM and CLEAR cannot be used in a procedure

The SETMEM command or CLEAR command was used in a procedure. These commands can only be used in the main body of the program, and should be used before DIM statements and type declarations.

Syntax error

The line is not a legal BASIC statement.

This is a general catch-all message for misformed lines. Examine the line carefully for problems, remembering that the problem may be subtle, like a comma used where a semicolon is expected.

`The file is already open`

An OPEN statement was used to open a file, but the file was already open.

Make sure the file is only opened one time, or that it is closed before another OPEN statement is used. You might want to close and reopen a file if it was originally open for input, and you need to write new information to the file, for example.

`The line is too long`

GSoft BASIC lines are limited to 255 tokenized bytes during the conversion from ASCII text to tokenized GSoft BASIC programs. The limit doesn't technically apply to a tokenized file, although GSoft BASIC doesn't give you any way to create a line longer than this limit. This error is flagged as a file is loaded from an editor or from a disk file, not while the program executes.

In general, tokens are shorter than ASCII text typed from the command line or in an editor, so this error won't occur until the typed line is far longer than 255 characters.

Shorten the line until this error does not occur. If you are using long strings, consider building the string from shorter strings on multiple lines using string concatenation, like this:

```
A$ = "Hello, "
A$ = A$ + "world."
```

`The program is too long`

The program is too big to fit in the program buffer.

Increase the size of the program buffer using the SETMEM command.

`The size of a value exceeded 32767 bytes`

Arrays and records are limited to 32767 bytes each. You can have multiple arrays or records that, added together, exceed this size, but no single array or record can be larger than 32767 bytes.

Reduce the size of the array or record. Consider using linked lists rather than a large array.

`Too many nested subroutine calls`

GOSUB or ON-GOSUB statements were nested more than 24 levels deep.

Check for unwanted recursive subroutine calls. If you need to nest subroutines more than 24 levels deep, switch to subroutines created with SUB.

`Too many open files`

GSoft BASIC can open up to 8 files at one time. This error occurs when an OPEN statement is used while 8 files are already open.

Reduce the number of open files. Keep in mind that you can close a file, open another, close it, then reopen the first—the limit is on the number of files that are open at one time, not on the total number of files opened by the program.

Too many subscripts

Fewer subscripts are expected for the array than are supplied. For example,

```
DIM A(5)
A(5, 2) = 6
```

would cause this error.

Supply the proper number of subscripts. Make sure you do not have more than one array with the same name.

Type mismatch

A value was used in an incorrect way, such as adding a string to a number or using a pointer that points to a different kind of value than expected.

Change the expression so the types match. If you need to use a value that is not the correct type, consider type casting or converting the value to the correct type with functions like CINT or STR$.

Undefined field

An attempt was made to reference a field of a record, but the field does not exist in the record. Check the spelling for the record field. Make sure you are using a value of the kind of record you expect.

Undefined function

A FN function was used, as in

```
PRINT FN SQUARE (4)
```

but no DEF FN statement had been found which defined the function.

Double-check the spelling in the function declaration and its use. Make sure the DEF FN appears in the same procedure as the line containing the DEF FN function call, or that both appear in the main program. Make sure the DEF FN declaration is executed before the function is used the first time.

Undefined statement

A command that uses a line number as a destination, such as

```
GOTO 500
```

referred to a line number that was not found.

Change the command to refer to a line number that exists, or add the appropriate line number to a statement. If the line number exists, but the line number is not in the same procedure as the statement that uses it, move both statements to the same procedure.

Undefined subroutine

A CALL statement was used, but the identifier that follows is not the name of a subroutine defined with a SUB statement.

Check spelling to make sure the subroutine name is spelled correctly. Make sure the subroutine is in the program file. Keep in mind that tool calls and GS/OS calls do not need a CALL statement, just the identifier.

Unimplemented command

A command that is recognized by some versions of GSoft BASIC is not available in this version.

You can generate this command deliberately with the ERROR command, but it is not generated by the commercial release of GSoft BASIC 1.0.

Unknown tool

A tool, user tool, or GS/OS call was encountered in the program, but the tool was not found in the tool symbol table.

Make sure the GSoftTools.int file is in the correct location. (It can appear in several places; see Chapter 19.) If the program uses additional tool files, be sure they are also in place. If the program was written with a later version of GSoft BASIC than the one you are using, upgrade your copy.

Unopened file referenced

The program used a file number in a command that only works on an open file, but no file with the given number was open.

Use OPEN to open a file before using any other command to manipulate the file. Make sure you have not closed the file with CLOSE.

WEND without WHILE

A WEND was encountered when no WHILE statement was active.

Make sure WEND is only used when a WHILE loop is active. Keep in mind that other mismatched statements can make a correct WHILE-WEND loop generate this error. See the example for "WHILE without WEND," for one way this can happen.

WHILE without WEND

The end of the program or subroutine was reached while a WHILE statement was active, but no WEND was found.

Make sure all WHILE statements have a matching WEND. Make sure the matching WEND is not hidden by another unfinished statement. For example,

```
P = CUSTOMERS
WHILE P <> NIL
    IF P.NAME = NAME THEN
        CALL MATCHFOUND(P)
        P = NIL
    ELSE
        P = P^.SUCC
WEND
```

will generate this error if P is NIL right away, even though the real error is that the IF has no matching END IF.

Appendix B – Console Control Codes

When you are writing programs that will be executed in the GSoft BASIC shell or from the Finder after using MakeRuntime, you have several special console control codes available. These are special characters which cause the console to take some action, like moving the cursor or turning the cursor off. This appendix lists those console control codes.

Keep in mind that these codes are specific to the console driver used with GSoft BASIC. While they are fairly common on Apple IIGS console drivers, they are not universal. These codes don't apply at all if you are writing text to the graphics screen, printing to a printer, or writing to a disk file.

Use the CHR$ function to convert these codes to characters, and any output statement to send them to the console. For example,

```
PRINT CHR$(7);
```

will beep the speaker.

Character codes in the range 0 to 31 that are not listed in the table are ignored.

Character codes from 32 to 127 are ASCII characters. Sending them to the console driver displays the character and moves the cursor forward one position. If the cursor starts in the rightmost position on a line, it moves to the first position of the following line, scrolling the screen if necessary.

If you send a code from 128 to 255 to the console driver, it starts by subtracting 128. The result is in the range 0 to 127, and is treated as described above.

In many cases, there is a built-in BASIC command that does the same thing as the control codes. You can find these commands scattered throughout Chapter 13, *Input and Output*.

Code	Description
5	Turn the cursor on. The character under the cursor is displayed as an inverse character. You would normally use this code to indicate that the user should type a character.
6	Turn the cursor off. This is the default; the cursor position is remembered, but there is no visible indication of the cursor position on the screen.
7	Beep the speaker.
8	Move the cursor one space to the left. Does nothing if the cursor is in the leftmost column.
10	Move the cursor down one line. The cursor is not moved to the leftmost column; it simply moves down. If the cursor starts in the lowest line, the screen scrolls up one line, losing the topmost line and replacing the bottommost line with spaces; in this case the relative position of the cursor does not change.

267

11   Clear to the end of the screen. All characters under the cursor, to the right of the cursor, or on lines below the cursor are replaced with spaces.

12   Clear the screen and home the cursor. All characters are replaced with spaces, and the cursor is moved to the top left position on the screen. This does the same thing as BASIC's HOME statement.

13   Carriage return. The cursor moves to the leftmost column on the current line.

14   Standard characters. All characters printed after sending this code use the standard display set. More precisely, the most significant bit is set on characters before they are displayed, giving standard ASCII characters. In most cases, it makes more sense to use BASIC's NORMAL statement.

15   Alternate characters. Characters are sent to the screen without setting the most significant bit, so they are displayed using either mousetext or inverse characters.

      Looking at *Text Screen Codes* in Appendix C, printing an ASCII character from the rightmost eight columns would display as the character eight columns to the left. For example, printing A would display an inverse A.

24   Mousetext off. Turns off the mousetext mode enabled with code 27. In most cases, it makes more sense to use BASIC's NORMAL statement.

25   Moves the cursor to the top left screen position. The screen is not cleared.

27   Mousetext on. After using this code, printing a character in the range '@' .. '_', which includes the uppercase alphabetic characters, displays the character as one of the mousetext characters. See Appendix C for a description of the mousetext character set.

      Sending this character code to the console does the same thing as BASIC's MOUSETEXT command.

28   Moves the cursor one column to the right. Does nothing if the cursor starts in the rightmost column.

29   Clear to the end of the line. The character beneath the cursor and all characters to the right of the cursor are replaced with spaces.

30   Position the cursor. The two following characters are used as the horizontal and vertical cursor position. After subtracting 31 from each character value, the cursor is moved to that location. For example,

      ```
      PRINT CHR$(30); CHR$(31 + 4); CHR$(31 + 5);
      ```

      moves the cursor to the fourth column, fifth line on the screen.

31   Move the cursor up one line. Does nothing if the cursor starts in the topmost line.

Appendix C – Character Sets

The ASCII Character Set

The ASCII character set establishes numeric equivalents for 95 printing characters. The tie between the ASCII character set and computers is so pervasive that virtually all keyboards built for use in the United States allow input of all of the ASCII characters—and only the ASCII characters. (Some software, like the Apple IIGS toolbox, adds extra characters by interpreting option keys, but these aren't actually on the keyboard.)

The ASCII character set also defines 33 nonprinting characters, numbered 0 to 31 and 127. All of these have a suggested meaning, and many are now nearly universal. You'll see many of these values used in the console control codes from Appendix B. This wasn't always true. To get an idea of how long the ASCII character set has been around, consider that character 127, rub, is used as a delete character. The reason it's character 127 is that this character is made up of seven 1 bits. When a mistake was made punching code into a paper tape—yes, a long yellow strip of paper used to store programs and data—deleting a character meant backing up and punching all seven holes out, or "rubbing out" the letter. And for the first 10 years or so of the microcomputer revolution, it was rare to find a keyboard with the entire ASCII character set available.

The complete ASCII character set is shown below. To find the number for a particular character, add the values to the top and left of the given character. For example, the ordinal value for the character A is 64 + 1, or 65.

	0	16	32	48	64	80	96	112
0	nul	dle		0	@	P	`	p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
8	bs	can	(	8	H	X	h	x
9	ht	em	)	9	I	Y	i	y
10	lf	sub	*	:	J	Z	j	z
11	vt	esc	+	;	K	[	k	{
12	ff	fs	,	<	L	\	l	|
13	cr	gs	-	=	M	]	m	}
14	so	rs	.	>	N	^	n	~
15	si	us	/	?	O	_	o	rub

Text Screen Codes

The text screen can display 256 different character values. Thirty-two of these are duplicates, so the total number of distinct characters it can display is 224.

Oddly enough, the Apple II has always displayed standard ASCII characters by adding 128 to the ASCII character value. When you use any Apple II print command, this mapping is done automatically, but if you decide to poke characters directly to the screen buffer you will need to account for this oddity.

The screen image below shows an actual screen dump of all of the characters the Apple IIGS can display using the standard US ROMs. To find the character number, add the row (counting from 0) to 16 times the column (again counting from 0). For example, a standard white on black lowercase a is $1 + 14 * 16$, or 225.

The Apple IIGS Text Screen Character Set

Toolbox Character Codes

Fonts in the Apple IIGS toolbox generally use the ASCII character set. Apple has also defined several other characters, usually to support alphabets for languages other than English. The table below shows the characters defined by Apple.

Keep in mind that these are guidelines, not requirements. A Greek font, for example, will generally dump the ASCII character to make room for the Greek characters. Some fonts implement radically different character sets, such as postal bar codes, hieroglyphics, or special symbols. Even fonts that implement the ASCII character set don't always add all of the characters shown below. Still, if the character is available, its character number almost always matches the number shown.

270

Appendix C: Character Sets

	00	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
0			spc	0	@	P	`	p	Ä	ê	†	∞	¿	—		
1			!	1	A	Q	a	q	Å	ë	°	±	¡	—		
2			"	2	B	R	b	r	Ç	í	¢	≤	¬	"		
3			#	3	C	S	c	s	É	ì	£	≥	√	"		
4			$	4	D	T	d	t	Ñ	î	§	¥	ƒ	'		
5			%	5	E	U	e	u	Ö	ï	•	µ	≈	'		
6			&	6	F	V	f	v	Ü	ñ	¶	∂	∆	÷		
7			'	7	G	W	g	w	á	ó	ß	∑	«			
8			(	8	H	X	h	x	à	ò	®	∏	»	◊		
9			)	9	I	Y	i	y	â	ô	©	π	…	ÿ		
A			*	:	J	Z	j	z	ä	ö	™	∫	spc			
B			+	;	K	[	k	{	ã	õ	´	ª	À			
C			,	<	L	\	l			å	ú	¨	º	Ã			
D			-	=	M	]	m	}	ç	ù	≠	Ω	Õ			
E			.	>	N	^	n	~	é	û	Æ	æ	Œ			
F			/	?	O	_	o		è	ü	Ø	ø	œ			

- The characters from the space ($20) to the tilde ($7E) are all standard printing ASCII characters.
- While they have standard definitions, the characters $11...$14, $AD, $B0...$B3, $B5...$BA, $BD, $C2...$C6 and $D6 tend to be rare in most fonts.
- Character $CA is the non-breaking space.

271

Appendix D – Writing User Tools for GSoft BASIC

The Role Of User Tools

While GSoft BASIC is a powerful, flexible language, there are some things it doesn't do well. Things like high-speed serial communications, accessing hardware, or implementing fast graphics routines work better in other languages, usually assembly language. User tools give you a way to write specific subroutines like these in a different language, then use them from GSoft BASIC.

There are two other benefits of user tools. First, they can be used from languages other than GSoft BASIC. Any Apple IIGS language that supports user tools, such as ORCA/Pascal, ORCA/C, ORCA/Modula-2 and ORCA/M, can use the same user tool you write for GSoft BASIC. Second, they give you a way to add features to GSoft BASIC through fast, efficient assembly language libraries.

Writing User Tools

Apple IIGS Toolbox Reference Volume 2

Appendix A of the *Apple IIGS Toolbox Reference, Volume 2* is the official documentation about how to write user tools. It tells you exactly how tools are organized, installed, and what the environment is when tool routines are called.

Avoiding Tool Number Conflicts

There are 256 available user tool numbers. While it is unlikely that there will be 256 publicly released user tools for the Apple IIGS, even approaching this number means there will be inevitable conflicts.

If you write your tools carefully, though, this does not need to be a serious problem.

The plain truth is that a tool doesn't need to know its own tool number. You can change the tool number for a carefully written tool by renaming the file and changing the interface file to use the new number. This makes it relatively easy for anyone using your tool to renumber it if there is a numbering conflict with some other existing tool.

There are two places where it might seem that a tool needs to use its own tool number. The first is if the tool needs to call itself. That's not really necessary, though. The tool can be organized internally as a series of subroutines, and calls from inside the tool can call the subroutines rather than using the tool call mechanism. In addition to avoiding a dependence on a specific tool number, this will also make the call faster.

The second possible problem area is tool error numbers. It is traditional to use the tool number as the most significant byte of the error number. In fact, that's exactly what the Game Paddle Library does—the parameter out of range error is the tool number multiplied by 256 and added to 1. If you change the tool number, though, you'll find that the most significant byte of the error number changes, too. That's because the error number is formed from the tool number passed to the Game Paddle Library when it is called, not from a fixed internal constant. Every tool call can look to see what the actual tool number is, rather than depending on a fixed tool number.

The Byte Works, Inc. is using user tools to implement a set of standard libraries. These libraries are numbered sequentially from 1. While all our libraries have numbers that can be changed, it is still best to avoid conflicts when possible, so we suggest that you use user tool numbers of 64 or higher for your own libraries. We will maintain a list of user tools developed for GSoft BASIC to advise you which numbers have been used.

The GSoft BASIC Interface

Writing the tool is the biggest part of creating a user tool for GSoft BASIC, but it's not the only part. You also need to create a tool interface file and compile the interface with CompileTool. This creates the .gst file that GSoft BASIC loads so it knows how to call your user tool.

See *CompileTool* in Chapter 4 for details.

Installing the Game Paddle Library

This section shows you how to install and make use of a user tool from GSoft BASIC. The description assumes you are installing and using a user tool that already has a GSoft BASIC interface file. If you have a user tool that does not have a GSoft BASIC interface file, see *CompileTool* in Chapter 4. CompileTool is the utility you use to create GSoft BASIC interface files.

There are three critical pieces to any user tool: The tool itself, the interface file that tells GSoft BASIC what's in the tool, and the documentation that tells you how to use the tool calls.

The documentation for the Game Paddle Library is in Chapter 18.

The next two paragraphs tell you how to manually install the Game Paddle Library. There is also an installer option to install the tool. If you used that option when you installed GSoft BASIC the files will already be in place.

To install the tool itself, copy the file UserTool001 from the folder :GSoft.Extras:System:Tools on the extras disk to the Tools folder, found inside your System folder. The disk containing this folder must be mounted when your program uses LOADLIBRARY to load the tool.

The interface file is in the :GSoft:Samples:GameTool folder on the program disk. Copy GameTool.gst to any of these locations.

The Folder Containing GSoft.Sys16

Copy the file to this folder if you are using the GSoft BASIC shell. This is the most common place to put the tool interface.

13:GSoftDefs:

Copy the file to this folder if you are using the version of GSoft BASIC that runs from the ORCA shell.

The Folder Containing Your Program

Copy the interface file to the folder containing your program if you don't want the user tool to be available for every program you create. If you copy the file to the local folder, though, don't copy it to the other locations. If you do, GSoft BASIC will load the file from each location. That won't actually cause your program to fail, but it takes extra time each time you start GSoft BASIC, and the files take up extra RAM.

Sample Source

There are some subtle tricks to writing user tools, like the fact that there are actually two return addresses on the stack when your tool is called, not one. It helps to have an example as you sort these issues out!

The complete source code for the Game Paddle Library and the Time Library are in your samples folder. They are written in assembly language, and require ORCA/M to assemble.

275

Appendix E – Converting Applesoft BASIC Programs to GSoft BASIC

The purpose of this appendix is to help you convert Applesoft BASIC programs to GSoft BASIC. It is also useful if you need to convert a GSoft BASIC program to Applesoft BASIC, or if you want to develop a program for both platforms simultaneously.

Applesoft BASIC Peeks, Pokes and Calls

This section lists PEEKs, POKEs and CALLs that are common in Applesoft BASIC programs. It tells what the statement is used for in Applesoft BASIC, and how to get the same effect in GSoft BASIC.

Some PEEK, POKE and CALL addresses are shown as negative numbers. This is an Applesoft BASIC convention for representing numbers in the range 32768 to 65535. The larger numbers actually worked in Applesoft BASIC, so, for example, you might occasionally see PEEK (49152) to read the keyboard instead of PEEK (-16384). To convert from negative numbers to positive, add the value to 65536. To convert from positive values to negative, subtract 65536 from the value. Hexadecimal notation is always used for the GSoft BASIC equivalents, although positive integers will work just as well—but *negative values will not work in GSoft BASIC!* The old 8 bit Apple][used a 16 bit address bus, which is why 65536 (2 raised to the power 16) is used to convert from negative to positive equivalent numbers. The Apple IIGS uses 24 bit numbers for addresses, which GSoft BASIC stores as 32 bit values, so this conversion doesn't work.

PEEKs and POKEs to soft switches still work, but there is often a better way to get the same effect in GSoft BASIC. CALL statements from Applesoft BASIC cannot be made from GSoft BASIC.

Call	Use and Conversion
CALL -10621	Clears the internal stack of all control information. There is no GSoft BASIC equivalent.
CALL -3288	Clears the internal stack in ONERR GOTO handlers. There is no GSoft BASIC equivalent.
CALL -3086	Clears the high resolution screen to black. There is no GSoft BASIC equivalent, but the Apple IIGS graphics screen can be cleared with HGR.
CALL -3082	Clears the high resolution screen to the color most recently used in an HPLOT statement.

277

Command	Use and Conversion
CALL -1998	Clears the low resolution screen to black or fills the text screen with inverse @ characters. There is no GSoft BASIC equivalent, but the Apple IIGS graphics screen can be rapidly painted any color with QuickDraw II's SETSOLIDPENPAT and PAINTRECT commands.
CALL -1994	Clears the upper half of the low resolution screen to black or fills the upper half of the text screen with inverse @ characters. There is no GSoft BASIC equivalent.
CALL -958	Clears the text screen from the current character position to the bottom right corner of the screen. Use `PRINT CHR$(11);` in GSoft BASIC.
CALL -922	Moves the cursor down one line. Use `PRINT CHR$(10);` in GSoft BASIC.
CALL -936	Erases the text screen and places the cursor at the top left of the screen. Use HOME in GSoft BASIC.
CALL -912	Scrolls the screen up one line. There is no GSoft BASIC equivalent, but you can get the same effect by moving the cursor to line 24, issuing a carriage return using a PRINT statement, then repositioning the cursor if you need to preserve the original location.
CALL -868	Clears the text screen from the cursor to the end of the line. Use `PRINT CHR$(29);` in GSoft BASIC.

PEEK	Use and Conversion
PEEK (36)	Returns the current horizontal cursor position.

Appendix E: Converting Applesoft BASIC Programs to GSoft BASIC

PEEK (36)
Use the POS function in GSoft BASIC. Keep in mind that Applesoft BASIC numbers the columns 0 to 79, but GSoft BASIC numbers them 1 to 80.

PEEK (37)
Returns the current vertical cursor position. Use the CSRLIN function in GSoft BASIC. Keep in mind that Applesoft BASIC numbers the lines 0 to 23, but GSoft BASIC numbers them 1 to 24.

PEEK (216)
Returns a value greater than 127 if an ONERR GOTO handler is active. There is no GSoft BASIC equivalent.

PEEK (219) * 256 + PEEK (218)
Used in ONERR GOTO handlers, this expression returns the line number where the error occurred. Use ERL in GSoft BASIC.

PEEK (222)
Used in ONERR GOTO handlers, this PEEK returns the error number. Use ERR in GSoft BASIC.

PEEK (-16384)
Returns the last character typed from the keyboard. Use

```
PEEK ($00C000)
```

in GSoft BASIC. Keep in mind that there are several ways to read the keyboard under GS/OS. If the Event Manager or the GNO shell are in use, reading the keyboard this way is not appropriate.

PEEK (-16336)
Clicks the speaker. Use

```
PEEK ($00C030)
```

in GSoft BASIC.

PEEK (-16352)
Clicks the cassette recorder output. The cassette recorder output is only present on the Apple][and Apple //e. Use ... in GSoft BASIC.

PEEK (-16320)
Triggers the game paddle port utility strobe. Use

```
PEEK ($00C040)
```

in GSoft BASIC.

PEEK (-16287)
Reads game paddle button 0. If the result is greater than 127, the button is being pressed; if not, the button is not being pressed. Use ... in GSoft BASIC.

PEEK (\$00C061)

in GSoft BASIC.

PEEK (-16286)   Reads game paddle button 1. If the result is greater than 127, the button is
being pressed; if not, the button is not being pressed.
Use

PEEK (\$00C062)

in GSoft BASIC.

PEEK (-16285)   Reads game paddle button 2. If the result is greater than 127, the button is
being pressed; if not, the button is not being pressed.
Use

PEEK (\$00C063)

in GSoft BASIC.

POKE	Use and Conversion
POKE 32 , L	Sets the left edge of the text screen. There is no GSoft BASIC equivalent.
POKE 33 , W	Sets the width of the text screen. There is no GSoft BASIC equivalent.
POKE 34 , T	Sets the top edge of the text screen. There is no GSoft BASIC equivalent.
POKE 35 , B	Sets the bottom edge of the text screen. There is no GSoft BASIC equivalent.
POKE 36 , CH	Sets the horizontal cursor position. Use HTAB in GSoft BASIC. Keep in mind that Applesoft BASIC numbers the columns 0 to 79, but GSoft BASIC numbers them 1 to 80.
POKE 37 , CV	Sets the horizontal cursor position. Use VTAB in GSoft BASIC. Keep in mind that Applesoft BASIC numbers the lines 0 to 23, but GSoft BASIC numbers them 1 to 24.
POKE 216 , 0	Turns any active ONERR GOTO handler off. Use

ONERR GOTO 0

Appendix E: Converting Applesoft BASIC Programs to GSoft BASIC

POKE -16368 , 0   Clears the keyboard strobe.
Use

```
POKE $00C010, 0
```

in GSoft BASIC.

POKE -16304 , 0   Switches the display from the text screen to one of the graphics modes.
See also POKE -16297 and POKE -16298.

Depending on how you are translating the program, you may want to
leave the POKE as is, or you may want to use GSoft BASIC's HGR
statement.

POKE -16303 , 0   Switches the display from one of the graphics modes to the text screen.

Depending on how you are translating the program, you may want to
leave the POKE as is, or you may want to use GSoft BASIC's TEXT
command.

POKE -16302 , 0   Switches the display to full screen graphics. See also POKE -16301.

There is no GSoft BASIC equivalent to mixed text and graphics.

Depending on how the text and graphics screens are accessed, you may be
able to omit the POKE altogether or use it as is.

POKE -16301 , 0   Switches the display to mixed text and graphics. See also POKE -16302.

There is no GSoft BASIC equivalent to mixed text and graphics.

POKE -16300 , 0   Switches the display from page 2 to page 1. See also POKE -16299.

You do not usually use page 2 with GSoft BASIC, but with adequate
preparation you can use this POKE as is. See *Low Resolution Graphics
and Text Screen Access*, later in this appendix, for details.

POKE -16299 , 0   Switches the display from page 1 to page 2. See also POKE -16300.

You do not usually use page 2 with GSoft BASIC, but with adequate
preparation you can use this POKE as is. See *Low Resolution Graphics
and Text Screen Access*, later in this appendix, for details.

POKE -16298 , 0   Displays the Apple][low resolution graphics screen, as opposed to the
high resolution graphics screen. See also POKE -16304 and POKE -
16297.

Depending on how you are translating the program, you may want to
leave the POKE as is, or you may want to use GSoft BASIC's HGR
statement.

POKE -16297 , 0   Displays the Apple][high resolution graphics screen. This is not the same as the Apple IIGS
low resolution graphics screen, as opposed to the
graphics screen normally used by GSoft BASIC and the Apple IIGS
toolbox. See also POKE -16304 and POKE -16298.

281

Depending on how you are translating the program, you may want to leave the POKE as is, or you may want to use GSoft BASIC's HGR statement.

POKE -16296 , 0   Turns off game paddle annunciator 0.
                   Use

                   POKE $00C058 , 0

                   in GSoft BASIC.

POKE -16295 , 0   Turns on game paddle annunciator 0.
                   Use

                   POKE $00C059 , 0

                   in GSoft BASIC.

POKE -16294 , 0   Turns off game paddle annunciator 1.
                   Use

                   POKE $00C05A , 0

                   in GSoft BASIC.

POKE -16293 , 0   Turns on game paddle annunciator 1.
                   Use

                   POKE $00C05B , 0

                   in GSoft BASIC.

POKE -16292 , 0   Turns off game paddle annunciator 2.
                   Use

                   POKE $00C05C , 0

                   in GSoft BASIC.

POKE -16291 , 0   Turns on game paddle annunciator 2.
                   Use

                   POKE $00C05D , 0

                   in GSoft BASIC.

POKE -16290 , 0   Turns off game paddle annunciator 3.

Use

```
POKE $00C05E , 0
```

in GSoft BASIC.

```
POKE -16289 , 0   Turns on game paddle annunciator 3.
```

Use

```
POKE $00C05F , 0
```

in GSoft BASIC.

Low Resolution Graphics and Text Screen Access

PEEKs and POKEs in the range $0400 to $07FF are reading and writing the first of two pages used for 40 column text or low resolution graphics. This memory area can be displayed as text, low resolution graphics, or a mixture with graphics on top and four lines of text at the bottom. The display method is set using soft switches. See the various POKE commands in the previous section for details about the locations to poke for various effects.

While GSoft BASIC does not support low resolution graphics directly, PEEKs and POKEs to the graphics page work fine, and PEEKs and POKEs to the text screen work fine, too.

Applesoft BASIC programs written for the Apple //e, Apple //c or Apple IIGS may also manipulate memory in the range $010400 to $0107FF, generally using bank switching, but occasionally by poking assembly routines into RAM. This gives access to the other 40 columns of text used when 80 columns of text are displayed. Bank switching will still work under GSoft BASIC. Poking assembly routines into RAM is not safe under GS/OS, but converting the program to use POKEs is generally easy. Under Applesoft BASIC, you could not POKE directly into memory bank 1, which is why these tricks are used in Applesoft BASIC. From GSoft BASIC you can POKE anywhere in RAM.

PEEKs and POKEs in the range $0800 to $0BFF are accessing the second 40 column text and low resolution graphics page. Accessing $010800 to $010BFF, again through bank switching or poking assembly language into RAM, is manipulating the second 40 columns of an 80 column display. Accessing this second page is generally not safe under GS/OS. If you must use the second display, you must reserve this memory first using the Memory Manager Tool Set. Since this range of memory is generally used for direct page memory by the first program GS/OS executes, this will be difficult—in fact, you may need to write a GS/OS init that will reserve the memory permanently at boot time.

Commands in GSoft BASIC That Are Not In Applesoft BASIC

These commands exist in GSoft BASIC but not in Applesoft BASIC. In some cases there are equivalent ways to accomplish the same thing, generally with the PEEKs, POKEs and CALLs shown earlier in this appendix.

If you are converting a GSoft BASIC program to Applesoft BASIC, you will have to remove each of these commands from the program.

!	ALLOCATE	BREAK	CASE	CDBL
CHDIR	CINT	CLOSE	CLNG	CSNG
CSRLIN	CURDIR$	DIR$	DISPOSE	DO
ERL	ERROR	ERR	EOF	FUNCTION
GSOS	KILL	LINE INPUT	LOADLIBRARY	LOC
LOF	LOOP	MKDIR	MOUSETEXT	NAME
NIL	OPEN	PUT	RMDIR	SEEK
SELECT	SETMEM	SIZEOF	SUB	TOOL
TOOLERROR	TYPE	UNLOADLIBRARY	LIBRARY	WHILE

Commands in Applesoft BASIC That Are Not In GSoft BASIC

These commands exist in Applesoft BASIC but not in GSoft BASIC.

&	COLOR=	DEL	DRAW	FLASH
GR	HGR2	HIMEM:	HLIN	IN#
LIST	LOAD	LOMEM:	NOTRACE	PDL
PLOT	PR#	RECALL	ROT=	SAVE
SCALE=	SCRN	SHLOAD	STORE	TRACE
USR	VLIN	XDRAW		

In some cases, like FLASH and SHLOAD, the commands actually aren't available in Applesoft BASIC on an Apple IIGS, either—the cassette port is missing, and the character ROM no longer supports flashing characters. STORE and RECALL were also used with cassette tape drives, and have no GSoft BASIC equivalent.

LOAD and SAVE worked with either cassette tape drives or disks, depending on whether a file name was used. GSoft BASIC supports LOAD and SAVE from the command line, but not from inside an executing program.

The & and USR commands were used to extend Applesoft BASIC using assembly language. Old & packages and USR subroutines cannot safely execute under GS/OS, so any Applesoft BASIC program that uses them would need to be converted. GSoft BASIC does support extensions via assembly language, but it uses LIBRARY to do so. Since the old

commands must be rewritten anyway, it makes sense to take advantage of the names and parameter passing available from user tools.

IN# and PR# are used in Applesoft BASIC to redirect input and output to hardware cards. The disk commands are used to handle all input and output in GSoft BASIC. GS/OS drivers can be written for practically any device, and have already been written for most of them.

DEL and LIST are available in GSoft BASIC, but not from inside a program. The GSoft BASIC shell has both commands, and they work just like they do in Applesoft BASIC.

HIMEM: and LOMEM: deal with memory allocation in a way that is not safe under GS/OS. GSoft BASIC supports SETMEM for the same purpose.

COLOR=, DRAW, GR, HGR2, HLIN, PLOT, ROT=, SCALE=, SCRN, VLIN and XDRAW are older graphics commands. GSoft BASIC has access to QuickDraw II, where you will find equivalents for most of these commands, as well as many new features.

TRACE and NOTRACE are used for debugging Applesoft BASIC programs. GSoft BASIC uses BREAK and source level debuggers (sold separately) for the same task.

The PDL command reads game paddle controls and joysticks. There is no equivalent command in GSoft BASIC, but there is a game port user tool that does the same thing. See appendix D for details.

Commands That Are Different in Applesoft BASIC and GSoft BASIC

These commands exist in both implementations of BASIC, and with the exception of CALL, generally do the same thing. Each command has some extended capabilities in GSoft BASIC that you will have to take into account when porting GSoft BASIC programs to Applesoft BASIC, though. CALL is the only statement that exists in both languages that will cause problems when you port an Applesoft BASIC program to GSoft BASIC.

CALL	The CALL statement is used to call machine language subroutines in Applesoft BASIC and SUB subroutines in GSoft BASIC.
DEF FN	GSoft BASIC supports multiple parameters and more return types.
DIM	GSoft BASIC supports named types using AS clauses.
FOR	GSoft BASIC supports all number types as FOR loop variables; Applesoft BASIC only allows SINGLE numbers.
GET	GSoft BASIC supports file numbers.
HCOLOR=	Applesoft BASIC uses the high resolution graphics screen, which displays a limited 6 colors on a 280 by 192 pixel screen. GSoft BASIC uses the Apple IIGS graphics screen, which supports 16 distinct colors (more with some tricks) on a 320 by 200 pixel screen.
HGR	Starts QuickDraw II in 320 by 200 pixel mode. See also HCOLOR=.
HPLOT	Draws a line on the QuickDraw II display. See also HCOLOR=.

IF	GSoft BASIC adds block IF-THEN-ELSE statements. All Applesoft BASIC IF statements will still work.
INPUT	GSoft BASIC supports file numbers.
PEEK	GSoft BASIC uses 24 bit addresses, while Applesoft BASIC uses 16 bit addresses. This causes conversion problems with Applesoft BASIC addresses that are negative numbers. These must be converted to positive numbers by adding 65536 before they can be used in GSoft BASIC. See the PEEK and POKE conversion table, earlier in this appendix, for a complete discussion.
POKE	See PEEK.
PRINT	GSoft BASIC supports PRINT USING and printing to files by file number.
PUT	GSoft BASIC supports file numbers.

Other Differences

Available Memory

Applesoft BASIC is limited to memory from $000800 to the beginning of the operating system in use. For ProDOS, that's $009600, giving you 35.5K of RAM. This memory is used both for the program and for variables. High resolution graphics, using page 2 of the text or low resolution graphics screens, & packages and machine language subroutines all took memory away.

GSoft BASIC doesn't have a fixed limit on memory. It's a rare Apple IIGS that can't set aside 256K each for the program and variable buffers and have room left over for dynamic memory allocated with ALLOCATE, and it's quite common for an Apple IIGS to have enough memory to set aside 1024K or more for each buffer. Compared to Applesoft BASIC, the memory available to GSoft BASIC programs is staggering. If you use a great deal of memory, it's unlikely you will be able to port your program to Applesoft BASIC without seriously crippling its capabilities.

Disk Input and Output

Applesoft BASIC does not have disk input and output commands. Disk input and output is handled by special print statements that start with CHR$(4). GSoft BASIC uses built-in disk commands that generally match those found on Microsoft BASIC implementations on DOS and Windows machines. While Applesoft BASIC and GSoft BASIC each have specific commands that are not available in the other implementation, you will find that common file operations are easy enough to accomplish from either BASIC. On the other hand, the commands are implemented in very different ways, so they will have to be translated as a program is ported in either direction.

Line Numbers

GSoft BASIC does not require line numbers. You can add them after the fact with the RENUMBER command, which is a good first step anytime you are porting a program from GSoft BASIC to Applesoft BASIC.

Numbers

Applesoft BASIC really only handles one number type, single precision real numbers. While it can store numbers in INTEGER format, these numbers are always converted to SINGLE values for calculations. This makes it faster to use SINGLE numbers in Applesoft BASIC, while it is much faster to use INTEGER values in GSoft BASIC.

GSoft BASIC also adds BYTE, LONG and DOUBLE numbers. These are not supported in Applesoft BASIC, so you will need to avoid them or convert them to port a GSoft BASIC program to Applesoft BASIC.

Finally, Applesoft BASIC uses a five byte SINGLE number that has over 9 decimal digits of precision, while GSoft BASIC uses a four byte SINGLE with over 7 decimal digits of precision. Some Applesoft BASIC programs make use of this extra precision. The easiest way to deal with this in GSoft BASIC is to convert the numbers to DOUBLE, which gives you even more precision—about 16 decimal digits of precision.

This extra precision gives Applesoft BASIC one other edge. In some situations, roundoff error shows up earlier in GSoft BASIC than it does in Applesoft BASIC. This is again caused by the extra precision from Applesoft BASIC's longer SINGLE numbers. Once again, in programs where this extra precision matters, you can switch to DOUBLE numbers.

Appendix F – Implementation Details

Memory Use

GSoft BASIC uses two large memory buffers which it subdivides for efficient internal use, the program buffer and the variable buffer. Programs can also allocate memory from outside this range using the ALLOCATE statement; see *Dynamic Memory*, later in this appendix.

Program Buffer

The program buffer is used to store the executing program. The program is stored as a series of tokenized lines, as described in *Organization in Memory*, later in this appendix.

If you are using GSoft BASIC from the GSoft BASIC shell, the program buffer is set using the SETMEM command. You will generally want to make the program buffer about 32K larger than the program to give you plenty of room to add new lines to the program. To get the approximate size of an existing program, multiply the number of blocks shown by the CATALOG command for the program file by 512.

If you are using GSoft BASIC from the ORCA shell, the program buffer is formed exactly to size as the program loads. From the ORCA shell, the program is stored as an ASCII text file. This file is loaded into RAM, then the program is converted to tokenized form. After this conversion, the program buffer's size is reduced to the exact size needed and the ASCII text file representation of the program is marked as purgeable. This releases the memory for use by your program, but leaves the file in memory for faster loading next time if the memory isn't needed by your program.

The most efficient of all is a program converted to run from the Finder using the MakeRuntime utility. These programs are stored on disk in tokenized format. When you run the program, GSoft BASIC allocates exactly the right amount of space and loads the program into RAM.

Variable Buffer

The variable buffer is used for global variables, subroutine parameters, local subroutine variables and strings. As the program starts to run, GSoft BASIC scans the file for SUB and FUNCTION statements, setting up a table in this buffer. Types, variables and DEF FN names from the main program are next. Finally, as each SUB or FUNCTION is called, a stack frame is created. The stack frame holds information about the call, such as the return address and the value of the parameters, as well as any local types, variables and DEF FN declarations from the procedure. This memory is released and later reused once the procedure finishes.

String variables are actually a pointer to a string stored in a string pool at the end of the variable buffer. As strings values are changed, empty memory can be created between the current string values. Eventually, there may not be enough memory left to store a string value, create a new subroutine, or create a new variable. When this happens, the strings are compacted, collecting all of the small pieces of memory into a single large piece. This process is called garbage collection. If garbage collection doesn't create enough free memory for the operation that is underway, your program stops with an out of memory error.

Garbage collection occurs automatically when needed. In some programs garbage collection can cause a delay at a critical time, such as a visible update on the screen or during communication with a time-sensitive external device. If this becomes a problem, use the FRE command to force garbage collection at a more convenient time and make sure the variable buffer is large enough that garbage collection will not be needed before you can force it again. Your program will be simpler, smaller and faster if you let garbage collection happen automatically, though, so in the vast majority of situations you should ignore garbage collection entirely and let it happen when needed.

Tool interface files don't take up space in this buffer. The space needed by tool interface files is allocated from main memory.

Dynamic Memory

The ALLOCATE statement gets memory from outside the program buffer and variable buffer. When you ask for a piece of memory larger than 2048 bytes, ALLOCATE gets a chunk of memory directly from Apple's Memory Manager Tool Set. For smaller chunks, GSoft BASIC allocates a 4096 byte chunk of memory and subdivides it.

Using ALLOCATE statements, a relatively small program with a small variable buffer can get access to all of the memory in the Apple IIGS. For some kinds of programs, this makes a great deal more sense than using fixed arrays in a large variable buffer.

Other Memory Locations

Applesoft BASIC programmers are used to owning the machine, using PEEK and POKE commands to hammer any memory they like. A memory map shows the places to stay away from, but any location not on the memory map is fair game. Not so with GSoft BASIC, which runs under GS/OS! The operating system, GSoft BASIC itself, other programs, inits, desk accessories and drivers all use memory. They claim memory using Apple's Memory Manager Tool Set, use the memory they get this way, and free the memory when it is no longer needed. With a few exceptions, it simply isn't safe for a GSoft BASIC program to use PEEK and POKE commands.

In general, your programs should use ALLOCATE to get memory and DISPOSE when the program is finished with the memory.

Tokenized Files

GSoft BASIC organizes tokenized files and files in memory as lines of tokenized symbols. Tokens are one, two or three byte numbers used to represent a reserved word or tool name. Using a number, usually a one byte number, makes programs much smaller, and allows the interpreter to run much faster, since it doesn't have to read and look up a sequence of characters to decide what command to execute. For example, a PRINT statement takes six bytes in an ASCII file—five for the letters and one for a space that invariably follows them—but only one byte with a value of 186 in a tokenized line.

This space saving is so dramatic that tokenized programs are usually shorter than the same program after it is compiled to machine code with a compiler. Of course the compiled program is faster, but in some cases space is more important than speed. This is one of the frustrating reasons why Applesoft BASIC programs generally could not be compiled. Due to the memory limits of an eight bit Apple][computer, even modest Applesoft BASIC programs frequently used all of the memory available to them. Compiling the program would frequently make it too large to fit in memory!

The Organization of Tokenized Programs

Whether the program is in a file or loaded in RAM, each program consists of a sequence of lines.

Each line starts with a two byte offset to the start of the following line. The end of the program is marked with two bytes of zero where this offset would normally appear. The offset is a two byte unsigned value stored least significant byte first. This means that each line must be smaller than 65535 bytes in length.

The next two bytes are the line number. Line numbers are optional in GSoft BASIC; if a line has no line number, the line number field is set to 0. Again, the number is a two byte unsigned value stored least significant byte first. This implies that the largest allowed line number is 65535.

Next comes the tokenized line. Every token that is a reserved word or the name of a tool call is converted to its equivalent token. Spaces and tabs are used to separate tokens from surrounding characters, but once their job is done they are dropped from the line. Identifiers, numbers, operators, data in DATA statements, strings and comments are stored in their original ASCII character form.

The end of the line is marked with a zero byte.

In a disk file, the end of file mark for the file appears right after the two zero bytes that mark the end of the sequence of lines. Since the last line ends in a zero, too, this means every program ends with three bytes of zero.

Programs in RAM are represented the same way, and also end with three zero bytes. The program buffer can extend further in memory; if it does, the contents of all bytes past the end marker are undefined.

Line Number Schemes

There are actually two incompatible line number schemes in use. In programs like those imported from Applesoft BASIC or typed from the GSoft BASIC shell which use line numbers on every line, line numbers must be sequential and unique. In programs that do not use line numbers on every line, line numbers do not have to be sequential. They do have to be unique within the main program or within any specific procedure, but you can safely use, say, 999 as an error exit in several different procedures. Even the limitation that line numbers within a procedure be unique is not enforced; if there are two identical line numbers in the same procedure, GSoft BASIC will find and use the first line number.

In a few cases GSoft BASIC needs some way to tell these two kinds of programs apart. This is important when the program is being edited, but not when it is running. It does so by scanning the program when it is first loaded. If there is any non-blank line in the program where a line number is missing or zero, the program is treated as if it uses optional line numbers. If every line has a non-zero line number, the program is treated as an old-style program with a line number on every line.

BASIC Tokens

The following table shows the tokens used by GSoft BASIC. These tokens are mixed with line lengths, line numbers, and ASCII characters as described in *File Organization and Organization in Memory*, later in this appendix.

The table shows the hexadecimal value for the token, the decimal value, and the ASCII characters printed by GSoft BASIC when the token is listed.

Token $FF (255) is a special token used to extend the number of available tokens from 128 to 383. It is the first byte of a two byte token; the next byte completes the pair. The decimal values shown assume that you read the complete two-byte token from memory as an INTEGER.

TOOL ($9F), LIBRARY ($A4) and GSOS ($90) start a three byte token sequence. For TOOL and LIBRARY, the byte immediately following this token is the tool number, and the third byte is the tool call number. For GSOS, the second byte is the least significant byte of the two-byte call number, and the last byte is the most significant byte of the two-byte call number.

$80	128	END
$81	129	FOR
$82	130	NEXT
$83	131	DATA
$84	132	INPUT
$85	133	CLNG
$86	134	DIM
$87	135	READ
$88	136	CSNG
$89	137	TEXT
$8A	138	PUT
$8B	139	SEEK
$8C	140	CALL
$8D	141	CDBL
$8E	142	CINT
$8F	143	CLOSE
$90	144	GSOS
$91	145	HGR
$92	146	HCOLOR=
$93	147	HPLOT
$94	148	CHDIR
$95	149	DIR$
$96	150	HTAB
$97	151	HOME
$98	152	MKDIR
$99	153	NAME

Hex	Dec	Token
$9A	154	OPEN
$9B	155	BREAK
$9C	156	LINE
$9D	157	NORMAL
$9E	158	INVERSE
$9F	159	TOOL
$A0	160	RMDIR
$A1	161	POP
$A2	162	VTAB
$A3	163	SETMEM
$A4	164	LIBRARY
$A5	165	ONERR
$A6	166	RESUME
$A7	167	unused
$A8	168	unused
$A9	169	SPEED=
$AA	170	LET
$AB	171	GOTO
$AC	172	CURDIR$
$AD	173	IF
$AE	174	RESTORE
$AF	175	EOF
$B0	176	GOSUB
$B1	177	RETURN
$B2	178	REM
$B3	179	STOP
$B4	180	ON
$B5	181	WAIT
$B6	182	KILL
$B7	183	LOF
$B8	184	DEF
$B9	185	POKE
$BA	186	PRINT
$BB	187	CONT
$BC	188	LOC
$BD	189	CLEAR
$BE	190	GET
$BF	191	unused
$C0	192	TAB
$C1	193	TO
$C2	194	FN
$C3	195	SPC
$C4	196	THEN
$C5	197	TYPE
$C6	198	NOT
$C7	199	STEP
$C8	200	unused
$C9	201	TOOLERROR
$CA	202	LOADLIBRARY
$CB	203	UNLOADLIBRARY

Hex	Dec	Token
$CC	204	SIZEOF
$CD	205	AND
$CE	206	OR
$CF	207	unused
$D0	208	unused
$D1	209	unused
$D2	210	SGN
$D3	211	INT
$D4	212	ABS
$D5	213	SUB
$D6	214	FRE
$D7	215	FUNCTION
$D8	216	POINTER
$D9	217	POS
$DA	218	SQR
$DB	219	RND
$DC	220	LOG
$DD	221	EXP
$DE	222	COS
$DF	223	SIN
$E0	224	TAN
$E1	225	ATN
$E2	226	PEEK
$E3	227	LEN
$E4	228	STR$
$E5	229	VAL
$E6	230	ASC
$E7	231	CHR$
$E8	232	LEFT$
$E9	233	RIGHT$
$EA	234	MID$
$EB	235	ERL
$EC	236	ERROR
$ED	237	ERR
$EE	238	CSRLIN
$EF	239	MOUSETEXT
$F0	240	WHILE
$F1	241	WEND
$F2	242	DO
$F3	243	LOOP
$F4	244	UNTIL
$F5	245	ELSE
$F6	246	SELECT
$F7	247	CASE
$F8	248	USING
$F9	249	AS
$FA	250	AT
$FB	251	NIL
$FC	252	ALLOCATE
$FD	253	DISPOSE

Appendices

$FE	254	unused
$FF	255	first of multi-byte
$FF01	511	BYTE
$FF02	767	LONG
$FF03	1023	SINGLE
$FF04	1279	DOUBLE
$FF05	1535	STRING
$FF06	1791	INTEGER
$FF07	2047	OUTPUT
$FF08	2303	BINARY
$FF09	2559	APPEND
$FF0A	2815	RANDOM
$FF0B	3071	TCP

Example of a Tokenized Program

Here's a short program, shown both in its text form and as a file dump that shows how it is represented in memory or on disk. The dump shows the offset within the file along the left side, followed by sixteen bytes per line in groups of four. The ASCII equivalents are shown to the right, where you can see that the numbers, operators and identifiers are represented as ASCII characters.

```
10 FOR I = 1 TO 10
20   PRINT I
30 NEXT
```

```
$000000  0C000A00 81493D31 C1313000 07001400   '  I=1 10  '
$000010  BA490006 001E0082 000000              '  I  '
```

Appendix G – Quick Reference to the Shell

BYE

    Exits GSoft BASIC.

CAT [pathname]
CATALOG [pathname]

    Catalogs a directory.

    If no path name is given, the current directory is cataloged. If given, the path name can be a full or partial path name, the name of a volume, or the name of a device.

    The abbreviation CAT can be used instead of the full name of CATALOG.

COPY from to

    Copies a file from one location to another.

CREATE pathname

    Creates a new directory.

DEBUG [linenumber | filename]

    Runs a program, with the same options as the RUN command. The difference is that DEBUG enters an ORCA compatible debugger (like ORCA/Debugger or Splat!), breaking on the first line executed.

    Do not use this command unless an ORCA compatible debugger is installed! ORCA compatible debuggers work by intercepting the 65816 COP instruction. There is no way for GSoft BASIC to tell if a debugger is installed or not, so it will issue the COP instruction whether or not a debugger is actually present. If there is no debugger installed, this causes the computer to crash. While this does no actual harm, the only way to recover is to reboot.

DEL start [',' end]

    Deletes a single line or a range of lines.

    The DEL command cannot be used with programs that do not use line numbers on every line.

`DELETE  filename`

Deletes the named file.

The file can be a directory. After checking to be sure the user really wants to delete the directory and its contents, all files in the directory and the directory itself are deleted.

`EDIT  [  filename  ]`

Enters an ORCA compatible editor, displaying the program in memory. If a file name is given, the file is loaded and edited exactly as if the commands

```
LOAD filename
EDIT
```

were used.

`LIST  [  line-number  [  ','  [  line-number  ]  ]  ]`

Lists the entire program, a single line, or a range of lines.

`LOAD  filename`

Loads a program from disk.

The program may be a GSoft BASIC tokenized file, a TXT file, or a BASIC SRC file. If the file is a TXT or SRC file, it is handled as if the NEW command was used, then each of the lines in the file was typed in turn.

`LOCK  filename`

Locks a file. Locked files cannot be renamed, deleted, or written to.

`MOVE  from  to`

Moves a file from one location to another.

`NEW`

The program is deleted from the workspace.

`PR  [  line-number  [  ','  [  line-number  ]  ]  ]`

Works like LIST, but sends the listing to a printer that is supported by the .PRINTER driver.

PREFIX [pathname]

Changes the default prefix (GS/OS prefix number 8) to the given path name.
If no prefix is given, the current value for the prefix is shown.
A path name consisting of two periods moves up one directory level.

RENAME old new

Renames a file.

RENUMBER first ',' step [',' start [',' end]]

Renumbers a program.

first	First line number to use.
step	Increment between new line numbers.
start	First line to renumber.
end	Last line to renumber.

RUN [line-number | filename]

Runs a program.
If a number is supplied as a parameter, program execution starts at that line.
If a file name is supplied as a parameter, the file is loaded and executed. The file may be a
GSoft BASIC tokenized file, a TXT file, or a BASIC SRC file. If the file is a TXT or SRC file,
it is handled as if the NEW command was used, then each of the lines in the file was typed in turn.

SAVE filename

Saves a program to disk.
The file is saved as a GSoft BASIC tokenized file.

SSAVE filename

Saves a program to disk.
The file is saved as an ORCA BASIC source file.

TSAVE filename

Saves a program to disk.
The file is saved as a text file.

UNLOCK   filename

Unlocks a file locked with the LOCK command. Locked files cannot be renamed, deleted, or written to.

Appendix H – Quick Reference to GSoft BASIC

Statements

! `any-ascii-characters`

The ! statement starts a comment. All characters from the ! character to the end of the line are ignored.

See also REM.

ALLOCATE `'(' l-value [ ',' expression ] ')'`

Allocates memory from the computer's memory. `l-value` is set to a pointer to the allocated memory. `expression` is the number of bytes of memory to reserve. If `expression` is not used, enough memory is reserved for one value of the type l-value.

See also DISPOSE, SIZEOF.

BREAK

Enters an ORCA compatible high level language source-level debugger, breaking on the current line.

Do not use this command unless an ORCA compatible debugger is installed! ORCA compatible debuggers work by intercepting the 65816 COP instruction. There is no way for GSoft BASIC to tell if a debugger is installed or not, so it will issue the COP instruction whether or not a debugger is actually present. If there is no debugger installed, this causes the computer to crash. While this does no actual harm, the only way to recover is to reboot.

CALL `identifier [ parameter-list ]`

Calls a subroutine defined by a SUB statement or a tool defined in a tool interface file. See SUB for details.

CASE

See SELECT.

CLEAR

Erases all types, variables and strings. Variables are removed whether they were created with the DIM statement or by being used without encountering a DIM statement.

CHDIR   pathname

Changes the default prefix to pathname.

CLOSE   ['#'   expression   ]

Closes a file.

If a file number is used, CLOSE closes the specific file specified by the expression. If no file number is used, CLOSE closes all files that have been opened by OPEN.

See also OPEN.

CONT

Continues execution after a STOP or END command.

DATA   any-ascii-characters

Creates DATA for READ statements. More than one piece of data can be created with a single DATA statement by separating the data with commas.

See also READ, RESTORE.

**DEF   FN   identifier   '('   identifier   [','   identifier   ]*   ')'
'='   expression**

Creates a local function.

Parameters and the value returned by the function can be any numeric or string type. Types are assigned using trailing type characters, as in A$ for a string.

When the function is called using a FN term in an expression, each parameter in the call is evaluated and assigned to the corresponding parameter variable. The expression is then evaluated. The expression must result in a value that is type compatible with the function name. The expression can use constants, parameter variables, other variables that do not have the same name as a parameter, and other functions.

Functions created with DEF FN are local to the main program or procedure in which they are created.

DIM identifier [subscript] [As type] [','
identifier [subscript] [As type]]*

Creates a variable. The variable can be an array or a single value. For arrays, a subscript is given. The result of each expression is converted to an integer and used as the maximum subscript value for the array. The minimum subscript value is always 0.

If no type is given, the last character of the name determines the variable type. Types assigned with the AS clause may be used with arrays or single values.

DISPOSE '(' l-value ')'

Disposes of memory previously allocated with ALLOCATE.

It is an error to dispose of memory using a pointer that was not assigned by ALLOCATE or to dispose of the same memory twice. BASIC cannot catch this error. An error of this type may eventually lead to corrupted memory or a crash.

See also ALLOCATE.

DO [WHILE expression | UNTIL expression]
LOOP [WHILE expression | UNTIL expression]

The DO and LOOP commands form a loop, executing all statements between them until certain conditions are met. Conditions can be used on either or both the DO and LOOP clause, or on neither one.

Execution begins at the DO statement. If the statement is a DO WHILE statement, the expression is evaluated, and if it is true (any non-zero numeric value is treated as true) the statements between DO and LOOP are executed. If the condition is false (zero is treated as false) execution continues with the first statement after the LOOP statement.

If the DO statement is a DO UNTIL statement, the expression is evaluated, but this time the statements between DO and LOOP are evaluated if the expression is false. If the expression is true, execution skips to the first statement after the LOOP statement.

If the LOOP statement is eventually executed, and there is no condition, execution jumps back to the DO statement.

If the LOOP statement is a LOOP WHILE, the expression is evaluated. If it is true, execution jumps back to the DO statement. If it is false, execution continues with the statement after the LOOP statement.

If the LOOP statement is a LOOP UNTIL, the expression is evaluated. If it is false, execution jumps back to the DO statement. If it is true, execution continues with the statement after the LOOP statement.

ELSE

See IF.

END

Stops execution of a program.

See also IF, SELECT CASE, FUNCTION, SUB, TYPE.

ERROR expression

Behaves exactly as if a run-time error occurred. The expression designates the error number, which can be read using the ERR function.

For a list of error messages by number, see the full documentation for the ERROR command.

See also ERR, ERL.

FOR identifier '=' expression TO expression [STEP
expression]
NEXT [identifier] [',' identifier]*

The FOR-NEXT loop executes a series of statements a specific number of times.

The expressions are evaluated. The first expression is assigned to the control variable, which is the identifier immediately after FOR. It must be a single numeric value; arrays, record fields and pointers are not allowed. The remaining expressions are evaluated once and the results stored. The statements between the FOR and NEXT statements are then executed.

When the NEXT statement is encountered, the value after STEP (or 1 if STEP is not used) is added to the loop control variable. If the step value is positive, and the control value is less than or equal to the value of the expression after TO, execution loops to the statement after the FOR statement; otherwise, execution continues with the statement after NEXT. If the step value is negative and the control value is greater than or equal to the expression after TO, execution loops to the statement after the FOR statement; otherwise, execution continues with the statement after NEXT.

The NEXT statement can be used to end more than one FOR statement. In this case, a comma is used. Generally the loop control variables are listed, but this is not required.

FUNCTION identifier [parameter-definition-list] [AS
type]
[statement]*
END FUNCTION

Defines a function.

The identifier is the name of the function, used when it is called. This is followed by the parameter list, if any, and the type returned by the function. The statements that appear between the FUNCTION statement and the END FUNCTION statement are executed as if they were a program, then the last value set for the function is returned to the caller.

The parameter list consists of one or more parameter declarations separated by commas. Each parameter declaration is a variable, optionally followed by AS and a type. If no type is given explicitly, the type is derived from the name of the variable. For example, I% would be an integer.

Arrays, records, pointers, strings and all numeric types are allowed as parameters. Pointers, strings and numeric types are allowed as return values. While records and arrays cannot be returned directly, you can return pointers to either type—but insure that the value is dynamically allocated, and not a local variable!

Inside the function, all parameters work as if they were variables preset to the value passed when the function is called. If the function is called with the name of a variable whose type exactly matches the parameter, and the value is changed inside the function, the value of the original variable is also changed. If the types do not match exactly, or if the function is called with an expression, or if the variable passed is surrounded by parentheses, the original value is not changed.

Variables declared inside the function survive until the function returns, but no longer. If the function is called again, an entirely new set of variables is allocated. Variables from outside the function cannot be accessed from inside, except for parameters, as noted above. Types defined in the main program are, however, available in the function as well as the program.

The value returned by the function is set by assigning a value to the function name. This can be done more than one time; the last value set is the one returned. If no value is set, 0 is returned for numeric functions, a null string for strings, and a null pointer for pointers.

GET ['#' expression ',' [expression] ','] l-value

Reads a single value from the keyboard or a disk file.

If no file is specified, the variable must be a string. A single character is read from the keyboard, converted to a string, and saved in the variable. If no characters have been typed, GET waits for a key before returning.

If a file is given, GET reads binary information from the file. While strings are still treated as single characters, any other data type can be read, including integers, real numbers, records or pointers.

See also PUT, INPUT.

GOSUB line-number

Control jumps to the first line whose number matches line-number. line-number must be an integer constant. When a RETURN statement is encountered, control jumps to the statement after GOSUB.

Subroutines can be nested up to 24 levels deep.

If GOSUB is used in a procedure, the destination line must be in the same procedure.

GOTO `line-number`

Control jumps to the first line whose number matches `line-number`. `line-number` must be an integer constant.

If GOTO is used in a procedure, the destination line must be in the same procedure.

HCOLOR= `expression`

Sets the pen color to one of the 16 colors used on the 320 mode graphics screen. Unless they have been deliberate changed with QuickDraw II calls, the colors are:

Number	Applesoft Color	GSoft BASIC Color
0	black	black
1	green	green
2	violet	purple
3	white	white
4	black	dark gray
5	orange	orange
6	blue	blue
7	white	red
8		beige
9		yellow
10		brown
11		light blue
12		lilac
13		Periwinkle blue
14		light gray
15		dark green

HGR

Starts QuickDraw in 320 graphics mode (if it has not already been started), switches the display to the graphics screen, clears the screen to black, and sets the pen color to white.

See HPLOT, HCOLOR=, and TEXT.

HOME

Clears the text screen and places the cursor at the top left of the screen.

See also HTAB, VTAB.

**HPLOT [expression ',' expression] [TO expression ','
expression]***

If the initial location is given, the pen is moved to that location and a single point is drawn. If one or more TO clauses follow, lines are drawn from the previous point to the location after TO.

See HGR and HCOLOR=.

HTAB expression

Sets the horizontal cursor position on the text screen. This changes the location of the flashing input cursor and the location where the next characters will be written on the text screen.

Columns are numbered from 1 at the left of the screen to 80 at the right. Numbers outside this range are legal, and are converted to the closest existing screen column.

The vertical position is not changed.

See also CSRLIN, HOME, POS, VTAB.

IF expression THEN statement

The expression is evaluated. If the result is not zero, the statement following THEN is executed. If the result of the expression is zero, the statement is not executed.

IF expression GOTO line-number

The expression is evaluated. If the result is not zero, the GOTO is executed, causing processing to skip to the specified line. If the result of the expression is zero, processing continues with the statement following the if statement.

line-number must be an integer constant.

IF expression THEN
[ELSE IF expression]*
[ELSE]
END IF

The expression following IF is evaluated. If the result is not zero, lines between this statement and the first ELSE are executed, and all others are skipped. If the result is zero, expressions in subsequent ELSE IF statements are evaluated, in turn, until one of them results in a non-zero value. When a non-zero expression result is found, the statements from that expression to the following ELSE or ELSE IF are executed, and all others are skipped.

If no expressions evaluate to non-zero, and there is an ELSE clause, the statements between ELSE and END IF are executed. If there is no ELSE clause, no statements are executed.

`INPUT [ '#' expression ',' ] [ string-expression ';' ] l-value [ ',' l-value ]*`

Reads a comma delimited value from the keyboard or a disk file.

If `string-expression` appears, it is used as a prompt string. It is written to standard output without a carriage return, then the input is read. If no prompt string is given and input is from the keyboard, a `?` character is written as a default prompt string.

One or more values can be read with a single INPUT statement by separating the variables with commas. These values can be any number type or a string.

Multiple input values are separated by carriage returns or commas. Any spaces appearing between input values are ignored.

See also LINE INPUT.

`INVERSE`

Subsequent characters are printed using the inverse character set.

See also MOUSETEXT, NORMAL.

`KILL filename`

Deletes the file or directory `filename`.

`[ LET ] l-value '=' expression`

The expression is evaluated, and the result stored in the location indicated by `l-value`.

`LINE INPUT [ '#' expression ',' ] [ string-expression ';' ] l-value [ ',' l-value ]*`

Works almost exactly like INPUT. The exception is how the two handle commas. LINE INPUT ignores them, reading all characters up to the end of a line.

See also INPUT.

`LOADLIBRARY expression`

Loads a user tool from disk.

`expression` is the tool number to load. GSoft BASIC looks for a file with the name UserToolXXX, where XXX is the tool number. It looks first in the local directory, which defaults to the location of the GSoft BASIC interpreter. The local directory can be changed before using LOADLIBRARY using the CHDIR command. If the tool is not found in the local directory, GSoft BASIC looks in the System directory at the path *:System:Tools:UserToolXXX.

See also UNLOADLIBRARY.

LOOP

    See DO.

MKDIR `pathname`

    Creates a new directory with the name `pathname`.

MOUSETEXT

    Subsequent characters are printed using the mouse text character set.

    See also INVERSE, NORMAL.

NAME `filename` **As** `filename`

    Renames the file, directory or disk. The first file name is the original file name, and the second is the new file name.

NEXT

    See FOR.

NORMAL

    Subsequent characters are printed using the normal character set.

    See also INVERSE, MOUSETEXT.

ON `expression` **GOTO** `line-number` **[** `','` `line-number` **]***

    The expression is evaluated and truncated to an integer. Numbering from one, the corresponding line number is selected from the list of line numbers, and execution jumps to that line.

    If there is no corresponding line number, execution continues with the statement after the ON-GOTO statement.

ON `expression` **GOSUB** `line-number` **[** `','` `line-number` **]***

    The expression is evaluated and truncated to an integer. Numbering from one, the corresponding line number is selected from the list of line numbers, and execution jumps to that line. Execution returns to the line after the ON-GOSUB statement when a RETURN statement is executed in the subroutine.

    If there is no corresponding line number, execution continues with the statement after the ON-GOSUB statement.

ONERR GOTO line-number

This statement has no immediate effect. If, later in the program, an error is encountered, execution jumps to the line line-number. From there, you can use ERR and ERL to identify the type and location of the error.

The destination line must appear in the main program, not in a subroutine or function.

See also ERR, ERL, RESUME.

OPEN filename FOR io-kind AS '#' expression [LEN expression]

Opens the file filename.

The file may be opened in any of the following ways by substituting the token shown for the io-kind field.

I/O kind	use
OUTPUT	The file is opened for output. If the file already exists, any old contents are lost.
INPUT	The file is opened for input. The file must already exist, but the file type does not matter. Input starts from the beginning of the file.
APPEND	The file is opened for output. If the file already exists, the old contents are not lost. New information is written after all of the old information.
RANDOM	The file is opened for random access. The file is opened for random access. The LEN field is required; each record written to or read from the file will use that number of bytes.
BINARY	The file is opened for input and output.

The value following # is used in subsequent file commands to identify the opened file. This value can range from 1 to 32767. No two open files may use the same file number, but once the file is closed, the number is available for use by another OPEN statement.

If used, the LEN expression gives the internal buffer size used to cache input and output. This field is required for random access files, and matches the length of one random access record. For all other file types, larger values use more RAM but generally result in faster disk input and output, while lower values save RAM but result in slower input and output.

See also CLOSE, GET, EOF, LOC, LOF, PRINT, PRINT USING, PUT, SEEK.

POKE expression ',' expression

The value of the second expression is converted to an integer. The least significant 8 bits are stored in the memory location specified by the first expression.

See also PEEK, WAIT.

POP

Removes one GOSUB return address from the stack. In effect, this turns the most recent GOSUB into a GOTO.

```
PRINT  [ '#' expression ]
       [ expression
       |   SPC '(' expression ')'
       |   TAB '(' expression ')'
       | ';'
       | ',' ]*
```

The various expressions are evaluated and printed in standard form. See PRINT on page 163 for a complete description of the formats used to print various kinds of information.

Output is normally printed to the text screen. Using # followed by a number sends the output to a file, instead.

SPC is a special function that prints spaces. The expression is evaluated and the specified number of spaces are printed.

TAB is a special function that prints spaces until the tab column is reached. For example, if the expression evaluates to 10, spaces are inserted until column 10 is reached, forcing the next character printed to show up in column 10. Columns number from 1.

The semicolon character is used to separate multiple expressions without printing characters between them. If used at the end of the line, a new line is not started before the next PRINT statement begins to print.

The comma character is used to separate expressions and provide easy column alignment. Spaces are inserted until a space is printed in a column divisible by 16.

```
PRINT  [ '#' expression ] USING format-string ';'
       expression [ ( ',' | ';' ) expression ]* ( ',' | ';' )
```

The format string is printed. Whenever a character that starts a format model is encountered in the format string, one expression from the list that follows the format string is evaluated. The result is rounded, truncated, or converted to fit in the space provided by the format model and printed.

See the PRINT USING on page 169 for a complete description of format models.

Output is normally printed to the text screen. Using # followed by a number sends the output to a file, instead.

```
PUT  '#' expression ',' [ expression ] ',' l-value
```

Writes values to files. It is usually used for binary or random access files, although technically it can be used with any file type.

The first expression is the file number, assigned when the file is opened with OPEN.

The next expression is the location in the file to write the value. For random access files, this is the record number; for all other files, this is a byte number. In both cases, the first value in the file is numbered 1.

`l-value` is the value to write to the file.

READ `l-value [ ',' l-value ]*`

Reads one or more values from a DATA statement. The READ and DATA statements must appear in the same subroutine or must both be in the main program.

See also DATA, RESTORE.

REM `any-ascii-characters`

The REM statement starts a comment. All characters following the command up to the end of the line are ignored

See also !.

RESTORE

Restores the DATA counter, so the next READ statement reads from the first DATA statement in the current subroutine.

See also DATA, READ.

RESUME

RESUME is used in ONERR-GOTO handlers to return to the line where the error occurred. It returns to the start of the offending line.

RETURN

Returns from the most recent GOSUB, transferring control to the statement following the GOSUB statement.

RMDIR `filename`

Deletes the file or directory `filename`.

SEEK `'#' expression ',' expression`

Sets the file so the next read or write occurs at the position indicated by the second expression. For random access files, the file is divided into chunks based on the length specified when the file is opened. For all other file types, the file is divided into bytes. In each case, the first chunk is numbered 1, with the remaining chunks numbered sequentially.

```
SELECT CASE expression
[ CASE case-range [ ',' case-range ]* ]*
[ CASE ELSE ]
END SELECT
```

The expression in the SELECT CASE statement is evaluated. Expressions in the subsequent CASE statements are examined; when an expression is found that matches the original, all statements between the matching CASE statement and the following CASE or END SELECT are executed.

Multiple expressions can be used on a single CASE statement, making it easy to use the same code for several different values. You can also specify a range of values by separating the lowest allowed value and highest allowed value with TO. Unlike many languages, expressions are not limited to scalar quantities; real numbers and strings are allowed as CASE values.

If no matching expression is found, and a CASE ELSE statement is used, statements between the CASE ELSE and END SELECT are executed. If no matching expression is found and no CASE ELSE is used, execution continues with the statement following END SELECT.

```
SETMEM '(' expression ',' expression ')'
```

Sets the size of a memory buffer. The first expression is the memory buffer to set; this is 0 for the variable buffer and 1 for the program buffer. The second expression is the new size for the buffer in bytes.

```
STOP
```

Stops execution of the program. If line numbers are used, the line number is printed.

You can stop a program at a problem point and examine or even change variables, then resume execution with the command CONT.

```
SPEED expression
```

Sets the output speed for characters written to the screen.

A speed of 255 writes characters as rapidly as possible; this is the default. A value of 0 introduces a long delay after each character is written. Intermediate values cause progressively longer or shorter delays.

```
SUB identifier [ parameter-definition-list ]
[ statement ]*
END SUB
```

Defines a subroutine.

The first identifier is the name of the subroutine, used when it is called. This is followed by the parameter list, if any. The statements that appear between the SUB statement and the END SUB statement are executed as if they were a program.

311

The parameter list consists of one or more parameter declarations separated by commas. Each parameter declaration is a variable, optionally followed by AS and a type. If no type is given explicitly, the type is derived from the name of the variable. For example, I% would be an integer.

Arrays, records, pointers, strings and all numeric types are allowed as parameters.

Inside the subroutine, all parameters work as if they were variables preset to the value passed when the subroutine is called. If the subroutine is called with the name of a variable whose type exactly matches the parameter, and the value is changed inside the function, the value of the original variable is also changed. If the types do not match exactly, or if the subroutine is called with an expression, or if the variable passed is surrounded by parentheses, the original value is not changed.

Variables declared inside the subroutine survive until the subroutine returns, but no longer. If the subroutine is called again, an entirely new set of variables is allocated. Variables from outside the subroutine cannot be accessed from inside, except for parameters, as noted above. Types defined in the main program are, however, available in the subroutine as well as the program.

Subroutines are called with the CALL statement.

TEXT

If the graphics screen is visible, the display is shifted back to the text screen. If the text screen is visible, nothing is changed.

TYPE identifier
```
[ ( field-name [ AS type ] )
| ( CASE [ expression ] ) ]+
```
END TYPE

Creates a new type with the name identifier. This type is a record, containing one or more fields. Each field has a distinct type, and the fields may have differing types.

Each field appears on a separate line. field-name is the name of the field, while type is the type for the field. If the type is not specified, the type is taken from the type character appearing at the end of the identifier, just as the type is derived for a variable. For example, I% is an integer field, while R is a single-precision real number field.

TYPE identifier AS type

Creates a new type with the name identifier.

UNLOADLIBRARY expression

Unloads the specified user tool, freeing the RAM used by the tool.

See also LOADLIBRARY.

VTAB expression

Sets the vertical cursor position on the text screen. This changes the location of the flashing input cursor and the location where the next characters will be written on the text screen.

Lines are numbered from 1 at the top of the screen to 24 at the bottom. Numbers outside this range are legal, and are converted to the closest existing screen line.

The horizontal position is not changed.

See also CSRLIN, HOME, HTAB, POS.

WAIT expression ',' expression

A logical and is performed between the byte at the memory address specified by the first expression and the value specified by the second expression. If the result is not zero, execution continues with the next statement. If the result is zero, the process repeats.

See also PEEK, POKE.

WHILE expression
WEND

The expression is evaluated. If it is not zero, the statements between WHILE and WEND are executed, and the process repeats. If the expression evaluates to zero, execution continues with the statement after WEND.

Functions

ABS '(' expression ')'

Returns the absolute value of expression.

ASC '(' string-expression ')'

Returns the ASCII numeric value for the first character in string-expression. ASC returns 0 if there are no characters in string-expression.

ATN '(' expression ')'

Returns the arc-tangent of expression.

CDBL '(' expression ')'

Converts expression to a double precision floating-point value.

CHR$ '(' expression ')'

Returns a string consisting of a single character whose ASCII value is expression.

CINT '(' expression ')'

Converts expression to an integer value.

CLNG '(' expression ')'

Converts expression to a long integer value.

COS '(' expression ')'

Returns the cosine of expression.

CSNG '(' expression ')'

Converts expression to a single precision floating-point value.

CSRLIN

Returns the line number where the next character will be printed. Lines are numbered from 1 to 24.

See also HTAB, POS, VTAB.

CURDIR$

Returns the name of the current directory.

DIR$ ['(' file-name ')']

Returns file names from a directory.

The first call should specify a parameter. This can be the name of a specific file or the wildcard character "*". Full or partial path names may be used. DIR$ will return the name of the file if there is a file by the given name, or the name of the first file in the directory if the wildcard character is used.

If the wildcard character is used, subsequent calls may be made without a parameter. These calls return the names of the remaining files in the directory. When all files have been returned, DIR$ returns an empty string.

EOF '(' expression ')'

Returns 0 if there is unread information in a file, and -1 if there is not.

See also GET, LOC, LOF, OPEN, SEEK.

ERL

Used in an ONERR-GOTO handler, ERL returns the line number of the line where the error occurred. If the line where the error occurred does not have a number, ERL returns 0.

ERR

Used in an ONERR-GOTO handler, ERR returns an error number indicating the type of the error.

See Appendix A for a list of the error numbers and their meanings.

EXP '(' expression ')'

Returns the exponent of expression.

FRE '(' expression ')'

Forces garbage collection on the string space, then returns the number of bytes remaining for variables, subroutine stacks, and strings.

The expression value should be 0, but is actually ignored.

INT '(' expression ')'

Returns the integer part of a number.

Unlike CINT, INT does not convert the result to an integer. Instead, the type of the result matches the type of the argument. For integer and long arguments, the expression result is returned unchanged. For single and double precision real numbers, the value returned is the largest integer that is less than or equal to the value of the expression.

LEFT$ '(' string-expression ',' expression ')'

Returns expression characters from the beginning of a string. If the length of the string is less than expression, the entire string is returned.

LEN '(' string-expression ')'

Returns the number of characters in string-expression.

LOC **`(`** **expression** **`)`**

Returns the number of records or bytes that have been read from or written to a file so far. At the beginning of a file, LOC returns 0.

LOF **`(`** **expression** **`)`**

Returns the number of records or bytes in a file.

LOG **`(`** **expression** **`)`**

Returns the natural logarithm of expression.

MID$ **`(`** **string-expression** **`,`** **expression** **`,`** **expression** **`)`**

Returns characters from any position in a string. The first expression is the index of the first character to return, numbering from 1. The second expression is the number of characters to return.

If there are not enough characters, all available characters are returned. If the character index is larger than the length of the string, a string with no characters is returned.

NIL

Returns a pointer value that is type compatible with all pointers, and that indicates a pointer which is not pointing to any memory location.

All pointers are initially set to NIL.

The ordinal value for NIL is 0.

PEEK **`(`** **expression** **`)`**

Returns the value of the byte located at the address expression.

See also POKE, WAIT.

POS **`(`** **expression** **`)`**

Returns the column where the next character will be printed. Columns are numbered starting from 1.

See also CSRLIN, HTAB, VTAB.

RIGHT$ **`(`** **string-expression** **`,`** **expression** **`)`**

Returns expression characters from the end of a string. If the length of the string is less than expression, the entire string is returned.

RND '(' expression ')'

Returns a random single precision number that is greater than or equal to 0.0 and less than 1.0.

If `expression` is a negative number, RND resets the random number generator seed using the argument as a seed value.

If `expression` is zero, RND returns the same value it returned on the previous call.

If `expression` is a positive number, a pseudo-random number is returned.

SGN '(' expression ')'

Returns -1 if `expression` is negative, 0 if `expression` is zero, and 1 if `expression` is positive.

SIN '(' expression ')'

Returns the sine of `expression`.

SIZEOF '(' (type | identifier) ')'

Returns the size required to store one value of a given type, or the size used by the variable `identifier`. The size is given in bytes.

SQR '(' expression ')'

Returns the square root of `expression`.

STR$ '(' expression ')'

Converts a numeric value to a string using the same formatting rules as the PRINT statement. See also VAL.

TAN '(' expression ')'

Returns the tangent of `expression`.

TOOLERROR

Returns the error code from the most recent tool, user tool or GS/OS call. A value of zero indicates there was no error.

VAL `'(' string-expression ')'`

Converts a string that represents a number into a value using the same rules as READ and INPUT.

See also STR$.

VERSION

Returns the GSoft BASIC version number encoded as a long integer. The format is VVMMBBTRR, where

VV   Major release number.

MM   Minor release number.

BB   Bug fix release number.

T   Release type; 0 for commercial, 1 for development, 2 for alpha and 3 for beta.

RR   Release number for the current type.

BNF Used in This Appendix

```
add-expression    ::=  mul-expression  [  addop  mul-expression
]*

addop  ::=  '+'  |  '-'

and-expression    ::=  relational-expression  [  AND
relational-expression  ]*

array-subscript  ::=  '('  expression  [  ','  expression  ]*
')'

case-range  ::=  expression  [  TO  expression  ]

digit  ::=  '0'..'9'

exp-expression  ::=  term  [  '^'  term  ]*

exponent-char  ::=  'e'  |  'E'  |  'd'  |  'D'

expression  ::=  and-expression  [  OR  and-expression  ]*

field-name  ::=  identifier

filename  ::=  string-constant

format-string  ::=  string-constant

function-name  ::=  identifier

hexadecimal-constant  ::=  '$'  [  '0'..'9'  |  'a'..'f'  |
'A'..'F'  ]+
```

```
identifier ::= letter [ letter | digit ] type-character

integer-constant ::= [ '0'..'9' ]+

io-kind ::= OUTPUT | INPUT | APPEND | RANDOM | BINARY

l-value ::= identifier [ array-subscript | '^' [ l-value
  ] | ',' l-value | '^' l-value ]

letter ::= 'A'..'Z' | 'a'..'z'

line-number ::= integer-constant

mul-expression ::= exp-expression [ mulop exp-expression
  ]*

mulop ::= '*' | '/'

parameter-definition ::= identifier [ As type ] [ '('
  ')' ]

parameter-definition-list ::= [ '(' parameter-definition
  [ ',' parameter-definition ]* ')' ]

parameter ::= '(' expression ')' | expression

parameter-list ::= [ '(' parameter [ ',' parameter ]*
  ')' ]

pathname ::= string-constant

pointer-name ::= identifier

read-constant ::= [ integer-constant ] [ '.' [ integer-
  constant ] ] [ exponent-char [ addop ] integer-
  constant ]

record-name ::= identifier

relational-expression ::= add-expression [ relop add-
  expression ]*

relop ::= '=' | '<' | '>' | '<=' | '=<' | '>=' | '=>'
  '<>' | '><' |

subscript ::= '(' expression [ ',' expression ]* ')'

statement ::= {any statement from those listen in this
  appendix}

string-constant ::= '"' {any characters except '"' } '"'

string-expression ::= expression
```

```
term       ::= addop term
           |   '@'  l-value
           |   NOT  term
           |   real-constant
           |   integer-constant
           |   hexadecimal-constant
           |   string-constant
           |   identifier  array-subscript
           |   record-name  '.'  l-value
           |   pointer-name  '^'
           |   pointer-name  '^'  '.'  l-value
           |   type-name  '('  expression  ')'
           |   function-name  parameter-list
           |   FN  function-name  parameter-list

type       ::= [ POINTER TO ] type-name

type-character ::= '~'  |  '%'  |  '&'  |  '!'  |  '#'  |  '$'

type-name  ::= BYTE  |  INTEGER  |  LONG  |  SINGLE  |  DOUBLE  |
               STRING  |  identifier
```