

2019 Edition

BASIC

Techniques and Utilities

Ethan Winer

Entire contents of this text and software Copyright © 1994 Ethan Winer.

Author's Note

This is a digital re-design of the disk version of *PC Magazine BASIC Techniques and Utilities*, which was originally published by Ziff-Davis Press in Emeryville, CA. When Ziff-Davis Press decided it was no longer profitable for them to continue printing it, they returned the rights to me. This digital version of my book is provided free as a service to the programming community. You are welcome to use any of the code fragments or complete programs in any way you see fit for no charge, including for commercial applications. However, the author retains all copyrights for the text and the programs. You may share this book and the accompanying programs with others, but only if you distribute the book and sources in its entirety, as originally uploaded by me to my web page (<http://www.ethanwiner.com>).

While I should not have to belabor the obvious: All of this software and the accompanying text are provided "as is", with no warranty expressed or implied. The author is not liable for any damages whatsoever, including incidental or consequential. Use this information at your own risk. If you wipe out your hard disk or CMOS memory, I am not responsible!

Special thanks are owed to Ariella Baston who spent more than half a year lovingly formatting every single page, creating a linked Table Of Content, and so much more.

You will notice a few comments here and there that were added to this digital version of my book only, and relate to VB/DOS, which was not covered in the original printed version. Since I do not use VB/DOS on a regular basis, I can't guarantee that all of the VB/DOS differences and features are documented completely. In most cases, however, the information about BASIC PDS applies equally to VB/DOS.

I will happily provide support for this book and answer questions as time permits via email sent to ethanw@ethanwiner.com. However, I much prefer to answer questions in the public forums rather than through e-mail, because public messages let others benefit from the answers.

– Ethan Winer

Notes on the 2019 recreation of *BASIC Techniques and Utilities*

By Ariella Baston

I didn't discover *BASIC Techniques and Utilities* until 2018, and I wish I had seen its original print in 1992. I feel it's the best programming book on QuickBASIC, and writing software for classic PCs. The depth of knowledge and practical application suggested in this work can light fires of discipline and craftsmanship no matter the decade or language. This book will remind you to care deeply about what your software is doing in its environment, and that there's always more you can do to improve its execution.

One of my hobbies is document conversion and archival, from modern works retrofitted for the past, to past works rejuvenated for the present. When I learned that this book's original publishing design files were lost, I felt impassioned to recreate them and give the book a new good-looking digital life. I reached out to Ethan and he was happy to support the effort. Ethan provided text files of the book's copy, and I consulted scans of the original 1992 printing to consider how it had once been put together, and what the images and tables looked like.

The work took 8 months of time spread across lunch breaks, evenings and weekends (I work full-time as a Business Analyst), and it was some of the best fun I've had in retro-computing. Finishing every chapter felt like a love-letter to my past in software development. I hope you enjoy the new edition as much as I enjoyed putting it together!

The recreation work involved:

- Assembly of all source text files into a word processor, including the fun cleanup of correcting CRLF to wrapping text, correcting paragraph breaks, cleaning up tabs and trailing white space, converting double spaces to singles, hyphens to em dashes, fixed-width preservation of code sections, applying styles, sections, and more.
- Recreation of all figures, tables, and screenshots. These took the most time, and involved a lot of extra software like virtual machines, MS-DOS apps, vector art apps, bitmap art apps, and spreadsheets.
- A new cover.
- New Part and Chapter layouts.
- Creation of new digital assets in modern and more accessible document formats like SVG, ODT and PNG.
- Fixing mistakes that were in the original book.

Software used: LibreOffice, GIMP, Inkscape, FreeDOS, Virtualbox, Linux Mint.

Hardware used: Unicomp Classic 101 keyboard, Acer laptop.

Feel free to say hello via email! I'd love to hear of your adventures with programming and retro-computing:

ariella@allthethings.ca



In memory of my father, Dr. Frank Winer

Acknowledgements

Many people helped me during the preparation of this book. First and foremost I want to thank my publisher, Cindy Hudson, for her outstanding support and encouragement, and for making it all happen. I also want to thank "black belt" editor Deborah Craig for a truly outstanding job. Never had I seen someone reduce a sentence from 24 words to less than half that, and improve the meaning in the process. Unfortunately, readers of this disk version are unable to benefit from Deborah's excellent work.

Several other people deserve praise as well:

Don Malin for his programming advice, and for eliminating all my GOTO statements.

Jonathan Zuck for his contribution on database and network programming, including all of the dBASE file access routines.

Paul Passarelli for unravelling the mysteries of floating point emulation, and for sharing that expertise with me.

Philip Martin Valley for his research and examples showing how to read Lotus 1-2-3 binary files.

Jim Mack for his skillful proof-reading of my manuscript, and countless good ideas.

My wife Elli for her support and encouragement during the eight long months it took to write this book.

Table of Contents

PREFACE.....	xiv
Introduction.....	xiv
Conventions used in this book.....	xv
How this book is organized.....	xv
A brief history of Microsoft Compiled BASIC.....	xviii
PART 1 Under the Hood	
1 An Introduction to Compiled BASIC.....	1
Compiler Fundamentals.....	1
Data Storage.....	2
Assembly Language Considerations.....	3
Compiler Optimization.....	6
Event and Error Checking.....	6
Event Trapping.....	7
Error Trapping.....	8
Overflow Errors.....	9
Compiler Optimization.....	10
The BASIC Run-time Libraries.....	11
Granularity.....	13
2 Variables and Data.....	15
Data Basics.....	15
Integers and Long Integers.....	16
Bits 'N' Bytes.....	17
Memory Addresses and Pointers.....	18
Integer Storage.....	19
Floating Point Values.....	20
Dynamic Strings.....	22
Far Strings in BASIC PDS.....	25
Fixed-Length Strings.....	27
User-Defined TYPE Variables.....	28
Arrays Within Types.....	30
Static versus Dynamic Data.....	30
Dynamic Arrays.....	31
Far Data Versus Near Data.....	34
Assessing Memory with FRE().....	35
SETMEM and STACK.....	36
VARPTR, VARSEG, and SADD.....	38
Constants.....	42
Passing Numeric Constants to a Procedure.....	44
String Constants.....	46
Passing String Constants to a Procedure.....	46

Unusual String Constants.....	47
Wouldn't It Be Nice If.....	48
Bit Operations.....	49
3 Programming Methods.....	52
Control Flow.....	52
The Dreaded GOTO.....	53
FOR/NEXT Loops.....	56
IF/THEN/ELSE and SELECT Case.....	57
ON GOTO and ON GOSUB Statements.....	66
A Comparison of Subroutine Methods.....	68
Subroutines Versus Functions.....	74
GOSUB Routines.....	75
Subprograms.....	75
DEF FN Functions.....	77
Formal Functions.....	78
Static Versus Non-static Procedures.....	78
Recursion.....	80
Passing Parameters To Procedures.....	82
Modular Programming.....	84
Include Files.....	85
Quick Libraries.....	85
Error and Event Handling.....	86
Event Handling here.....	88
Programming Style.....	89
PART 2 Programming Hands On	
4 Debugging Strategies.....	94
Common Programming Errors.....	94
Logic Errors.....	96
Understanding BASIC's Quirks.....	98
Debugging and Testing Techniques.....	100
Using the QB and QBX Editing Environments.....	101
Step and Trace Debugging.....	102
Watch Variables and Breakpoints.....	103
Using /d to Detect Errors.....	105
Advanced Debugging.....	106
Creating an Assembly Language Source Listing.....	106
Using Microsoft CodeView.....	108
5 Compiling and Linking.....	117
An Overview of Compiling and Linking.....	117
Compiling.....	120
Compiler Options.....	123
Compiler Metacommands.....	137

Linking.....	139
Link Options.....	141
Stub Files (PDS and later).....	147
Quick Libraries.....	148
Creating a Quick Library.....	150
Response Files.....	151
Linking With Batch Files.....	152
Linking With Overlays (PDS and VB/DOS Pro Only).....	153
Other LINK Details.....	155
Maintaining Libraries.....	156
LIB Options.....	159
Useful BC, LINK, and LIB Environment Parameters.....	161
6 File and Device Handling.....	163
Disk File Fundamentals.....	163
Disk-Like Devices.....	166
Exploring Data Files.....	168
File Buffers.....	168
File Access Methods.....	170
Files Manipulation Statements.....	173
Opening and Closing Files.....	173
Reading and Writing Data.....	175
Sequential Output.....	175
Sequential Input.....	178
Random Access.....	179
Binary Access.....	185
Navigating Your Files.....	194
Advanced File Techniques.....	198
Speeding Up File Access.....	199
Processing Large Files.....	203
Minimizing Disk Usage.....	211
Avoiding BASIC's Limitations.....	213
Advanced Device Techniques.....	218
The Printer Device.....	218
The Screen Device.....	219
The Keyboard Device.....	226
Redirection.....	231
7 Network and Database Programming.....	233
Data Files versus Data Management.....	233
Data-Driven Programming.....	234
The dBASE III File Structure.....	236
dBASE File Access Tools.....	238
dBASE Utility Programs.....	243
Limitations of the dBASE III Structure.....	248

Indexing Techniques.....	249
Sorted Lists.....	249
Expression Evaluation.....	251
Relational Databases.....	251
SQL: The Black Box.....	254
Programming for a Network.....	255
File Sharing and Locking.....	255
Record Locking.....	256
Additional Network Considerations.....	258
Operating System Confirmation.....	259
Third-Party Database Tools.....	260
AJS Publishing's db/LIB.....	260
Novell's Btrieve.....	261
CDP Consultants' Index Manager.....	261
Ocelot.....	261
8 Sorting and Searching.....	263
Sorting Fundamentals.....	264
Indexed Sorts.....	266
Data Manipulation Techniques.....	268
The Quick Sort Algorithm.....	273
An Assembly Language Quick Sort.....	276
Sorting on Multiple Keys.....	280
Indexed Sorting on Multiple Keys.....	287
Sorting Files.....	289
Searching Fundamentals.....	296
Binary Searches.....	301
Numeric Arrays.....	303
Soundex.....	305
Linked Data.....	308
Array Element Insertion and Deletion.....	310
PART 3 Beyond BASIC	
9 Program Optimization.....	314
Programming Shortcuts and Speed Improvements.....	314
Predefining Variables.....	315
Integer and Long Integer Assignments.....	317
Short Circuit Expression Evaluation.....	320
Miscellaneous Tips and Techniques.....	322
Formatting and Rounding.....	322
String Tricks and Minimizing Parameters.....	323
Word Wrapping.....	325
Unusual Ways to Access Display Memory.....	326
Rebooting a PC.....	327
Integer Values Greater Than 32K.....	328

Benchmarking.....	329
10 Key Memory Areas in the PC.....	334
Improving PEEK and POKE.....	334
Low Memory Addresses.....	335
Communications Port Addresses.....	337
Printer Port Addresses.....	338
System Data.....	338
Keyboard Data.....	341
The Keyboard Buffer.....	344
Diskette Data.....	346
Display Adapter Data.....	347
System Timer Data.....	348
Printer Timeout Data.....	349
EGA and VGA Data.....	351
Miscellaneous Data.....	351
Input/Output Ports.....	353
Keyboard Ports.....	355
11 Accessing DOS and BIOS Services.....	357
What is an Interrupt?.....	357
Registers.....	359
Accessing the BIOS.....	361
The BIOS Video Interrupt.....	362
Accessing DOS Interrupts.....	364
Accessing the Default Drive.....	365
Determining if a File Exists.....	367
The Carry Flag.....	369
Improving On Interrupt.....	369
Obtaining the Current Directory.....	369
Reading Files and Directory Names.....	371
Managing Files.....	378
Accessing The Mouse.....	391
Mouse Services.....	392
Determining if a Mouse is Present.....	400
Controlling the Text Cursor.....	402
Reading the Mouse Buttons and Cursor Position.....	403
Changing the Mouse Cursor Shape.....	404
Controlling the Mouse Cursor Position and Range.....	405
Accessing the Mouse Driver.....	406
Accessing Expanded Memory.....	406
EMS Services.....	408
Determining if EMS is Present.....	413
Determining Available EMS Memory.....	414
Storing and Retrieving Data.....	414

Detecting EMS Errors.....	415
Suggested Enhancements.....	415
12 Assembly Language Programming.....	417
As Easy as BASIC.....	418
Spaghetti Code?.....	420
The Registers.....	421
Variables in Assembly Language.....	424
Calculations in Assembly Language.....	426
String Processing Instructions.....	427
Movs and Cmps.....	428
Scas and Stos.....	428
Repeating String Operations.....	428
The Stack.....	430
Passing Parameters.....	431
Procedures in Assembly Language.....	435
Simplified Directives.....	436
Calling Interrupts.....	438
DOS and BIOS Services.....	438
Accessing BASIC Strings in Assembly Language.....	440
BASIC Near Strings.....	440
Conditional Jump Instructions.....	442
DOS Strings.....	444
BASIC Fixed-Length Strings.....	444
Far Strings in BASIC PDS.....	446
Accessing Arrays.....	447
Dynamic Arrays.....	448
Huge Arrays.....	450
Assembler Functions.....	452
String Functions.....	452
Far String Functions.....	454
Floating Point Functions.....	455
Floating Point Comparisons.....	458
Exploiting MASM's Features.....	460
Conditional Assembly.....	460
Comment Blocks.....	461
Quoted Strings.....	462
Length and Address Self-Calculation.....	462
Defining Data Structures.....	463
Minimizing DGROUP Usage.....	463
Local Variables.....	464
Storing Data in the Code Segment.....	465
MASM Macros.....	468
Segment Naming.....	471
Accessing BASIC Internals.....	471

BASIC's Internal Data.....	472
BASIC's Internal Routines.....	474
Finding Other Routines.....	477
Reading the Array Descriptor.....	477
Handling Interrupts.....	481
Debugging With CodeView.....	484
MASM 6.0 Enhancements.....	485
Improved Assembly Optimizations.....	486
New Simplified Directives.....	487
Tricks of the Trade.....	488
Minimize Code to Access Parameters.....	488
Byte Savers.....	489
Cycle Savers.....	492
Miscellaneous Techniques.....	493
Appendix.....	497
Overview of the Accompanying Files.....	497
Starting BASIC.....	497
Linking.....	498

PREFACE

Introduction

BASIC has always been the most popular language for personal computers. It is easy to learn and use, extremely powerful, and some form of BASIC is included for free with nearly every PC. Although BASIC is often associated with beginners and students, it is in fact ideally suited for a wide range of programming projects. Because it offers the best features of a high-level language coupled with direct access to DOS and BIOS system services, BASIC is fast becoming the language of choice for beginners and professional developers alike. This book is about power programming using Microsoft compiled BASIC.

It is intended for people who already possess a fundamental understanding of BASIC programming concepts, but want to achieve the best performance possible from their BASIC compiler.

Power programming is knowing when and how to use BASIC commands such as CALL INTERRUPT, VARSEG and VARPTR, and even PEEK and POKE effectively. It involves understanding the PC's memory organization sufficiently to determine how much stack space is needed for a recursive sub program or function. A power programmer knows how to translate a time-critical portion of a BASIC program into assembly language when needed. Finally, and perhaps most importantly, power programming means knowing enough about BASIC's internal operation to determine which sequence of instructions is smaller or faster than another.

This book will show you how to go beyond creating programs that merely work. Because it explains how the compiler operates and how it interacts with the BASIC runtime language library, this book will teach you how to write programs that are as small and fast as possible. Although the emphasis here is on Microsoft QuickBASIC and the BASIC Professional Development System (PDS), much of the information will apply to other BASIC compilers such as Power Basic from Spectra Publishing.

Despite what you may have read, BASIC is the most capable and easy to learn of the high-level languages. Modern BASIC compilers are highly optimizing, and can thus create extremely efficient executable programs. In addition, you can often achieve with just a few BASIC statements what would take many pages of code in another high-level language. Moreover, beginners can be immediately productive in BASIC, while serious programmers have a wealth of powerful capabilities at their disposal.

Microsoft BASIC has many capabilities that are not available in any other high-level language. Among these are dynamic (variable-length) strings, automatic memory allocation and heap management, built-in support for sophisticated graphics, and interrupt-driven communications. Add to that huge arrays, network file handling, music and sound, and protection against inadvertently overwriting memory, and you can see why BASIC is so popular.

This book aims to provide intermediate to advanced programmers with information that is not available elsewhere. It does not, however, cover elementary topics such as navigating the QuickBASIC editor, loading and saving files, or using the Search and Replace feature. That information is readily available elsewhere. Rather, it delves into previously uncharted territory, and examines compiled BASIC at its innermost layer. Besides the discussions and programs in the text, this book includes a companion ZIP file that contains all of the subroutines and other code listed in this book, including several useful utilities. Installing these programs is described in the Appendix.

Conventions used in this book

This book uses the terms QuickBASIC and QB to mean the Microsoft QuickBASIC4.x and 7.x editing environments. BC and Compiler indicate the BC.EXE command-line compiler that comes with QuickBASIC, Microsoft BASIC PDS, and the now-discontinued BASIC 6.0. When a distinction is necessary, QBX will refer to the QuickBASIC Extended editor that comes with the BASIC Professional Development System (PDS). In most cases, the discussions will be the same for all of these versions of BASIC. When a difference does occur, the PDS and QBX exceptions will be indicated as call-out boxes, or [in square brackets].

Code sections are presented in fixed width format.

Technical terms that are introduced and useful to learn are *italicized*.

How this book is organized

This book is divided into parts, and each part contains several chapters that discuss a specific aspect of BASIC programming. You needn't fully understand an entire chapter before moving on to the next one. Each topic will be covered in great depth, and in many cases you will want to return to a given chapter as your knowledge and understanding of the subject matter increases.

PART 1 Under the Hood and its three chapters describe in detail how your BASIC source code is manipulated throughout the compiling and linking process.

Chapter 1 An Introduction to Compiled BASIC presents an overview of compilers in general, and BASIC compilers in particular. It discusses what BASIC compilers are all about and how they work, and how the compiled code that is generated interacts with routines in the runtime libraries.

Chapter 2 Variables and Data discusses variables, constants, and other program data, and how they fit within the context of the PC's memory organization. This chapter also covers bit manipulation using AND, OR, and XOR.

Chapter 3 Programming Methods examines the various control flow methods available in BASIC, showing which statements and procedure constructs are appropriate in different situations. In particular, you will learn the relative advantages and disadvantages of each method, based on their capabilities, code size, and speed.

PART 2 Programming Hands On examines programming techniques, and shows specific examples of writing effective code and also making it work.

Chapter 4 Debugging Strategies explores program debugging using the facilities built into the QuickBASIC editing environment, as well as the CodeView utility that comes with Microsoft BASIC PDS. This chapter also discusses common programming problems, along with the appropriate solutions.

Chapter 5 Compiling and Linking explains compiling and linking, both from within the QB environment, and directly from DOS. A number of compiler options are inadequately documented by Microsoft, and each is discussed here in great detail. A thorough discussion of the LIB.EXE utility program included with BASIC explains how libraries are manipulated and organized.

Chapter 6 File and Device Handling covers all aspects of file and device handling, and discusses the many different ways in which data may be read and written. The emphasis here is on speeding file handling as much as possible, and storing data on disk efficiently. Because input/output (I/O) devices are accessed similarly, they too are described here in detail.

Chapter 7 Network and Database Programming explains the basics of writing database and network applications, and discusses file locking strategies using practical programming examples. A series of subroutines show how to read and write files using the popular dBASE format, and these may be incorporated into programs that you write.

Chapter 8 Sorting and Searching shows how to sort and search array data as quickly as possible. Several methods are examined including conventional and indexed sorting, and many useful subroutines are presented.

The final part, **PART 3 Beyond BASIC**, includes information that is rarely covered in books about BASIC. Its three chapters go far beyond the information provided in any of the Microsoft manuals.

Chapter 9 Program Optimization covers several general optimization techniques that reduce the space of programs and make them run faster. The material is organized into three principle categories: programming shortcuts and speed improvements, miscellaneous tips and techniques, and benchmarking.

Chapter 10 Key Memory Areas in the PC identifies many of the key memory areas in the PC, and shows when and how they can be manipulated in a BASIC program.

Chapter 11 Accessing DOS and BIOS Services presents an in-depth discussion of accessing DOS and BIOS services using CALL INTERRUPT. These services offer a wealth of functionality that BASIC cannot otherwise provide directly.

Chapter 12 Assembly Language Programming is an introduction to assembly language, from a BASIC programmer's perspective. This chapter presents many useful subroutines, and includes a thorough discussion of how they work.

Finally, the *Appendix* describes the additional source files that accompany this book.

A brief history of Microsoft Compiled BASIC

In March of 1982, IBM released the first BASIC compiler for the IBM PC. This compiler, BASCOM 1.0, was written by Microsoft for IBM using code and methods developed by Bill Gates, Greg Whitten, and others. Although Microsoft had already written BASIC compilers for the Apple II and CP/M computers, BASCOM 1.0 was the most powerful they had produced so far. Compared to the Microsoft BASIC interpreters available at that time, BASCOM 1.0 offered many additional capabilities, and also an enormous increase in program execution speed. Line numbers were no longer mandatory, program statements could exceed 255 characters, and a single string could be as long as 32,767 characters. Further, assembly language subroutines could be linked directly to a compiled BASIC application. Over the next few years, Microsoft continued to enhance the compiler, and in 1985 it was released by IBM as BASCOM 2.0. This version offered many improvements over the older BASCOM 1.0. Among the most important were multi-line DEF FN functions, dynamic arrays, descriptive line labels (as opposed to numbers), network record locking, and an ISAM file handler. With named subroutines programmers were finally able to exceed the 64K code size limitation, by writing separate modules that could then be linked together. The inclusion of subroutine parameters—long overdue for BASIC—was an equally important step toward fostering structured programming techniques in the language.

At the same time that IBM released BASCOM 2.0, Microsoft offered essentially the same product as QuickBASIC 1.0, but without the ISAM file handler. However, there was one other big difference between these compilers: QuickBASIC 1.0 carried a list price of only \$99. This low price was perhaps the most important feature of all, because high-performance BASIC was finally available to everyone, and not just professional developers.

Encouraged by the tremendous acceptance of QuickBASIC 1.0, Microsoft quickly followed that with QuickBASIC version 2.0 in early 1986. This important new release added an integrated editing environment, as well as EGA graphics capabilities. The editor was especially welcome, because it allowed programs to be developed and tested very rapidly. The environment was further enhanced with the advent of Quick Libraries, which allowed assembly language subroutines to be easily added to a BASIC program. Quick Libraries also helped launch the start of a new class of BASIC product: third-party add-on libraries.

In early 1987 Microsoft released the next major enhancement to QuickBASIC, version 3.0. QuickBASIC 3.0 included a limited form of step and trace debugging, as well as the ability to monitor a variable's value continuously during program execution. Also added was support for the EGA's 43-line text mode, and several new language features. Perhaps most impressive of the new features was the control flow statements DO and LOOP, and SELECT CASE. Beyond merely providing a useful alternative to the IF statement, these constructs also let the compiler generate more efficient code.

Also added with version 3.0 was optional support for an 8087 numeric coprocessor. In order to support a coprocessor, however, Microsoft had to abandon their own proprietary numeric format.

Both the Microsoft and IEEE methods for storing single and double precision numbers use four bytes and eight bytes respectively, but the bits are organized differently. Although the IEEE format which the 8087 requires is substantially slower than Microsoft's own, it is the current standard. Therefore, a second version of the compiler was included solely to support IEEE math.

By the time QuickBASIC 4.0 was announced in late 1987, hundreds of thousands of copies of QuickBASIC were already in use world-wide. With QuickBASIC 4.0, Microsoft had created the most sophisticated programming environment ever seen in a main-stream language: the threaded p-code interpreter. This remarkable technology allowed programmers to enjoy the best features of an interpreted language, but with the execution speed of a compiler.

In addition to an Immediate mode whereby program statements could be executed one by one, QuickBASIC 4.0 also supported program break-points, monitoring the value of multiple variables and expressions, and even stepping backwards through a program. This greatly enhanced the debugging capabilities of the language, and increased programmer productivity enormously.

Also new in QuickBASIC 4.0 was support for inter-language calling. Although this meant that a program written in Microsoft BASIC could now call subroutines written in any of the other Microsoft languages, it also meant that IEEE math was no longer an option—it became mandatory. When a QuickBASIC 4.0 program was run on a PC equipped with a coprocessor, floating point math was performed very quickly indeed. However, it was very much slower on every other computer! This remained a sore point for many BASIC programmers, until Microsoft introduced BASIC 6.0 later that year. That version included an alternate math library that was similar to their original proprietary format.

Also added in QuickBASIC 4.0 were huge arrays, long (4-byte) integer variables, user-defined TYPE variables, fixed-length strings, true functions, and support for CodeView debugging. With the introduction of huge arrays, BASIC programmers could create arrays that exceeded 64K in size, with only a few restrictions. TYPE variables let the programmer define a composite data type comprised of any mix of BASIC's intrinsic data forms, thus adding structure to a program's data as well as to its code. The newly added FUNCTION procedures greatly improved on BASIC's earlier DEF FN-style functions by allowing recursion, the passing of TYPE variables and entire arrays as arguments, and the ability to modify an incoming parameter.

Although BASIC 6.0 provided essentially the same environment and compiler as QuickBASIC 4.0, it also included the ability to create programs that could be run under OS/2. Other features of this release were a utility program to create custom run-time libraries, and a copy of the Microsoft Programmer's Editor. The custom run-time utility was particularly valuable, since it allowed programmers to combine frequently-used subroutines with the BRUN.EXE language library, and then share those routines among any number of chained modules.

QuickBASIC 4.5 was introduced in 1988, although the only major enhancement over the earlier 4.0 version was a new help system and slightly improved pull-down menus. Unfortunately, the new menus required much more memory than QuickBASIC 4.0, and the "improved" environment reduced the

memory available for programs and data by approximately 40K. To this day, many programmers continue to use QuickBASIC 4.0 precisely because of its increased program capacity.

In answer to programmer's demands for more string memory and smaller, more efficient programs, Microsoft released the BASIC Professional Development System version 7.0 in late 1989. This was an enormous project even for a company the size of Microsoft, and at one point more than fifty programmers were working on the new compiler and QBX environment. PDS version 7.0 finally let BASIC programmers exceed the usual 64K string memory limit, albeit with some limitations.

Other features introduced with that version were an ISAM file handler, improved library granularity, example tool box packages for creating simple graphics and pull-down menus, local error handling, arrays within TYPE variables, and greatly improved documentation. Because the QBX editor uses expanded memory to store subprograms and functions, much larger programs could be developed without resorting to editing and compiling outside of the environment.

Sixth months later PDS version 7.1 was released, with the long-overdue ability to redimension an array but without destroying its contents. Also added in that version were support for passing fixed-length string arrays to subprograms and functions, and an option to pass parameters by value to BASIC procedures. Although the BYVAL option had been available since QuickBASIC 4.0, it was usable only with subroutines written in non-BASIC languages. With this mechanism, BASIC can now create more efficient object code than ever before.

```
Just as the initial print version of this book was
being completed, Microsoft released Visual Basic for
DOS. Although this book does not address VB/DOS
specifically, most of the information about BASIC PDS
applies to VB/DOS. One notable exception is that VB/DOS
supports far strings only, where BASIC PDS lets you
specify either near strings or far. Because far strings
are stored in a separate "far" area of DOS memory, it
takes slightly longer to access those strings.
Therefore, a VB/DOS program that is string-intensive
will not be as fast as an equivalent compiled with
QuickBASIC or with PDS near strings. This book also
does not cover the pseudo event-driven forms used by
VB/DOS.
```



PART 1
Under the Hood

1

An Introduction to Compiled BASIC

This chapter explores the internal workings of the BASIC compiler. Many people view a compiler simply as a "black box" which magically transforms BASIC source files into executable code. Of course, magic does not play a part in any computer program, and the BC compiler that comes with Microsoft BASIC is no exception. It is merely a program that processes data in the same way any other program would. In this case, the data is your BASIC source code.

You will learn here what the BASIC compiler does, and how it does it. You will also get an inside glimpse at some of the decisions a compiler must make, as it transforms your code into the assembly language commands the CPU will execute. By truly understanding the compiler's role, you will be able to exploit its strengths and also avoid its weaknesses.

Compiler Fundamentals

No matter what language a program is written in, at some point it must be translated into the binary codes that the PC's processor can understand. Unlike BASIC commands, the CPU within every PC is capable of acting on only very rudimentary instructions. Some typical examples of these instructions are "Add 3 to the value stored in memory location 100", and "Compare the value stored at address 4012 to the number -12 and jump to the code at address 2015 if it is less". Therefore, one very important value of a high-level language such as BASIC is that a programmer can use meaningful names instead of memory addresses when referring to variables and subroutines. Another is the ability to perform complex actions that require many separate small steps using only one or two statements.

As an example, when you use the command `PRINT X%` in a program, the value of `X%` must first be converted from its native two-byte binary format into an ASCII string suitable for display. Next, the current cursor location must be determined, at which point the characters in the string are placed into the screen's memory area. Further, the cursor position has to be updated, to place it just past the digits that were printed. Finally, if the last digit happened to end up at the bottom-right corner of the screen, the display must also be scrolled up a line. As you can see, that's an awful lot of activity for such a seemingly simple statement!

A compiler, then, is a program that translates these English-like BASIC source statements into the many separate and tiny steps the microprocessor requires. The BASIC compiler has four major responsibilities:

Translate BASIC statements into an equivalent series of assembly language commands.

1. Assign addresses in memory to hold each of the variables being used by the program.

2. Remember the addresses in the generated code where each line number or label occurs, for GOTO and GOSUB statements.
3. Generate additional code to test for events and detect errors when the /v, /w, or /d compile options are used.

As the compiler processes a program's source code, it translates only the most basic statements directly into assembly language. For other, more complex statements, it instead generates calls to routines in the BASIC run-time library that is supplied with your compiler. When designing a BASIC program you would most likely identify operations that need to be performed more than once, and then create subprograms or functions rather than add the same code in-line repeatedly. Likewise, the compiler takes advantage of the inherent efficiency of using called subroutines.

For example, when you use a BASIC statement such as `PRINT Work$`, the compiler processes it as if you had used `CALL PRINT(Work$)`. That is, `PRINT` really is a called subroutine. Similarly, when you write `OPEN FileName$ FOR RANDOM AS #1 LEN = 1024`, the compiler treats that as a call to its `Open` routine, and it creates code identical to `CALL OPEN(FileName$, 1, 1024, 4)`. Here, the first argument is the file name, the second is the file number you specified, the third is the record length, and the value 4 is BASIC's internal code for `RANDOM`. Because these are BASIC key words, the `CALL` statement is of course not required. But the end result is identical. While the BC compiler could certainly create code to print the string or open the file directly, that would be much less efficient than using subroutines. Indeed, all of the subroutines in the Microsoft-supplied libraries are written in assembly language for the smallest size and highest possible performance.

Data Storage

The second important job the compiler must perform is to identify all of the variables and other data your program is using, and allocate space for them in the object file. There are two kinds of data that are manipulated in a BASIC program—static data and dynamic data. The term static data refers to any variable whose address and size does not change during the execution of a program. That is, all simple numeric and `TYPE` variables, and static numeric and `TYPE` arrays. String constants such as "Press a key to continue" and `DATA` items are also considered to be static data, since their contents never change.

Dynamic data is that which changes in size or location when the program runs. One example of dynamic data is a dynamic array, because space to hold its contents is allocated when the program runs. Another is string data, which is constantly moved around in memory as new strings are assigned and old ones are erased. Variable and array storage is discussed in depth in Chapter 2, so I won't belabor that now. The goal here is simply to introduce the concept of variable storage. The important point is that BC deals only with static data, because that must be placed into the object file.

As the compiler processes your source code, it must remember each variable that is encountered, and allocate space in the object file to hold it. Further, all of this data must be able to fit into a single 64K segment, which is called `DGROUP` (for Data Group). Although the compiled code in each object file

may be as large as 64K, static data is combined from all of the files in a multi-module program, and may not exceed 64K in total size. Note that this limitation is inherent in the design of the Intel microprocessors, and has nothing to do with BC, LINK, or DOS.

As each new variable is encountered, room to hold it is placed into the next available data address in the object file. (In truth, the compiler retains all variable information in memory, and writes it to the end of the file all at once following the generated code.) For each integer variable, two bytes are set aside. Long integer and single precision variables require four bytes each, while double precision variables occupy eight bytes. Fixed-length string and TYPE variables use a varying number of bytes, depending on the components you have defined.

Static numeric and TYPE arrays are also written to the object file by the compiler. The number of bytes that are written of course depends on how many elements have been specified in the DIM statement. Also, notice that no matter what type of variable or array is encountered, only zeroes are written to the file. The only exceptions are quoted string constants and DATA items, in which case the actual text must be stored. Unlike numeric, TYPE, and fixed-length variables, strings must be handled somewhat differently. For each string variable a program uses, a four-byte table called a *string descriptor* is placed into the object file. However, since the actual string data is not assigned until the program is run, space for that data need not be handled by the compiler. With string arrays—whether static or dynamic—a table of four-byte descriptors is allocated.

Finally, each array in the program also requires an array descriptor. This is simply a table that shows where the array's data is located in memory, how many elements it currently holds, the length in bytes of each element, and so forth.

Assembly Language Considerations

In order to fully appreciate how the translation process operates, you will first need to understand what assembly language is all about. Please understand that there is nothing inherently difficult about assembly language. Like BASIC, assembly language is comprised of individual instructions that are executed in sequence. However, each of these instructions does much less than a typical BASIC statement. Therefore, many more steps are required to achieve a given result than in a high-level language. Some of these steps will be shown in the following examples. If you are not comfortable with the idea of tackling assembly language concepts just yet, please feel free to come back to this section at a later time.

Let's begin by examining some very simple BASIC statements, and see how they are translated by the compiler. For simplicity, I will show only integer math operations. The 80x86 family of microprocessors can manipulate integer values directly, as opposed to single and double precision numbers which are much more complex. The short code fragment in Listing 1-1 shows some very simple BASIC instructions, along with the resulting compiled assembly code. In case you are interested, disassemblies such as those you are about to see are easy to create for yourself using the

Microsoft CodeView utility. CodeView is included with the Macro Assembler as well as with BASIC PDS.

```
A% = 12
  MOV  WORD PTR [A%],12      ;move a 12 into the word variable A%

X% = X% + 1
  INC  WORD PTR [X%]        ;add 1 to the word variable X%

Y% = Y% + 100
  ADD  WORD PTR [Y%],100    ;add 100 to the word variable Y%

Z% = A% + B%
  MOV  AX,WORD PTR [B%]     ;move the contents of B% into AX
  ADD  AX,WORD PTR [A%]     ;add to that the value of A%
  MOV  WORD PTR [Z%],AX     ;move the result into Z%
```

Listing 1-1: These short examples show the compiled results of some simple BASIC math operations.

The first statement, $A\% = 12$, is directly translated to its assembler equivalent. Here, the value 12 is *moved* into the word-sized address named A%. Although an integer is the smallest data type supported by BASIC, the microprocessor can in fact deal with variables as small as one byte. Therefore, the WORD PTR (word pointer) argument is needed to specify that A% is a full two-byte integer, rather than a single byte. Notice that in assembly language, brackets are used to specify the contents of a memory address. This is not unlike BASIC's PEEK() function, where parentheses are used for that purpose.

In the second statement, $X\% = X\% + 1$, the compiler generates assembly language code to increment, or add 1 to, the word-sized variable in the location named X%. Since adding or subtracting a value of 1 is such a common operation in all programming languages, the designers of the 80x86 included the INC (and complementary DEC) instruction to handle that.

$Y\% = Y\% + 100$ is similarly translated, but in this case to assembler code that adds the value 100 to the word-sized variable at address Y%. As you can see, the simple BASIC statements shown thus far have a direct assembly language equivalent. Therefore, the code that BC creates is extremely efficient, and in fact could not be improved upon even by a human hand-coding those statements in assembly language.

The last statement, $Z\% = A\% + B\%$, is only slightly more complicated than the others. This is because separate steps are required to retrieve the contents of one memory location, before manipulating it and assigning the result to another location. Here, the value held in variable B% is moved into one of the processor's *registers* (AX). The value of variable A% is then added to AX, and finally the result is moved into Z%. There are about a dozen registers within the CPU, and you can think of them as special variables that can be accessed very quickly.

The next example in Listing 1-2 shows how BASIC passes arguments to its internal routines, in this case PRINT and OPEN. Whenever a variable is passed to a routine, what is actually sent is the address (memory location) of the variable. This way, the routine can go to that address, and read the value that

is stored there. As in Listing 1-1, the BASIC source code is shown along with the resultant compiler-generated assembler instructions.

It may also be worth mentioning that the order in which the arguments are sent to these routines is determined by how the routines are designed. In BASIC, if a SUB is designed to accept, say, three parameters in a certain order, then the caller must pass its arguments in that same order.

Parameters in assembler routines are handled in exactly the same manner. Of course, any arbitrary order could be used, and what's important is simply that they match.

```
PRINT Work$
  MOV AX,OFFSET Work$      ;move the address of Work$ into AX
  PUSH AX                  ;push that onto the CPU stack
  CALL B$PESD              ;call the string printing routine

OPEN FileName$ FOR OUTPUT AS #1
  MOV AX,OFFSET FileName$ ;load the address of FileName$
  PUSH AX                  ;push that onto the stack
  MOV AX,1                 ;load the specified file number
  PUSH AX                  ;and push that as well
  MOV AX,-1                ;-1 means that a LEN= was not given
  PUSH AX                  ;and push that
  MOV AX,2                 ;2 is the internal code for OUTPUT
  PUSH AX                  ;pass that on too
  CALL B$OPEN              ;finally, call the OPEN routine
```

Listing 1-2: Many BASIC statements create assembler code that passes arguments to internal routines, as shown above.

When you tell BASIC to print a string, it first loads the address of the string into AX, and then pushes that onto the stack. The stack is a special area in memory that all programs can access, and it is often used in compiled languages to hold the arguments being sent to subroutines. In this case, the OFFSET operator tells the CPU to obtain the address where the variable resides, as opposed to the current contents of the variable. Notice that the words offset, address, and memory location all mean the same thing. Also notice that calls in assembly language work exactly the same as calls in BASIC. When the called routine has finished, execution in the main program resumes with the next statement in sequence.

Once the address for Work\$ has been pushed, BASIC's B\$PESD routine is called. Internally, one of the first things that B\$PESD does is to retrieve the incoming address from the stack. This way it can locate the characters that are to be printed. B\$PESD is responsible for printing strings, and other BASIC library routines are provided to print each type of data such as integers and single precision values.

In case you are interested, PESD stands for Print End-of-line String Descriptor. Had a semicolon been used in the print statement—that is, PRINT Work\$;—then B\$PSSD would have been called instead (Print Semicolon String Descriptor). Likewise, printing a 4-byte long integer with a trailing comma as in PRINT Value&, would result in a call to B\$PCI4 (Print Comma Integer 4), where the 4 indicates the integer's size in bytes.

In the second example of Listing 1-2 the OPEN routine is set up and called in a similar fashion, except that four parameters are required instead of only one. Again, each parameter is pushed onto the stack in turn, followed by a call to the routine. Most of BASIC's internal routines begin with the characters "B\$", to avoid a conflict with subroutines of your own. Since a dollar sign is illegal in a BASIC procedure name, there is no chance that you will inadvertently choose one of the same names that BASIC uses.

As you can see, there is nothing mysterious or even difficult about assembly language, or the translations performed by the BASIC compiler. However, a sequence of many small steps is often needed to perform even simple calculations and assignments. We will discuss assembly language in much greater depth in Chapter 12, and my purpose here is merely to present the underlying concepts.

Please note that variable names are not retained after a program has been compiled. Once BC has finished its job, all references to each variable name have been replaced with an equivalent memory addresses in the object file. Further, once LINK has joined the object files and linked them to the BASIC language libraries, the procedure names are lost as well. These issues will be explored in much greater detail in Chapter 12.

Compiler Optimization

As you have seen, some code is translated by the compiler into the equivalent assembly language statements, while other code is instead converted to calls to the language routines in the BASIC libraries. Some statements, however, are not translated at all. Rather, they are known as *compiler directives* that merely provide information to the compiler as it works. Some examples of these non-executable BASIC statements include DEFINT, OPTION BASE, and REM, as well as the various "meta commands" such as \$INCLUDE and \$DYNAMIC. Some others are SHARED, BYVAL, DATA, DECLARE, CONST, and TYPE.

For our purposes here, it is important to understand that DIM when used on a static array is also a non-executable statement. Because the size of the array is known when the program is compiled, BC can simply set aside memory in the object file to hold the array contents. Therefore, code does not need to be generated to actually create the array. Similarly, TYPE/END TYPE statements also merely define a given number of bytes that will ultimately end up in the program file when the TYPE variable is later dimensioned by your program.

Event and Error Checking

The last compiler responsibility I will discuss here is the generation of additional code to test for events and debugging errors. This occurs whenever a program is compiled using the /d, /w, or /v command

line switches. Although event trapping and debugging are entirely separate issues, they are handled in a similar manner. Let's start with event trapping.

When the IBM PC was first introduced, the ability to handle interrupt-driven events distinguished it from its then-current Apple and Commodore counterparts. Interrupts can provide an enormous advantage over polling methods, since polling requires a program to check constantly for, say, keyboard or communications activity. With polling, a program must periodically examine the keyboard using `INKEY$`, to determine if a key was pressed. But when interrupts are used, the program can simply go about its business, confident that any keystrokes will be processed. Here's how that works:

Each time a key is pressed on a PC, the keyboard generates a hardware interrupt that suspends whatever is currently happening and then calls a routine in the ROM BIOS. That routine in turn reads the character from the keyboard's output port, places it into the PC's keyboard buffer, and returns to the interrupted application. The next time a program looks for a keystroke, that key is already waiting to be read. For example, a program could begin writing a huge multi-megabyte disk file, and any keystrokes will still be handled even if the operator continues to type.

Understand that hardware interrupts are made possible by a direct physical connection between the keyboard circuitry and the PC's microprocessor. The use of interrupts is a powerful concept, and one which is important to understand. Unfortunately, BASIC does not use interrupts in most cases, and this discussion is presented solely in the interest of completeness.

Event Trapping

BASIC provides a number of event handling statements that perhaps could be handled via interrupts, but aren't. When you use `ON TIMER`, for example, code is added to periodically call a central event handler to check if the number of seconds specified has elapsed. Because there are so many possible event traps that could be active at one time, it would be unreasonable to expect BASIC to set up separate interrupts to handle each possibility. In some situations, such as `ON KEY`, there is a corresponding interrupt. In this case, the keyboard interrupt. However, some events such as `ON PLAY(Count)`, where a `GOSUB` is made whenever the `PLAY` buffer has fewer than `Count` characters remaining, have no corresponding physical interrupt. Therefore, polling for that condition is the only reasonable method.

The example in Listing 1-3 shows what happens when you compile using the `/v` switch. Notice that the calls to `B$EVCK` (Event Check) are not part of the original source code. Rather, they show the additional code that `BC` places just before each program statement.

```
DEFINT A-Z
  CALL B$EVCK           'this call is generated by BC
ON TIMER(1) GOSUB HandleTime
  CALL B$EVCK           'this call is generated by BC
TIMER ON
  CALL B$EVCK           'this call is generated by BC
X = 10
```

```

        CALL B$EVCK           'this call is generated by BC
Y = 100
        CALL B$EVCK           'this call is generated by BC
END

HandleTime:
        CALL B$EVCK           'this call is generated by BC
BEEP
        CALL B$EVCK           'this call is generated by BC
RETURN

```

Listing 1-3: When the /v compiler switch is used, BC generates calls to a central event handler at each BASIC statement.

At five bytes per call, you can see that using /v can quickly bloat a program to an unacceptable size. One alternative is to instead use /w. In fact, /w can be particularly attractive in those cases where event handling cannot be avoided, because it lets you specify where a call to B\$EVCK is made: at each line label or line number in your source code. The only downside to using line numbers and labels is that additional working memory is needed by BC to remember the addresses in the code where those labels are placed. This is not usually a problem, though, unless the program is very large or every line is labelled.

All of the various BASIC event handling commands are specified using the ON statement. It is important to understand, however, that ON GOTO and ON GOSUB do not involve events. That is, they are really just an alternate form of GOTO and GOSUB respectively, and thus do not require compiling with /w or /v.

Error Trapping

The last compiler option to consider here is the /d switch, because it too generates extra code that you might not otherwise be aware of. When a program is compiled with /d, two things are added. First, for every BASIC statement a call is made to a routine named B\$LINA, which merely checks to see if Ctrl-Break has been pressed. Normally, a compiled BASIC program is immune to pressing the Ctrl-C and Ctrl-Break keys, except during an INPUT or LINE INPUT statement. Since much of the purpose of a debugging mode is to let you break out of an errant program gone berserk, the Ctrl-Break checking must be performed frequently. These checks are handled in much the same way as event trapping, by calling a special routine once for each line in your source code.

Another important factor resulting from the use of /d is that all array references are handled through a special called routine which ensures that the element number specified is in fact legal. Many people don't realize this, but when a program is compiled without /d and an invalid element is given, BASIC will blindly write to the wrong memory locations. For example, if you use DIM Array%(1 TO 100) and then attempt to assign, say, element number 200, BASIC is glad to oblige. Of course, there is no element 200 in that case, and some other data will no doubt be overwritten in the process.

To prevent these errors from going undetected, BC calls the B\$HARY (Huge Array) routine to calculate the address based on the element number specified. If B\$HARY determines that the array reference is out of bounds, it invokes an internal error handler and you receive the familiar "Subscript out of range" message. Normally, the compiler accesses array elements using as little code as possible, to achieve the highest possible performance. If a static array is dimensioned to 100 elements and you assign element 10, BC knows at the time it compiles your program the address at which that element resides. It can therefore access that element directly, just as if it were a non-array variable.

Even when you use a variable to specify an array element such as `Array%(X) = 12`, the starting address of the array is known, and the value in X can be used to quickly calculate how far into the array that element is located. Therefore, the lack of bounds checking in programs that do not use /d is not a bug in BASIC. Rather, it is merely a trade-off to obtain very high performance. Indeed, one of the primary purposes of using /d is to let BC find mistakes in your programs during development, though at the cost of execution speed.

The biggest complication from BASIC's point of view is when huge (greater than 64K) arrays are being manipulated. In fact, B\$HARY is the very same routine that BC calls when you use the /ah switch to specify huge arrays (hence the name HARY). Since extra code is needed to set up and call B\$HARY compared to the normal array access, using /ah also creates programs that are larger and slower than when it is not used. Further, because B\$HARY is used by both /d and /ah, invalid element accesses will also be trapped when you compile using /ah.

Overflow Errors

The final result of using /d is that extra code is generated after certain math operations, to check for overflow errors that might otherwise go undetected. Overflow errors are those that result in a value too large for a given data type. For example, if you multiply two integers and the result exceeds 32767, that causes an overflow error. Similarly, an underflow error would be created by a calculation resulting a value that is too small.

When a floating point math operation is performed, errors that result from overflow are detected by the routines that perform the calculation. When that happens there is no recourse other than halting your program with an appropriate message. Integer operations, however, are handled directly by 80x86 instructions. Further, an out of bounds result is not necessarily illegal to the CPU. Thus, programs compiled without the /d option can produce erroneous results, and without any indication that an error occurred.

To prove this to yourself, compile and run the short program shown in Listing 1-4, but without using /d. Although the correct result should be 90000, the answer that is actually displayed is 24464. And you will notice that no error message is displayed!

As with illegal array references, BC would rather optimize for speed, and give you the option of using /d as an aid for tracking down such errors as they occur. If you compile the program in Listing 1-4 with the /d option, then BASIC will report the error as expected.

Since an overflow resulting from integer operations is not technically an error as far as the CPU is concerned, how, then, can BASIC trap for that? Although an error in the usual sense is not created, there is a special flag variable within the CPU that is set whenever such a condition occurs. Further, a little-used assembler instruction, INTO (Interrupt 4 if Overflow), will generate software Interrupt 4 if that flag is set. Therefore, all BC has to do is create an Interrupt 4 handler, and then place an INTO instruction after every integer math operation in the compiled code. The interrupt handler will receive control and display an "Overflow" message whenever an INTO calls it. Since the INTO instruction is only one byte and is also very fast, using it this way results in very little size or performance degradation.

```
X% = 30000
Y% = X% * 3
PRINT Y%
```

Listing 1-4: This brief program illustrates how overflow errors are handled in BASIC.

Compiler Optimization

Designing a compiler for a language as complex as BASIC involves some very tricky programming indeed. Although it is one thing to translate a BASIC source file into a series of assembly language commands, it is another matter entirely to do it well! Consider that the compiler must be able to accept a BASIC statement such as `X! = ABS(SQR((Y# + Z!) ^VAL(Work$)))`, and reduce that to the individual steps necessary to arrive at the correct result.

Many, many details must be accounted for and handled, not the least of which are syntax or other errors in the source code. Moreover, there are an infinite number of ways that a programmer can accomplish the same thing. Therefore, the compiler must be able to recognize many different programming patterns, and substitute efficient blocks of assembler code whenever it can. This is the role of an *optimizing compiler*.

One important type of optimization is called *constant folding*. This means that as much math as possible is performed during compilation, rather than creating code to do that when the program runs. For example, if you have a statement such as `X = 4 * Y * 3` BC can, and does, change that to `X = Y * 12`. After all, why multiply 3 times 4 later, when the answer can be determined now? This substitution is performed entirely by the BC compiler, without your knowing about it.

Another important type of optimization is BASIC's ability to remember calculations it has already performed, and use the results again later if possible. BC is especially brilliant in this regard, and it can look ahead many lines in your source code for a repeated use of the same calculations. Listing 1-5 shows a short fragment of BASIC source code, along with the resultant assembler output.

```

X% = 3 * Y% * 4
    MOV  AX,12                ;move the value 12 into AX
    IMUL WORD PTR [Y%]      ;Integer-Multiply that times Y%
    MOV  WORD PTR [X%],AX   ;assign the result in AX to X%

A% = S% * 100
    MOV  BX,AX               ;save the result from above in BX
    MOV  AX,100              ;then assign AX to 100
    IMUL WORD PTR [S%]      ;now multiply AX times S%
    MOV  WORD PTR [A%],AX   ;and assign A% from the result

Z% = Y% * 12
    MOV  WORD PTR [Z%],BX   ;assign Z% from the earlier result

```

Listing 1-5: These short code fragments illustrate how adept BC is at reusing the result of earlier calculations already performed.

As you can see in the first part of Listing 1-5, the value of 3 times 4 was resolved to 12 by the compiler. Code was then generated to multiply the 12 times Y%, and the result is in turn assigned to X%. This is similar to the compiled code examined earlier in Listing 1-1. Notice, however, that before the second multiplication of S% is performed, the result currently in AX is saved in the BX register. Although AX is destroyed by the subsequent multiplication of S% times 100, the result that was saved earlier in BX can be used to assign Z% later on. Also notice that even though 3 * 4 was used first, BC was smart enough to realize that this is the same as the 12 used later.

While the compiler can actually look ahead in your source code as it works, such optimization will be thwarted by the presence of line numbers and labels, as well as IF blocks. Since a GOTO or GOSUB could jump to a labelled source line from anywhere in the program, there is no way for BC to be sure that earlier statements were executed in sequence. Likewise, the compiler has no way to know which path in an IF/ELSE block will be taken at run time, and thus cannot optimize across those statements.

The BASIC Run-time Libraries

Microsoft compiled BASIC lets you create two fundamentally different types of programs. Those that are entirely self-contained in one .EXE file are compiled with the /o command line switch. In this case, the compiler creates translations such as those we have already discussed, and also generates calls to the BASIC language routines contained in the library files supplied by Microsoft. When your compiled program is subsequently linked, only those routines that are actually used will be added to your program.

When /o is not used, a completely different method is employed. In this case, a special .EXE file that contains support for every BASIC statement is loaded along with the BASIC program when the program is run from the DOS command line. As you are about to see, there are advantages and disadvantages to each method. For the purpose of this discussion I will refer to stand-alone programs as BCOM programs, after the BCOMxx.LIB library name used in all versions of QuickBASIC. Programs that instead require the BRUNxx.EXE library to be present at run time will be called BRUN programs.

Beginning with BASIC 7 PDS, the library naming conventions used by Microsoft have become more obscure. This is because PDS includes a number of variations for each method, depending on the type of "math package" that is specified when compiling and whether you are compiling a program to run under DOS or OS/2. These variations will be discussed fully in Chapter 5, when we examine all of the possible options that each compiler version has to offer. But for now, we will consider only the two basic methods—BCOM and BRUN, which have the following primary differences:

- BCOM programs require less memory, run faster, and do not require the presence of the BRUNxx.EXE file when the program is run.
- BRUN programs occupy less disk space, and also allow subsequent chaining to other programs that can share the common library code which is already resident. Chained-to programs also load quickly because the BRUN library is already in memory.

Stand-alone BCOM programs are always larger than an equivalent BRUN program because the library code for PRINT, INSTR, and so forth is included in the final .EXE file. However, less memory will be required when the program runs, since only the code that is really needed is loaded into the PC. Likewise, a BRUN program will take less disk space, because it contains only the compiled code. The actual routines to handle each BASIC statements are stored in the BRUNxx.LIB library, and that library is loaded automatically when the main program is run from DOS.

You might think that since a BRUN program is physically smaller on disk it will load faster, but this is not necessarily true. When you execute a BRUN program from the DOS command line, one of the first things it does is load the BRUN.EXE support file. Since this support file is fairly large, the overall load time will be much greater than the compiled BASIC program's file size would indicate. However, if the main program subsequently chains to another BASIC program, that program will load quickly because the BRUN file does not need to be loaded a second time.

One other important difference between these two methods is the way that the BASIC language routines are accessed. When a BCOM program is compiled and linked, the necessary routines are called in the usual fashion. That is, the compiler generates code that calls the routines in the BCOM library directly. When the program is subsequently linked, the procedure names are translated by LINK into the equivalent memory addresses. That is, a call to PRINT is in effect translated from `CALL B$PESD` to `CALL #####:####`, where `#####:####` is a segment and address.

BRUN programs, on the other hand, instead use a system of interrupts to access the BASIC language routines. Since there is no way for LINK to know exactly where in memory the BRUNxx.EXE file will be ultimately loaded, the interrupt vector table located in low memory is used to hold the various routine addresses. Although many of these interrupt entries are used by the PC's system resources, many others are available. Again, I will defer a thorough treatment of call methods and interrupts until Chapter 11. But for now, suffice it to say that a direct call is slightly faster than an indirect call, where the address to be called must first be retrieved from a table.

As an interesting aside, the routines in the BRUNxx.EXE file in fact modify the caller's code to perform a direct call, rather than an interrupt instruction. Therefore, the first time a given block of code is executed, it calls the run-time routines through an interrupt instruction. Thereafter, the address where the BRUN file has been loaded is known, and will be used the next time that same block of code is executed. In practice, however, this improves only code that lies within a FOR/NEXT, WHILE, or DO loop. Further, code that is executed only once will actually be much slower than in a BCOM program, because of the added self-modification (the program changes itself) instructions.

Notice that when BC compiles your program, it places the name of the appropriate library into the object file. The name BC uses depends on which compiler options were given. This way you don't have to specify the correct name manually, and LINK can read that name and act accordingly. Although QuickBASIC provides only two libraries—one for BCOM programs and one for BRUN—BASIC PDS offers a number of additional options. Each of these options requires the program to be linked with a different library. That is, there are both BRUN and BCOM libraries for use with OS/2, for near and far strings, and for using IEEE or Microsoft's alternate math libraries. Yet another library is provided for 8087-only operation.

Granularity

Until now, we have examined only the actions and methods used by the BC compiler. However, the process of creating an .EXE file that can be run from the DOS command line is not complete until the compiled object file has been linked to the BASIC libraries. I stated earlier that when a stand-alone program is created using the /o switch, only those routines in the BCOM library that are actually needed will be added to the program. Unfortunately, that is not entirely accurate. While it is true that LINK is very smart and will bring in only those routines that are actually called, there is one catch.

Imagine that you have written a BASIC program which is comprised of two separate modules. In one file is the main program that contains only in-line code, and in the other are two BASIC subprograms. Even if the main program calls only one of those subprograms, both will be added when the program is linked. That is, LINK can resolve routines to the source file level only, but cannot extract a single routine from an object module which contains multiple routines. Since an .LIB library file is merely a collection of separate object modules, all of the routines that reside in a given module will be added to a program, even if only one has been accessed. This property is called *granularity*, and it determines how finely LINK can remove routines from a library.

In the case of the libraries supplied with BASIC, the determining factor is which assembly language routines were combined with which other routines in the same source file by the programmers at Microsoft. In QuickBASIC 4.5, for example, when a program uses the CLS statement, the routines that handle COLOR, CSRLIN, POS(0), LOCATE, and the function form of SCREEN are also added. This is true even if none of those other statements have been used. Fortunately, Microsoft has done much to improve this situation in BASIC PDS, but there is still room for improvement. In BASIC PDS, CLS is stored in a separate file, however POS(0), CSRLIN, and SCREEN are still together, as are COLOR and LOCATE.

Obviously, Microsoft has their reasons for doing what they do, and I won't attempt to second guess their expertise here. The BASIC language libraries are extremely complex and contain many routines. (The QuickBASIC 4.5 BCOM45.LIB file contains 1,485 separate assembler procedures.) With such an enormous number of assembly language source files to deal with, it no doubt makes a lot of sense to organize the related routines together. But it is worth mentioning that Crescent Software's P.D.Q. library can replace much of the functionality of the BCOM libraries, and with complete granularity. In fact, P.D.Q. can create working .EXE programs from BASIC source that are less than 800 bytes in size.

Summary

In this chapter, you learned about the process of compiling, and the kinds of decisions a sophisticated compiler such as Microsoft BASIC must make. In some cases, the BASIC compiler performs a direct translation of your BASIC source code into assembly language, and in others it creates calls to existing routines in the BCOM libraries. Besides creating the actual assembler code, BASIC must also allocate space for all of the data used in a program.

You also learned some basics about assembly language, which will be covered in more detail in Chapter 12. However, examples in upcoming chapters will also use brief assembly language examples to show the relative efficiency of different coding styles. In Chapter 2, you will learn how variables and other data are stored in memory.

2

Variables and Data

Data Basics

In Chapter 1 you examined the role of a compiler, and learned how it translates BASIC source code into the assembly language commands a PC requires. But no matter how important the compiler is when creating a final executable program, it is only half of the story. This chapter discusses the equally important other half: data. Indeed, some form of data is integral to the operation of every useful program you will ever write. Even a program that merely prints "Hello" to the display screen requires the data "Hello".

Data comes in many shapes and sizes, starting with a single bit, continuing through eight-byte double precision variables, and extending all the way to multi-megabyte disk files. In this chapter you will learn about the many types of data that are available to you, and how they are manipulated in a BASIC program. You will also learn how data is stored and assigned, and how BASIC's memory management routines operate.

Compiled BASIC supports two fundamental types of data (numeric and string), two primary methods of storage (static and dynamic), and two kinds of memory allocation (near and far). Of course, the myriad of data types and methods is not present to confuse you. Rather, each is appropriate in certain situations. By fully understanding this complex subject, you will be able to write programs that operate as quickly as possible, and use the least amount of memory.

I will discuss each of the following types of data: integer and floating point numeric data, fixed-length and dynamic (variable-length) string data, and user-defined TYPE variables. Besides variables which are identified by name, BASIC supports named constant data such as literal numbers and quoted strings.

I will also present a complete comparison of the memory storage methods used by BASIC, to compare near versus far storage, and dynamic versus static allocation. It is important to understand that near storage refers to variables and other data that compete for the same 64K data space that is often referred to as Near Memory or Data Space. By contrast, far storage refers to the remaining memory in a PC, up to the 640K limit that DOS imposes.

The distinction between dynamic and static allocation is also important to establish now. Dynamic data is allocated in whatever memory is available when a program runs, and it may be resized or erased as necessary. Static data, on the other hand, is created by the compiler and placed directly into the .EXE file. Therefore, the memory that holds static data may not be relinquished for other uses.

Each type of data has its advantages and disadvantages, as does each storage method. To use an extreme example, you could store all numeric data in string variables if you really wanted to. But this would require using STR\$ every time a value was to be assigned, and VAL whenever a calculation had to be made. Because STR\$ and VAL are relatively slow, using strings this way will greatly reduce a program's performance. Further, storing numbers as ASCII digits can also be very wasteful of memory. That is, the double precision value 123456789.12345 requires fifteen bytes, as opposed to the usual eight.

Much of BASIC's broad appeal is that it lets you do pretty much anything you choose, using the style of programming you prefer. But as the example above illustrates, selecting an appropriate data type can have a decided impact on a program's efficiency. With that in mind, let's examine each kind of data that can be used with BASIC, beginning with integers.

Integers and Long Integers

An integer is the smallest unit of numeric storage that BASIC supports, and it occupies two bytes of memory, or one "word". Although various tricks can be used to store single bytes in a one-character string, the integer remains the most compact data type that can be directly manipulated as a numeric value. Since the 80x86 microprocessor can operate on integers directly, using them in calculations will be faster and require less code than any other type of data. An integer can hold any whole number within the range of -32768 to 32767 inclusive, and it should be used in all situations where that range is sufficient. Indeed, the emphasis on using integers whenever possible will be a recurring theme throughout this book.

When the range of integer values is not adequate in a given programming situation, a long integer should be used. Like the regular integer, long integers can accommodate whole numbers only. A long integer, however, occupies four bytes of memory, and can thus hold more information. This yields an allowable range of values that spans from -2147483648 through 2147483647 (approximately +/- 2.15 billion). Although the PC's processor cannot directly manipulate a long integer in most situations, calculations using them will still be much faster and require less code when compared to floating point numbers.

Regardless of which type of integer is being considered, the way they are stored in memory is very similar. That is, each integer is comprised of either two or four bytes, and each of those bytes contains eight bits. Since a bit can hold a value of either 0 or 1 only, you can see why a larger number of bits is needed to accommodate a wider range of values. Two bits are required to count up to three, three bits to count to seven, four bits to count to fifteen, and so forth.

A single byte can hold any value between 0 and 255, however that same range can also be considered as spanning from -128 to 127. Similarly, an integer value can hold numbers that range from either 0 to 65535 or -32768 through 32767, depending on your perspective. When the range is considered to be 0 to 65535 the values are referred to as *unsigned*, because only positive values may be represented.

BASIC does not, however, support unsigned integer values. Therefore, that same range is used in BASIC programs to represent values between -32768 and 32767. When integer numbers are considered as using this range they are called *signed*.

If you compile and run the short program in the listing that follows, the transition from positive to negative numbers will show how BASIC treats values that exceed the integer range of 32767. Be sure not to use the /d debugging option, since that will cause an overflow error to be generated at the transition point. The BASIC environment performs the same checking as /d does, and it too will report an error before this program can run to completion.

```
Number% = 32760
FOR X% = 1 TO 14
    Number% = Number% + 1
    PRINT Number%,
NEXT
```

Displayed result:

32761	32762	32763	32764	32765
32766	32767	-32768	-32767	-32766
-32765	-32764	-32763	-32762	-32761

As you can see, once an integer reaches 32767, adding 1 again causes the value to "wrap" around to -32768. When Number% is further incremented its value continues to rise as expected, but in this case by becoming "less negative". In order to appreciate why this happens you must understand how an integer is constructed from individual bits. I am not going to belabour binary number theory or other esoteric material, and the brief discussion that follows is presented solely in the interest of completeness.

Bits 'N' Bytes

Sixteen bits are required to store an integer value. These bits are numbered 0 through 15, and the least significant bit is bit number 0. To help understand this terminology, consider the decimal number 1234. Here, 4 is the least significant digit, because it contributes the least value to the entire number. Similarly, 1 is the most significant portion, because it tells how many thousands there are, thus contributing the most to the total value. The binary numbers that a PC uses are structured in an identical manner. But instead of ones, tens, and hundreds, each binary digit represents the number of ones, twos, fours, eights, and so forth that comprise a given byte or word.

To represent the range of values between 0 and 32767 requires fifteen bits, as does the range from -32768 to -1. When considered as signed numbers, the most significant bit is used to indicate which range is being considered. This bit is therefore called the sign bit. Long integers use the same method except that four bytes are used, so the sign bit is kept in the highest position of the fourth byte.

Selected portions of the successive range from 0 through -1 (or 65535) are shown in Table 2-1, to illustrate how binary counting operates. When counting with decimal numbers, once you reach 9 the

number is wrapped around to 0, and then a 1 is placed in the next column. Since binary bits can count only to one, they wrap around much more frequently. The Hexadecimal equivalents are also shown in the table, since they too are related to binary numbering. That is, any Hex value whose most significant digit is 8 or higher is by definition negative.

Signed Decimal	Unsigned Decimal	Binary				Hex
0	0	0000	0000	0000	0000	00000
1	1	0000	0000	0000	0001	00001
2	2	0000	0000	0000	0010	00002
3	3	0000	0000	0000	0011	00003
4	4	0000	0000	0000	0100	00004
.	.					.
.	.					.
32765	32765	0111	1111	1111	1101	7FFD
32766	32766		0111	1111	1111 1110	7FFE
32767	32767	0111	1111	1111	1111	7FFF
-32768	32768	1000	0000	0000	0000	8000
-32767	32769	1000	0000	0000	0001	8001
-32766	32770	1000	0000	0000	0010	8002
.	.					.
.	.					.
-4	65531	1111	1111	1111	1100	FFFB
-3	65532	1111	1111	1111	1101	FFFC
-2	65533	1111	1111	1111	1110	FFFD
-1	65534	1111	1111	1111	1111	FFFE
0	65535	0000	0000	0000	0000	FFFF

Table 2-1: When a signed integer is incremented past 32767, its value wraps around and becomes negative.

Memory Addresses and Pointers

Before we can discuss such issues as variable and data storage, a few terms must be clarified. A *memory address* is a numbered location in which a given piece of data is said to reside. Addresses refer to places that exist in a PC's memory, and they are referenced by those numbers. Every PC has thousands of memory addresses in which both data and code instructions may be stored.

A *pointer* is simply a variable that holds an address. Consider a single precision variable named Value that has been stored by the compiler at memory address 10. If another variable—let's call it Address%—is then assigned the value 10, Address% could be considered to be a pointer to Value. Pointer variables are the bread and butter of languages such as C and assembler, because data is often read and written by referring to one variable which in turn holds the address of another variable.

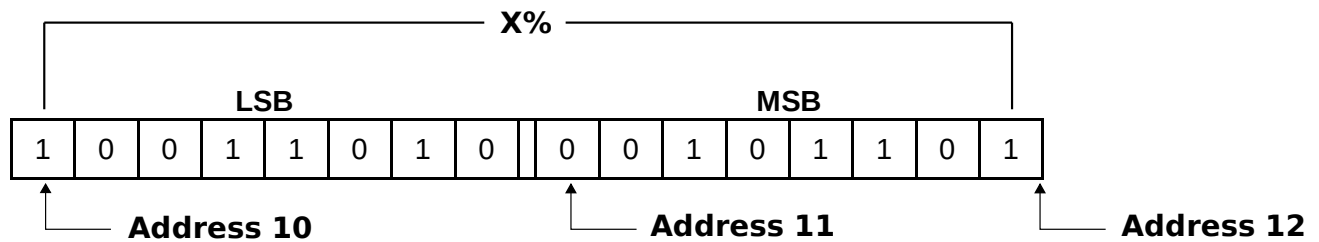


Figure 2-1: An integer is stored in two adjacent memory locations, with the Least Significant Byte at the lower address, and the Most Significant Byte at the higher.

Although BASIC shields you as the programmer from such details, pointers are in fact used internally by the BASIC language library routines. This method of using pointers is sometimes called *indirection*, because an additional, indirect step is needed to first go to one variable, get an address, and then go to that address to access the actual data. Now let's see how these memory issues affect a BASIC program.

Integer Storage

When a conventional two-byte integer is stored in the PC's memory, the lower byte is kept in the lower memory address. For example, if X% is said to reside at address 10, then the least significant byte is at address 10 and the most significant byte is at address 11. Likewise, a long integer stored at address 102 actually occupies addresses 102 through 105, with the least significant portion at the lowest address. This is shown graphically in Figure 2-1.

This arrangement certainly seems sensible, and it is. However, some people get confused when looking at a range of memory addresses being displayed, because the values in lower addresses are listed at the left and the higher address values are shown on the right. For example, the DEBUG utility that comes with DOS will display the Hex number ABCD as CD followed by AB. I mention this only because the order in which digits are displayed will become important when we discuss advanced debugging in Chapter 4.

In case you are wondering, the compiler assigns addresses in the order in which variables are encountered. The first address used is generally 36 Hex, so in the program below the variables will be stored at addresses 36, 38, 3A, and then 3C. Hex numbering is used for these examples because that's the way DEBUG and CodeView report them.


```

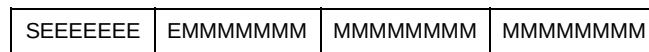
A% = 1      'this is at address &H36
B% = 2      'this is at address &H38
C% = 3      'this is at address &H3A
D% = 4      'this is at address &H3C

```

Floating Point Values

Floating point variables and numbers are constructed in an entirely different manner than integers. Where integers and long integers simply use the entire two or four bytes to hold a single binary number, floating point data is divided into portions. The first portion is called the *mantissa*, and it holds the base value of the number. The second portion is the *exponent*, and it indicates to what power the mantissa must be raised to express the complete value. Like integers, a *sign bit* is used to show if the number is positive or negative.

The IEEE format:



The MBF format:

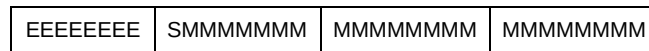


Figure 2-2: A single precision value is comprised of a Sign bit, eight Exponent bits, and 23 bits to represent the Mantissa. Each letter shown here represents one bit, and the bytes on the left are at higher addresses.

The structure of single precision values in both IEEE and the original proprietary Microsoft Binary Format (MBF) is shown in Figure 2-2. For IEEE numbers, the sign bit is in the most significant position, followed by eight exponent bits, which are in turn followed by 23 bits for the mantissa. Double precision IEEE values are structured similarly, except eleven bits are used for the exponent and 52 for the mantissa.

Double precision MBF numbers use only eight bits for an exponent rather than eleven, trading a reduced absolute range for increased resolution. That is, there are fewer exponent bits than the IEEE method uses, which means that extremely large and extremely small numbers cannot be represented. However, the additional mantissa bits offer more absolute digits of precision.

Notice that with IEEE numbers, the exponent spans a byte boundary. This undoubtedly contributes to the slow speed that results from using numbers in this format when a coprocessor is not present. Contrast that with Microsoft's MBF format in which the sign bit is placed between the exponent and mantissa. This allows direct access to the exponent with fewer assembler instructions, since the various bits don't have to be shifted around.

The IEEE format is used in QuickBASIC 4.0 and later, and BASIC PDS unless the /fpa option is used. BASIC PDS uses the /fpa switch to specify an alternate math package which provides increased speed but with a slightly reduced accuracy. Although the /fpa format is in fact newer than the original MBF used in interpreted BASIC and QuickBASIC 2 and 3, it is not quite as fast.

As was already mentioned, double precision data requires twice as many bytes as single precision. Further, due to the inherent complexity of the way floating point data is stored, an enormous amount of assembly language code is required to manipulate it. Common sense therefore indicates that you would use single precision variables whenever possible, and reserve double precision only for those cases where the added accuracy is truly necessary. Using either floating point variable type, however, is still very much slower than using integers and long integers. Worse, rounding errors are inevitable with any floating point method, as the following short program fragment illustrates.

```
FOR X% = 1 TO 10000
    Number! = Number! + 1.1
NEXT
PRINT Number!
```

Displayed result:

```
10999.52
```

Although the correct answer should be 11000, the result of adding 1.1 ten thousand times is incorrect by a small amount. If you are writing a program that computes, say, tax returns, even this small error will be unacceptable. Recognizing this problem, Microsoft developed a new Currency data type which was introduced with BASIC PDS version 7.0.

The Currency data type is a cross between an integer and a floating point number. Like a double precision value, Currency data also uses eight bytes for storage. However, the numbers are stored in an integer format with an implied scaling of 10000. That is, a binary value of 1 is used to represent the value .0001, and a binary value of 20000 is treated as a 2. This yields an absolute accuracy to four decimal places, which is more than sufficient for financial work. The absolute range of Currency data is plus or minus 9.22 times 10^{14} ($\pm 9.22E14$ or 922,000,000,000,000.0000), which is very wide indeed. This type of storage is called *Fixed-Point*, because the number of decimal places is fixed (in this case at four places).

Currency data offers the best compromise of all, since only whole numbers are represented and the fractional portion is implied. Further, since a separate exponent and mantissa are not used, calculations involving Currency data are extremely fast. In practice, a loop that adds a series of Currency variables will run about half as fast as the same loop using long integers. Since twice as many bytes must be manipulated, the net effect is an overall efficiency that is comparable to long integers. Compare that to double precision calculations, where manipulating the same eight bytes takes more than six times longer.

As you have seen, there is a great deal more to "simple" numeric data than would appear initially. But this hardly begins to scratch the surface of data storage and manipulation in BASIC. We will continue our tour of BASIC's data types with conventional dynamic (variable-length) strings, before proceeding to fixed-length strings and TYPE variables.

Dynamic Strings

One of the most important advantages that BASIC holds over all of the other popular high-level languages is its support for dynamic string data. In Pascal, for example, you must declare every string that your program will use, as well as its length, before the program can be compiled. If you determine during execution of the program that additional characters must be stored in a string, you're out of luck.

Likewise, strings in C are treated internally as an array of single character bytes, and there is no graceful way to extend or shorten them. Specifying more characters than necessary will of course waste memory, and specifying too few will cause subsequent data to be overwritten. Since C performs virtually no error checking during program execution, assigning to a string that is not long enough will corrupt memory. And indeed, problems such as this cause untold grief for C programmers.

Dynamic string memory handling is built into BASIC, and those routines are written in assembly language. BASIC is therefore extremely efficient and very fast in this regard. Since C is a high-level language, writing an equivalent memory manager in C would be quite slow and bulky by comparison. I feel it is important to point out BASIC's superiority over C in this regard, because C has an undeserved reputation for being a very fast and powerful language.

Compiled BASIC implements dynamic strings with varying lengths by maintaining a *string descriptor* for each string. A string descriptor is simply a four-byte table that holds the current length of the string as well as its current address. The format for a BASIC string descriptor is shown in Figure 2-3. In QuickBASIC programs and BASIC PDS when far strings are not specified, all strings are stored in an area of memory called the *near heap*. The string data in this memory area is frequently shuffled around, as new strings are assigned and old ones are abandoned.

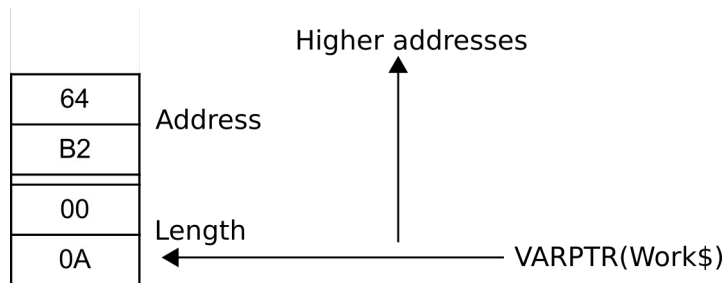


Figure 2-3: Each string in a QuickBASIC program has a corresponding string descriptor, which holds its current length and address. The string in this example has a length of ten characters (0A Hex) and its data is presently at address 25778 (64B2 Hex).

The lower two bytes in a string descriptor together hold the current length of the string, and the second two bytes hold its address. The memory location at the bottom of Figure 2-3 is at the lowest address. The short program below shows how you could access a string by peeking at its descriptor.

```
DEFINT A-Z

Test$ = "BASIC Techniques and Utilities"
Descr = VARPTR(Test$)
Length = PEEK(Descr) + 256 * PEEK(Descr + 1)
Addr = PEEK(Descr + 2) + 256 * PEEK(Descr + 3)

PRINT "The length is"; Length
PRINT "The address is"; Addr
PRINT "The string contains ";
FOR X = Addr TO Addr + Length - 1
    PRINT CHR$(PEEK(X));
NEXT
```

Displayed result:

```
The length is 17
The address is 15646 (this will vary)
The string contains BASIC Techniques and Utilities
```

Each time a string is assigned or reassigned, memory in the heap is claimed and the string's descriptor is updated to reflect its new length and address. The old data is then marked as being abandoned, so the space it occupied may be reclaimed later on if it is needed. Since each assignment claims new memory, at some point the heap will become full. When this happens, BASIC shuffles all of the string data that is currently in use downward on top of the older, abandoned data. This heap compaction process is often referred to colorfully as *garbage collection*.

In practice, there are two ways to avoid having BASIC claim new space for each string assignment—which takes time—and you should consider these when speed is paramount. One method is to use LSET or RSET, to insert new characters into an existing string. Although this cannot be used to make a string longer or shorter, it is very much faster than a straight assignment which invokes the memory

management routines. The second method is to use the statement form of MID\$, which is not quite as fast as LSET, but is more flexible.

Microsoft BASIC performs some additional trickery as it manages the string data in a program. For example, whenever a string is assigned, an even number of bytes is always requested. Thus, if a five-character string is reassigned to one with six characters, the same space can be reused. Since claiming new memory requires a finite amount of time and also causes garbage collection periodically, this technique helps to speed up the string assignment process.

For example, in a program that builds a string by adding new characters to the end in a loop, BASIC can reduce the number of times it must claim new memory to only every other assignment. Another advantage to always allocating an even number of bytes is that the 80286 and later microprocessors can copy two-byte words much faster than they can copy the equivalent number of bytes. This has an obvious advantage when long strings are being assigned.

In most cases, BASIC's use of string descriptors is much more efficient than the method used by C and other languages. In C, each string has an extra trailing CHR\$(0) byte just to mark where it ends. While using a single byte is less wasteful than requiring a four-byte table, BASIC's method is many times faster. In C the entire string must be searched just to see how long it is, which takes time. Likewise, comparing and concatenating strings in C requires scanning both strings for the terminating zero character. The same operations in BASIC require but a single step to obtain the current length.

Pascal uses a method that is similar to BASIC's, in that it remembers the current length of the string. The length is stored with the actual string data, in a byte just before the first character. Unfortunately, using a single byte limits the maximum length of a Pascal string to only 255 characters. And again, when a string is shortened in Pascal, the extra characters are not released for use by other data. But it is only fair to point out that Pascal's method is both fast and compact. And since strings in C and Pascal never move around in memory, garbage collection is not required.

Although a BASIC string descriptor uses four bytes of additional memory beyond that needed for the actual data, this is only part of the story. An additional two bytes are needed to hold a special "variable" called a *back pointer*. A back pointer is an integer word that is stored in memory immediately before the actual string data, and it holds the address of the data's string descriptor. Thus, it is called a back pointer because it points back to the descriptor, as opposed to the descriptor which points to the data.

Because of this back pointer, six additional bytes are actually needed to store each string, beyond the number of characters that it contains. For example, the statement `Work$ = "BASIC"` requires twelve bytes of data memory: five for the string itself, one more because an even number of bytes is always claimed, four for the descriptor, and two more for a back pointer. Every string that is defined in a program has a corresponding descriptor which is always present, however a back pointer is maintained only while the string has characters assigned to it. Therefore, when a string is erased the two bytes for its back pointer are also relinquished.

I won't belabour this discussion of back pointers further, because understanding them is of little practical use. Suffice it to say that a back pointer helps speed up the heap compaction process. Since the address portion of the descriptor must be updated whenever the string data is moved, this pointer provides a fast link between the data being moved and its descriptor. By the way, the term "pointer" refers to any variable that holds a memory address, regardless of what language is being considered.

Far Strings in BASIC PDS

BASIC PDS offers an option to specify *far strings*, whereby the string data is not stored in the same 64K memory area that holds most of a program's data. The method of storage used for far strings is of necessity much more complex than near strings, because both an address and a segment must be kept track of. Although Microsoft has made it clear that the structure of far string descriptors may change in the future, I would be remiss if this undocumented information were not revealed here. The following description is valid as of BASIC 7.1 [it is still valid for VB/DOS too].

For each far string in a program, a four-byte descriptor is maintained in near memory. The lower two bytes of the descriptor together hold the address of an integer variable that holds yet another address: that of the string length and data. The second pair of bytes also holds the address of a pointer, in this case a pointer to a variable that indicates the segment in which the string data resides. Thus, by retrieving the address and segment from the descriptor, you can locate the string's length and data, albeit with an extra level of indirection.

It is interesting to note that when far strings are being used, the string's length is kept just before its data, much like the way Pascal operates. Therefore, the address pointer holds the address of the length word which immediately precedes the actual string data.

The short program that follows shows how to locate all of the components of a far string based on examining its descriptor and related pointers. Notice that long integers are used to avoid the possibility of an overflow error if the segment or addresses happen to be higher than 32767. This way you can run the program in the QBX [or VB/DOS] editing environment. Figure 2-4 in turn illustrates the relationship between the address and pointer information graphically.

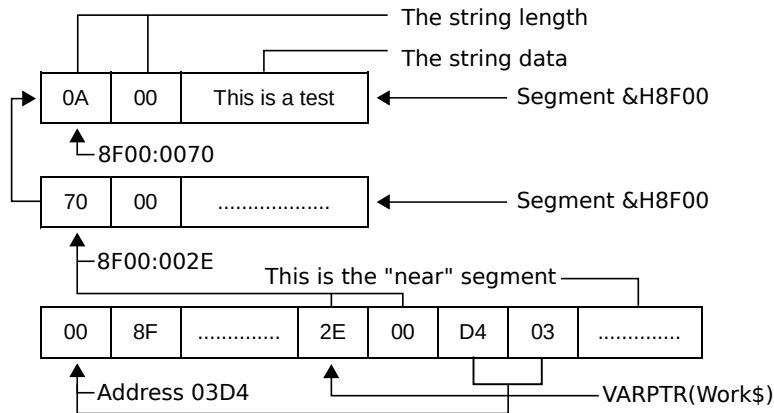


Figure 2-4: A far string descriptor holds the addresses of other addresses, in this case addresses that hold a far string's segment and its length and actual data.

```

DEF FNPeekWord& (A&)
  FNPeekWord& = PEEK(A&) + 256& * PEEK(A& + 1)
END DEF

Work$ = "This is a test"

DescAddr& = VARPTR(Work$)
AddressPtr& = FNPeekWord&(DescAddr&)
SegmentPtr& = FNPeekWord&(DescAddr& + 2)
Segment& = FNPeekWord&(SegmentPtr&)

DEF SEG = Segment&
DataAddr& = FNPeekWord&(AddressPtr&)
Length% = FNPeekWord&(DataAddr&)
StrAddr& = DataAddr& + 2

PRINT "The descriptor address is:"; DescAddr&
PRINT "    The data segment is:"; Segment&
PRINT "    The length is:"; Length%
PRINT "The string data starts at:"; StrAddr&
PRINT "    And the string data is: ";

FOR X& = StrAddr& TO StrAddr& + Length% - 1
  PRINT CHR$(PEEK(X&));
NEXT

```

Displayed result (the addresses may vary):

```

The descriptor address is: 17220
    The data segment is: 40787
    The length is: 14
The string data starts at: 106
    And the string data is: This is a test

```

```
DATA items in VB/DOS programs are still kept in near
memory, but quoted strings are now kept in a separate
segment.
```

Because two bytes are used to hold the segment, address, and length values, we must PEEK both of them and combine the results. This is the purpose of the PeekWord function that is defined at the start of the program. Note the placement of an ampersand after the number 256, which ensures that the multiplication will not cause an overflow error. I will discuss such use of numeric constants and type identifiers later in this chapter.

Even in a far-string program, some of the string data will be near. For example, DATA items and quoted string constants are stored in the same 64K DGROUP data segment that holds simple numeric and TYPE variables. The same "indirect" method is used, whereby you must look in one place to get the address of another address. In this case, however, the "far" segment that is reported is simply the normal near data segment.

One final complication worth mentioning is that strings within a FIELD buffer (and possibly in other special situations) are handled slightly differently. Since all of the strings in a FIELD buffer must be contiguous, BASIC cannot store the length word adjacent to the string data. Therefore, a different method must be used. This case is indicated by setting the sign bit (the highest bit) in the length word as a flag. Since no string can have a negative length, that bit can safely be used for this purpose. When a string is stored using this alternate method, the bytes that follow the length word are used as additional pointers to the string's actual data segment and address.

Fixed-Length Strings

One of the most important new features Microsoft added beginning with QuickBASIC 4.0 was fixed-length string and TYPE variables. Although fixed-length strings are less flexible than conventional BASIC strings, they offer many advantages in certain programming situations. One advantage is that they are static, which means their data does not move around in memory as with conventional strings. You can therefore obtain the address of a fixed-length string just once using VARPTR, confident that this address will never change. With dynamic strings, SADD must be used each time the address is needed, which takes time and adds code. Another important feature is that arrays of fixed-length strings can be stored in far memory, outside of the normal 64K data area. We will discuss near and far array memory allocation momentarily.

With every advantage, however, comes a disadvantage. The most severe limitation is that when a fixed-length string is used where a conventional string is expected, BASIC must generate code to create a temporary dynamic string, and then copy the data to it. That is, all of BASIC's internal routines that operate on strings expect a string descriptor. Therefore, when you print a fixed-length string, or use MID\$ or INSTR or indeed nearly any statement or function that accepts a string, it must be copied to a form that BASIC's internal routines can accept. In many cases, additional code is created to delete the

temporary string afterward. In others, the data remains until the next time the same BASIC statement is executed, and a new temporary string is assigned freeing the older one.

To illustrate, twenty bytes of assembly language code are required to print a fixed-length string, compared to only nine for a conventional dynamic string. Worse, when a fixed-length string is passed as an argument to a subprogram or function, BASIC not only makes a copy before passing the string, but it also copies the data back again in case the subroutine changed it! The extra steps the compiler performs are shown as BASIC equivalents in the listing that follows.

```
'----- This is the code you write:

DIM Work AS STRING * 20
CALL TestSub(Work$)

'----- This is what BASIC actually does:

Temp$ = SPACE$(20)      'create a temporary string
LSET Temp$ = Work$      'copy Work$ to it
CALL TestSub(Temp$)     'call the subprogram
LSET Work$ = Temp$      'copy the data back again
Temp$ = ""              'erase the temporary data
```

As you can imagine, all of this copying creates an enormous amount of additional code in your programs. Where only nine bytes are required to pass a conventional string to a subprogram, 64 are needed when a fixed-length string is being sent. But you cannot assume unequivocally that conventional strings are always better or that fixed-length strings are always better. Rather, I can only present the facts, and let you decide based on the knowledge of what is really happening. In the discussion of debugging later in Chapter 4, you will learn how to use CodeView to see the code that BASIC generates. You can thus explore these issues further, and draw your own conclusions.

User-Defined TYPE Variables

As I mentioned earlier, the TYPE variable is an important and powerful addition to modern compiled BASIC. Its primary purpose is to let programmers create composite data structures using any combination of native data types. C and Pascal have had such user-defined data types since their inception, and they are called Structures and Records respectively in each language.

One immediately obvious use for being able to create a new, composite data type is to define the structure of a random access data file. Another is to simulate an array comprised of varied types of data. Obviously, no language can support a mix of different data types within a single array. That is, an array cannot be created where some of the elements are, say, integer while others are double precision. But a TYPE variable lets you do something very close to that, and you can even create arrays of TYPE variables.

In the listing that follows a TYPE is defined using a mix of integer, single precision, double precision, and fixed-length string components. Also shown below is how a TYPE variable is dimensioned, and how each of its components are assigned and referenced.

```
TYPE MyType
  I AS INTEGER
  S AS SINGLE
  D AS DOUBLE
  F AS STRING * 20
END TYPE

DIM MyData as MyType

MyData.I = 12      'assign the integer portion
MyData.S = 100.09 'and then the single part
MyData.D = 43.2E56 'and then the double
MyData.F = "Test" 'and finally the string

PRINT MyData.F    'now print the string
```

Once the TYPE structure has been established, the DIM statement must be used to create an actual variable using that arrangement. Although DIM is usually associated with the definition of arrays, it is also used to identify a variable name with a particular type of data. In this case, DIM tells BASIC to set aside an area of memory to hold that many bytes. You may also use DIM with conventional variable types. For example, DIM LastName AS STRING or DIM PayPeriod AS DOUBLE lets you omit the dollar sign and pound sign when you reference them later in the program. In my opinion, however, that style leads to programs that are difficult to maintain, since many pages later in the source listing you may not remember what type of data is actually being referred to.

As you can see, a period is needed to indicate which portion of the TYPE variable is being referenced. The base name is that given when you dimensioned the variable, but the portion being referenced is identified using the name within the original TYPE definition. You cannot print a TYPE variable directly, but must instead print each component separately. Likewise, assignments to a TYPE variable must also be made through its individual components, with two exceptions. You may assign an entire TYPE variable from another identical TYPE directly, or from a dissimilar TYPE variable using LSET.

For example, if we had used DIM MyData AS MyType and then DIM HisData AS MyType, the entire contents of HisData could be assigned to MyData using the statement MyData = HisData. Had HisData been dimensioned using a different TYPE definition, then LSET would be required. That is, LSET MyData = HisData will copy as many characters from HisData as will fit into MyData, and then pad the remainder, if any, with blanks.

It is important to understand that this behavior can cause strange results indeed. Since CHR\$(32) blanks are used to pad what remains in the TYPE variable being assigned, numeric components may receive some unusual values. Therefore, you should assign differing TYPE variables only when those overlapping portions being assigned are structured identically.

Arrays Within Types

With the introduction of BASIC PDS, programmers may also establish static arrays within a single TYPE definition. An array is dimensioned within a TYPE as shown in the listing that follows. As with a conventional DIM statement for an array, the number of elements are indicated and a non-zero lower bound may optionally be specified. Please understand, though, that you cannot use a variable for the number of elements in the array. That is, using `PayHistory(1 TO NumDates)` would be illegal.

```
TYPE ArrayType
  AmountDue AS SINGLE
  PayHistory(1 TO 52) AS SINGLE
  LastName AS STRING * 15
END TYPE

DIM TypeArray AS ArrayType
```

There are several advantages to using an array within a TYPE variable. One is that you can reference a portion of the TYPE by using a variable to specify the element number. For example, `TypeArray.PayHistory(PayPeriod) = 344.95` will assign the value 344.95 to element number `PayPeriod`. Without the ability to use an array, each of the 52 components would need to be identified by name. Further, arrays allows you to define a large number of TYPE elements with a single program statement. This can help to improve a program's readability.

Static versus Dynamic Data

Preceding sections have touched only briefly on the concept of static and dynamic memory storage. Let's now explore this subject in depth, and learn which methods are most appropriate in which situations.

By definition, static data is that which never changes in size, and never moves around in memory. In compiled BASIC this definition is further extended to mean all data that is stored in the 64K near memory area known as DGROUP. This includes all numeric variables, fixed-length strings, and TYPE variables. Technically speaking, the string descriptors that accompany each conventional (not fixed-length) string are also considered to be static, even though the string data itself is not. The string descriptors that comprise a dynamic string array, however, are dynamic data, because they move around in memory (as a group) and may be resized and erased.

Numeric arrays that are dimensioned with constant (not variable) subscripts are also static, unless the '\$DYNAMIC metacommand has been used in a preceding program statement. That is, `DIM Array#(0 TO 100)` will create a static array, while `DIM Array#(0 TO MaxElements)` creates a dynamic array. Likewise, arrays of fixed-length strings and TYPE variables will be static, as long as numbers are used to specify the size.

There are advantages and disadvantages to each storage method. Access to static data is always faster than access to dynamic data, because the compiler knows the address where the data resides at the time

it creates your program. It can therefore create assembly language instructions that go directly to that address. In contrast, dynamic data always requires a pointer to hold the current address of the data. An extra step is therefore needed to first get the data address from that pointer, before access to the actual data is possible. Static data is also in the near data segment, thus avoiding the need for additional code that switches segments.

The overwhelming disadvantage of static data, though, is that it may never be erased. Once a static variable or array has been used in a program, the memory it occupies can never be released for other uses. Again, it is impossible to state that static arrays are always better than dynamic arrays or vice versa. Which you use must be dictated by your program's memory requirements, when compared to its execution speed.

Dynamic Arrays

You have already seen how dynamic strings operate, by using a four-byte pointer table called a string descriptor. Similarly, a dynamic array also needs a table to show where the array data is located, how many elements there are, the length of each element, and so forth. This table is called an array descriptor, and it is structured as shown in Table 2-2.

There is little reason to use the information in an array descriptor in a BASIC program, and indeed, BASIC provides no direct way to access it anyway. But when writing routines in assembly language for use with BASIC, this knowledge can be quite helpful. As with BASIC PDS far string descriptors, none of this information is documented, and relying on it is most certainly not endorsed by Microsoft. Perhaps that's what makes it so much fun to discuss!

Technically speaking, only dynamic arrays require an array descriptor, since static arrays do not move or change size. But BASIC creates an array descriptor for every array, so only one method of code generation is necessary. For example, when you pass an entire array to a subprogram using empty parentheses, it is the address of the array descriptor that is actually sent. The subprogram can then access the data through that descriptor, regardless of whether the array is static or dynamic.

Offset	Size	Description
00	02	Address where array data begins
00	02	Segment where that address resides
04	02	Far heap descriptor, pointer
06	02	Far heap descriptor, block size
08	01	Number of dimensions in the array
09	01	Array type and storage method: Bit 0 set = far array Bit 1 set = huge (/ah) array Bit 6 set = static array Bit 7 set = string array
0A	02	Adjusted Offset
0C	02	Length in bytes of each element
0E	02	Number of elements in the last dimension (UBOUND - LBOUND + 1)
10	02	First element number in that dimension (LBOUND)
12	03	Number of elements in the second from last dimension
14	02	First element number in that dimension. Repeat number of elements and first element number as necessary, through the first dimension

Table 2-2: Every array in a BASIC program has an associated array descriptor such as the one shown here. This descriptor contains important information about the array.

The first four bytes together hold the segmented address where the array data proper begins in memory. Following the standard convention, the address is stored in the lower word, with the segment immediately following.

The next two words comprise the Far Heap Descriptor, which holds a pointer to the next dynamic array descriptor and the current size of the array. For static arrays both of these entries are zero. When multiple dynamic arrays are used in a program, the array descriptors are created in static DGROUP memory in the order BC encounters them. The Far Heap Pointer in the first array therefore points to the next array descriptor in memory. The last descriptor in the chain can be identified because it points to a word that holds a value of zero.

The block size portion of the Far Heap Descriptor holds the size of the array, using a byte count for string arrays and a "paragraph" count for numeric, fixed-length, and TYPE arrays. For string arrays—whether near or far—the byte count is based on the four bytes that each descriptor occupies. With numeric arrays the size is instead the number of 16-byte paragraphs that are needed to store the array.

The next entry is a single byte that holds the number of dimensions in the array. That is, `DIM Array(1 TO 10)` has one dimension and `DIM Array(1 TO 10, 2 TO 20)` has two.

The next item is also a byte, and it is called the Feature byte because the various bits it holds tell what type of array it is. As shown in the table, separate bits are used to indicate if the array is stored in far memory, whether or not `/ah` was used to specify huge arrays, if the array is static, and if it is a string array. Multiple bits are used for each of these array properties, since they may be active in combination. However, BASIC never sets the far and huge bits for string arrays, even when the PDS `/fs` option is used and the strings are in fact in far memory.

Of particular interest is the Adjusted Offset entry. Even though the segmented address where the array data begins is the first entry in the descriptor, it is useful only when the first element number in the array is zero. This would be the case with `DIM Array(0 TO N)`, or simply `DIM Array(N)`. To achieve the fastest performance possible when retrieving or assigning a given element, the Adjusted Offset is calculated when the array is dimensioned to compensate for an `LBOUND` other than 0.

For example, if an integer array is dimensioned starting at element 1, the Adjusted Offset is set to point two bytes before the actual starting address of the data. This way, the compiler can take the specified element number, multiply that times two (each element comprises two bytes), and then add that to the Adjusted Offset to immediately point at the correct element in memory. Otherwise, additional code would be needed to subtract the `LBOUND` value each time the array is accessed. Since the array's `LBOUND` is simply constant information, it would be wasteful to calculate that repeatedly at run time. Of course, the Adjusted Offset calculation is correspondingly more complex when dealing with multi-dimensional arrays.

The remaining entries identify the length of each element in bytes, and the upper and lower bounds. String arrays always have a 4 in the length location, because that's the length of each string descriptor. A separate pair of words is needed for each array subscript, to identify the `LBOUND` value and the number of elements. The `UBOUND` is not actually stored in the array descriptor, since it can be calculated very easily when needed. Notice that for multi-dimensional arrays, the last (right-most) subscript is identified first, followed by the second from the last, and continuing to the first one.

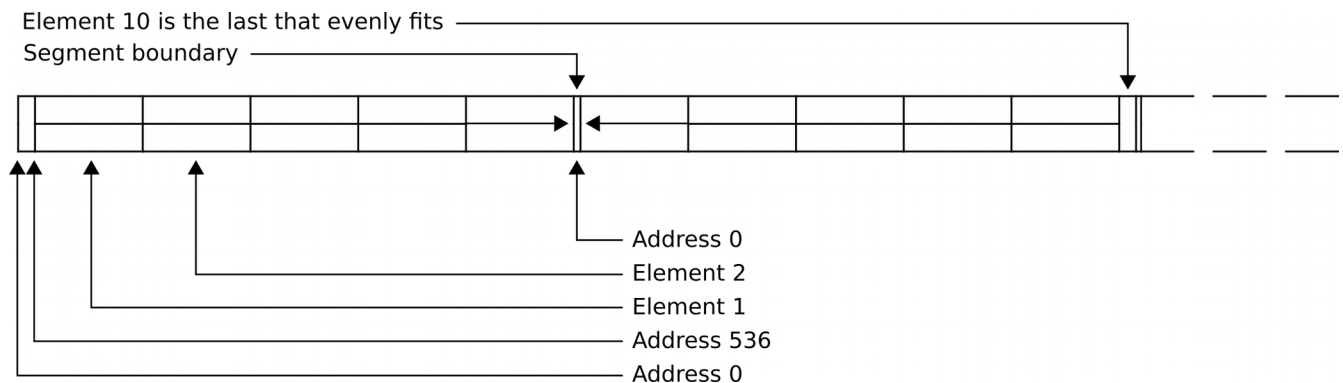


Figure 2-5 How BASIC handles arrays whose element lengths are not a power of two

One final note worth mentioning about dynamic array storage is the location in memory of the first array element. For numeric arrays, the starting address is always zero, within the specified segment. (A new segment can start at any 16-byte address boundary, so at most 15 bytes may be wasted.) However, BASIC sometimes positions fixed-length string and TYPE arrays farther into the segment. BASIC will not allow an array element to span a segment boundary under any circumstances. This could never happen with numeric data, because each element has a length that is a power of 2. That is, 16,384 long integer elements will exactly fit in a single 64K segment. But when a fixed-length string or TYPE array is created, nearly any element length may be specified.

For example, if you use `REDIM Array(1 TO 10) AS STRING * 13000`, 130,000 bytes are needed and element 6 would straddle a segment. To prevent that from happening, BASIC's dynamic DIM routine fudges the first element to instead be placed at address 536. Thus, the last byte in element 5 will be at the end of the 64K segment, and the first byte in element 6 will fall exactly at the start of the second 64K code segment. The only limitation is that arrays with odd lengths like this can never exceed 128K in total size, because the inevitable split would occur at the start of the third segment. Arrays whose element lengths are a power of 2, such as 32 or 4096 bytes, do not have this problem. (Bear in mind that 1K is actually 1,024 bytes, so 128K really equals 131,072 bytes.) This is shown graphically above in Figure 2-5.

Far Data Versus Near Data

We have already used the terms "near" and "far" to describe BASIC's data, and now let's see exactly what they mean. The 8086 family of microprocessors that are used in IBM PC and compatible computers use what is called a *segmented architecture*. This means that while an 8086 can access a megabyte of memory, it can do so only in 64K blocks at a time. Before you think this is a terrible way to design a CPU, consider the alternative.

For example, the 68000 family used in the Apple Macintosh and Atari computers use linear addressing, whereby any data anywhere may be accessed without restriction. But the problem is that with millions of possible addresses, many bytes are needed to specify those addresses. Because the data segment is implied when dealing with an 80x86, a single integer can refer to any address quickly and with very little code. Therefore, assembler instructions for the 68000 that reference memory tend to be long, making those programs larger.

Since being able to manipulate only one 64K segment is restrictive, the 8086's designers provided four different segment registers. One of these, the DS (Data Segment) register, is set to specify a single segment, which is then used by the program as much as possible. This data segment is also named DGROUP, and it holds all of the static data in a BASIC program. Again, data in DGROUP can be accessed much faster and with less code than can data in any other segment. In order to assign an element in a far array, for example, BASIC requires two additional steps which generates additional code. The first step is to retrieve the array's segment from the array descriptor, and the second is to assign the ES (Extra Segment) register to access the data.

Far data in a BASIC program therefore refers to any data that is outside of the 64K DGROUP segment. Technically, this could encompass the entire 1 Megabyte that DOS recognizes, however the memory beyond 640K is reserved for video adapters, the BIOS, expanded memory cards, and the like. BASIC uses far memory (outside the 64K data segment but within the first 640K) for numeric, fixed-length string, and TYPE arrays, although BASIC PDS can optionally store conventional strings there when the /fs (Far String) option is used. Communications buffers are also kept in far memory, and this is where incoming characters are placed before your program actually reads them.

Near memory is therefore very crowded, with many varied types of data competing for space. Earlier I stated that all variables, static arrays, and quoted strings are stored in near memory (DGROUP). But other BASIC data is also stored there as well. This includes DATA items, string descriptors, array descriptors, the stack, file buffers, and the internal working variables used by BASIC's run-time library routines.

When you open a disk file for input, an area in near memory is used as a buffer to improve the speed of subsequent reads. And like subprograms and functions that you write, BASIC's internal routines also need their own variables to operate. For example, a translation table is maintained in DGROUP to relate the file numbers you use when opening a file to the file handles that DOS issues.

One final note on the items that compete for DGROUP is that in many cases data is stored twice. When you use READ to assign a string from a DATA item, the data itself remains at the DATA statement, and is also duplicated in the string being assigned. There is simply no way to remove the original data. Similarly, when you assign a string from a constant as in `Message$ = "Press any key"`, the original quoted string is always present, and `Message$` receives a second copy. When string space is very tight, the only purely BASIC solution is to instead store the data in a disk file.

Speaking of DATA, bear in mind that reading numeric variables is relatively slow and often even more wasteful. Since all DATA items are stored as strings, each time you use READ the VAL routine is called internally by BASIC. VAL is not a particularly fast operation, because of the complexity of what it must do. Worse, by storing numbers as strings, even more memory can be wasted than you might think. For example, storing an integer value such as -20556 requires six bytes as a string, even though it will be placed ultimately into a two-byte integer.

Assessing Memory with FRE()

Since memory is very important to the operation of most programs, it is often useful to know how much of it is available at any given moment. BASIC provides the FRE function to do this, however there are a number of variations in its use. Let's take an inside look at the various forms of FRE, and see how they can be put to good use.

There are no less than six different arguments that can be used with FRE. The first to consider is FRE(0), which reports the amount of free string space but without first compacting the string pool. Therefore, the value returned by FRE(0) may be much lower than what actually could be available.

FRE when used with a string argument, for example FRE("") or FRE(Temp\$), also returns the amount of DGROUP memory that is available, however it first calls the heap compaction routines. This guarantees that the size reported accurately reflects what is really available.

Although FRE(0) may seem to be of little value, it is in fact much faster than FRE when a string argument is given. Therefore, you could periodically examine FRE(0), and if it becomes unacceptably low use FRE("") to determine the actual amount of memory that is available. With BASIC PDS far strings, FRE(0) is illegal, FRE("") reports the number of bytes available for temporary strings, and FRE(Any\$) reports the free size of the segment in which Any\$ resides. Temporary strings were discussed earlier, when we saw how they are used when passing fixed-length string arguments to procedures.

FRE(-1) was introduced beginning with QuickBASIC 1, and it reports the total amount of memory that is currently available for use with far arrays. Thus, you could use it in a program before dimensioning a large numeric array, to avoid receiving an "Out of memory" error which would halt your program. Although there is a distinction between near and far memory in any PC program, BASIC does an admirable job of making available as much memory as you need for various uses. For example, it is possible to have plenty of near memory available, but not enough for all of the dynamic arrays that are needed. In this case, BASIC will reduce the amount of memory available in DGROUP, and instead relinquish it for far arrays.

FRE(-1) is also useful if you use SHELL within your programs, because at least 20K or so of memory is needed to load the necessary additional copy of COMMAND.COM. It is interesting to observe that not having enough memory to execute a SHELL results in an "Illegal function call" error, rather than the expected "Out of memory".

FRE(-2) was added to QuickBASIC beginning with version 4.0, and it reports the amount of available stack space. The stack is a special area within DGROUP that is used primarily for passing the addresses of variables and other data to subroutines. The stack is also used to store variables when the STATIC option is omitted from a subprogram or function definition. I will discuss static and non-static subroutines later in Chapter 3, but for now suffice it to say that enough stack memory is necessary when many variables are present and STATIC is omitted.

FRE(-3) was added with BASIC PDS, mainly for use within the QBX editing environment. This newest variant reports the amount of expanded (EMS) memory that is available, although EMS cannot be accessed by your programs directly using BASIC statements. However, QBX uses that memory to store subroutines and optionally numeric, fixed-length, and TYPE arrays. The ISAM file handler that comes with BASIC PDS can also utilize expanded memory, as can the PDS overlay manager.

SETMEM and STACK

Besides the various forms of the FRE function, SETMEM can be used to assess the size of the far heap, as well as modify that size if necessary. The STACK function is available only with BASIC PDS, and it

reports the largest possible size the stack can be set to. Let's see how these functions can be useful to you.

Although SETMEM is technically a function (because it returns information), it is also used to re-size the far heap. When given an argument of zero, SETMEM returns the current size of the far heap. However, this value is not the amount of memory that is free. Rather, it is the maximum heap size regardless of what currently resides there. The following short program shows this in context.

```
PRINT SETMEM(0)           'display the heap size
REDIM Array!(10000)      'allocate 40,000 bytes
PRINT SETMEM(0)         'the total size remains
```

Displayed result (the numbers will vary):

```
276256
276256
```

When a program starts, the far heap is set as large as possible by BASIC and DOS, which is sensible in most cases. But there are some situations in which you might need to reduce that size, most notably when calling C routines that need to allocate their own memory. Also, BASIC moves arrays around in the far heap as arrays are dimensioned and then erased. This is much like the near heap string compaction that is performed periodically. If the far heap were not rearranged periodically, it is likely that many small portions would be available, but not a single block sufficient for a large array.

In some cases a program may need to claim memory that is guaranteed not to move. Therefore, you could ask SETMEM to relinquish a portion of the far heap, and then call a DOS interrupt to claim that memory for your own use. (DOS provides services to allocate and release memory, which C and assembly language programs use to dimension arrays manually.) Unlike BASIC, DOS does not use sophisticated heap management techniques, therefore the memory it manages does not move. I will discuss using SETMEM this way later on in Chapter 12.

Finally, the STACK function will report the largest amount of memory that can be allocated for use as a stack. Like SETMEM, it doesn't reflect how much of that memory is actually in use. Rather, it simply reports how large the stack could be if you wanted or needed to increase it. Because the stack resides in DGROUP, its maximum possible size is dependent on how many variables and other data items are present.

When run in the QBX environment, the following program fragment shows how creating a dynamic string array reduces the amount of memory that could be used for the stack. Since the string descriptors are kept in DGROUP, they impinge on the potentially available stack space.

```
PRINT STACK
REDIM Array$(1000)
PRINT STACK
ERASE Array$
PRINT STACK
```

Displayed result:

```
47904
43808
47904
```

Since BASIC PDS does not support FRE(0), the STACK function can be used to determine how much near memory is available. The only real difference between FRE(0) and STACK is that STACK includes the current stack size, where FRE(0) does not. The STACK function is mentioned here because it relates to assessing how much memory is available for data. Sizing the stack will be covered in depth in Chapter 3, when we discuss subprograms, functions, and recursion.

VARPTR, VARSEG, and SADD

One of the least understood aspects of BASIC programming is undoubtedly the use of VARPTR and its related functions, VARSEG and SADD. Though you probably already know that VARPTR returns the address of a variable, you might be wondering how that information could be useful. After all, the whole point of a high-level language such as BASIC is to shield the programmer from variable addresses, pointers, and other messy low-level details. And by and large, that is correct. Although VARPTR is not a particularly common function, it can be invaluable in some programming situations.

VARPTR is a built-in BASIC function which returns the address of any variable. VARSEG is similar, however it reports the memory segment in which that address is located. SADD is meant for use with conventional (not fixed-length) strings only, and it tells the address where the first character in a string begins. In BASIC PDS, SSEG is used instead of VARSEG for conventional strings, to identify the segment in which the string data is kept. Together, these functions identify the location of any variable in memory.

The primary use for VARPTR in purely BASIC programming is in conjunction with BSAVE and BLOAD, as well as PEEK and POKE. For example, to save an entire array quickly to a disk file with BSAVE, you must specify the address where the array is located. In most cases VARSEG is also needed, to identify the array's segment as well. When used on all simple variables, static arrays, and all string arrays, VARSEG returns the normal DGROUP segment. When used on a dynamic numeric array, it instead returns the segment at the which the specified element resides.

The short example below creates and fills an integer array, and then uses VARSEG and VARPTR to save it very quickly to disk.

```
REDIM Array%(1 TO 1000)

FOR X% = 1 TO 1000
    Array%(X%) = X%
NEXT

DEF SEG = VARSEG(Array%(1))
BSAVE "ARRAY.DAT", VARPTR(Array%(1)), 2000
```

Here, DEF SEG indicates in which segment the data that BSAVE will be saving is located. VARPTR is then used to specify the address within that segment. The 2000 tells BSAVE how many bytes are to be written to disk, which is determined by multiplying the number of array elements times the size of each element. We will come back to using VARPTR repeatedly in Chapter 11 when we discuss accessing DOS and BIOS services with CALL Interrupt. However, it is important to point out here exactly how VARPTR and VARSEG work with each type of variable.

When VARPTR is used with a numeric variable, as in `Address = VARPTR(Value!)`, the address of the first byte in memory that the variable occupies is reported. Value! is a single-precision variable which spans four bytes of memory, and it is the lowest of the four addresses that is returned. Likewise, VARPTR when used with static fixed-length string and TYPE variables reports the lowest address where the data begins. But when you ask for the VARPTR of a string variable, what is returned is the address of the string's descriptor.

To obtain the address of the actual data in a string requires the SADD (String Address) function. Internally, BASIC simply looks at the address portion of the string descriptor to retrieve the address. Likewise, the LEN function also gets its information directly from the descriptor. When used with any string, VARSEG always reports the normal DGROUP data segment, because that is where all strings and their descriptors are kept.

Beginning with BASIC PDS and its support for far strings, the SSEG function was added to return the segment where the string's data is stored. But even when far strings are being used, VARSEG always returns the segment for the descriptor, which is in DGROUP.

SADD is not legal with a fixed-length string, and you must instead use VARPTR. Perhaps in a future version BASIC will allow either to be used interchangeably. SADD is likewise illegal for use with the fixed-length string portion of a TYPE variable or array. Again, VARPTR will return the address of any component in a TYPE, within the segment reported by VARSEG.

Another important use for VARPTR is to assist passing arrays to assembly language routines. When a single array element is specified using early versions of Microsoft compiled BASIC, the starting address of the element is sent as expected. Beginning with QuickBASIC 4.0 and its support for far data residing in multiple segments, a more complicated arrangement was devised. Here's how that works.

When an element in a dynamic array is passed as a parameter, BASIC makes a copy of the element into a temporary variable in near memory, and then sends the address of the copy. When the routine returns, the data in the temporary variable is copied back to the original array element, in case the called routine changed the data. In many cases this behavior is quite sensible, since the called routine can assume that the variable is in near memory and thus operate that much faster.

Further, BASIC subroutines require a non-array parameter (not passed with empty parentheses) to be in DGROUP. That is, any time a single element in an integer array is passed to a routine, that routine

would be designed to expect a single integer variable. This is shown in the brief example below, where a single element in an array is passed, as opposed to the entire array.

```
REDIM Array%(1 TO 100)
Array%(25) = -14
CALL MyProc(Array%(25))      'pass one element
.
.
.
SUB MyProc(IntVar%) STATIC  'this sub expects a
  PRINT IntVar%             ' single variable
END SUB
```

Displayed result:

```
-14
```

Unfortunately, this copying not only generates a lot of extra code to implement, it also takes memory from DGROUPE to hold the copy, and that memory is taken permanently. Worse still, each occurrence of an array element passed in a CALL statement reserves however many bytes are needed to store the element. For a large TYPE structure this can be a lot of memory indeed!

So you won't think that I'm being an alarmist about this issue, here are some facts based on programs compiled using BASIC 7.1 PDS. These examples document the amount of additional code that is generated to pass a near string array element as an argument to a subprogram or function.

Passing a string array element requires 56 bytes when a copy is made, compared to only 17 when it is not. The same operations in QuickBASIC 4.5 create 47 and 18 bytes respectively, so QB 4.5 is actually better when making the copy, but a tad worse when not. The code used in these examples is shown below, and Array\$ is a dynamic near string array. (I will explain the purpose of BYVAL in just a moment.) Again, the difference in byte counts reflects the additional code that BC creates to assign and then delete the temporary copies.

```
CALL Routine(Array$(2))
CALL Routine(BYVAL VARPTR(Array$(2)))
```

Worse still, with either compiler 73 bytes of code are created to pass an element in a TYPE array the usual way, compared to 18 when the copying is avoided. And this byte count does not include the DGROUPE memory required to hold the copy. Is that reduction in code size worth working for? You bet it is! And best of all, hardly any extra effort is needed to avoid having BASIC make these copies—just the appropriate knowledge.

The key, as you can see, is VARPTR. If you are calling an assembly language routine that expects a string and you want to pass an element from a string array, you must use BYVAL along with VARPTR. CALL Routine(BYVAL VARPTR(Array\$(Element))) is functionally identical to CALL Routine(Array\$(Element)), although they sure do look different. In either case, the integer address of a string is passed to the routine.

Unlike the usual way that BASIC passes a variable by sending its address, BYVAL instead sends the actual data. In this case, the value of an address is what we wanted to begin with anyway. (Without the BYVAL, BASIC would make a temporary copy of the integer value that VARPTR returns, and send the address of that copy.) Best of all, asking for the address directly defeats the built-in copying mechanism. Although creating a copy of a far numeric array element is sensible as we saw earlier, it is not clear to me why BC does this with string array data that is in DGROUP already.

Although you can't normally send an integer—which is what VARPTR actually returns—to a BASIC subprogram that expects a string, you can if that subprogram is in a different file and the files are compiled separately. This will also work if the BASIC code has been pre-compiled and placed in a Quick Library.

But there is another, equally important reason to use VARPTR with array elements. If you are calling an assembler routine that will sort an array, it must have access to the array element's address, and not the address of a copy. All of the elements in any array are contiguous, and a sort routine would need to know where in memory the first element is located. From that it can then access all of the successive elements. With VARPTR we are telling BASIC that what is needed is the actual address of the specified element.

Bear in mind that this relates primarily to passing arrays to assembly language (and possibly C) routines only. After all, if you are designing a sort routine using purely BASIC commands, you would pass and receive the array using empty parentheses. Indeed, this is yet another important advantage that BASIC holds over C and Pascal, since neither of those languages have array descriptors. Writing a sort routine in C requires that you do all of the work to locate and compare each element in turn, based on some base starting address.

There is one final issue that we must discuss, and that is passing far array data to external assembly language routines. I already explained that by making a copy of a far array element, the called routine does not have to be written to deal with far (two-word segmented) addresses. But in some cases, writing a routine that way will be more efficient. Further, like C, assembly language routines thrive on manipulating pointers to data. Although an assembler routine could be written to read the segment and address from the array descriptor, this is not a common method. One reason is that if Microsoft changes the format of the descriptor, the routine will no longer work. Another is that it is frankly easier to have the caller simply pass the full segmented address of the first element.

This brings us to the SEG directive, which is a combination of BYVAL and VARPTR and also BYVAL and VARSEG. As with BYVAL VARPTR, using SEG before a variable or array element in a call tells BASIC that the value of the array's full address is needed. A typical example would be `CALL Routine(SEG Array#(1))`, and in this case, BASIC sends not one address word but two to the routine.

You could also pass the full address of an array element by value using VARSEG and VARPTR, and this next example produces the identical result: `CALL Routine(BYVAL VARSEG(Array#(1)), BYVAL VARPTR(Array#(1)))`. Using SEG results in somewhat less code, though, because

BASIC will obtain the segment and address in a single operation. In fact, this is one area where the compiler does a poor job of optimizing, because using VARSEG and VARPTR in a single program statement generates a similar sequence of code twice.

There is one unfortunate complication here, which arises when SEG is used with a fixed-length string array. What SEG should do in that case is pass the segmented address of the specified element. But it doesn't. Instead, BASIC creates a temporary copy of the specified element in a conventional dynamic string, and then passes the segmented address of the copy's descriptor. Of course, this is useless in most programming situations.

There are two possible solutions to this problem. The first is to use the slightly less efficient BYVAL VARSEG and BYVAL VARPTR combination as shown above. The second solution is to create an equivalent fixed-length string array by using a dummy TYPE that is comprised solely of a single string component. Since TYPE variables are passed correctly when SEG is used, using a TYPE eliminates the problem. Both of these methods are shown in the listing that follows.

```
'----- this creates more code and looks clumsy
REDIM Array(1 TO 1000) AS STRING * 50
CALL Routine(BYVAL VARSEG(Array(1)), BYVAL VARPTR(Array(1)))

'----- this creates less code and reads clearly

TYPE FLen
  S AS STRING * 100
END TYPE
REDIM Array(1 TO 1000) AS FLen
CALL Routine(SEG Array(1))
```

Although SEG looks like a single parameter is being passed, in fact two integers are sent to the called routine—a segment and an address. This is why a single SEG can replace both a VARSEG and a VARPTR in one call. Chapter 12 will return to BYVAL, VARPTR, and SEG, though the purpose there will be to learn how to write routines that accept such parameters.

Constants

The final data type to examine is constants. By definition, a constant is simply any value that does not change, as opposed to a variable that can. For example, in the statement `I% = 10`, the value 10 is a constant. Similarly, the quoted string "Hello" is a constant when you write `PRINT "Hello"`.

There are two types of constants that can appear in a BASIC program. One is simple numbers and quoted strings as described above, and the other is the named constant which is defined using a `CONST` statement. For example, you can write `CONST MaxRows = 25` as well as `CONST Message$ = "Insert disk in drive"`, and so forth. It is even possible to define one `CONST` value based on a previous one, as in `CONST NumRows = 25, ScrnSize = NumRows * 80`. Then, you could use these meaningful names later in the program, instead of the values they represent.

It is important to understand that using named constants is identical to using the numbers themselves. The value of this will become apparent when you see the relative advantages and disadvantages of using numbers as opposed to variables. Let's begin this discussion of numbers with how they are stored by the compiler. Or rather, how they are sometimes stored.

When a `CONST` statement is used in a BASIC program, BASIC does absolutely nothing with the value, other than to remember that you defined it. Therefore, you could have a hundred `CONST` statements which are never used, and the final `.EXE` program will be no larger than if none had been defined. If a `CONST` value is used as an argument to, say, `LOCATE` or perhaps as a parameter to a subroutine, BASIC simply substitutes the value you originally gave it. When a variable is assigned as in `Value% = 100`, BASIC sets aside memory to hold the variable. With a constant definition such as `CONST Value% = 100`, no memory is set aside and BASIC merely remembers that any use of `Value%` is to be replaced by the number 100. But how are these numbers represented internally?

When you create an integer assignment such as `Count% = 5`, the BASIC compiler generates code to move the value 5 into the integer variable, as you saw in Chapter 1. Therefore, the actual value 5 is never stored as data anywhere. Rather, it is placed into the code as part of an assembly language instruction.

Now, if you instead assign a single or double precision variable from a number—and again it doesn't matter whether that number is a literal or a `CONST`—the appropriate floating point representation of that number is placed in `DGROUP` at compile time, and then used as the source for a normal floating point assignment. That is, it is assigned as if it were a variable.

There is no reasonable way to embed a floating point value into an assembly language instruction, because the CPU cannot deal with such values directly. Therefore, assigning `X% = 3` treats the number 3 as an integer value, while assigning `Y# = 3` treats it as a double precision value. Again, it doesn't matter whether the 3 is a literal number as shown here, or a `CONST` that has been defined. In fact, if you use `CONST Three! = 3`, a subsequent assignment such as `Value% = Three!` treats `Three!` as an integer resulting in less resultant code. As you can see, the compiler is extremely smart in how it handles these constants, and it understands the context in which they are being used.

In general, BASIC uses the minimum precision possible when representing a number. However, you can coerce a number to a different precision with an explicit type identifier. For example, if you are calling a routine in a separate module that expects a double precision value, you could add a pound sign (`#`) to the number like this: `CALL Something(45#)`. Without the double precision identifier, BASIC would treat the 45 as an integer, which is of course incorrect.

Likewise, BASIC can be forced to evaluate a numeric expression that might otherwise overflow by placing a type identifier after it. One typical situation is when constructing a value from two byte portions. The usual way to do this would be `Value& = LoByte% + 256 * HiByte%`. Although the result of this expression can clearly fit into the long integer no matter what the values of `LoByte%`

and HiByte% might be, an overflow error can still occur. (But as we saw earlier, this will happen only in the QB environment, or if you have compiled to disk with the /d debugging option.)

The problem arises when HiByte% is greater than 127, because the result of multiplying HiByte% times 256 exceeds the capacity of a regular integer. Normally, BASIC is to be commended for the way it minimizes overhead by reducing calculations to the smallest possible data type. But in this case it creates a problem, because the result cannot be expressed as an integer.

The solution, then, is to add an ampersand after the 256, as in `Value& = LoByte% + 256& * HiByte%`. By establishing the value 256 as a long integer, you are telling BASIC to perform the calculation to the full precision of a long integer. And since the result of the multiplication is treated as a long integer, so is the addition of that result to LoByte%. A single precision exclamation point could also be used, but that would require a floating point multiplication. Since a long integer multiply is much faster and needs less code, this is the preferred solution.

One final item worth noting is the way the QB and QBX editing environments sometimes modify constants. For example, if you attempt to enter a statement such as `Value! = 1.0`, you will see the constant changed to read `1!` instead. This happens when you press Enter to terminate the line. Similarly, if you write `D# = 1234567.8901234`, BASIC will add a trailing pound sign to the number. This behavior is your clue that these numbers are being stored internally as single and double precision values respectively.

Passing Numeric Constants to a Procedure

Normally, any constant that could be an integer is passed to a subprogram or function as an integer. That is, calling an external procedure as in `CALL External(100)` passes the 100 as an integer value. If the called routine has been designed to expect a variable of a different type, you must add the appropriate type identifier. If a long integer is expected, for example, you must use `CALL External(100&)`. If, on the other hand, the called routine is in the same module (that is, the same physical source file), QB will create a suitable `DECLARE` statement automatically. This lets QB and BC know what is expected so they can pass the value in the correct format. Thus, BASIC is doing you a favor by interpreting the constant's type in a manner that is relevant to your program.

This "favor" has a nasty quirk, though. If you are developing a multi-module program in the QuickBASIC editor, the automatic type conversion is done for you automatically, even when the call is to a different module. Your program uses, say, `CALL Routine(25)`, and QB or QBX send the value in the correct format automatically. But when the modules are compiled and linked, the same program that had worked correctly in the environment will now fail.

Since each module in a multi-module program is compiled separately, BC has no way to know what the called routine actually expects. In fact, this is one of the primary purposes of the `DECLARE` statement—to advise BASIC as to how arguments are to be passed. For example, `DECLARE SUB Marine(Trident!)` tells BASIC that any constant passed to Marine is to be sent as a single

precision value. You could optionally use the AS SINGLE directive, thus: `DECLARE SUB Marine(Trident AS SINGLE)`. In general, I prefer the more compact form since it conveys the necessary information with less clutter.

Another important use for adding a type identifier to a numeric constant is to improve a program's accuracy. Running the short program below will illustrate this in context. Although neither answer is entirely accurate, the calculation that uses the double precision constant is much closer. In this case, a decimal number that does not have an explicit type identifier is assumed to have only single precision accuracy. That is, the value is stored in only four bytes instead of eight.

```
FOR X% = 1 TO 10000
  Y# = Y# + 1.1
  Z# = Z# + 1.1#
NEXT
PRINT Y#, Z#
```

Displayed result:

```
11000.00023841858      11000.000000000204
```

You have already learned that BASIC often makes a temporary copy of a variable when calling a subprogram or function. But you should know that this also happens whenever a constant is passed as an argument. For example, in a function call such as `Result = Calculate!(Value!, 100)`, where `Calculate!` has been declared as a function, the integer value 100 is copied to a temporary location. Since BASIC procedures require the address of a parameter, a temporary variable must be created and the address of that variable passed. The important point to remember is that for each occurrence of a constant in a `CALL` or function invocation, a new area of `DGROUP` is taken.

You might think that BASIC should simply store a 100 somewhere in `DGROUP` once, and then pass the address of that value. Indeed, this would save an awful lot of memory when many constants are being used. The reason this isn't done, however, is that subroutines can change incoming parameters. Therefore, if a single integer 100 was stored and its address passed to a routine that changed it, subsequent calls using 100 would receive an incorrect value.

The ideal solution to this problem is to create a variable with the required value. For example, if you are now passing the value 2 as a literal many times in a program, instead assign a variable, perhaps named `Two%`, early in your program. That is, `Two% = 2`. Then, each time you need that value, instead pass the variable. For the record, six bytes are needed to assign an integer such as `Two%`, and four bytes are generated each time that variable is passed in a call.

Contrast that to the 10 bytes generated to create and store a temporary copy and pass its address, not including the two bytes the copy permanently takes from near memory. Even if you use the value only twice, the savings will be worthwhile (24 vs. 30 bytes). Because a value of zero is very common, it is also an ideal candidate for being replaced with a variable. Even better, you don't even have to assign it! That is, `CALL SomeProc(Zero%)` will send a zero, without requiring a previous `Zero% = 0` assignment.

String Constants

Like numeric constants, string constants that are defined in a `CONST` statement but never referenced will not be added to the final `.EXE` file. Constants that are used—whether as literals or as `CONST` statements—are always stored in `DGROUP`. If your program has the statement `PRINT "I like BASIC"`, then the twelve characters in the string are placed into `DGROUP`. But since the `PRINT` statement requires a string descriptor in order to locate the string and determine its length, an additional four bytes are allocated by `BASIC` just for that purpose. Variables are always stored at an even-numbered address, so odd-length strings also waste one extra byte.

Because string constants have a ferocious appetite for near memory, `BC` has been designed to be particularly intelligent in the way they are handled. Although there is no way to avoid the storage of a descriptor for each constant, there is another, even better trick that can be employed. For each string constant you reference in a program that is longer than four characters, `BC` stores it only once. Even if you have the statement `PRINT "Press any key to continue"` twenty-five times in your program, `BC` will store the characters just once, and each `PRINT` statement will refer to the same string.

In order to do this, the compiler must remember each string constant it encounters as it processes your program, and save it in an internal working array. When many string constants are being used, this can cause the compiler to run out of memory. Remember, `BC` has an enormous amount of information it must deal with as it processes your `BASIC` source file, and keeping track of string constants is but one part of the job.

To solve this problem Microsoft has provided the `/s` (String) option, which tells `BC` not to combine like data. Although this may have the net effect of making the final `.EXE` file larger and also taking more string space, it may be the only solution with some large programs. Contrary to the `BASIC` documentation, however, using `/s` in reality often makes a program smaller. This issue will be described in detail in Chapter 5, where all of the various `BC` command line options are discussed.

Passing String Constants to a Procedure

As you have repeatedly seen, `BASIC` often generates additional code to create copies of variables and constants. It should come as no surprise, therefore, to learn that this happens with string constants as well. When you print the same string more than once in a program, `BASIC` knows that its own `PRINT` routine will never change the data. But as with numeric constants, if you send a string constant to a subprogram or function, there is no such guarantee.

For example, if you have a statement such as `CALL PrintIt(Work$)` in your program, it is very possible—even likely—that the `PrintIt` routine may change or reassign its incoming parameter. Even if you know that `PrintIt` will not change the string, `BASIC` has no way to know this. To avoid any

possibility of that happening, BASIC generates code to create a temporary copy of every string constant that is used as an argument. And this is done for every call. If the statement `CALL PrintMessage("Press a key")` appears in your program ten times, then code to copy that message is generated ten times.

Beginning with BASIC 7.1 PDS, you can now specify that variables are to be sent by value to BASIC procedures. This lets you avoid the creation of temporary copies, and this subject will also be explored in more detail in Chapter 3.

With either QuickBASIC 4.5 or BASIC PDS, calling a routine with a single quoted string as an argument generates 31 bytes of code. Passing a string variable instead requires only nine bytes. Both of these byte counts includes the five bytes to process the call itself. The real difference is therefore 4 bytes vs. 26—for a net ratio of 6.5 to 1. (Part of those 31 bytes is code that erases the temporary string.) So as with numeric constants that are used more than once, your programs will be smaller if a variable is assigned once, and that variable is passed as an argument.

While we are on the topic of temporary variables, there is yet another situation that causes BASIC to create them. When the result of an expression is passed as an argument, BASIC must evaluate that expression, and store the result somewhere. Again, since nearly all procedures require the address of a parameter rather than its value, an address of that result is needed. And without storing the result, there can of course be no address.

When you use a statement such as `CALL Home(Elli + Lou)`, BASIC calculates the sum of Elli plus Lou, and stores that in a reserved place in DGROUP which is not used for any other purpose. That address is then sent to the Home routine as if it were a single variable, and Home is none the wiser. Likewise, a string concatenation creates a temporary string, for the same reason. Although the requisite descriptor permanently steals four bytes of DGROUP memory, the temporary string itself is erased by BASIC automatically after the call. Thus, the first example in the listing below is similar in efficiency to the second. The four-byte difference is due to BASIC calling a special routine that deletes the temporary copy it created, as opposed to the slightly more involved code that assigns Temp\$ from the null string ("") to erase it.

```
CALL DoIt(First$ + Last$)    'this makes 41 bytes
Temp$ = First$ + Last$      'this makes 45 bytes
CALL DoIt(Temp$)
Temp$ = ""
```

Unusual String Constants

One final topic worth mentioning is that QuickBASIC also lets you embed control and extended characters into a string constant. Consider the program shown below. Here, several of the IBM extended characters are used to define a box, but without requiring CHR\$ to be used repeatedly. Characters with ASCII values greater than 127 can be entered easily by simply pressing and holding the

Alt key, typing the desired ASCII value on the PC's numeric key-pad, and then releasing the Alt key. This will not work using the number keys along the top row of the keyboard.

```
DIM Box$(1 TO 4)           'define a box

Box$(1) = "  "
Box$(2) = "  "
Box$(3) = "  "
Box$(4) = "  "

FOR X = 1 TO 4             'now display the box
  PRINT Box$(X)
NEXT
```

To enter control characters (those with ASCII values less than 32) requires a different trick. Although the Alt-keypad method is in fact built into the BIOS of all PCs, this next one is specific to QuickBASIC, QBX, and some word processor programs. To do this, first press Ctrl-P, observing the ^P symbol that QB displays at the bottom right of the screen. This lets you know that the next control character you press will be accepted literally. For example, Ctrl-P followed by Ctrl-L will display the female symbol, and Ctrl-P followed by Ctrl-[will enter the Escape character.

Bear in mind that some control codes will cause unusual behavior if your program is listed on a printer. For example, an embedded CHR\$(7) will sound the buzzer if your printer has one, a CHR\$(8) will back up the print head one column, and a CHR\$(12) will issue a form feed and skip to the next page. Indeed, you can use this to advantage to intentionally force a form feed, perhaps with a statement such as REM followed by the Ctrl-L female symbol.

I should mention that different versions of the QB editor respond differently to the Ctrl-P command. QuickBASIC 4.0 requires Ctrl-[to enter the Escape code, while QBX takes either Ctrl-[or the Escape key itself. I should also mention that you must never embed a CHR\$(26) into a BASIC source file. That character is recognized by DOS to indicate the end of a file, and BC will stop dead at that point when compiling your program. QB, however, will load the file correctly.

Wouldn't It Be Nice If...

No discussion of constants would be complete without a mention of initialized data. Unfortunately, as of this writing BASIC does not support that feature. The concept is simple, and it would be trivial for BASIC's designers to implement. Here's how initialized data works.

Whenever a variable requires a certain value, the only way to give it that value is to assign it. Some languages let you declare a variable's initial value in the source code, saving the few bytes it takes to assign it later. Since space for every variable is in the .EXE file anyway, there would be no additional penalty imposed by adding this capability. I envision a syntax such as DIM X = 3.9 AS SINGLE, or perhaps simply DIM Y% = 3, or even DIM PassWord\$ = "GuessThis". Where Y% = 3 creates a six-byte code sequence to put the value 3 into Y%, what I am proposing would have the compiler place that value there at the time it creates the program.

Equally desirable would be allowing string constants to be defined using CHR\$ arguments. For example, `CONST EOF$ = CHR$(26)` would be a terrific enhancement to the language, and allowing code such as `CONST CRLF$ = CHR$(13) + CHR$(10)` would be even more powerful. Again, we can only hope that this feature will be added in a future version.

Yet another constant optimization that BASIC could do but doesn't is constant string function evaluation. In many programming situations the programmer is faced with deciding between program efficiency and readability. A perfect example of this is testing an integer value to see whether it represents a legal character. For instance, `IF Char < 65` is not nearly as meaningful as `IF Char < ASC("A")`.

Clearly, BC could and should resolve the expression `ASC("A")` while it is compiling your program, and generate simple code that compares two integers. Instead, it stores the "A" as a one-byte string (which with its descriptor takes five bytes), and generates code to call the internal ASC function before performing the comparison. The point here is that no matter how intelligent BC is, folks like us will always find some reason to complain!

Bit Operations

The last important subject this chapter will cover is bit manipulation using AND, OR, XOR, and NOT. These logical operators have two similar, but very different, uses in a BASIC program. The first use—the one I will discuss here—is to manipulate the individual bits in an integer or long integer variable. The second use is for directing a program's flow, and that will be covered in Chapter 3.

Each of the bit manipulation operators performs a very simple Binary function. Most of these functions operate on the contents of two integers, using those bits that are in an equivalent position. The examples shown in Figure 2-6 use a single byte only, solely for clarity. In practice, the same operations would be extended to either the sixteen bits in an integer, or the 32 bits in a long integer.

The examples given here use the same decimal values 13 and 25, and these are also shown in their Binary equivalents. What is important when viewing Binary numbers is to consider the two bits in each vertical column. In the first example, the result in a given column is 1 (or True) only when that bit is set in the first number AND the same bit is also set in the second. This condition is true for only two of the bits in these particular numbers. The result bits therefore represent the answer in Binary, which in this case is `13 AND 25 = 9`. What is important here is not that `13 AND 25` equals 9, but how the bits interact with each other.

```

13 = 0000 1101
25 = 0001 1001
-----
    0000 1001

```

result when AND is used

↑ ↑
both of the bits are set
In each column

```

13 = 0000 1101
25 = 0001 1001
-----
    0001 1101

```

result when OR is used

↑ ↑ ↑ ↑
one or both bits are set
In each column

```

13 = 0000 1101
25 = 0001 1001
-----
    0001 0100

```

result when XOR is used

↑ ↑
the bits are different
In each column

```

13 = 0000 0000 0000 1101
    1111 1111 1111 0010
-----

```

result after using NOT

Figure 2-6 Bit functions

The second example shows OR at work, and it sets the result bits for any position where a given bit is set in one byte OR that bit is set in the other. Of course, if both are set the OR result is also true. In this case, four of the columns have one bit or the other (or both) set to 1. By the way, these results can be proven easily in BASIC by simply typing the expression. That is, `PRINT 13 OR 25` will display the answer 29.

The third example is for XOR, which stands for Exclusive Or. XOR sets a result bit only when the two bits being compared are different. Here, two of the bits are different, thus `13 XOR 25 = 20`. Again, it is not the decimal result we are after, but how the bits in one variable can be used to set or clear the bits in another.

The NOT operator uses only one value, and it simply reverses all of the bits. Any bit that was a 1 is changed to 0, and any bit that had been 0 is now 1. A full word is used in this example, to illustrate the fact that NOT on any positive number makes it negative, and vice versa. As you learned earlier in this chapter, the highest, or left-most bit is used to store the sign of a number. Therefore, toggling this bit also switches the number between positive and negative. In this case, `NOT 13 = -14`.

All of the logical operators can be very useful in some situations, although admittedly those situations are generally when accessing DOS or interfacing with assembly language routines. For example, many DOS services indicate a failure such as "File not found" by setting the Carry flag. You would thus use

AND after a CALL Interrupt to test that bit. Another good application for bit manipulation is to store True or False information in each of the sixteen bits in an integer, thus preserving memory. That is, instead of sixteen separate Yes/No variables, you could use just one integer.

Bit operations can also be used to replace calculations in certain situations. One common practice is to use division and MOD to break an integer word into its component byte portions. The usual way to obtain the lower byte is $LoByte\% = Word\% \text{ MOD } 256$, where MOD provides the remainder after dividing. While there is nothing wrong with doing it that way, $Word\% = LoByte\% \text{ AND } 255$ operates slightly faster. Division is simply a slower operation than AND, especially on the 8088. Newer chips such as the 80286 and 80386 have improved algorithms, and division is not nearly as slow as with the older CPU. Chapter 3 will look at some other purely BASIC uses of AND and OR.

Summary

As you have seen in this chapter, there is much more to variables and data than the BASIC manuals indicate. You have learned how data is constructed and stored, how the compiler manipulates that data, and how to determine for yourself the amount of memory that is needed and is available. In particular, you have seen how data is copied frequently but with no indication that this is happening. Because such copying requires additional memory, it is a frequent cause of "Out of memory" errors that on the surface appear to be unfounded.

You have also learned about BASIC's near and far heaps, and how they are managed using string and array descriptors. With its dynamic allocation methods and periodic rearrangement of the data in your program, BASIC is able to prevent memory from becoming fragmented. Although such sophisticated memory management techniques require additional code to implement, they provide an important service that programmers would otherwise have to devise for themselves.

Finally, you have learned how the various bit manipulation operations in BASIC work. This chapter will prove to be an important foundation for the information presented in upcoming chapters. Indeed, a thorough understanding of data and memory issues will be invaluable when you learn about accessing DOS and BIOS services in Chapter 11.

3

Programming Methods

In Chapters 1 and 2 you learned how the BASIC compiler translates a source file into the equivalent assembly language statements, and how it allocates memory to store variables and constants. In particular, you saw that the BC compiler generates assembly language code directly for some statements, while for others it creates calls to routines in the BASIC libraries. Most of the code examples presented in that chapter dealt with simple variable assignments and calculations.

Of course, the compiler must do much more than merely assign and manipulate variables and other data. Equally important is controlling how your program operates, and determining which paths are to be taken as it progresses. In this chapter we will delve into the inner workings of control flow structures, with an eye toward writing programs that are as efficient as possible. As with the earlier chapters, this discussion includes numerous disassemblies of compiled BASIC code. Thus, you will see exactly what the compiler does, and how each control flow statement is handled.

This chapter also discusses the design of both static and non-static subprograms and functions, and compares the relative merits of each method. Many programmers do not fully understand the term Static, and find the related subject of recursive subroutines especially difficult to grasp.

BASIC supports four types of subroutines, and each will be described in this chapter: GOSUB routines, subprograms, DEF FN functions, and what I call "formal functions". You will notice that I use the terms subroutine and procedure interchangeably, to indicate a single block of code that may be executed more than once. You will also learn how parameters are passed to these procedures.

Finally, in this chapter I will discuss programming style. Programming in any language is arguably as much of an art as it is a science. But unlike, say, music, where a composer can write any sequence of notes and proclaim them acceptable, a computer program must at least work correctly. There are an infinite number of ways to accomplish any programming task, and I can make recommendations only. Which approach you choose will reflect both your own personal taste and style, as well as your current level of competence and understanding of programming in general.

Control Flow

All programs—regardless of the language in which they are written—require a mechanism for testing certain conditions and then performing different actions based on those conditions. Although there are many ways to perform tests and branches in a BASIC program, all of them do essentially the same thing. The BASIC control flow statements are GOTO, DO/LOOP, WHILE/WEND, IF/THEN/ELSE,

FOR/NEXT, SELECT CASE, ON GOTO, and ON GOSUB. Because the capabilities of WHILE/WEND are also available with a DO/LOOP construct, the two will be discussed together.

In almost all cases, the BASIC compiler directly generates the code that controls a program's flow. One exception is when floating point values are used as a FOR counter, or as a WHILE or UNTIL condition. In those situations, calls are made to the floating point comparison routines in the BASIC runtime library. Another place is when you have a statement such as `CASE ASC(X$), or IF LEFT$(X$, 10) = Y$`. `ASC` and `LEFT$` are also subroutines in the BASIC language library, and they too are invoked by calls.

It is important to reiterate that when dealing with integer test conditions, BC will in many cases create assembly language code that is as good as a human programmer would write. In the short program fragment that follows, all of the BASIC source code is shown translated to the equivalent assembly language statements. This listing was derived by compiling and linking the BASIC program for Microsoft CodeView, and then using CodeView to display the resultant code.

This is what you write:

```
DO
  X% = X% + 1
LOOP WHILE X% < 100
```

This is the result after compilation:

```
30:
  INC  WORD PTR [X%]           ;X% = X% + 1
  CMP  WORD PTR [X%],64       ;compare X% to 100
  JL   30                     ;jump if less to 30
```

Here the variable `X%` is incremented, and then compared to the value 100. (64 is the Hex equivalent to 100, which is how CodeView displays values.) If `X%` is indeed less than 100, the program jumps back to address 30 and continues processing the loop. Notice that while this example does not use a named label in the BASIC source code as the target for a `GOTO`, the equivalent assembly language code does. In this case, the label is the code at address 30. Do not confuse the addresses that assembly language must use as jump targets with the numbered labels that in BASIC are optional.

The Dreaded GOTO

Modern programming philosophy dictates that `GOTO` and `GOSUB` statements should be avoided at all cost, in favor of `DO` and `WHILE` loops. However, all of these methods result in nearly identical code. Indeed, there is nothing inherently wrong with using `GOTO` when circumstances warrant it. By examining the program listing below, you will see that BASIC generates code that is identical for a `GOTO` as for a `DO` loop.

This is what you write:

```

Label:
  X% = X% + 1
  IF X% < 100 THEN GOTO Label

```

This is the result after compilation:

```

30:
  INC WORD PTR [X%]           ;X% = X% + 1
  CMP WORD PTR [X%],64       ;compare X% to 100
  JL 30                       ;jump if less to 30

```

Since GOTO and DO/LOOP produce the same results, which one is better, and why? In general, a DO/LOOP is preferable for two reasons. First, it is a nuisance to have to create a new and unique label name for every location that a program may need to branch to. Admittedly, in a short program this will not be a problem. But in a large application with many small loops that test for keyboard input, you end up creating many labels with names such as GetKey1, GetKey2, and so forth. And if you inadvertently use the wrong label name, your program will not work correctly.

More important, however, is that for each label you define in a program, the BC compiler must remember its name and the equivalent address in the object code that the label identifies. Since label names can be as long as 40 characters and memory addresses require 2 bytes each to identify, a finite number of label names can be accommodated. By avoiding unnecessary labels, you are giving BC that much more memory to use for compiling your program.

There are several situations in which GOTO is preferable to a DO or WHILE loop. Indeed, one of my personal pet peeves is when a programmer tries to shoehorn structure into a program no matter what the cost. Consider the three different code fragments below; each waits for a key press and then assigns it to the variable Ky\$.

This approach is the worst:

```

Ky$ = ""
WHILE Ky$ = ""
  Ky$ = INKEY$
WEND

```

This method is better:

```

Label:
  Ky$ = INKEY$
  IF Ky$ = "" GOTO Label

```

And this is better still:

```

DO
  Ky$ = INKEY$
LOOP WHILE Ky$ = ""

```

In the first example, an extra step is needed solely to clear Ky\$ to a null string, so the initial WHILE will be true and execute at least once. Every string assignment adds 13 bytes to a program, and those 13 bytes can add up quickly in a large application.

The second example avoids the unnecessary assignment, but adds a label for GOTO to jump to. Although this label does require a small amount of additional memory while the program is being compiled, it does not increase the size of the final executable program file.

The last example is better still, because it avoids the need for a line label and also avoids an extra string assignment. Since a DO loop allows the test to be placed at either the top or bottom of the loop, you can force the loop to be executed at least once by putting the test at the bottom as shown here.

However, even this can be improved upon by eliminating the string comparison that checks if Ky\$ is equal to a null string. If we replace LOOP WHILE Ky\$ = "" with LOOP UNTIL LEN(Ky\$), only 13 bytes of code are generated instead of 15. When two strings are compared (Ky\$ and ""), each must be passed to the string comparison routine. Since LEN requires only one argument, the code to pass the second parameter is avoided. There are some situations for which the GOTO is ideally suited. In the first two examples below, a complex expression is used as the condition for executing a DO WHILE loop, and the same expression is then used again within the loop.

```
DO WHILE (X% + Y%) * Z% > 13
  IF (X% + Y%) * Z% = 100 THEN PRINT
  ...
  ...
LOOP

DO WHILE ASC(MID$(S$, A%, B%)) > 13
  IF ASC(MID$(S$, A%, B%)) > 100 THEN PRINT
  ...
  ...
LOOP

Label:
  Temp% = ASC(MID$(S$, A%, B%))
  IF Temp% > 13 THEN
    IF Temp% > 100 THEN PRINT
    ...
    ...
  GOTO Label
END IF
```

In the first example, BASIC remembers the results of its test that checks if a $(X\% + Y\%) * Z\%$ is greater than 13, and it uses the result it just calculated in the next test that compares the same expression to 100. This is one more example of the kinds of optimizations BC performs as it compiles your programs. String expressions such as those used in the second example are of necessity more complex, and require calls to library routines. With this added complexity, BASIC unfortunately cannot retain the result of the earlier comparison, and it generates identical code a second time.

A more elegant solution in this case is therefore the GOTO as shown in the last example. Because the result of evaluating the expression is saved manually, it may be reused within the loop. As proof, the

second DO WHILE example above requires 73 bytes to implement, as opposed to only 53 when Temp% and GOTO are used.

I should also point out that the most common and valuable use for GOTO is to get out of a deeply nested series of IF or other blocks of code. It is not uncommon to have a FOR/NEXT loop that contains a SELECT CASE block, and within that a series of IF/ELSE tests. The only way to jump out of all three levels at once is with a GOTO.

FOR/NEXT Loops

Unlike WHILE and DO loops that can test for nearly any condition and at either the top or bottom of the loop, a FOR/NEXT loop is intended to perform a block of statements a fixed number of times. A FOR/NEXT loop could also be replaced with code that compares a value and uses GOTO to reenter the loop if needed, but that is hardly necessary. My point is to yet again illustrate that all of BASIC's seemingly fancy constructs are no more than tests and GOTOs deep down at the assembly language level.

A FOR/NEXT loop determines the number of iterations that will be executed once ahead of time, before the loop begins. For example, the listing below shows a loop that changes the upper limit inside the loop. However the loop still executes 10 times.

```
Limit% = 10
FOR X% = 1 TO Limit%
  Limit% = 5
  PRINT Limit%
NEXT
```

The code that BASIC produces for the FOR/NEXT loop in the previous example is translated to the following equivalent during the compilation process.

```
Limit% = 10
Temp% = Limit%
X% = 1
GOTO Next:
For:
  Limit% = 5
  PRINT Limit%
  X% = X% + 1
Next:
  IF X% <= Temp% THEN GOTO For
```

Please understand that changing a loop condition inside the loop is considered bad practice, because the program becomes difficult to understand. If you really need to alter the limit inside a loop, the loop should be re-coded to use WHILE or DO instead. Another good reason for avoiding such code is because it is possible that future versions of BASIC will behave differently than the one you are using now. If Microsoft were to modify BASIC such that the limit condition were reevaluated at the NEXT statement, your code would no longer work. It is also considered bad practice to modify the loop

counter variable itself (X% in the previous examples). However, this causes no real harm, and you should not be afraid to do that if the situation warrants it. Of course, changing the loop counter will affect the number of times the loop is executed.

IF/THEN/ELSE and SELECT Case

BASIC provides two methods for testing conditions in a program, and executing different blocks of code based on the result. The most common method is the IF test, which can be used on a single variable, the result of an expression, the returned value from a function, or any combination of these. I won't belabor the most common uses for IF here, but I do want to point out some of its less obvious properties. Also, there are some situations where IF and ELSEIF are appropriate, and others where their counterpart, SELECT CASE, is better.

As you have already learned, a simple IF test will in most cases be translated into the equivalent assembler instructions directly. In some cases, however, the condition you specify is tested, while in others the opposite condition is tested. If you say IF X > 10 THEN GOTO Label, BASIC may change that to IF X <= 10 GOTO [next statement]. Which BASIC uses depends on what you will do if the condition is true, and how far away in the generated code the statements that will be executed are located. When a GOTO is to be performed if the test passes, then the relative position of the target label is also a factor.

A jump to a location either ahead in the code or more than 128 bytes backwards requires BASIC to generate more code. The 128 byte displacement is significant, because the 80x86 can perform a *conditional jump* to an address only a limited distance away. That is, after a comparison is made, the target address for a conditional jump such as "Jump if Greater" must be no more than that many bytes distant. However, an unconditional jump can be to any address within the same 64K code segment. (Bear with me for a moment, because the significance of this will soon become apparent.) This is shown in the next listing:

```
IF X% = 100 THEN
  CMP Word Ptr [X%],64 ;compare X% to 100
  JE 003A ;jump ahead if equal
  JMP Label ;else, skip ahead
003A: ;BASIC made this label
Y% = 2
MOV Word Ptr [Y%],2
END IF

Label:
IF X > 8 GOTO Label
CMP Word Ptr [X%],8 ;compare X% to 8
JG Label ;jump back if greater
```

In the first example above, BASIC compares the value of X% to 100 (64 Hex), and if equal jumps ahead to a label it created at address 003A Hex. Otherwise, a jump is made to the next statement in the program, which in this case is a named label. Although using two jumps may seem unnecessarily convoluted, it is necessary because BASIC has no way of knowing how many statements will follow at

the time it compiles the IF test. Thus, it also cannot know whether the statement following the END IF will end up being 128 or more bytes ahead.

By jumping to another, unconditional jump, BC is assured that the generated code will be legal. (When BC finally encounters the END IF, it goes back to the code it created earlier, and completes the portion of the unconditional jump instruction that tells how far to go.) Some compilers avoid this situation and create the longer, two-jump code on a trial basis, but then go back and change it to the shorter form if possible. These are called two-pass compilers, because they process your source code in two phases. Unfortunately, current versions of Microsoft BASIC do not use more than one pass.

In the second example Label has already been encountered, and BC knows that the label is within 128 bytes. Therefore, it can translate the IF statement directly, without having to conditionally jump to yet another jump. Had the earlier label been farther away, though, an extra jump would have been needed. It is important to understand that forward jumps are always handled with more code than is likely necessary, because BASIC does not know how far ahead the jump must go. In fact, this same issue must be dealt with when writing in assembly language, since the conditional jump distance limitation is inherent in the 80x86 microprocessor.

The bottom line, therefore, is that you can in many cases reduce the size of your programs by controlling in which direction a conditional jump will be performed. For example, almost all programs must at some point sit in a loop waiting until a key is pressed. The next listing shows two common ways to do this, with one testing for a key press at the top of the loop, and the other doing the test at the bottom.

```
DO UNTIL LEN(INKEY$)      ;this comprises 18 bytes
0030:
  CALL B$INKY             ;call INKEY$
  PUSH AX                 ;pass the result to LEN
  CALL B$FLEN             ;AX now holds the length
  AND AX,AX               ;see if it's zero
  JZ 0042                 ;yes, jump to LOOP
  JMP 0044                 ;no, jump out of loop
0042:
LOOP
  JMP 0030                 ;jump back to DO

0044:
DO
LOOP UNTIL LEN(INKEY$)    ;this is only 15 bytes
  CALL B$INKY             ;call INKEY$
  PUSH AX                 ;as above
  CALL B$FLEN             ;
  AND AX,AX               ;
  JZ 0044                 ;jump back if zero
```

Viewed from a purely BASIC perspective, these two examples operate identically. But as you can see, the code that BASIC creates is more efficient for the second example. When BASIC encounters the first DO statement, it has no idea how many more statements there will be until the terminating LOOP. Therefore, it has no recourse but to create an extra jump. In the second example, the location of the DO

is already known to be within 128 bytes, so the LOOP test can branch back using the shorter and more direct method.

An ELSEIF statement block is handled in a similar fashion, with code that directly compares each condition and branches accordingly. Because the code to be executed if the IF is true is always after the IF test itself, the less efficient two-jump code must be generated. A simple IF/ELSEIF follows, shown as a mix of BASIC and assembly language statements.

```
IF X% > 9 THEN
  CMP Word Ptr [X%],9 ;compare X% to 9
  JG 003A ;assign Y% if greater
  JMP 0043 ;else jump to next test
003A:
Y% = 1
  MOV Word Ptr [Y%],1 ;assign Y%
  JMP 0066 ;jump out of the block
ELSEIF X% > 5 THEN
0043:
  CMP Word Ptr [X%],5 ;as above
  JG 004D
  JMP 0066
004D:
Y% = 2
  MOV Word Ptr [Y%],2
END IF
0066:
...
...
```

Aside from the additional jumping over jumps that are added to all forward address references, this code is translated quite efficiently. In this situation, the compiled output is identical to that produced had SELECT CASE been used. However, there is one important situation in which SELECT CASE is more efficient than IF and ELSEIF.

For each ELSEIF test condition, code is generated to create a separate comparison. When a simple comparison such as $X\% > 9$ is being made, only one assembly language statement is needed. But when an expression is tested—for example, $ABS((X\% + Y\%) * Z\%) > 9$ —identical code is generated repeatedly. This is illustrated in the listing that follows.

```
IF ABS((X% + Y%) * Z%) = 5 THEN
  A% = 1
ELSEIF ABS((X% + Y%) * Z%) = 6 THEN
  A% = 2
ELSEIF ABS((X% + Y%) * Z%) = 7 THEN
  A% = 3
END IF
```

Each time BC encounters the expression $ABS((X\% + Y\%) * Z\%)$, it duplicates the same assembly language statements. But when SELECT CASE is used, the expression is evaluated once, and used for each subsequent test. The first example in the next listing shows how SELECT CASE could be used to provide the same functionality as the preceding IF/ELSEIF block, but with much less code. The second example then shows what SELECT CASE really does, using an IF/ELSEIF equivalent.

You write it this way:

```
SELECT CASE ABS((X% + Y%) * Z%)
CASE 5: A% = 1
CASE 6: A% = 2
CASE 7: A% = 3
CASE ELSE
END SELECT
```

BASIC really does this:

```
Temp% = ABS((X% + Y%) * Z%)
IF Temp% = 5 THEN
  A% = 1
ELSEIF Temp% = 6 THEN
  A% = 2
ELSEIF Temp% = 7
  A% = 3
END IF
```

As you can see, SELECT CASE evaluates the expression once, stores the result in a temporary variable, and then uses that variable repeatedly for all subsequent comparisons. Therefore, when the same expression is to be tested multiple times, SELECT CASE will be more efficient than IF and ELSEIF. This is also true for string expressions and other functions. For example, SELECT CASE LEFT\$(Work\$, 10) will result in less code and faster performance than using IF and ELSEIF with that same expression more than once.

Another important feature of SELECT CASE is its ability to use either variable or constant test conditions, and to operate on a range of values. For example, the C language Switch statement which is the equivalent of BASIC's SELECT CASE can use only constant numbers for each test. BASIC is particularly powerful in this regard, and allows any legal expression for each CASE condition. For example, CASE IS > (Y AND Z) is valid, and so is CASE 0 TO Max. CASE also accepts multiple conditions separated by commas such as CASE 1, 3, 4 TO 100, -10 TO -1. In this case, the statements that follow will be executed if the selected expression equals 1, 3, any value between 4 and 100 inclusive, or any value between -10 and -1 inclusive.

It is also worth mentioning here that QuickBASIC version 4.0 contains an interesting and irritating quirk that requires a CASE ELSE in the event that none of the tests match. Had the CASE ELSE been omitted from the previous example and the value of the expression was not between 5 and 7, QuickBASIC 4.0 would issue a "CASE ELSE expected" error at run time. Fortunately, this has been repaired in QuickBASIC 4.5 and later versions.

Notice that this is not a bug in QuickBASIC. Rather, it is the behavior described in the ANSI (American National Standards Institute) specification for BASIC. At the time QuickBASIC 4.0 was introduced, Microsoft mistakenly believed the then-proposed ANSI standard for BASIC would be significant. As that standard approached fruition, it became clear to Microsoft that the only standard most programmers really cared about was Microsoft's.

One final point I cannot make often enough is the inherent efficiency of integer operations and comparisons. This is especially true in the comparisons that are made in both IF and CASE tests. In the first example below, each of the characters in a string is tested in turn. The second example shows a much better way to write such a test, by obtaining the ASCII value once and using that for subsequent integer comparisons.

Not recommended:

```
FOR X = 1 TO LEN(Work$)
  SELECT CASE MID$(Work$, X, 1)
    CASE CHR$(9): PRINT "Tab key"
    CASE CHR$(13): PRINT "Enter key"
    CASE CHR$(27): PRINT "Escape key"
    CASE "A" TO "Z", "a" TO "z": PRINT "Letter"
    CASE "0" TO "9": PRINT "Number"
  END SELECT
NEXT
```

Much more efficient:

```
FOR X = 1 TO LEN(Work$)
  SELECT CASE ASC(MID$(Work$, X, 1))
    CASE 9: PRINT "Tab key"
    CASE 13: PRINT "Enter key"
    CASE 27: PRINT "Escape key"
    CASE 65 TO 90, 97 TO 122: PRINT "Letter"
    CASE 48 TO 57: PRINT "Number"
  END SELECT
NEXT
```

In the first program the SELECT itself generates 27 bytes, which is comprised of a call to the MID\$ function and then a call to the string assign routine. A string assignment is needed to save the MID\$ result in a temporary variable for the subsequent tests that follow. Each CASE test that uses CHR\$ adds 27 bytes, and this includes the call to CHR\$ as well as an additional call to the string comparison routine. Testing for the letters adds 75 bytes, and testing for the numbers adds 39 more. This results in a total code size of 222 bytes, not counting the FOR/NEXT loop.

Contrast that with only 131 bytes for the second example, in which the SELECT portion requires only 26 bytes. Although an extra call is needed to obtain the ASCII value of the extracted character, the lack of a subsequent string assignment more than makes up for that. Further, the tests for 9, 13, and 27 require only 13 bytes each, compared to 27 when CHR\$ values were used. The letters test requires 43 bytes, and the numbers test only 23.

Clearly this is a significant improvement, especially in light of the small number of tests that are being performed here. In a real program that performs hundreds of string comparisons, replacing those with integer comparisons where appropriate will yield a substantial size reduction.

AND, OR, EQV, and XOR

When you use AND or OR in an IF test, what is really being compared is either 0 or -1. That is, BASIC evaluates the truth of each expression being tested on both sides of the AND or OR, and a truth in BASIC always results in one or the other of these values. Once each expression has been evaluated, the results are combined using an assembly language AND or OR instruction, and a branch is then made accordingly. Remember that when integers are treated as unsigned, setting all of the bits to 1 results in a value of -1.

In chapter 2 I showed how the various logical operators are used to manipulate bits in an integer or long integer variable. The concept is identical when these operators are used for decision-making in a BASIC program. The difference is really more a matter of semantics than definition. That is, the same bit manipulation is performed, only in this case on the result of the truth of a BASIC expression. This is shown in context below, where two test expressions are combined using AND.

```

IF X > 1 AND Y < 2 THEN
  CMP Word Ptr [X%],1 ;compare X% to 1
  MOV AX,0 ;assume False
  JLE 003B ;we assumed correctly
  DEC AX ;wrong, decrement to -1
003B:
  CMP Word Ptr [Y%],2 ;now compare Y% to 2
  MOV CX,0000 ;assume False
  JGE 0046 ;we assumed correctly
  DEC CX ;wrong, decrement to -1
0046:
  AND CX,AX ;combine the results
  AND CX,CX ;(this is redundant)
  JNZ 004F ;if not 0 assign Z%
  JMP 0055 ;else jump past END IF
Z = 3
004F:
  MOV Word Ptr [Z%],3 ;assign Z%
END IF
0055:
  ...
  ...

```

The result of the first comparison is saved in the AX register as either 0 or -1, and the second is saved in CX using similar code. Once both tests have been performed and AX and CX are holding the appropriate values, the registers are then tested against each other using AND. The instruction AND CX, AX not only combines the results, but it also sets the CPU's Zero Flag to indicate if the result was zero or not. Therefore, the second test that uses AND to compare CX against itself to check for a zero result is redundant. At only 2 additional bytes, the impact on a program's size is not terribly significant. However, this shows first-hand the difference between code written by a compiler and code written by a person.

OR conditions are handled similarly, except the assembly language OR instruction is used instead of AND. When multiple conditions are being tested using combinations of AND and OR and perhaps nested parentheses as well, additional similar code is employed.

There are many situations where all that is really necessary is to test for a zero or non-zero condition. For example, it is common to use an integer variable as a True/False "flag" which can be set in one part of a program, and tested in another. By understanding the underlying code that BASIC creates, you can help BASIC to reduce the size of your programs enormously. In particular, avoiding a comparison with an explicit value lets BASIC generate fewer comparison instructions. The listing below shows how you can test multiple flags using AND, but with much less resulting code than using an explicit comparison.

```

IF Flag1% AND Flag2% THEN
  MOV  AX,[Flag2%]      ;move Flag2% into AX
  AND  AX,[Flag1%]     ;AND that with Flag1%
  AND  AX,AX           ;(this is redundant)
  JNZ  0063            ;if not zero assign Z%
  JMP  0069            ;else skip past END IF
Z% = 3
0063:
  MOV  Word Ptr [Z%],3
END IF
0069:
  ...
  ...

```

The key here is that zero is always used to represent False, and -1 to represent a True condition. That is, instead of writing `IF Flag1% = -1 AND Flag2% = -1`, using `IF Flag1% AND Flag2%` provides the same results. At only 20 bytes of generated code, this method is far superior to tests for an explicit -1 which require 37 bytes. If you recall, in Chapter 2 I showed how the various bits in a variable can be turned on or off with AND. Thus, `1111 AND 1111` equals `1111`, while `1111 AND 0000` equals `0`.

Notice that using 0 and -1 has many other benefits as well. For example, the NOT operator which was also described in Chapter 2 can toggle a variable between those values. If all of the bits in a variable are presently zero, then `NOT Variable%` results in all ones (-1). This property can also be used to enhance a program's readability, by using NOT much like you would in an English sentence. For example, the code following the line `IF NOT Flag% THEN` will be executed if Flag% is 0 (False), but it will not be executed if Flag% is -1 (True).

In fact, an explicit comparison is optional if you need to test only for a non-zero value. `IF Variable <> 0 THEN` can be reduced to `IF Variable THEN`, and the statements that follow will be executed as long as Variable is not 0. Notice that the only saving here is in the BASIC source, since either comparison creates ten bytes of assembler code. But when using long integers, the short form saves five bytes—14 bytes versus 19 for an explicit comparison to zero.

NOT is equally valuable when toggling a flag variable between two values. If you have, say, an input routine that keeps track of the Insert key status, then you could use `Insert% = NOT Insert%` each time you detect that the Insert key was pressed. The first time the operator presses that Key, the Insert flag will be switched from the default start-up value of 0 to -1. Then using `Insert% = NOT Insert%` a second time will revert the bits back to all zeros. In fact, it is a common technique to define True and False variables (or constants) in a program using this:

```
False% = 0
True% = NOT False%
```

Most programmers understand how to use parentheses to force a particular order of evaluation. By default, BASIC performs multiplication and division before it does addition and subtraction. When operators of the same precedence are being used, then BASIC simply works from left to right. However, the order in which logical comparisons are made is not always obvious. This can become particularly tricky if you are using some of the shorthand methods I described earlier.

For example, consider the statements `IF X AND Y > 12`, `IF NOT X OR Y`, and `IF X AND Y OR Z`. In the first example, the truth of the expression `Y > 12` is evaluated first, with a result of either 0 or -1. Then, that result is combined logically with the value of `X` using `AND`. The resulting order of evaluation is performed as if you had used `IF X AND (Y > 12)`. The other expressions are evaluated as `IF (NOT X) OR Y` and `IF (X AND Y) OR Z`.

The last logical operators we will consider are `EQV` and `XOR`. These are used rarely by most BASIC programmers, probably because they are not well understood. However, `EQV` can dramatically reduce the size of a program in certain circumstances. It is not uncommon to test if two conditions are the same, whether True or False. `EQV` stands for Equivalent, meaning it tests if the expressions are the same—either both true or both false. All three program fragments below serve the same purpose, however the first generates 57 bytes, while the second and third create only 16 bytes.

```
IF (X = -1 AND Y = -1) OR (X = 0 AND Y = 0) THEN
  . . .
END IF

IF X EQV Y THEN
  . . .
END IF

IF NOT (X XOR Y) THEN
  . . .
END IF
```

Although these examples could be replaced with a simple comparison that tests if `X` equals `Y`, `EQV` can reduce other, more elaborate `AND` and `OR` tests. For example, you could replace this:

```
IF (X = 10 AND Y = 100) OR (X <> 10 AND Y <> 100)
```

with this:

```
IF X = 10 EQV Y = 100
```

and gain a handsome reduction in code size. Notice that because of the way `EQV` works, the third example in the listing above results in identical assembly language code as the second. `XOR` is true only when the two conditions are different, thus `NOT XOR` is true when they are the same.

One final point worth mentioning is that you can assign a variable based on the truth of one or more expressions. As you saw earlier, every IF test that is used in a BASIC program adds a minimum of 3 extra bytes for a second, unconditional jump. That additional code can be avoided in many cases by assigning a variable based on whether a particular condition is true or not. In the code examples that follow, both program fragments do the same thing, except the first requires 25 bytes compared to only 14 for the second.

```
IF Variable = 20 THEN
  Flag = -1
ELSE
  Flag = 0
END IF
```

```
Flag = (Variable = 20)
```

In either case, the truth of the expression `Variable = 20` must be evaluated. However, the IF method adds code to jump around to different addresses that assign either -1 or 0 to Flag. The second example simply assigns Flag directly from the 0 or -1 result of the truth test. Other variants on this type of programming are statements such as `A = (B = C)`, and `Flag = (LEN(Temp$) <> 0 AND Variable < 50)`. Note that the surrounding parentheses are shown here for clarity only, and BASIC produces the same results without them.

Short Circuits

There is one important point regarding AND testing you should be aware of. Although the code that BASIC creates to implement these logical tests is very efficient, in some cases a different approach can yield even better results. When many conditions are tested, QuickBASIC creates assembly language code to evaluate all of them before making a decision. This can be wasteful, because often one of the conditions will be false, negating a need to test the remaining conditions. For example, this statement:

```
IF Any$ = "Quit" AND IntVar% > 100 AND Float! <> 0 THEN PRINT "True"
```

requires that all three conditions be tested before the program can proceed. But if Any\$ is not equal to "Quit", there is no need reason to spend time evaluating the other tests.

The solution is to instead use nested IF tests, preferably placing the most likely (or simplest) tests first, as shown below.

```
IF Any$ = "Quit" THEN
  IF IntVar% > 100 THEN
    IF Float! <> 0 THEN
      PRINT "True"
    END IF
  END IF
END IF
```

Here, if the first test fails, no additional time is wasted testing the remaining conditions. Further, using the nested IF tests with QuickBASIC also results in less code: 50 bytes versus 64. Note, however, that

BASIC PDS, and VB/DOS, incorporate a technique known as *short circuit expression evaluation*, which generates slightly more efficient code when AND is used. With the newer compilers, each condition is tested in sequence, and the first one that fails causes the program to skip over the code that prints "True". But even with this improved code generation, you should still place the most likely tests first.

ON GOTO and ON GOSUB Statements

The last non-procedural control flow statements I will discuss here—ON GOTO and ON GOSUB—are used infrequently by many BASIC programmers. But when you need to test many different values and those values are sequential, ON GOTO and ON GOSUB can reduce substantially the amount of code that BASIC generates. For clarity, I will use ON GOTO for most of the examples that follow. Both work in a similar fashion except with ON GOSUB, execution resumes at the next BASIC statement when the subroutine returns.

You have already seen that IF/ELSEIF and SELECT CASE blocks are not as efficient as they could be, because the compiler does not know how far ahead the END IF or END SELECT statements are located. Therefore, no matter how trivial the IF or CASE tests being performed are, a pair of jumps is always created even when a single jump would be sufficient. Further, when many tests are necessary, there is no avoiding at least some amount of code for each comparison. This is where ON GOTO can help.

Rather than perform a series of separate tests for each value being compared, ON GOTO uses a lookup table which is embedded in the code segment. This table is merely a list of addresses to branch to, based on the value of the variable or expression being evaluated. If the value being tested is 1, then a branch is taken to the first label in the list. If it is 2, the code at the second label is executed, and so forth.

As many as 60 labels can be listed in an ON GOTO statement, although the number being tested can range from 0 to 255. If the value is 0 or higher than the number of items in the list, the ON GOTO command is ignored, and execution resumes with the statement following the ON GOTO. Negative values or values higher than 255 cause an "Illegal function call" error. A simple example showing the basic syntax for ON GOTO is shown below.

```
INPUT "Enter a value between 1 and 3: ", X
ON X GOTO Label1, Label2, Label3
PRINT "Illegal entry!"
END
```

```
Label1:
  PRINT "You pressed 1"
  END
```

```
Label2:
  PRINT "You pressed 2"
  END
```

```
Label3:
  PRINT "You pressed 3"
  END
```

Notice that the more labels there are, the bigger the savings in code size. ON GOTO adds a fixed overhead of 70 bytes, 61 of which is the size of the library routine that evaluates the value and actually jumps to the code at the appropriate label. The remaining 9 bytes are needed to load the value being tested and pass that on to the ON GOTO routine. However, for each label in the list, only 2 bytes are required in the lookup table to hold the address.

Compare that to SELECT CASE which requires 6 bytes of set-up code (when an integer is being tested), and 13 bytes more to process each CASE. Thus, the crossover point at which ON GOTO is more efficient is when there are 6 or more comparisons. Notice that if ON GOTO is used in more than one place in a program, the savings are even greater because the 61-byte library routine is added only once.

Again, ON GOTO has the important restriction that all of the values must be sequential. However, this limitation can also be turned into a feature by taking advantage of the inherent efficiency of lookup tables.

Using a lookup table is a very powerful technique, because you can determine a result using an index rather than actually calculating the answer. A lookup table is commonly used to determine log and factorial functions, since those calculations are particularly tedious and time consuming. With a lookup table you would calculate all of the values once ahead of time, and fill an array with the answers. Then, to determine the factorial for, say, the number 14, you would simply read the answer from the fourteenth element in the array.

You can apply this same technique in BASIC using a combination of INSTR and ON GOTO or ON GOSUB. Although INSTR is intended to find the position of one string within another, it is also ideal for looking up characters in a table. Imagine you have written an input routine that must handle a number of different keys, and branch according to which one was pressed. One way would be to use an IF/ELSEIF or SELECT CASE block, with one section devoted to each possible key. But as you saw earlier, once there are more than 5 keys to be recognized, either of those constructs are less efficient than ON GOTO.

The approach I often use is to combine INSTR and ON GOSUB to branch according to which function key was pressed. The beauty of this method is that a value of zero (or one that is out of range) causes control to fall through to the next statement. Therefore any keys that are not explicitly being tested for are simply ignored. This is shown in context below.

```
DO
  DO                                'wait for a key press
    K$ = INKEY$
    Length% = LEN(K$)
  LOOP UNTIL Length%
```



```

IF Length% = 2 THEN      'it's an extended key
  Code$ = RIGHT$(K$, 1) 'isolate the key code and branch accordingly
  ON INSTR(";<=>?@ABCD", Code$) GOSUB ...
END IF

```

```

LOOP UNTIL K$ = CHR$(27) 'until they press Esc

```

Here, extended keys are identified by a length of 2, and the key code is then isolated with RIGHT\$. The punctuation and letters within the quotes are characters 59 through 68, which correspond to the extended codes for F1 through F10. (A list of all the extended key codes is in your BASIC owner's manual.) Of course, any arbitrary list of key codes could be used. Further, the key codes do not need to be contiguous. For example, to branch on the Up arrow, Down arrow, Ins, Del, PgUp, and PgDn keys you would use "HPRSIQ" as the source string. Any other mix of characters could also be used, including Alt keys.

Another interesting and clever trick that combines INSTR and ON GOTO lets you test multiple keys regardless of capitalization. The short program below accepts a character, and uses INSTR to look it up in a table of upper and lower case character pairs.

```

PRINT "Yes/No/Load/Save/Retry/Quit? ";
DO
  K$ = INKEY$
  LOOP UNTIL LEN(K$) = 1
  ON (INSTR("YyNnLlSsRrQq", K$) + 1) \ 2 GOTO ...

```

After adding 1 and dividing that by 2, the result will indicate in which character pair the choice was found. This technique could also be extended to include 3 or 4-character groups, or any other combination of characters. Since any value between 0 and 255 is legal for an ASCII character, INSTR can be used in other, more general lookup situations as well.

A Comparison of Subroutine Methods

There are four primary subroutine types that BASIC supports: GOSUB subroutines, DEF FN functions, called subprograms, and what I refer to as "formal functions". Each has its own advantages and disadvantages, which I will describe momentarily. But I would first like to introduce several terms that will be used throughout the discussion that follows.

Module

The first is *module*, which is a series of BASIC program statements kept in their own separate source file. All modules have a main portion, and some also have procedures within a SUB or FUNCTION block. The main portion of a program is that which receives control when the program is first run. When a program is comprised of multiple modules, each additional module has a main portion, although code within that portion is rarely executed. In fact, there are only two ways to access code in

the main portion of an ancillary module: One is to create a line label and use that as the target for ON ERROR or another "ON" event. The other is to define a DEF FN function and invoke the function.

Variable Scope

The second term is *variable scope*, which indicates where in a program a variable may be accessed. Variables that are used in the main portion of a program are accessible anywhere else in the main, but not within a SUB or FUNCTION block. Likewise, a variable that is defined within a SUB or FUNCTION is by default private to that procedure. The overwhelming advantage of private variables is that you do not have to worry about errors caused by inadvertently using the same variable name twice.

SHARED

The third term is *SHARED*, and it overrides the default private scope of a variable used in a procedure. SHARED may be used in either of two ways. If it is specified with a DIM statement in the main body of a program—that is, DIM SHARED Variable—the variable is established as being shared throughout the entire source file. Even though DIM is usually associated with arrays, it can be used this way to extend a variable's scope.

SHARED may also be used within a subroutine to share one or more variables with the main portion. Notice that the statement SHARED Variable inside a procedure defines the variable as being shared with the main portion of the program only. SHARED used within a procedure does not share the named variable with any other procedures. The only exception is when other procedures also use SHARED with the same variable name. In that case they are shared between procedures, as well as with the main program.

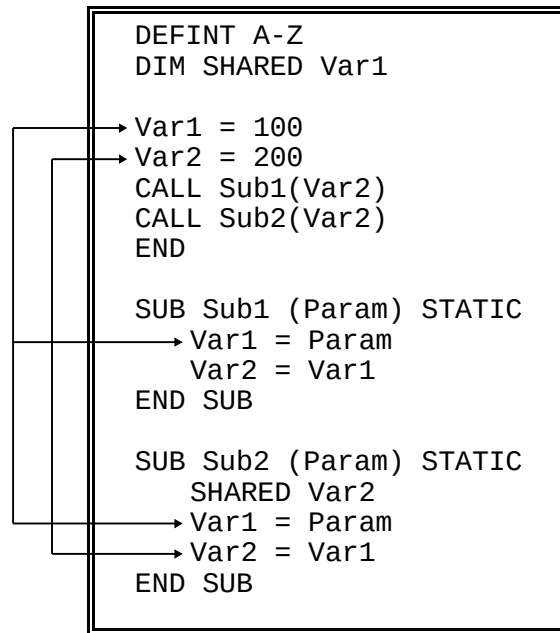


Figure 3-1: How SHARED and DIM SHARED affect variable scope. Variables that share the same identity are shown connected.

COMMON

The fourth term is *COMMON*, which is related to *SHARED* in that it also lets you share variables among procedures. However, *COMMON* has the additional property of allowing variables to be shared by procedures that are not in the same physical source file. When BC compiles your program, it translates your variable names to memory addresses. Thus, those names are not available when the program is linked to other object files. Variables that are listed in a *COMMON* statement are placed in a separate portion of the data segment which is reserved just for that purpose. Therefore, other program modules using *COMMON* can also access those variables in that portion of DGROUP.

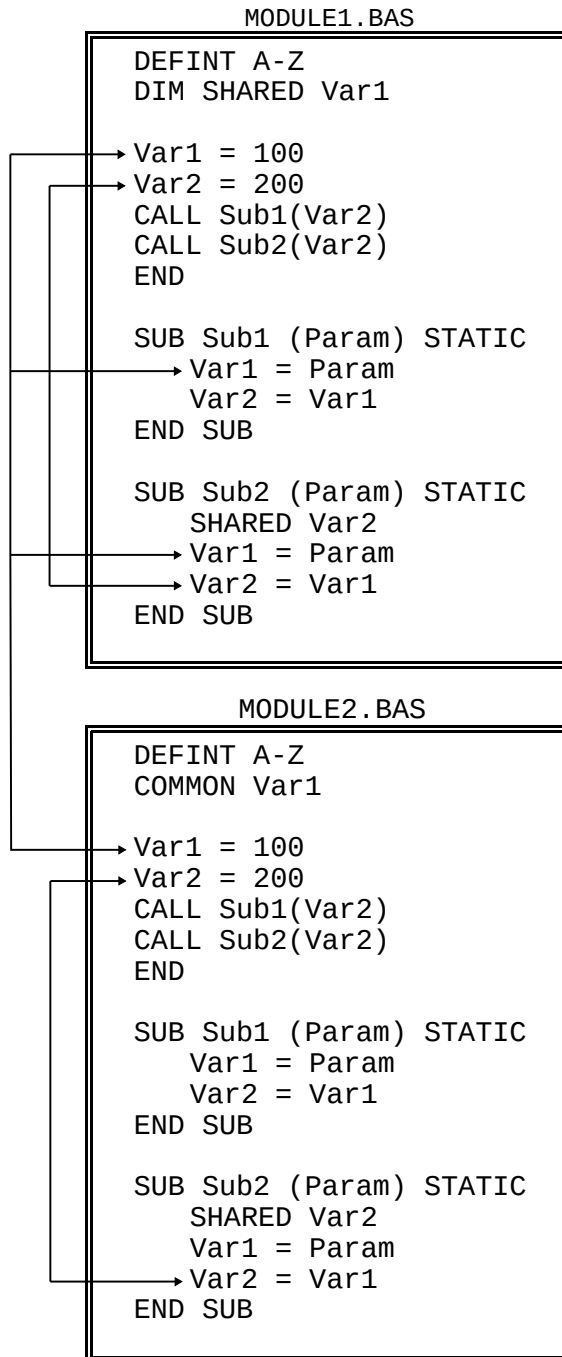


Figure 3-2: How COMMON and COMMON SHARED affect variable scope. Variables that share the same identity are shown connected.

COMMON can also be combined with SHARED, to specify that one or more variables be shared throughout the main program as well as with other modules. That is, the statement `COMMON SHARED Variable` tells BASIC that Variable is to be both DIM SHARED and COMMON. To establish a TYPE variable as COMMON, you must state the type name as well: `COMMON TypeVar AS MyType`. In all cases, COMMON statements must precede the executable statements in a program. The only statements that may appear before COMMON are other non-executable statements such as DECLARE, CONST, and '\$STATIC.

Because the variable names listed in a COMMON statement are not stored in the final program, the names used in one module do not need to be the same as the corresponding names in another module. You could, for example, have `COMMON X%, Y$, Z#` in one file, and `COMMON A%, B$, C#` in another. Here, X% refers to the same memory location as A%; Y\$ is the same variable as B\$, and so forth. It is imperative, however, that the order and type of variables match. If one file has an integer followed by a string followed by a double precision variable, then all other files containing a COMMON statement must have their COMMON variables in that same order.

This is one good reason for storing all COMMON statements in a single include file, which is included by each module that needs access to the COMMON variables.

One or more arrays may also be listed as COMMON; however, the rules are different for static and dynamic arrays. When a dynamic array is to be made COMMON, it should be dimensioned in the main program only, following the COMMON statement. (But you may use REDIM in another module if necessary, to change the array's size.) Static arrays must be dimensioned in each module, before the associated COMMON declaration. Of course, all array types must match across modules—you may not list a static array as the first COMMON item in one file, and then list a dynamic array in that same position in another file.

There are actually two forms of COMMON statement: the blank COMMON and the named COMMON. The examples shown thus far are blank COMMON statements. A named COMMON block lets you specify selected variable groups as COMMON, to avoid having to list many variables when all of them are not needed in a given module. A COMMON block is named by preceding the variable list with a name surrounded by slash characters. For instance, this line:

```
COMMON /IntVars/ X%, Y%, Z%
```

establishes a named COMMON block called IntVars. By creating several such named blocks you may share only those that are actually needed in a given module.

In this case, the block name is stored in the object file, and LINK ensures that the COMMON variables in each module share the same addresses. One important limitation of a named COMMON block is that it cannot be used to pass information between programs that use CHAIN.

STATIC

The fifth term is *STATIC*, which I described in a slightly different context in the section about data in Chapter 2. When you add the *STATIC* option to a *SUB* or *FUNCTION* definition, BASIC treats the variables within that procedure very differently than when *STATIC* is omitted. With *STATIC*, memory in *DGROUP* is allocated by the compiler for each variable, and that memory is permanently reserved for use by those variables.

When *STATIC* is not specified, the variables in the routine are by default placed onto the system stack. This means that sufficient stack memory must be available, although that memory can then be used again later for variables in other procedures. An important side effect of using the stack for variable storage is that the memory is cleared each time the subprogram or function is entered. Therefore, all numeric variables are initialized to zero, and strings are initialized to null. Any arrays within a non-static procedure are by default dynamic, which means they are created upon entry to the routine and erased when the routine exits.

STATIC also has an additional meaning in subprograms and functions; it can establish variables as being private to a procedure. If a variable has been declared as shared throughout a module by using *DIM SHARED* in the main portion of the program, using the statement *STATIC Variable* inside the subroutine will override that property. Thus, *Variable* will be local to the procedure, and will not conflict with a global shared variable of the same name. *STATIC* within a subprogram or function also lets you use the same name for a variable that was already given to a named constant.

Many programmers find the use of the term *STATIC* for two very different purposes confusing, and rightly so. It would have made more sense to use a different keyword, perhaps *LOCAL*, to limit a variable's scope. And to further confuse the issue, the '\$*STATIC* metacommand is used to establish the memory storage method for arrays. None the less, *STATIC* always indicates that memory for a variable is permanently allocated, and it may also specify that a variable is private to a procedure.

Recursion

The final term I want to introduce now is *recursion*. The classic definition of a recursive procedure is that it may call itself. While this is certainly true, that doesn't really explain what recursion is all about, or how it could be useful. I will cover recursion in depth momentarily, but for now suffice it to say that recursion is often helpful when manipulating tree-structured information.

For example, a program that lists all of the files on a hard disk would most likely be based on a recursive subroutine. Such a program would first change to the root directory, and then call the routine to read and display all of the file names it finds there. Then for each directory under the current one, the routine would change to that directory and call itself again to read and display the files in that directory. And if more directories were found at the next level down, the routine would call itself yet again to process all of those files too. This continues until all of the files in all directories on the hard disk have been processed.

Another application for recursion is a subroutine that sorts an array on more than one key. For example, consider a TYPE array in which each element has components for a first name, a last name, and address fields. You might want to be able to sort that array first by last name, then by first name, and then by Zip code. That is, all of the Smiths would be grouped together, and within that group Adam would be listed before John. All of the John Smiths would in turn be sorted in Zip code order.

By employing recursion, the routine would first sort the entire array based on the last name only. Next, it would identify each range of elements that contain identical last names. The routine would then call itself to sort that subgroup, and call itself again to sort the subgroup within that group based on Zip code.

Subroutines Versus Functions

There is a fundamental difference between subroutines and functions. A subroutine is accessed with either a CALL or GOSUB statement, and a function is invoked by referencing its name. In general, a subroutine is used to perform an action such as opening a group of files, or perhaps updating a screen-full of information. A function, on the other hand, returns a value such as the result of a calculation. A string function also returns information, although in this case that information is a string.

Notice that the type of information returned by a function is independent of the type of parameters, if any, that are passed to it. For example, BASIC's native STR\$ function accepts a numeric argument but returns a string. Likewise, a numeric function such as INSTR accepts two strings and returns a single integer. This is also true for functions that you design using either DEF FN or FUNCTION.

Although a function is primarily used for calculations and a subroutine for performing one or more actions, there is no hard and fast distinction between the two. You could easily design a subroutine that multiplies three numbers and returns the answer in one of the parameters. Similarly, a function could be written to clear the screen and then open a file. Which you use and when will depend on your own programming style. However, there are definite advantages to using functions where appropriate.

One immediately obvious benefit of a function is that a value can be returned without requiring an additional passed parameter. Each variable that is passed as a parameter requires 4 bytes of code for setup, plus an additional 5 bytes within the subroutine each time it is accessed.

Another important advantage of using a function is BASIC's automatic type conversion. If you assign a single precision variable from the result of an integer function, BASIC will convert the data from one format to the other transparently. In fact, a simple assignment from a variable of one type to that of another type is also handled for you by the compiler. But if a routine is written to pass the value back as a parameter, then you must use whatever type of data the subprogram expects.

Although most high-level languages require the programmer to match explicitly the types of data being assigned, Microsoft BASIC has done this automatically since its inception. When you write `Var1! =`

Var2%, BASIC treats that as $\text{Var1!} = \text{CSNG}(\text{Var2}\%)$. Object oriented programming languages use the term *polymorphism* to describe such automatic type conversion.

GOSUB Routines

The primary advantage a GOSUB routine holds over all of the other subroutine types is that it can be accessed very quickly. Translated to assembly language a GOSUB statement is but three bytes in length, and its speed is surpassed only by a GOTO. When the only thing that matters is how fast a subroutine can be called, GOSUB has the clear advantage. However, there are many limitations inherent in a GOSUB.

The most important restriction is that arguments cannot be passed using GOSUB. Therefore, any variables must be assigned before invoking the routine, and possibly reassigned when it returns. For example, if a subroutine requires two parameters—perhaps a row and column at which to print a message—those variables must be assigned before the GOSUB can be used. And if a value is being returned, your program must know the name of the variable that was assigned within the GOSUB routine.

Another important limitation is that the target line label must be in the same block of code as the GOSUB. Although a GOSUB is legal within a SUB or FUNCTION, both the GOSUB and the routine it calls must be located in the same procedure. Likewise, a GOSUB in the main body of a program cannot access a subroutine inside a procedure, or vice versa.

```
And of course you cannot invoke a GOSUB routine that is
located in a different source module.
```

Both of these problems restrict your ability to reuse a subroutine in more than one program. One of the goals of modern structured programming is the ability to design a routine for one application, and also use it again later in other programs. The only way to do that using GOSUB routines is to establish a variable naming convention, and always use variables and line labels with those unique names.

Subprograms

Subprograms were introduced with QuickBASIC version 2.0, and they improve greatly on GOSUB routines in many respects. The most important advantages of a subprogram are that it accepts passed parameters, and that variables used within the subprogram are local by default. Besides the obvious benefit of not having to worry about variable naming conflicts, these properties allow you to create your own toolbox of useful subroutines, and use them repeatedly in different programming projects. I will discuss this use of subprograms in detail later in this chapter.

A subprogram is accessed using the CALL statement, and any number of arguments may optionally be passed to the routine. A subprogram is defined with a statement of the form `SUB SubName`

(Param1, Param2, ...) STATIC. The parameters and surrounding parentheses are optional, as is the STATIC directive. Of course, the number of arguments passed to a subprogram must match the number of parameters it expects.

As you can see, subprograms have many advantages over GOSUB routines. However, they are not a magical panacea for every programming problem. Each subprogram includes a fixed amount of overhead just to enter and exit it. Because of the complexities of accessing incoming parameters, a *stack frame* must be created by the compiler upon entry. A stack frame is simply a fancy name for an area of memory that holds the addresses of the incoming parameter. However, this requirement adds a fair amount of code to each subprogram.

Eight bytes of code are needed to set up and call the internal BASIC routine that creates the stack frame, and the routine itself comprises another 35 bytes. Eight more bytes are needed to call the routine that exits a subprogram, and that routine adds contains 26 bytes. Finally, all but the last subprogram in a source file needs a 3-byte jump to skip over the other subprograms that follow. Therefore, a total of 80 bytes are added to any program that uses a subprogram rather than a GOSUB routine. It is important to point out, however, that the 61 bytes used by the library routines to enter and exit a subprogram are added to the final .EXE file only once.

It is also worth mentioning that BASIC PDS provides the /Ot switch, which eliminates the usual overhead incurred from calling the routines needed to enter and exit a subprogram. Although using /Ot avoids the code that is otherwise added, there is one important restriction: You may not use a GOSUB within the subprogram. When a program performs a GOSUB, the address to return to is placed onto the stack, for retrieval later when the subroutine returns. Likewise, when a subprogram is called, both a segment and address to return to are put on the stack.

If a GOSUB were used inside the subprogram and an EXIT SUB was then encountered within the GOSUBed subroutine, the return addresses on the stack would be out of order. Thus, the subprogram would return to the wrong place, with undoubtedly disastrous consequences. To avoid this, BASIC by default saves the address to return to when the subprogram is first entered, and uses that when it is exited. Therefore, when the compiler sees that a GOSUB is being used, it does not use the abbreviated method even if /Ot has been specified.

Although using /Ot makes a subprogram (and function) much faster by eliminating the overhead to call the entry and exit routines, there is no actual savings in code size. A series of assembler NOP (No Operation) instructions are placed where the entry and exit code would have been. However, those empty instructions are never executed. We can only hope that in future releases of BASIC PDS Microsoft will improve BC's code generation to eliminate these unnecessary instructions.

Another problem with subprograms is that programmers tend to use them to excess. For example, I have seen people create subprograms to increment and decrement integer variables even though it is far more efficient to do that with in-line code. The statement $X\% = X\% + 1$ creates only 4 bytes of code, compared to 9 for a single call to a subprogram to do the same thing! However, incrementing long integer or floating point variables does take more code than invoking a subprogram with a single

parameter, so a subprogram could be useful in that case. Only by counting the number of times a subprogram will be used and comparing that to the overhead incurred can you determine whether there will be any savings.

DEF FN Functions

Although a DEF FN function is designed to return a result, it is more closely related to a GOSUB subroutine in actual operation. Like a GOSUB routine it is invoked with a 3-byte assembly language "near" call, as opposed to the 5-byte "far" call that subprograms and formal functions require. And while a DEF FN function can accept incoming parameters, variables within the function definition are by default shared with the main portion of the program.

As I already explained, variables used in a DEF FN function can be made private to the function only by explicitly declaring them as `STATIC`. However, at least it is possible to employ local variables. Further, a DEF FN function can return a result, which makes it an ideal replacement for GOSUB when speed is paramount.

Internally, parameters are passed to a DEF FN function very differently than to a called subprogram or formal function. Arguments are passed to a subprogram by placing their addresses on the stack. With a DEF FN function, however, a copy of each parameter is created, and the function directly manipulates those copies. Therefore, it is impossible for a DEF FN function to modify an incoming parameter directly. This behavior is neither good nor bad. Rather, it is simply different and thus important to understand. It is also important to understand that a DEF FN function can be used only in the module in which it is defined. If the same function is needed in different modules, the same code must be duplicated again and again.

In the manuals that come with QuickBASIC and BASIC PDS, Microsoft advises against using DEF FN functions, in favor of the newer, more powerful formal functions. Because of this favoritism, Microsoft will probably never correct one disturbing anomaly that is present in all DEF FN functions. When a string is passed as an argument to a DEF FN function, a copy is made for the function to manipulate. Unfortunately, the copy is never deleted. Therefore, if you pass, say, a 10,000 byte string to a DEF FN function, that amount of memory is permanently taken until the function is invoked again later. The short listing below proves this behavior.

```
DEF FnWaste (A$)
  FnWaste = ASC(A$)
END DEF

Big$ = SPACE$(10000)
PRINT FRE(Big$)
X = FnWaste(Big$)
PRINT FRE(Big$)
```

Notice that running this program in the QuickBASIC editing environment will not give the expected (memory-wasting) result. However, in a separately compiled program the 10000 byte loss will be evident.

As with subprograms, there is a fixed amount of overhead required to enter and exit a DEF FN function. For each function that has been defined, 5 bytes are needed to call the Enter and Exit routines. Further, these routines are 14 and 24 bytes in length respectively. But again, the routines themselves are added to a program only once when it is linked.

There are two final limitations of DEF FN functions worth mentioning here. The first is that arrays and TYPE variables may not be passed as parameters to them. Since by design a copy is made of every incoming parameter, there is no reasonable way to do that with an entire array. The second limitation is that the function definition must be physically positioned in the source file before any references are made to it.

Formal Functions

A formal function is nearly identical to a called subprogram, and it requires the exact same amount of overhead to enter and exit. Also like subprograms, nearly any type of data may be passed to a function, including TYPE variables and arrays. The only limitation is that a fixed-length string may not be used directly as a parameter. If a fixed-length string is passed to a subprogram or function that expects a string, a copy is made and assigned to a conventional string. This copying was described in detail in Chapter 2.

Because a formal function is invoked by referencing its name in an assignment or PRINT statement, it is essential that it be declared. After all, how else could BASIC know that the statement `PRINT MyFunc` means to call a function and display the result, as opposed to printing the variable named `MyFunc`? When a BASIC function is created in the BASIC editing environment, a corresponding DECLARE statement is generated automatically. But when a function is written in another language or kept in a Quick Library, an explicit declaration is mandatory.

Like subprograms, formal functions are ideally suited to modular, reusable programming methods. Furthermore, a function may be accessed from any module in an entire application, even those in other source files. Indeed, the only difference between a subprogram and a function is that a function returns a result. The assembly language code that BASIC generates is in all other respects identical.

Static Versus Non-static Procedures

As I stated earlier, when the STATIC keyword is appended to a SUB or FUNCTION declaration, all of the variables within the routine are assigned a permanent address in DGROUP. And when STATIC is omitted, the variables are instead stored on the stack and cleared to zeros or null strings each time the routine is entered. There are several important ramifications of this behavior. Non-static procedures allocate new stack memory each time they are invoked, and then release that memory when they exit. It is therefore possible to exhaust the available stack space when the subroutine calls are deeply nested.

For example, if you call one subprogram that then calls another which in turns calls yet another, sufficient stack memory must be available for all of the variables in all of the subprograms. Besides the memory needed for each variable in a subprogram or function, other data is also placed onto the stack as part of the call. For each parameter that is passed, 2 bytes are taken to hold its address. Add to that 4 bytes to store the segment and address to return to in the calling program. Finally, temporary variables that BASIC creates for its own purposes are also stored on the stack in a non-static subprogram or function.

Another important consideration when `STATIC` is omitted is that every string variable must be deleted before the subprogram exits. Because of the way BASIC's string management routines operate, memory that holds string descriptors and string data cannot simply be abandoned. Every string must be released explicitly by a called routine, at a cost of 9 bytes per string. Please understand that you do not have to delete these strings. Rather, this is another case where BASIC creates additional code without telling you.

Again, I would love to be able to tell you that using `STATIC` is always desirable, or that never using it always makes sense. But unfortunately, it just isn't that simple. When a program becomes very large and complex, only by counting variables can you be absolutely certain how much stack space is really needed. Although the `FRE(-2)` function may be used to determine how much stack memory is currently available, it does not tell how much memory is actually needed by each routine.

To summarize the trade-offs between static and non-static variables: Static variables are allocated permanently by the compiler, and the memory they occupy can never be used for any other purpose. Non-static variables are placed onto the stack, and exist only while the subprogram or function is in use. Remember that you can also have a mix of static and non-static variables in the same procedure. By omitting `STATIC` after the subroutine name, all variables will by default be non-static. You can then override that property for selected variables by using the `STATIC` keyword. In the section on debugging in Chapter 4, you will learn how to use CodeView to determine the stack requirements for a procedure's variables.

Controlling the Stack Size

There are several ways to control the amount of memory that is dedicated for use by the stack. All versions of BASIC support the `CLEAR` command, which takes an optional argument that sets the stack size. The statement `CLEAR , , StackSize` sets aside `StackSize` bytes for the stack. Unfortunately, `CLEAR` also clears all of the data in a program, closes any open files, and erases all arrays. If you know ahead of time how much stack memory will be needed, then using `CLEAR` as the first statement in a program will not cause a problem.

Even when `CLEAR` is used as the first statement in a program, there is still one situation where that will not be acceptable. When you use `CHAIN` to execute a subsequent program, a `CLEAR` statement in that program will clear all of the variables that have been declared `COMMON`. Fortunately, there are two solutions to this problem: BASIC PDS offers the `STACK` statement, which lets you establish the size of the stack but without the side effects of `CLEAR`. For example, the statement `STACK 5000` sets

aside 5000 bytes for the stack. The other solution is to use the /STACK: link switch, which reserves a specified number of bytes. All of the options that LINK supports are described in Chapter 5.

Recursion

I have already illustrated some of the situations in which a recursive subprogram or function could be useful. Now let's look at some actual programming examples. The Evaluate function in the listing below uses recursion to reinvoke itself for each new level of parentheses it encounters.

```
DECLARE FUNCTION Evaluate# (Formula$)

INPUT "Enter an expression: ", Expr$
PRINT "That evaluates to"; Evaluate#(Expr$)

FUNCTION Evaluate# (Formula$)
  'Search for an operator using INSTR as a table lookup. If found,
  'remember which one and its position in the string.
  FOR Position% = 1 TO LEN(Formula$)
    Operation% = INSTR("+-*/", MID$(Formula$, Position%, 1))
  IF Operation% THEN EXIT FOR
  NEXT

  'Get the value of the left part, and a tentative value for the
  'right part.
  LeftVal# = VAL(Formula$)
  RightVal# = VAL(MID$(Formula$, Position% + 1))

  'See if there's another level to evaluate.
  Paren% = INSTR(Position%, Formula$, "(")

  'There is, call ourselves for a new RightVal#.
  IF Paren% THEN RightVal# = Evaluate#(MID$(Formula$, Paren% + 1))
  'No more to evaluate, do the appropriate operation and exit.
  SELECT CASE Operation%
    CASE 1 'addition
      Evaluate# = LeftVal# + RightVal#
    CASE 2 'subtraction
      Evaluate# = LeftVal# - RightVal#
    CASE 3 'multiplication
      Evaluate# = LeftVal# * RightVal#
    CASE 4 'division
      Evaluate# = LeftVal# / RightVal#
  END SELECT
END FUNCTION
```

When you run this program, enter an expression like $15 * (12 + (100 / 8))$. To keep the code to a minimum, Evaluate accepts only simple, two-number expressions. That is, it will not work with more than one math operator within each pair of parentheses as in $10 * (3 + 4 + 5)$. However, the parentheses may be nested to nearly any level.

This function begins by examining each character in the incoming formula string for a math operator. If it finds one the operator number (1 through 4) is remembered, as well as its position in the formula string. Next, VAL is used to obtain the value of the digits to the left of the operator, as well as the digits

to the right. Notice that it was not necessary to use LEFT\$ to isolate the left-most portion of the string, because VAL stops examining the string when it encounters any non-digit character such as the "+" or "(".

Once these values have been saved, the next test determines if any more parentheses follow in the formula. If so, Evaluate calls itself, passing only those characters that are beyond the next parenthesis. Thus, the same routine evaluates each new level, returning to the level above only after all levels have been examined. I encourage you to run this program in the QuickBASIC editing environment, and step through each statement one by one with the F8 Trace command. In particular, use the Watch Variable feature to view the value of Position% and LeftVal# as the function recurses into subsequent invocations.

It is important to understand the need for stack variables in this program, and why STATIC must not be used in the function definition. When Evaluate walks through the incoming string and determines which math operator is specified, that operator must be remembered throughout the course of the function. If a static variable were used for Operation%, then its previous value would be destroyed when Evaluate calls itself. Likewise, LeftVal# cannot be overwritten either, or it would not hold the correct value when Evaluate returns to itself from the level below. Therefore, as you step through this program you will observe that each new invocation of Evaluate creates a new set of variables.

As you can see, stack variables are necessary for the proper functioning of a subprogram or function that calls itself. They are also necessary when one procedure calls another procedure which in turn calls the first one again. The key point is that each time a non-static routine is invoked, new and unique variables must be created. Otherwise, the variable contents from a previous level above will be overwritten.

Although recursion is a powerful and necessary technique, it should be used only when necessary. There is a substantial amount of overhead needed to allocate stack memory and clear it to zeros, so invoking a non-static routine is relatively slow. And as I described earlier, every non-static string variable must be deleted when the routine exits, at a cost of 9 bytes apiece.

Some programmers use recursion even when there are other, more efficient ways to solve a problem. For example, the QuickBASIC manual shows a recursive function that calculates a factorial. (A factorial is derived by multiplying a number by all of the whole numbers less than itself. That is, the factorial of 4 equals $4 * 3 * 2 * 1$.) However, a factorial can be calculated faster and with less code using a simple FOR/NEXT loop as shown below. This version of Factorial is 20 percent faster than the example given in the QuickBASIC manual.

```
FUNCTION Factorial#(Number%) STATIC
  Seed# = 1
  FOR X% = 1 TO Number%
    Seed# = Seed# * X%
  NEXT
  Factorial# = Seed#
END FUNCTION
```

Passing Parameters To Procedures

As you have already learned, BASIC normally passes data to a subprogram or function by placing its address on the stack. And when an entire array is specified, the address of the array descriptor is sent instead. But there are some cases where BASIC imposes restrictions on how variables and arrays may be passed to a procedure. Let's look now at some of the ways to get around those restrictions.

When using versions of BASIC earlier than PDS 7.1, it is not legal to pass an array of fixed-length strings. In fact, it is also impossible to pass a single fixed-length string directly. As you saw in Chapter 2, BASIC copies every fixed-length string argument to a regular string, which adds a lot of code and also wastes string memory.

The simplest solution for fixed-length strings is to define an equivalent TYPE that is comprised of a single string component. Since a TYPE variable or array can legally be passed, this is the easiest and most direct approach, as shown here.

```
TYPE FLen
  S AS STRING * 100
END TYPE
DIM MyString AS FLen
CALL Subprogram(MyString)

SUB Subprogram(FLString AS FLen)
  ...
  ...
END SUB
```

If the subprogram being called is in a separate module, then the TYPE definition must also be present in that file. However, the DIM statement is needed only in the program that passes the string. This also works with fixed-length string arrays, except that the DIM would have to be changed to DIM MyArray(1 TO NumElements) AS FLen, and the subprogram's definition would be changed to SUB Subprogram(FLString() AS FLen).

BASIC PDS 7.1 supports passing a fixed-length string array directly, so this work-around is not needed with that version. Curiously, a single fixed-length string may not be passed as a parameter in BASIC 7.1. Since a fixed-length string is closely related to a TYPE variable, this limitation seems arbitrary at best.

BASIC 7.1 also supports the use of BYVAL (By Value) when passing numeric arguments to procedures. This is a particularly powerful feature, because it can greatly reduce the amount of code needed to access those values within the routine. It also eliminates the need to make copies when a constant is passed as an argument. To take advantage of this feature, you simply specify BYVAL in both the calling and receiving argument list, as shown below.

```
DECLARE SUB Subroutine(BYVAL Arg1%, BYVAL Arg2%)
CALL Subroutine(Var1%, Var2%)

SUB Subroutine(BYVAL X%, BYVAL Y%)
```

```
...  
...  
END SUB
```

Because the actual value of the argument is being passed, there is no way to return information back to the caller. But in those situations where an assignment to the original variable from within the routine is not needed, BYVAL can eliminate a lot of compiler-generated code when dealing with integers. Of course, you may use a mix of BYVAL and non-BYVAL parameters if you need the benefits of both methods in a single call. As proof of this savings, disassemblies of a one-statement subprogram designed both ways is presented below, to show how an integer parameter is accessed when it is passed by address and by value.

```
SUB ByAddress(Param%) STATIC  
LocVar% = Param%  
  MOV SI,[Param%] ;get the address of Param%  
  MOV AX,[SI] ;then read the value there  
  MOV LocVar%,AX ;assign that to LocVar%  
END SUB
```

```
SUB ByValue(BYVAL Param%) STATIC  
LocVar% = Param%  
  MOV AX,Param% ;read Param% directly  
  MOV LocVar%,AX ;and assign it to LocVar%  
END SUB
```

Note that the savings are only within the subroutine, and not when it is called. That is, 4 bytes are needed to pass an integer variable whether by address or by value. In fact, passing larger data types requires more code to pass by value. Any variable can be passed by address with 4 bytes of compiler-generated code, because what is sent is a single address. But to pass a double precision number by value requires 16 bytes, since 4 bytes of code are needed for each 2-byte portion of the number.

In general, passing variables as parameters to a subprogram or function is preferable to sharing them. When many variables are shared throughout a program, you run the risk of introducing bugs caused by accidentally using the same variable name more than once. However, sharing has some definite advantages in at least two situations.

The first is when a procedure must be accessed as quickly as possible. Since a finite amount of code is needed to pass each parameter, some amount of time is also required to execute that code. Therefore, sharing a few, carefully selected variables can improve the speed of your programs and reduce their size as well. Another important use for SHARED is to conserve data memory. Nearly all programs use at least a few temporary scratch variables, perhaps as FOR/NEXT loop counters. By dimensioning several such variables as being shared throughout a program, the same variables can be used repeatedly. I often begin programs with a DIM SHARED statement such as DIM SHARED X, Y, Z, and then use those variables as often as possible.

One final trick I want to share is how to pass a large number of parameters using less code than would normally be necessary. Each argument that is passed to a procedure requires 4 bytes of code. In a

complicated routine that needs many parameters, this can quickly add up. Worse, these bytes are added for every call. Therefore, a subprogram that accepts 10 parameters and is called 20 times will add 800 bytes to the final executable file just to handle the parameters!

One solution is to use an array, which is ideal when all of the parameters are the same type of data. An entire array can be passed as a single parameter since only the array descriptor's address is needed. Even better, however, is to create a TYPE variable, and then assign all of the parameters to it. A TYPE variable can hold nearly any amount and type of data, and it too can be passed using only 4 bytes. Although this does require a separate assignment for each TYPE component, you simply use the TYPE where the regular variables would have been assigned. By eliminating the added code to pass many parameters, programs that use a TYPE this way will also be much faster.

Modular Programming

QuickBASIC versions 4.0 and later let you load subprograms and functions from multiple files into the editing environment at the same time. This further enhances their reusability, since the different modules can be treated as "black boxes" whose purpose is already known. Once a routine has been developed and debugged, it can be used again and again, without further regard for the names of the variables within the routines. Indeed, many of the utility routines included with this book are provided as separate modules, intended to be loaded along with your programs. Any variable name can be passed as an argument to a procedure, even if a different name is used to represent the same variable within the procedure. If you have defined a subprogram such as `SUB MySub (X%, Y!, Z$)`, then you could call it using `CALL MySub (A%, B!, C$)`. Of course, the variables you pass must be of the same data type as the subroutine expects.

Because reusability is an important consideration in the design of any procedure, it generally makes sense to store it in its own source file. This lets you combine the same module repeatedly with any number of programs. The alternative would be to merge the file into each program that needs it. But maintaining multiple copies of the same code wastes disk space. Further, if a bug is found in the routine, you will have to identify all of the programs that contain it, and manually correct each one of them.

Another important advantage of using separate files is that you can exceed the usual 64K code size barrier. Unlike the data segment which is comprised of the sum of all data in all modules, an .EXE file can contain multiple code segments. Each BASIC module has a single code segment, and each of these can be as large as 64K. In fact, dividing a program into separate files is the only way to exceed the usual 64K code size limitation.

Although using a separate source file for each subprogram makes sense in many situations, there is one slight disadvantage. When all of the various program modules are linked together, each separate module adds approximately 100 bytes of overhead. None the less, for all but the smallest programming projects, the advantages of using separate modules will probably outweigh the slight increase in code size.

Include Files

Another useful BASIC feature that can help you to create modular programs is the Include file. An Include file is a separate file that is read and processed by BASIC at a specified place in your program. The statement `'$INCLUDE: 'filename'` tells QB or BC to add the statements in the named file to your source code, as if that code had been entered manually. If a file extension is not given, then .BAS is assumed. Many of the files that Microsoft provides with QuickBASIC use a .BI extension, which stands for "BASIC Include". Some programmers use .INC, and you may use whatever seems appropriate to the contents of the file.

Include files are ideal for storing DECLARE, CONST, TYPE, and COMMON statements. Except for COMMON, none of these statements add to the size of your program, and none of them create any executable code. Therefore, you could create a single include file that is used for an entire project, and add an appropriate '\$INCLUDE directive to the beginning of each program source file. Unused DECLARE and CONST statements and TYPE definitions are ignored by BASIC if they are not referenced. However, they do impinge slightly on available memory within the QuickBASIC editor, since BASIC has no way to know that they are not being used. Similarly, BC must keep track of the information in these statements as it compiles your program. But again, there is no impact on the size of your final executable program.

In general, I recommend that you avoid placing any executable statements into an include file. Because the code in an include file is normally hidden from your view, it is easy to miss a key statement that is causing a bug. Likewise, a '\$DYNAMIC or '\$STATIC command hidden within an include file will obscure the true type of any arrays that are subsequently dimensioned. Perhaps worst of all is placing a DEFINT or other DEFtype statement there, for the same reason.

Quick Libraries

Quick Libraries contribute to modular programming in two important ways. Perhaps the most important use for a Quick Library is to allow access to subprograms and functions that are not written in BASIC. All DOS programs and subroutines—regardless of the language they were originally written in—end up as .OBJ files suitable for LINK to join together. But the QB and QBX editing environments manipulate BASIC source code, and interpret the commands rather than truly compile them. Therefore, the only way you can access a routine written in assembly language or C within QuickBASIC is by placing the routine into a Quick Library.

Quick Libraries also let you store completed BASIC subprograms and functions out of the way from the rest of your program. If you have a large number of subroutines in one program, the list of names displayed when F2 is pressed can be very long and confusing. Since QuickBASIC does not display the routines in a Quick Library, there will be that many fewer names to deal with. Another advantage of placing pre-compiled BASIC routines into a Quick Library is that they can take less memory than when

the BASIC source code is loaded as a module. This is true especially when you have many comments in the program, since comments are of course not compiled.

Be aware that there are a few disadvantages to placing BASIC code into a Quick Library. One is that you cannot step and trace through the code, since it is not in its original BASIC source form. Another is that Quick Libraries are always stored in normal DOS memory, as opposed to expanded memory which QBX [and VB/DOS] can use. When a BASIC subprogram or function is less than 16K in size and EMS is present, QBX and VB/DOS will place its source code in expanded memory to free up as much conventional memory as possible.

Error and Event Handling

As a BASIC programmer, there are several types of errors that you must deal with in a program. These errors fall into two general categories: compile errors and runtime errors. Compile errors are those that QB or BC issue, such as "Syntax error" or "Include file not found". Generally, these are easy to understand and correct, because the QuickBASIC editor places the cursor beneath the offending statement. In some cases, however, the error that is reported is incorrect. For example, if your program uses a function in a Quick Library that expects a string parameter and you forgot to declare it, BASIC reports a "Type mismatch" error. After all, with a statement such as `X = FuncName%(Some$)`, how could BASIC know that `FuncName%` is not simply an integer array? Assuming that it is an array, BASIC rejects `Some$` as being illegal for an element number.

Runtime errors are those such as "File not found" which are issued when your program tries to open a file that doesn't exist, or is not in the specified directory. Other common runtime errors are "Illegal function call", "Out of string space", and "Input past end". Many of these errors can be avoided by an explicit test. If you are concerned that string space might be limited you can query the `FRE("")` function before dimensioning a dynamic string array. However, some errors are more difficult to anticipate. For example, to determine if a particular directory exists you must use `CALL Interrupt` to query a DOS service.

The conventional way to handle errors is to use `ON ERROR`, and design an error handling subroutine. There are a number of problems with using `ON ERROR`, and most professional programmers try to avoid using it whenever possible. But `ON ERROR` does work, and it is often the simplest and most direct solution in many programs. The short listing below shows the minimum steps necessary to implement an error handler using `ON ERROR`.

```
ON ERROR GOTO HandleErr
FILES "*.XYZ"
END

HandleErr:
SELECT CASE ERR
CASE 53: PRINT "File not found"
CASE 68: PRINT "Device unavailable"
CASE 71: PRINT "Disk not ready"
CASE 76: PRINT "Path not found"
```

```
    CASE ELSE: PRINT "Error number"; ERR
END SELECT
RESUME NEXT
```

The statement `ON ERROR GOTO HandleErr` tells BASIC that if an error occurs, the program should jump to the `HandleErr` label. Without `ON ERROR`, the program would display an error message and then end. Since it is unlikely that you have any files with an `.XYZ` extension, BASIC will go to the error handler when this program is run. Within the error handling routine, the program uses the `ERR` function to determine the number of the error that occurred. Had line numbers been used in the program, the line number in which the error occurred would also be available with the `ERL` function.

In this brief program fragment, the most likely error numbers are filtered through a `SELECT CASE` block, and any others will be reported by number. Regardless of which error occurred, a `RESUME NEXT` statement is used to resume execution at the next program statement.

`RESUME` can also be used with an explicit line label or number to resume there; if no argument is given BASIC resumes execution at the line that caused the error. In many cases a plain `RESUME` will cause the program to enter an endless loop, because the error will keep happening repeatedly.

In this case, the file will not exist no matter how many times BASIC tries to find it. Therefore, a plain `RESUME` is not appropriate following a "File not found" or similar error. Had the error been "Disk not ready", you could prompt the user to check the drive and then press a key to try again. In that case, then, `RESUME` would make sense. Although BASIC's `ON ERROR` can be useful, it does have a number of inherent limitations.

Perhaps the worst problem with `ON ERROR` is that it often increases the program's size. When you use `RESUME NEXT`, you must also use the `/x` compile switch. Unfortunately, `/x` adds internal address labels to show where each statement begins, so the `RESUME` statement can find the line that caused the error. These labels are included within the compiled code and therefore increases its size.

Another problem with `ON ERROR` is that it can hide what is really happening in a program. I recommend strongly that you `REM` out all `ON ERROR` statements while working in the QuickBASIC editing environment. Otherwise, an Illegal function call or other error may cause QuickBASIC to go to your error handler, and that handler might ignore it if the error is not one you were expecting and testing for. If that happens and your program uses `RESUME NEXT`, you might never even know that an error occurred!

Yet another problem with `ON ERROR` is that it's frankly a clumsy way to program. Most languages let you test for the success or failure of the most recent operation, and act on or ignore the results at your discretion. Pascal, for example, uses the `IOResult` function to indicate if an error occurred during the last input or output operation.

Finally, BASIC generates errors for many otherwise proper circumstances, such as the `FILES` statement above. You might think that if no files were found that matched the `.XYZ` extension given, then BASIC would simply not display anything. Indeed, an important part of toolbox products such as Crescent

Software's QuickPak Professional are the routines that replace BASIC's file handling statements. By providing replacement routines that let you test for errors without an explicit ON ERROR statement, an add-on library can help to improve the organization of your programs.

As I mentioned earlier, some errors can be avoided by using CALL Interrupt to access DOS directly. (One important DOS service lets you see if a file exists before attempting to open it.) But critical errors such as those caused by an open drive door require assembly language. In Chapter 11 you will learn how to bypass BASIC and access DOS directly using CALL Interrupt.

Event Handling here

BASIC includes several forms of event handling, and like ON ERROR, these too are avoided when possible by many professional programmers. Event handling lets your programs perform a GOSUB automatically and without any action on your part, based on one or more conditions. Some of the more commonly used event statements are ON KEY, ON TIMER, and ON COM. With ON KEY, you can specify that a particular key or combination of keys will temporarily halt the program, and branch to a GOSUB routine designated as the ON KEY handler. ON TIMER is similar, except it performs a GOSUB at regular intervals based on BASIC's TIMER function. Likewise, ON COM performs a GOSUB whenever a character is received at the specified communications port.

The concept of event handling is very powerful indeed. For example, ON COM allows your program to go about its business, and also handle characters as they arrive at the communications port. ON TIMER lets you simulate a crude form of multi-tasking, where control is transferred to a separate subroutine at one second intervals. Unfortunately, BASIC's event handling is not truly interrupt driven, and the resulting code to implement it adds considerably to a program's size.

When any of the event handling methods are used, BASIC calls an interval event dispatcher periodically in your program. These calls add five bytes apiece, and one is added at either every statement, or at every labelled statement depending on whether you compiled using /v or /w respectively. This can increase your program's size considerably. Even worse, the repeated calls have an adverse effect on the speed of most programs. Like ON ERROR, BASIC's event handling statements provide a simple solution that is effective in many programming situations. And also like ON ERROR, they are best avoided in important programming projects.

Using purely BASIC techniques, the only alternative to event trapping is polling. Polling simply means that your program manually checks for events, instead of letting BASIC do it automatically. The primary advantage of polling is that you can control when and where this checking occurs. The disadvantage is that it requires more effort by you.

To see if any characters have been received from a communications port but are still waiting to be read you would use the LOF function. And to see if a given amount of time has elapsed you must query the TIMER function periodically. If true interrupt driven event handling were available in BASIC, that

would clearly be preferable to either of the two available methods. However, only with Crescent's P.D.Q. product can such capability be added to a BASIC program.

Programming Style

Programming style is a personal issue, and every programmer develops his or her own particular methods over time. Some aspects of programming style have little or no impact on the quality of the final result. For example, the number of columns you indent a FOR/NEXT loop will not affect how quickly a sort routine operates. But there are style factors that can help or harm your programs. One is that clearly commenting your code will help you to understand and improve it later. Another is when more than one programmer is working on a large project simultaneously. If neither programmer can figure out what the other is doing, the program's quality will no doubt suffer.

Clearly, no one can or even should try to force a particular style or ideology upon you. However, I would like to share some of the decisions that I have made over the years, and explain why they make sense to me. Of course, you are free to use or not use these opinions as you see fit. Programmers are as unique and varied as any other discipline, and no one set of rules could possibly serve everyone equally. Whatever conventions you settle upon, be consistent above all else.

The most important convention that I follow is to use `DEFINT A-Z` as the first statement in every program. For me, using integers verges on religion, and my fingers could type `DEFINT` even if I were asleep. As I have stated repeatedly, integers should be used whenever possible, unless you have a compelling reason not to. Integers are much faster and smaller than any other variable type BASIC offers. Nearly all of the available third party add-on products use integers parameters wherever possible, and so should the routines you write. The only reasonable exception to this is when writing financial or scientific programs, or other math-intensive applications.

Equally important is adding sufficient and appropriate comments. Some programmers like to use comment headers that identify each related block of code; others prefer to comment every line. I recommend doing both, especially if other people will be reading your programs. I also prefer using an apostrophe as a comment delimiter, rather than the more formal `REM`. There are only so many columns available for each comment line, and it seems a shame to waste the space `REM` requires.

When writing a subprogram or function that you plan to use again in other projects, include a complete heading comment that shows the purpose of the routine and the parameters it expects. If each parameter is listed neatly at the beginning of the file, you can create a hard copy index of routines by printing that section of each file.

Avoid comments that are obvious or redundant, such as this:

```
Count = Count + 1 'increment Count
```

If Count is keeping track of the number of lines read from a file, a more appropriate comment would be: `'show that another line was read`. Also avoid comments that are too cute or flip. Simply state clearly what is happening so you will know what you had in mind when you come back to the program next month or next year.

Selecting meaningful variable names is equally valuable in the overall design of a program. If you are keeping track of the current line in a file, use a variable name such as `CurLine`. Although BASIC in some cases lets you use a reserved word as a variable name, I recommend against that. Over the years, different versions of BASIC have allowed or disallowed different keywords for variables. While QuickBASIC 4.5 lets you use `Name$` as a variable, there is no guarantee that the next version will. Also, be aware that variables names which begin with the letters `Fn` are illegal, because BASIC reserves that for user-defined functions. Using the variable `FName$` to hold a file name may look legal, but it isn't.

Don't be ashamed to use `GOTO` when it is appropriate. There are many places where `GOTO` is the most direct way to accomplish something. As I showed earlier in this chapter, `GOTO` when used correctly can sometimes produce smaller and faster code than any other method.

Use line labels instead of line numbers. The statement `GOSUB 1020` doesn't provide any indication as to what happens at line 1020. `GOSUB OpenFile`, on the other hand, reads like plain English. The only exception to this is when you are debugging a program that crashes with the message "Illegal function call at line no line number". In that case, you should add line numbers to your program and run it again. A program that reads a source file and prints each line to another file with sequential numbers is trivial to write. I will also discuss debugging in depth in Chapter 4.

Even though using `DEFINT` is supposed to force all subsequent `CONST`, `DEF FN`, and `FUNCTION` declarations to be integer, a bug in QuickBASIC causes untyped names to occasionally assume the single precision default. Therefore, I always use an explicit percent sign (%) to establish each function's type. In fact, I use whatever type identifier is appropriate for functions and `CONST` statements, to make them easily distinguishable in the program listing. For example, in the statement `IF CurRow > MaxRows% THEN CurRow = MaxRows%`, I know that `MaxRows%` has been defined as a constant. Some people prefer to use all upper-case letters for constants, though I prefer to reserve upper case for BASIC keywords.

Although BASIC supports the optional `AS INTEGER` and `AS SINGLE` directives when defining a subprogram or function, that wastes a lot of screen space. I greatly prefer using the variable type identifiers. That is, I will use `SUB MySub (A%, B!)` rather than `SUB MySub (A AS INTEGER, B AS SINGLE)`. The same information is conveyed but with a lot less effort and screen clutter.

A well-behaved subroutine will restore the PC to the state it was when called. If you have subprogram that prints a string centred on the bottom line of the screen, use `CSRLIN` and `POS(0)` to read the current cursor location before you change it. Then restore the cursor before you exit.

I like to indent two spaces within FOR/NEXT and IF/THEN blocks. Although some people prefer indenting four or even eight columns for each level, that can quickly get out of hand when the blocks are deeply nested. Nothing is harder to read than code that extends beyond the edge of the screen. But whatever you do, please do not change the tab stop settings in the QuickBASIC editor, unless you are the only one who will ever have to look at your code. Even though the program may look fine on your screen, the indentation will be completely wrong on everyone else's PC.

When creating a dynamic array I prefer REDIM to a previous '\$DYNAMIC statement. REDIM is clearer because it shows at the point in the source where the array is dimensioned that this is a dynamic array. Otherwise you have to scan backwards through your source code looking for the most recent '\$DYNAMIC or '\$STATIC, to see what type of array it really is. By the same token, using ever-changing DEFtype statements throughout your code is poor practice. Further, if a variable is a string, always include the dollar sign (\$) suffix when you reference it. If you use DEFSTR S or even worse, DIM xxx AS STRING and then omit the dollar sign, nobody else will understand your program.

I also prefer to explicitly dimension all arrays, and not let BC create them with the 11-element default (including element zero). If you need less than 11 elements, the memory is wasted. And if you need more, then your program will behave unpredictably. Not dimensioning every array is sloppy programming. Period.

Avoid repeated calls to BASIC's internal functions if possible. In the listing below, the first example creates 61 bytes of code, while the second generates only 46 bytes.

Not recommended:

```
IF CSRLIN = 1 OR CSRLIN = 6 OR CSRLIN = 12 THEN
  . . .
END IF
```

Much better:

```
Temp = CSRLIN
IF Temp = 1 OR Temp = 6 OR Temp = 12 THEN
  . . .
END IF
```

As I stated earlier in this chapter, using SELECT CASE instead of IF will also eliminate this problem. Many BASIC statements are translated into calls, and each call takes a minimum of five bytes.

Your programs will be easier to read if you evaluate temporary expressions separately. Even though BASIC lets you nest parentheses to nearly any level, nothing is gained by packing many expressions into a single statement. In the examples below that strip the extension from a file name, the first creates only a few bytes less code. Although this may seem counter to the other advice I have given, a slight code increase is often more than offset by a commensurate improvement in clarity.

```
File$ = LEFT$(File$, INSTR(File$, ".") - 1)
```



```
Dot = INSTR(File$, ".")  
File$ = LEFT$(File$, Dot - 1)
```

The last issue I want to discuss is how to pronounce BASIC keywords and variable names. Don't laugh, but many programmers have no idea how to communicate the words LEFT\$ or VARSEG over the telephone. Some people say "X dollar" for X\$ even though "X string" is so much easier to say. Another keyword that's hard to verbalize is VARPTR. I prefer "var pointer" since it is, after all, a pointer function. CHR\$(13) is pronounced "character string thirteen", again because that's the clearest and most straight forward interpretation. Likewise, INSTR is pronounced "in string" and LEFT\$ would be said as "left string". If you're not sure how to pronounce something, use the closest equivalent English wording you can think of.

Summary

In this chapter you have learned how BASIC's control flow statements are constructed, and how the compiler-generated code is similar regardless of which statements are used. You also learned where GOSUB and GOTO should be used, and when subprograms and functions are more appropriate. The discussion on logical operations showed how AND, OR, EQV, and XOR operate, and how they can be used to advantage in your programs.

I have explained in detail exactly what recursion is, and how recursive subroutines can perform services that are not possible using any other technique. You have also learned about the importance of the stack in recursive and other non-static subroutines. Passing parameters to subprograms and functions has also been described in detail, along with some of the principles of modular program and event handling.

Finally, I have shared with you some of my own personal preferences regarding programming style, and when and how such conventions can make a difference. Although this is a personal issue, I firmly believe it is important to develop a consistent style and stick with it.

In Chapter 4 you will learn debugging methods using both the QuickBASIC editing environment and Microsoft's CodeView debugger. The successful design of a program is but one part of its development. Once it has been written, it must also be made to work correctly and reliably. As you will learn, there are many techniques that can be used to identify and correct common programming errors.



PART 2
Programming Hands On

4

Debugging Strategies

There are many individual components which contribute to a completed application. The logical flow of the program must be determined, the user interface must be designed, and appropriate algorithms must be selected. But no matter how much effort you devote to the design and implementation of a program, the bottom line is it must also work correctly.

In an ideal scenario, you would begin writing a program by first jotting down some notes that describe its operation. Next, you would create an outline listing each of the program's major components. You would then determine all of the subroutines and functions that are needed, and perhaps even create a flow chart showing each of the paths that could be taken. Properly prepared for any situation that might arise, you finally write the actual code and find that it works perfectly. Now, what's wrong with this picture? Few people actually program that way!

In practice, many programmers simply start coding with little forethought and no detailed plan. They begin with the first statement and continue to the last, occasionally reworking portions into subroutines as necessary. After all, planning is not nearly as much fun as programming, and everyone knows that fun is the most important part. Believe it or not, I agree. There's nothing really wrong with plodding through a program, stabbing here and there until it works. Indeed, some great algorithms developed out of aimless doodling. I have personally never drawn a flow chart, and I have no plans to start now.

What I will address here is how to find and correct problems when they do occur. There are more things that can go wrong with a program than can go right, and tracking down an elusive "Illegal function call" error that appears only occasionally is definitely not much fun. How quickly you can solve these problems is directly related to your understanding of programming in general, and to your familiarity with the tools available.

In this chapter you will learn how to identify problems in your programs, and also how to solve them. Programming errors, or bugs, can be as simple as a misspelled variable name, and as complex and ornery as an internal flaw in BASIC itself. The BASIC editing environment provides a wealth of powerful debugging features, and understanding how to use them will help you produce programs that are reliable and error free.

Common Programming Errors

There are three distinct types of programming errors: simple misspellings and other naming or syntax errors, incorrect logic such as misunderstanding or incorrectly coding an algorithm, and failing to understand some of the finer points of the BASIC language. No matter how carefully you type, no

matter how much forethought you apply to a particular problem, and no matter how often you read the BASIC manuals, it is impossible to completely avoid making mistakes.

The first category includes those errors caused by simple mistakes such as misspelling a variable or procedure name. Trying to call a subprogram that doesn't exist will be immediately obvious, because BASIC gives you an error message before the program can be run. But an incorrect variable name will return the wrong results with no warning.

Passing the wrong number of arguments to a procedure may or may not be reported, depending on whether the routine has been declared. Assembly language routines in a Quick Library can be particularly pesky in this regard. Although BASIC automatically generates a DECLARE statement for BASIC subprograms and functions you have loaded in source form, it does not do this for routines in a Quick Library. If you call an assembly language routine incorrectly, you will probably crash the PC. However, it is also possible to corrupt string memory and not know it. Worse, a "String space corrupt" error is often not reported until much later in the program. If you run the short program below in the QuickBASIC 4.5 editor, it will appear to operate correctly.

```
X$ = SPACE$(1000)           'create a string
POKE SADD(X$) - 2, 100     'corrupt string memory
PRINT "Testing"
X% = 1
PRINT "More testing"
X% = 2
PRINT "Yet more testing"
X% = 3
```

Here, the POKE statement is overwriting the back pointer that belongs to X\$, which is one type of string corruption that can occur. But QuickBASIC doesn't know that this has happened, because it has no reason to check the integrity of its string memory until another string assignment is made. However, adding the statement PRINT FRE("") anywhere after the POKE command causes BASIC to check string memory, and report the error. Even if your program does not use POKE, calling a procedure incorrectly can cause it to overwrite memory in this fashion.

Another simple error is inadvertently using the same variable name twice, or omitting a type declaration character from a variable name. For example, if you are using a variable named Bytes& to track how many bytes of a file have been read, accidentally using Bytes later on will give the wrong results. If a DEFINT statement is in effect, then Bytes will be an integer variable. Otherwise, it will be single precision which is also incorrect. Unless you use the DIM. . . AS statement to declare a variable explicitly, BASIC lets you have different variables with the same name. That is, Var%, Var!, and Var# can all coexist in the same program, and each is a unique variable.

Similarly, using the wrong variable entirely will cause your program to operate incorrectly, and again with no error message displayed. More than once I have had a program with one FOR loop nested within another, and used the outer loop counter variable when I meant to use the inner one.

Another common situation is caused by changing the name of a variable during the course of writing a program. For example, you may have a variable named BPtr that tracks where you are reading within a

buffer. If you later decide to change that name to `BufPointer` because it is more meaningful, you must also remember to change all occurrences of the name. Of course, BASIC's search and replace feature minimizes that problem. More important, though, you must make a mental note to use the new name as you continue to develop the program.

Forgetting to declare a function can also lead to incorrect results that produce no warning. If an integer function is not declared, then BASIC will dimension an array with that name if the function expects a numeric argument. When BASIC encounters the statement `X = FuncName%(Y%)` it assumes that `FuncName%` is an integer array, and create an array containing the default 11 elements. In this case `X` will be assigned a value of zero, or you will receive a "Subscript out of range" error if `Y%` is not between 0 and 11. I once observed an unexplainable "Out of string space" error that was caused by the statement `Size = ScreenSize%(ULRow, ULCol, LRRow, LRCol)`. `ScreenSize%` was a function present in a Quick Library, but without a `DECLARE` statement BASIC created a 4-dimensional integer array.

Logic Errors

The second cause of bugs is logic errors, and these include adding when you meant to subtract, or using the wrong variable altogether. Programs that manipulate pointers (variables that hold the addresses of other variables) are particularly prone to errors in logic. Another common logic error is forgetting to trim the leading or trailing blanks from a file or directory name before using it. If the operator enters `" c:\thisfile.dat "` and you try to open that file, BASIC will report a "Bad file name" error.

Another cause of logic errors is failing to consider all of the things a user may enter. An inexperienced operator is likely to enter data that you as the programmer would never consider, or select menu items in an order that makes no sense. Indeed, never underestimate the value of beta testers. After you have exhausted all of the possibilities you can think of, give the program to a 4 year old child, and ask him or her to try it while you watch. Your uncle Ernie would be a good beta tester too, and the less he knows about your program, the more valuable his contribution will be. People who know absolutely nothing about computers have an uncanny knack for creating "Illegal function call" errors in a program that you just know is perfect.

Similarly, you must consider all of the possible error conditions that could happen in a program. In an error handler that has a `CASE` statement for each possibility you anticipate, also include a `CASE ELSE` clause for those you haven't thought of. The short listing that follows shows a typical error handler that incorporates this added safety measure.

```
ON ERROR GOTO HandleErr
...
...
HandleErr:
  SELECT CASE ERR
    CASE 7, 14
      PRINT "Out of memory"
    CASE 24, 25, 27
      PRINT "Fix the printer"
```

```

CASE 53
  PRINT "File not found"
CASE ELSE
  PRINT "Error number"; ERR
END SELECT
...
...

```

The CASE ELSE clause lets you accommodate any possibility, and your user can then at least report to you what the error number was. This simple example doesn't include all of the possibilities, but you can certainly see the general concept.

Another common logic error is using the same file number twice. When a file has been opened as #1, that number remains in use until the file is closed. This can be problematical when writing reusable modules, since there is no way to know which files may be in use by the main program. Some programmers use #99 or another unlikely number in a routine that will be reused in many programs. But even that approach is flawed, because you have to remember which numbers are used by which routines.

BASIC's FREEFILE function is intended to solve this problem, and it returns the next available file number. Be sure to save the results FREEFILE returns, however, since the value will change as soon as the next file is opened. The code below shows both the wrong and right ways to use FREEFILE.

Wrong:

```

OPEN "accounts.dat" FOR INPUT AS #FREEFILE
INPUT #FREEFILE, X$      'FREEFILE has changed!
CLOSE #FREEFILE

```

Right:

```

FileNum = FREEFILE      'get and save the number
OPEN "accounts.dat" FOR INPUT AS #FileNum
INPUT #FileNum, X$
CLOSE #FileNum

```

In the first example if FREEFILE returns, say, a value of 2, then it will return 3 at the INPUT statement which is of course incorrect. Therefore, you must save the value FREEFILE returns, and use that for all subsequent file accesses. This situation also occurs with INKEY\$, because once a character has been returned it is no longer available unless you saved it.

Two other frequent problems are attempting to use LSET to assign characters into a string that does not exist, and failing to clear a counter variable within a static subprogram or function. The second problem can be especially frustrating, because the routine will work correctly the first time it is invoked. In the function below, a counter returns the number of embedded control characters it finds in a string.

```

FUNCTION CtrlCount%(Work$) STATIC
FOR X% = 1 TO LEN(Work$)
  IF ASC(MID$(Work$, X%, 1)) < 32 THEN

```

```

        Count% = Count% + 1
    END IF
NEXT

    CtrlCount% = Count%    'return the count
END FUNCTION

```

The problem here is that `Count%` retains its value between function invocations. Therefore, each time `CtrlCount%` is used it will return ever higher values. One solution is to add the statement `Count% = 0` at the beginning of the function. Another is to omit the `STATIC` option from the function definition.

Understanding BASIC's Quirks

The third type of error is caused by not understanding some of BASIC's finer points and quirks. For example, some people do not realize that omitting the third argument from `MID$` causes it to return all of the remaining characters in a string. To see if a drive letter was given as part of a file name and if so extract it, you might use a statement such as `IF MID$(FileName$, 2) = ":" THEN Drive$ = LEFT$(FileName$, 1)`. But since the number of characters was not specified to `MID$`, it returned all but the first character in the string. Unless the string was a drive letter and colon only ("C:"), the test for a colon could never work. The solution, of course, is to use `MID$(FileName$, 2, 1)`.

Another instance in which an intimate knowledge of BASIC's idiosyncrasies comes into play can affect the earlier example of a file name that contains leading blanks. Most programmers do not use `INPUT` to accept information, unless the program is very simple and it will be used only occasionally. However, asking for a file name with `INPUT` is one way to avoid that problem, because `INPUT` strips all leading and trailing blank spaces, as well as `CHR$(9)` tab characters. The more useful `LINE INPUT`, on the other hand, does not strip leading blanks and tabs. Most programmers would never be so foolish as to enter a file name with leading blanks. So this is yet another situation where it is important to consider all of the possibilities.

It is also possible to crash a program by using the `ASC` function when the string might be null. Again, you would never press `Enter` alone in response to a prompt for a file name or other mandatory information, but someone else might.

Another BASIC quirk is caused by rounding errors. As you saw in Chapter 2, adding or multiplying many numbers in succession can produce results that are not precisely correct. Instead of checking to see if a value is zero, it is often better to compare it to a very small number. That is, instead of `IF Value# = 0` you would use `IF Value# < .000001` or `IF Value# < .000001 AND Value# > -.000001` or something similar. Also, some numbers simply cannot be represented at all. If you try to enter the statement `X# = .000000000001` in the QuickBASIC 4.5 editor, the value will be converted to `9.999999999999999D-12` as soon as you press `Enter`.

Although not technically a BASIC quirk, many programmers forget that variables within a `DEF FN` function are by default global. Unless you include an explicit `STATIC` statement listing each variable

that is to be local to the function, it is likely that an unexpected change will be made to a variable in the main program.

Some programming situations require that you obtain the address of a string variable using SADD. However, SADD is not legal for use with a fixed-length string or the string portion of a TYPE variable. More important, when using BASIC PDS for strings you must also remember to use SSEG to get the string's data segment. Using VARSEG will not create an error; however, the program will not work correctly.

Related to that, it is important to remember that strings and dynamic arrays move around in memory—often at unexpected times. The program below appends a zero character to one string for each zero that is found in another string. Since BASIC may move Work\$ during the course of assigning Zero\$, this code will fail eventually:

```
Address = SADD(Work$)
FOR Y = Address TO Address + LEN(Work$) - 1
  IF PEEK(Y) = 48 THEN Zero$ = Zero$ + "0"
NEXT
```

Another particularly insidious bug can result if you inadvertently add parentheses around a variable that is passed to a subprogram or function. In the example below, a subprogram that intentionally modifies a parameter has been declared and is then called without the CALL keyword.

```
DECLARE SUB Square(Param%)
Square (Value%)

SUB Square(Value%) STATIC
  Value% = Value% * Value%
END SUB
```

Because of the unnecessary and incorrect use of parentheses, a copy of the argument is sent to Square instead of the argument itself, with the result that Value% is never actually changed. The fix is to either remove the parentheses, or add the word CALL. Another, related issue is placing a DEFINT after DECLARE statements. In the example below, the parameters X, Y, and Z are assumed by BASIC to be single precision, even though this is clearly not what was intended.

```
DECLARE SUB (X, Y, Z) 'X, Y, and Z are singles!
DEFINT A-Z
:
```

The final issue I want to address here is potential overflow errors. The statement `IF IntVar% * 14 > 1000000` can never be true, because BASIC performs integer math assuming an integer range only. Unless you compile your program using the /d debug option, the error will be unreported in a compiled program. If this statement is executed within the QB environment, BASIC will report an overflow error, even though the instruction certainly appears to be legal. But since integer math assumes an integer result, the product of IntVar% times 14 will overflow the range of integer values if IntVar% is greater than 2,340.

One solution is to use a long integer for IntVar, and BASIC will then use the range of long integers for the comparison. Using a long integer wastes memory, however, and calculations on long integers are slower and require more code to implement. A much better solution is to use CLNG (Convert to Long), which tells BASIC to assume a long integer result.

The statement `IF CLNG(IntVar%) * 14 > 1000000` will create a long integer version of IntVar%, and then multiply the result times 14 and use that for the subsequent comparison. Unlike the copies that BASIC makes which steal DGROUP memory, the long integer conversion in this instance is handled within the CPU's registers. CLNG when used this way is really just a compiler directive, as opposed to a called library routine. Another solution is to add an ampersand after the constant 14, thus: `IF IntVar% * 14& > 1000000`. Again, no additional DGROUP memory is used to handle 14 as a long integer value.

Another interesting use of CLNG and CINT—unrelated to debugging but worth mentioning none the less—is to reduce the size of comparison code. When you use a statement such as `IF X% > VAL(Some$)`, a floating point comparison is performed even if Some\$ holds an integer value. By replacing that example with `IF X% > CINT(VAL(Some$))` 6 bytes of code can be saved. The CINT tells BASIC that it will not have to perform any floating point rounding when it compares the two values.

Debugging and Testing Techniques

When you are developing a large application that is comprised of many individual modules, there are several useful debugging techniques you can employ. One is to create short test-bed programs that exercise each subprogram and function. Finding an error in a complex program with many interdependencies between subroutines can be a tedious prospect at best. If you instead create a small program whose sole purpose is to test a particular subprogram, you will be better able to focus on just that routine.

Another useful technique for detecting and preventing sporadic errors is to test your code on "boundary conditions". If you have a routine that reads and processes a file in 4K (4096 byte) increments, test it with a file that is exactly 4096 bytes long, as well as with other test files that are 4095 and 4097 bytes long.

Perhaps nothing is more frustrating than having a program fail with the message "xxx at line No line number". This message is a throw-back to the days when all BASIC programs had to use line numbers. Now that line numbers are not required in modern compiled BASIC, most programmers do not use them, opting instead for more descriptive line labels when labels are needed at all. When an error does occur and the program has been compiled with /d, BASIC reports the number of the nearest numbered line preceding the line in which the error occurred.

A good solution to track down the cause of such errors is to use a variant on a hardware debugging technique known as the *cut in half* method. In a complex electronic circuit that does not work, using this technique means that the circuit is first checked at its mid-point for the correct signal. If the circuit tests correctly at that point, then the error is in the second half. Therefore, the test engineer would *cut in half* again, and test at a point halfway between the middle and the end. If the test fails there, then the problem must lie between the middle of the circuit and that point.

In a purely software situation, you would add a line number to a line that falls approximately half-way through the program. If that number is reported, then the problem is occurring in the second half of the program. An enhancement to this technique that I recommend is to add, say, ten line numbers in evenly spaced increments throughout the program. This will let you quickly isolate the problem to a much smaller portion of the program.

Besides the line number (or lack of line number) that BASIC reports, the segment and address at which the error occurred is also reported. This information is frankly useless in a purely BASIC environment. You must either use CodeView to identify the line that is associated with the error, or view the assembly language output that BC can optionally generate. These will be described in the section on advanced debugging later in this chapter.

Finally, it is important to point out that you should never use ON ERROR while a program is being developed. ON ERROR can hide programming errors that you need to know about. As an example, a LOCATE statement with incorrect values will generate an "Illegal function call" error. But if ON ERROR is in effect and your program uses RESUME NEXT for errors it is not expecting, you may never even know that an error occurred. If you run the complete program below you can see that there is no indication that an error occurred at the obviously illegal LOCATE statement.

```
CLS
ON ERROR GOTO HandleErr
LOCATE 100, -90
PRINT "My program seems to work fine."
END

HandleErr:
RESUME NEXT
```

Using the QB and QBX Editing Environments

The single most powerful debugging feature that is available to you is the BASIC editing environment. More than just an editor that you can use to enter program statements, the QB environment is exactly that: a complete editing environment for developing and testing BASIC programs. The BASIC editor lets you enter program statements, single-step through a program, examine variable values, and much more. Besides being able to execute commands singly and in sequence, you can also trace into subroutines and functions, and even run your program in reverse.

The primary advantage of using the QB environment instead of a separate editor is the enhanced debugging capabilities. In most high-level languages, you first write a program using an editor, and then compile and run it to see if it works correctly. If an error occurs, you must start the editor again, load your program, and study the code to see what went wrong. In contrast, QB lets you run your program at the same time it is being edited. You can even modify the program while it is running and then resume execution, view and change variable values, and change the order in which statements are executed.

Further, BASIC can be instructed to stop and return to the edit mode when the program reaches a certain statement, or when a particular logical condition becomes true. For example, you can tell BASIC to halt the program when a variable takes on a specified value. These are extremely powerful debugging tools which have no equal in any other language. In the sections that follow, I will describe each of these capabilities in detail.

Step and Trace Debugging

Early versions of Microsoft BASIC offered a very primitive trace capability that displayed the line numbers of the currently executing statements. Although this was better than nothing, interpreting a blur of line numbers flashing by on the screen required a lot of mental effort. When Microsoft introduced QuickBASIC version 3.0 they added greatly improved debugging in the form of a step and trace feature. To activate step and trace you would enter a STOP statement at a selected point in the source code. When the program reached that point you could then execute each statement in sequence by pressing a function key. QuickBASIC 3 also provided the ability to display continuously the value of a single variable in a window at the top of the screen.

QuickBASIC 4.0 offered an improved version of this feature, using additional function keys to control how a program proceeds. This method has been continued with little change through current versions of QuickBASIC and BASIC PDS. Of course, the primary reason you would want to step through a program one statement at a time is to determine why it is not working. For example, if you have code that opens a file for output but the file is never created, you would step through that portion of the code to see which statements are being executed and which are not. In particular, stepping through a program lets you see which path an IF or CASE test is taking.

Two function keys are used to single-step through a program, and four additional options are available to assist program debugging. Each time the F10 key is pressed, the current statement is executed and the program advances to the next statement. If you have just loaded the program being tested, you will press F10 once to get to the first instruction. Pressing F10 again executes that statement, and continues to the next one. If the current statement is related to screen activity, the screen is switched momentarily to display the program's output rather than the source code. The screen is also switched during a CALL statement or function invocation, in case that routine performs screen output. You can optionally toggle between viewing the output and edit screens manually by pressing F4.

In some cases you may want to treat a subroutine as a single statement, which is what F10 does. That is, `CALL MySub` is handled as single statement, and all of the statements within the routine are executed as one operation. In other cases, however, you may need to trace into a subprogram, `GOSUB` routine, `DEF FN`, or function, to step through its statements as well. This is what F8 is for. When F8 is pressed at a `CALL` or `GOSUB` statement or function invocation, BASIC traces into the procedure and lets you watch as it executes each statement individually.

Two additional capabilities let you navigate a program more quickly. Pressing F7 tells BASIC to execute all of the statements up to the current cursor location. This way, you are spared from having to watch a long sequences of commands that you know are working correctly. For example, stepping through a `FOR/NEXT` loop that initializes 1000 elements in an array is usually pointless. Therefore, when you reach that spot in the program you would manually move the cursor to the statement following the `NEXT`, and press F7.

It is also possible to force execution to a particular point in the program using the "Set next statement" option of the Debug menu. Unlike F7, though, the statements that precede the selected line will not be executed. Therefore, this option is equivalent to adding a temporary `GOTO` to the program, causing it to jump to the specified line.

One of the most powerful features of the BASIC editor is that you can actually modify your program, then resume execution. In earlier versions of QuickBASIC, making even the slightest change to a program—even if only to a single comment—the entire program would have to be recompiled. BASIC can now preserve variable values and indeed the entire program state during most types of editing operations.

The last important step operation I want to mention now is the History feature. This too must be selected from a menu, and using it will slow your program's operation considerably. When the History option is selected from the Debug menu, BASIC remembers the last 25 program statements, and lets you step through your program in reverse. For example, if a variable has taken on an incorrect value, you can walk backwards through the program to see what statements caused that to happen. Where F8 steps forward through your program, Shift-F8 instead steps backward.

Watch Variables and Breakpoints

As powerful as BASIC's single-step feature is, it is only half of the story. Equally important is the Watch capability that lets you view a program's variables in real time. One or more variables may be placed into a special Watch window at the top of the editing screen, and their values will be displayed and updated after each statement is executed. Between the Step and Watch features, you can observe all aspects of your program's operation as it is executing.

Besides watching variable values, you can also monitor complex expressions and function results. For example, you could watch the value of `X% * Y% + Z%`, `ASC(Work$)`, or the result of a function such as `StrFunction$(Array$(), Count%)`. Because each variable or expression is updated

after every program statement, your program will run more slowly when many items are displayed in the watch window. However, this is seldom a problem in a debugging situation, and the ability to see precisely what is happening far outweighs the minor speed penalty.

Being able to watch the results of expressions as well as simple variables offers some useful and interesting techniques. As an example, suppose you are watching a string variable named `Buffer$`. If `Buffer$` is very long, you can use `LEFT$` or `MID$` to watch just a portion of the string: `MID$(Buffer$, CurPointer%, 70)`. This expression displays the 70-character portion of `Buffer$` that is currently pointed to by `CurPointer%` (assuming, of course, you are using variables with those names).

Likewise, if you are observing a string but nothing is showing in the watch window, you could watch `"{" + Work$ + "}"`. This displays `"{}"` if the string is null, and shows if there are leading or trailing blanks or `CHR$(0)` bytes. Adding braces also lets you see if the string contains characters that begin past the edge of the visible window.

One particularly powerful use of BASIC's Watch capability is related to the fact that all of the expressions are evaluated anew at each statement. Earlier I mentioned how insidious "String space corrupt" errors can be, because BASIC checks the integrity of its string memory only when a string is being assigned. Therefore, watching the expression `FRE(Any$)` tells BASIC to evaluate string memory after every source line. Thus, as soon as string memory is corrupted it will be immediately reported. This technique can be extended to identify a "Far heap corrupt" error as well, by watching the expression `FRE(-1)`.

Besides the Step and Watch capabilities, there are two additional features you should understand: *Break Points* and *Watch Points*. When a program is very large and complex, it becomes impractical to step and trace through every statement. Also, in some cases you may not know at which statement an error is occurring.

Pressing F9 sets up a Break Point which tells BASIC to halt when it reaches that point in the program, regardless of how it arrived there. You can have multiple break points, and the program will run normally until the specified statement is about to be executed. Simply place the cursor on the line at which the program is to stop, and press F9. That line will be highlighted to show that it is currently a Break Point. Pressing F9 again removes the Break Point.

A Watch Point tells BASIC to execute the program, until a certain condition becomes true. Some examples of Watch Points are `X% = 100`, `ABS(Total#) > 1000`, and `FRE("") < 1000`. In the first example you are telling BASIC to stop the program and return to the editor when `X%` equals 100. The second example will stop the program when the absolute value of `Total#` exceeds 1000, and the third halts it when there are less than 1000 bytes of string space remaining.

Considered together, these debugging features are extremely powerful. You can tell BASIC, in effect, *Run until the value of Count% hits 14; then stop the program, and let me walk backwards through the program to see how that happened.*

Using /d to Detect Errors

Another very powerful debugging solution at your disposal is to compile your program with the /d debug option. When creating an .EXE file in the BASIC environment from the Run menu, you would select the "Produce debug code" option. Compiling with /d tells BC to add three important safeguards to the code it generates. Some of these debugging issues were described in Chapter 1, but they deserve elaboration here.

The first code addition is a call to a central event handler prior to every BASIC program statement, to detect if Ctrl-Break was pressed. Normally, a compiled BASIC program is immune from pressing Ctrl-Break and Ctrl-C, unless the program is processing an INPUT statement. BASIC adds break checking to let you get out of an endless loop or other similar situation, without having to reboot your computer.

The second addition is an overflow test following each integer and long integer addition, subtraction, and multiplication, to detect results that exceed the range of legal values. If you have a statement such as `X% = Y% * Z%` and the result after multiplying is greater than 32767, the overflow test will detect that and produce an error message. Otherwise, X% would be assigned an erroneous value and your program would have no way to detect it. Floating point operations do not need any additional testing, because overflows are detected and reported whether or not /d is used.

The last additional code that BASIC adds when /d is used is array element bounds checking. If you have dimensioned an array and attempt to assign an element that doesn't exist, a compiled BASIC program will normally ignore the error. For example, if an array has been dimensioned using `DIM Array%(1 TO 100)` and you then have the statement `Array%(200) = 12`, BASIC will store the value 12 at what would have been the 200th element. This can lead to disastrous consequences such as overwriting an element in another array, or corrupting string memory. When /d is used BASIC adds additional code to check every array element referenced, and reports an error if that element does not exist.

Because of the added checking for overflow errors and illegal element numbers, a program compiled with /d will be larger and run more slowly than one in which /d is not used. Therefore, you should not release a program for general use that has been compiled with the debug option. One exception worth noting is that QuickBASIC versions 4.0 and 4.5 contain a bug that generates incorrect code for certain long integer array operations. The only solution when that happens is to use /d. This way, the routine that calculates element addresses and checks for illegal element numbers is used, rather than the incorrect in-line code that BC produces directly.

You could also compile with the /ah (huge array) switch, which uses the same routine to calculate and check array element addresses. Using /ah has an advantage over /d in this case, because your program will not be halted if Ctrl-Break is pressed. Using /ah also avoids the extra code and time to check for

overflow errors. However, /ah affects dynamic arrays only, and errors with static arrays will not be prevented.

When a program is run in the BASIC editor, the same protection that /d provides is employed. This added debug testing within the editor is one more contributor to its slowness when compared to a fully compiled program.

Advanced Debugging

Although being able to step through your program and watch its variables in the BASIC editing environment is very powerful, there are still some limitations inherent in that process. For example, it is possible that a program will work perfectly in the editor, but not when it has been compiled to an .EXE program. Microsoft has tried to make the BASIC editor as compatible with BC as possible, but the editor is an interpreter and not a true compiler. There are bound to be some differences in how the program runs. Another limitation is that some programs are just too large to be run within the editor. Finally, if you receive an error message from an executable program that lists only a segment and address, there is no way to determine where the error occurred using the editor.

In these cases you will need to work with the actual compiled program. To relate an error address to the original BASIC source statement you must be able to see the assembly language code that BC generates, along with the original BASIC source. One way to do this is with the Microsoft CodeView debugger. CodeView comes with BASIC PDS and VB/DOS Professional Edition, as well as with Microsoft's Macro Assembler. CodeView provides a debugging environment that is similar to the QB editor, except it is intended for tracing through a program that has already been compiled.

Another way is to instruct BC to generate an assembly language source listing as it compiles your program. This listing shows a mix of BASIC source statements and the resultant assembly language code and addresses. However, the listing is not as clear or easy to follow as the display that CodeView presents. But if you do not have CodeView, this is your only choice. I will describe this method first.

Creating an Assembly Language Source Listing

To create an assembly language list file you use the compiler's /a switch, and then specify a list file name. The syntax is shown below, followed by a sample list file that is generated.

You enter this:

```
bc program /a [/other options] , , listfile;
```

LISTFILE.LST contains this:

```
PAGE    1  
25 June 91
```

14:28:08

Microsoft (R) QuickBASIC Compiler Version 4.50

```
Offset Data Source Line
0030 0006 CLS
0030 0006 INPUT Count%
0030 ** I00002: mov ax,0FFFFh
0033 ** push ax
0034 ** call B$SCLS
0039 ** mov ax,offset <const>
003C ** push ax
003D ** call 0000h
0040 ** pop ax
0041 ** add ax,000Dh
0044 ** push cs
0045 ** push ax
0046 ** call B$INPP
004B ** jmp $+04h
004D ** dw 0002h
004F ** db 00h
0050 ** db 02h
0051 ** mov bx,offset COUNT%
0054 ** push ds
0055 ** pop es
0056 ** push es
0057 ** push bx
0058 ** call B$RDI2
005D 0008 IF Count% < 100 THEN
005D 0008 Count% = 100
005D 0008 END IF
005D ** call B$PEOS
0062 ** cmp word ptr COUNT%,64h
0067 ** jl $+03h
0069 ** jmp I00003
006C ** mov COUNT%,0064h
0072 0008 PRINT Count%
0072 0008 END
0072 0008
0072 0008 I00003: push COUNT%
0076 ** call B$PEI2
007B ** call B$CEND
0080 ** call B$CENP
0085 0008
```

43981 Bytes Available

43643 Bytes Free

0 Warning Error(s)

0 Severe Error(s)

Here, the list file shows the original BASIC source code, as well as the generated assembly language instructions. The column at the left holds the code addresses, and these correspond to the addresses that BASIC displays when a program crashes with an error message. Unfortunately, several BASIC statements are grouped together, so it is not immediately apparent which address goes with which source statement. For example, after the BASIC statement `INPUT Count%`, the earlier assembly language instructions that clear the screen are shown. Similarly, the call to `B$PEOS` is actually part of the `INPUT` code, although it is listed following the `IF` test.

When BASIC displays an error message and ends your program by displaying a segmented address, only the address portion is meaningful. The segment in which a program is running will depend on many factors, including the DOS version (and thus its size), the FILES= and BUFFERS= values specified in CONFIG.SYS, and whether TSR programs and device drivers are loaded. Each of these factors cause the program to be loaded at a higher segment, although the addresses within that segment never change. Also, in a multi-module program, a different segment is used for each module's source file. Therefore, if the message is "Illegal function call in module XYZ at address 3456:1234", you would compile XYZ.BAS to create a list file instead of the main program. The code in the vicinity of address 1234 will be where the error occurred.

Using Microsoft CodeView

Although compiling with the /a switch lets you view the assembly language code that BASIC creates, there is little you can actually do with that information. CodeView is a much more powerful debugging tool, and it lets you step through an .EXE file as it is running. This lets you follow the compiled program's execution path, and also view its assembly language instructions. Further, CodeView can trace into BASIC's library routines, as well as calls to C or assembly language routines that you have written.

CodeView can also be used to see how many bytes of code are generated for each BASIC statement. This is a good way to compare the relative efficiency of different programming methods, to see which ones produce less code. It is important to understand that the size of the assembly language code generated for a given BASIC statement is a combination of two factors: the number of bytes the compiler generates for each occurrence of the statement, and the size of the called routine within BASIC's runtime library. Of course, the called routine is added to your program only once. However, the code that sets up and calls the routine is added each time the statement is encountered.

Compiling a program for use with CodeView is very simple, and merely requires the addition of special compiler and linker option switches. Note that you cannot compile a program for CodeView from within the QuickBASIC editor; you must compile and link manually from the DOS command line, as shown below. Also notice that the BASIC program must be saved as ASCII text, and not with the special "Fast Load" method that QB optionally uses.

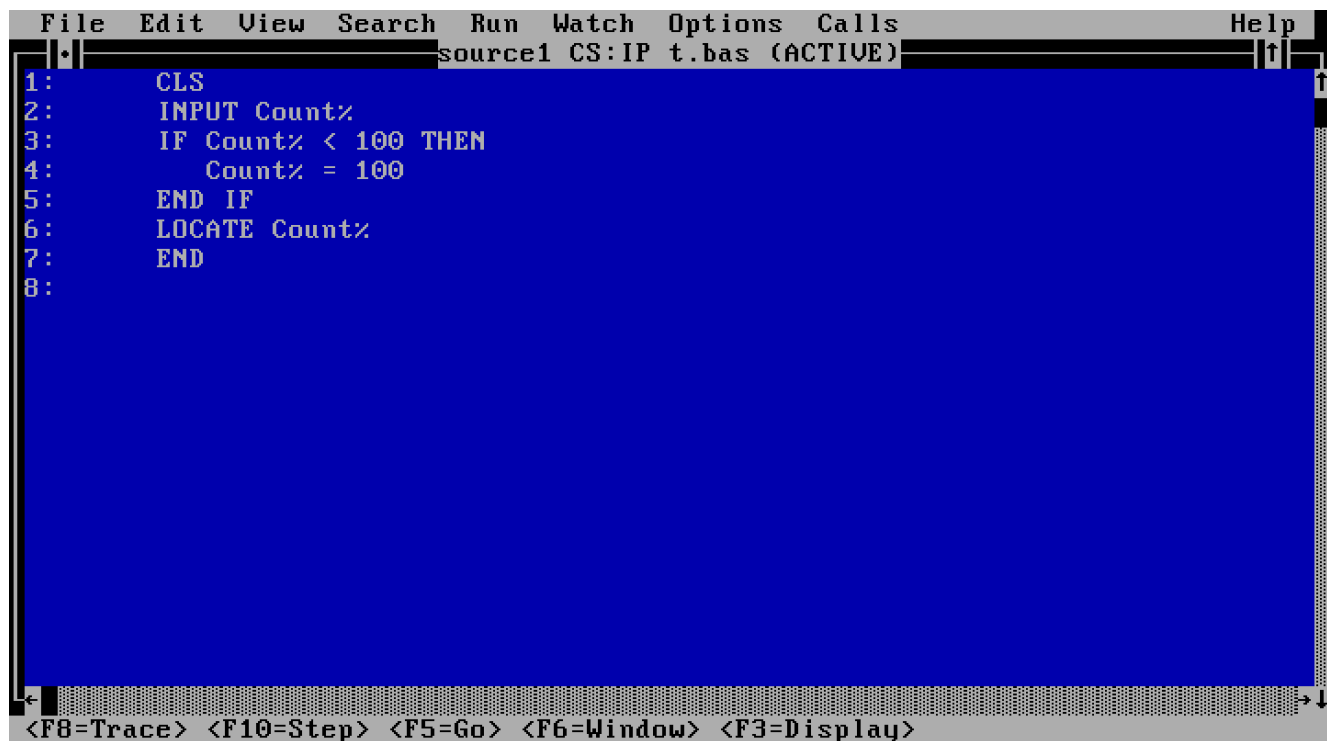
```
bc program /zi [/other options];  
link program /co [/other options];  
cv program
```

The /zi option tells BC to write additional information into the object file, which is used by LINK and CodeView to relate each line of BASIC source code to its resultant assembly code. The more meaningfully named /co switch is required so LINK will know to do likewise. You may be interested to know that /zi is named after Microsoft legend Mark Zibikowski, whose initials (MZ) also appear as the first two bytes in every DOS .EXE file.

Once the program has been compiled and linked, start CodeView by entering CV followed by the file's first name (that is, without the .BAS or .EXE extension). You will then be presented with a screen very similar to that of the QB editor. Most versions of CodeView initially show the BASIC source code. In other versions, you must press Alt-R-R to "restart" the program and bring it to the first source line. I should point out that CodeView is a quirky program, and it is often referred to as the program that people love to hate. It has some glaring omissions, many aspects of its interface are inconsistent and downright obnoxious, and I personally would be lost without it.

When the BASIC source is displayed, you may press F4, F7, F8, and F10, which perform the same functions as their BASIC editor counterparts. One important difference, however, is that you may also press F3 to show a mix of BASIC and assembly language code. Stepping through the program with F8 and F10 will execute either a single BASIC statement or a single assembler command, depending on the context. That is, if you are in the BASIC view mode, then you will step through the BASIC code. If the assembly language code is being displayed, then you will step through that instead.

Figure 4-1 shows a screen snapshot of a short sample program as displayed by CodeView when it is first started in the BASIC view mode. Figure 4-2 shows the same program after pressing F10 to execute up to the first statement, followed by F3 to view a mix of BASIC and assembly language. This screen is in a 50-line mode to allow the entire program to be displayed. Although it is not shown here, CodeView can continuously display the processor's registers in a small window at the right side of the screen. The register display is alternately activated and deactivated by pressing F2.



```
File Edit View Search Run Watch Options Calls Help
source1 CS:IP t.bas (ACTIVE)
1: CLS
2: INPUT Count%
3: IF Count% < 100 THEN
4:     Count% = 100
5: END IF
6: LOCATE Count%
7: END
8:
<F8=Trace> <F10=Step> <F5=Go> <F6=Window> <F3=Display>
```

Figure 4-1: The CodeView display when using the BASIC view mode.

```

File Edit View Search Run Watch Options Calls Help
source1 CS:IP t.bas (ACTIVE) reg
1 : CLS
5FEE:0030 B8FFFF MOV AX,FFFF
5FEE:0033 50 PUSH AX
5FEE:0034 9A5732F65F CALL B$SCLS (5FF6:3257)
2 : INPUT Count%
5FEE:0039 B84000 MOV AX,b$STRTAB_END+10 (0040)
5FEE:003C 50 PUSH AX
5FEE:003D E80000 CALL B$LENDRW+31 (0040)
5FEE:0040 58 POP AX
5FEE:0041 050D00 ADD AX,b$STRTAB+d (000D)
5FEE:0044 0E PUSH CS
5FEE:0045 50 PUSH AX
5FEE:0046 9AE82CF65F CALL B$INPP (5FF6:2CE8)
5FEE:004B EB04 JMP B$LENDRW+42 (0051)
5FEE:004D 0200 ADD AL,Byte Ptr [BX+SI]
5FEE:004F 0002 ADD Byte Ptr [BP+SI],AL
5FEE:0051 BB3600 MOV BX,COUNT% (0036)
5FEE:0054 1E PUSH DS
5FEE:0055 07 POP ES
5FEE:0056 06 PUSH ES
5FEE:0057 53 PUSH BX
5FEE:0058 9A922EF65F CALL B$RDIZ (5FF6:2E92)
5FEE:005D 9ADA26F65F CALL B$PEOS (5FF6:26DA)
3 : IF Count% < 100 THEN
5FEE:0062 833E360064 CMP Word Ptr [COUNT% (0036)],+64
5FEE:0067 7C03 JL B$LENDRW+5d (006C)
5FEE:0069 E90600 JMP B$LENDRW+63 (0072)
4 : Count% = 100
5FEE:006C C70636006400 MOV Word Ptr [COUNT% (0036)],b$HEAP_F
6 : LOCATE Count%
5FEE:0072 B80100 MOV AX,b$STRTAB+1 (001)
5FEE:0075 50 PUSH AX
5FEE:0076 FF363600 PUSH Word Ptr [COUNT% (0036)]
5FEE:007A B80200 MOV AX,b$STRTAB+2 (0002)
5FEE:007D 50 PUSH AX
5FEE:007E 9AE431F65F CALL B$LOCT (5FF6:31E4)
7 : END
5FEE:0083 9AA71AF65F CALL B$CEND (5FF6:1AA7)
9:
5FEE:0088 9A901AF65F CALL B$CENP (5FF6:1A90)
5FEE:008D 00C6 ADD DH,AL
5FEE:008F 06 PUSH ES
5FEE:0090 E001 LOOPNE B$LENDRW+84 (0093)
AX = 0000
BX = 0000
CX = 0000
DX = 0000
SP = 0C00
BP = 0000
SI = 0000
DI = 0000
DS = 0D10
ES = 0D10
SS = 0EB8
CS = 0DB2
IP = 00CC
FL = 0200
NU UP EI PL
NZ NA PO NC
<F8=Trace> <F10=Step> <F5=Go> <F6=Window> <F3=Display>

```

Figure 4-2: The CodeView display for the same program, but using the assembly language view mode.

Notice in Figure 4-2 that CodeView displays each BASIC statement indented and with a line number. This lets you identify where each BASIC command starts, and also which block of assembly language code it is associated with. The numbers at the left edge of the display show the segment and address of each instruction in hexadecimal notation. The segment value never changes within a single program module, although the addresses increase based on the number of bytes in each assembly language instruction. As you can see, some assembly language commands are as short as one byte, and others are as long as six.

In the first instruction, CLS, a value of -1 (FFFF hex) is passed to the CLS routine as a flag to show that no argument was given. Had the BASIC statement been CLS 2, then a value of 2 would have been moved into AX instead. Nine bytes of code are generated each time CLS is used, not counting the code within B\$\$CLS. Besides showing the B\$\$CLS routine name, CodeView also shows the segment and address at which B\$\$CLS resides. Knowing the routine's address is of little practical use in this situation, and it is displayed solely for informational purposes.

The INPUT statement is fairly complicated to set up, and I won't belabor what every assembly language instruction does, but several items are worth discussing. The first is that CodeView attempts to relate every number it encounters to a variable or procedure address. In many cases this is confusing, because some numbers are simply that, and have no relationship to a variable or procedure address.

For example, at address 39 the assembly language command `MOV AX,40` is shown as `MOV AX,b$STRTAB_END+10 (0040)`, as if there was some significance to the fact that the value 40 is an address ten bytes past the end of an internal string table. Likewise, two instructions later the value 40 is represented as being 31 bytes past the beginning of the B\$LENDRW procedure. Two instructions past that the value 13 (0D hex) is added to AX, and again CodeView tries to establish a significance where none exists.

In not one of these cases are the values shown related to the named address, and you should therefore treat those named labels with skepticism. The only symbolic names that are meaningful in most cases are variable and procedure names that do not have an extra value added to them. In the instruction `MOV Word Ptr [COUNT% (0036)],b$HEAP_FIRST (0064)` at address 6C, the address for Count % (36) is valid, while the value 64 named b\$HEAP_FIRST is meaningless. In this case, 64 hex represents the value 100 in the BASIC statement `Count% = 100`. Whatever b\$HEAP_FIRST may represent, it has no meaning here.

I suggest that you enter this short program and then step through it one statement at a time, just to get a feel for how CodeView operates. You should also try tracing into some of the BASIC library calls, as well as into a simple subprogram or two of your own. Again, you may use either F10 or F8 to step through the code, but only F8 will trace into code that is being called. You can also use F8 to trace into some BIOS interrupts, but you should never try to trace through a DOS interrupt (21 hex). Many DOS services never return, or return in a non-standard manner, and a locked-up PC is the likely result. You will not hurt anything if you do trace into a DOS interrupt, but be prepared to press Ctrl-Alt-Del.

Besides being able to view and step through the assembly language code that BASIC creates, you can also view and modify your program's data directly. If you have pressed F2 to display the CPU's registers, CodeView will show the value currently in every memory address that is about to be accessed. For example, if the next statement to be executed is `MOV Word Ptr [COUNT%],10`, CodeView will show the current contents of the variable `COUNT%`.

A range of memory addresses may be displayed by entering commands into the immediate window at the bottom of the screen. When CodeView is first started, the cursor is placed at the bottom line in that window. As with the BASIC editor, the F6 key is used to toggle between the code output and immediate windows. Unlike the BASIC editor, however, you may type commands regardless of which window is active.

The three primary commands you will find useful are D, U, and R. The D (Dump) command tells CodeView to display a range of memory, starting at a given address. For example, `D 0` means to show the 32 bytes that start at address 0 in the default data segment. Likewise, `D ES:100` means to start at address 100 in the segment held in the ES register. Unfortunately, CodeView is particularly obtuse in this regard, because in some cases the numbers you enter are assumed to be decimal while in others it assumes hexadecimal. Which is which depends on your view perspective (selected with F3), and I won't even begin to offer a reason or explain the confusing rules. If you don't get what you expect, try adding an "&H" prefix to the number. And if you start by using &H and CodeView reports a syntax error, then try it without the &H.

When the contents of memory are displayed, they are shown as individual bytes, rather than as integer words which is generally more useful. In the listing below, two string constants have been displayed in response to the command `D &H40`.

```
5676:0040 02 00 44 00 48 69 23 00 4A 00 41 42 43 44 45 46
5676:0050 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56
```

As you learned in Chapter 2, BASIC near strings have a 4-byte descriptor, with the first two bytes holding the string's current length, and the second two bytes its current address. Beginning with the first two numbers displayed, the `02 00` represents the length of a 2-character string, and the `44 00` indicates the address which is 44. The data itself is a `CHR$(&H48)` followed by a `CHR$(&H69)` ("Hi"), and it immediately follows the string descriptor. When two bytes are used to store an integer word, the least significant byte is kept in the lower memory address. Therefore, the value 0002 is actually listed as `02 00` (CodeView adds an extra blank between bytes for clarity).

Immediately following the six bytes for the string "Hi" and its descriptor is another descriptor. This one shows that the string has a length of 23 Hex bytes, and its data starts at address 4A Hex. Again, the value 0023 is shown as `23 00`, and the address 004A is displayed as `4A 00`. This string contains the data "ABCDEFGHIJKLMNOPQRSTUVWXYZ".

The U (Unassemble) command can be used to show the assembly language source code at any arbitrary segment and address. The command `U 2000:1000` will unassemble the code at address 2000:1000, though again you may need to use `U &H2000:&H1000` in some view modes. The U command is not

used that frequently, since CodeView is used most often to step through code in sequence, rather than to examine an arbitrary block of instructions.

The R command lets you change the contents of a register, and this might be useful when debugging your own assembly language subroutines. When you type, for example, `RCX` and press Enter, the current value of the CX register is displayed and you are prompted for a new value. Pressing Enter alone cancels the command and leaves the current register contents intact. Otherwise, the value you enter will be assigned to CX. This is similar to BASIC's immediate window, in which you can assign new values to a variable.

The last CodeView features worth describing here are Watch Variables and Watch Points, which are similar to the same features in QB. Unlike QB, though, you cannot use an expression as the target of a Watch; it must be a simple variable name, array element, or address. Watch Variables may be added using the pull-down menu, or by pressing Alt-W and then typing the variable name. If you are in the BASIC view mode you may add only BASIC variables; in the assembly language view mode you can add only assembly language variables. To monitor the contents of a memory address requires the W command. For example, `W 40` will set up address 40 as the target of a Watch.

Although CodeView does support Watch points, whereby the program will run continuously until a given expression is true, you won't want to use that feature. Asking CodeView to stop when, say, CX becomes greater than 100 will cause your program to run at less than one thousandth its normal speed. Therefore, I have never found using Watch Points effective in any situation—it is always too slow.

I have avoided discussing the latest versions of CodeView, in favor of focusing on those features which are common to all versions. CodeView 3.10 which is included with BASIC 7.1 has several new convenience features, and a few new bugs as well. Many of the commands that in earlier versions have to be entered manually are now available by simply typing new values onto the display. For instance, where older versions of CodeView required you to enter Dump commands repeatedly, the new version updates the displayed values in a range of addresses constantly. And to change the address range, you may now simply move the cursor to the segment and address numbers and type new ones. An option to display memory values as words or even single and double precision values is also present in version 3.10.

Now that you have seen what CodeView is all about and how to use it, I want to conclude this chapter with a practical example. As I mentioned in Chapter 3, the amount of stack memory that is needed in a non-static subprogram or function can be difficult to determine. The calculation itself is trivial: simply add up the number of bytes needed by every variable in the routine. Each integer requires two bytes, single precision, long integer, and string variables need four bytes, and so forth. The problem, of course, is who wants to do all that counting, especially when there may be hundreds of variables. Counting is what computers are for, no?

The solution is that BASIC knows how many bytes are needed for the subprogram, and the very first thing a subprogram does when it is invoked is to call another routine that allocates the necessary stack space. So rather than use trial and error methods to increase the stack in small increments, you can use

CodeView to directly see how many bytes of stack space are being requested. Here's how that's done, using the example program shown below.

```
DEFINT A-Z
DECLARE SUB StackTest (Dummy)
Test = 10
CALL StackTest (Test)
END

SUB StackTest (AnyVar)
  X = 100
  Y = 10
  Z = AnyVar
END SUB
```

Save this program as an ASCII file using the name TEST.BAS, and then compile it with the /o and /zi options. Next, link TEST.OBJ for CodeView using the /co option. Then start CodeView by entering CV TEST. Once you are in CodeView and viewing the BASIC source, press F10 to skip past BASIC's start-up code. At this point the cursor should be on the first statement, `Test = 10`. Finally, press F3 to show a mix of BASIC and assembly language source code. The display should look similar to that shown in Figure 4-3.

```

File Edit View Search Run Watch Options Calls Help
source1 CS:IP test.bas (ACTIVE)
3:      Test = 10
6557:0030 C7063600A00    MOV      Word Ptr [TEST% (0036)],b$STRTAB+a (000A)
4:      CALL TestStack(Test)
6557:0036 B83600              MOV      AX,TEST% (0036)
6557:0039 50                  PUSH     AX
6557:003A 9A47005765         CALL     TESTSTACK (6557:0047)
5:      END
6557:003F 9A430F5D65         CALL     B$CEND (655D:0F43)
6557:0044 E92200              JMP      TESTSTACK+22 (0069)
TESTSTACK:
7:      SUB TestStack (AnyVar)
6557:0047 B90600              MOV      CX,b$STRTAB+6 (0006)
6557:004A 9A9F165D65         CALL     B$ENRA (655D:169F)
8:      X = 100
6557:004F C746F46400         MOV      Word Ptr [X%],B$MSINA+3 (0064)
9:      Y = 100
6557:0054 C746F26400         MOV      Word Ptr [Y%],B$MSINA+3 (0064)
10:     Z = AnyVar
6557:0059 8B7606              MOV      SI,Word Ptr [ANYVAR%]
6557:005C 8B04                MOV      AX,Word Ptr [SI]
6557:005E 8946F0              MOV      Word Ptr [Z%],AX
11:     END SUB
6557:0061 9A74165D65         CALL     B$EXSA (655D:1674)
6557:0066 CA0200              RETF     b$STRTAB+2 (0002)
13:
6557:0069 9A2C0F5D65         CALL     B$CENP (655D:0F2C)
6557:006E 56                  PUSH     SI
6557:006F 8BDD                MOV      BX,BP
6557:0071 8BCB                MOV      CX,BX
6557:0073 8B1F                MOV      BX,Word Ptr [BX]
6557:0075 3B1EF004           CMP      BX,Word Ptr [b$curframe (04F0)]
6557:0079 75F6                JNZ      TESTSTACK+2a (0071)
6557:007B 8BD9                MOV      BX,CX
6557:007D 33C9                XOR      CX,CX
6557:007F 8B5F04              MOV      BX,Word Ptr [BX+04]
6557:0082 98                  CBW
<F8=Trace> <F10=Step> <F5=Go> <F6=Window> <F3=Display>

```

Figure 4-3: How to determine the amount of stack memory needed for a non-static procedure.

Notice the first statement within the TestStack subprogram at line 7, where the value 6 (erroneously labelled b\$STRTAB+6) is assigned to the CX register. This is the number of bytes of stack space being requested from the B\$ENRA routine which is called in the next instruction. B\$ENRA is the routine that actually allocates the stack memory, and it uses the value BASIC sends in CX to know how many bytes are needed. TestStack has three local variables and each is a two-byte integer, hence six bytes are required to store them on the stack.

For a very large program, the value assigned to CX will of course be much larger. Further, if one subprogram calls another, it will be up to you to add up all of the CX values to determine the total stack memory requirements. But this is very much easier than counting variables.

Summary

In this chapter you have learned how to identify and correct common programming errors. You have also learned the importance of understanding BASIC's various quirks, and how some statements do not always do exactly what you thought they would. I have shown several debugging strategies, including a software adaptation of the "cut in half" hardware technique.

Perhaps your most powerful debugging ally is the QuickBASIC and QBX editing environments. These powerful editors let you single step through a program, monitor variable values and function results, and halt your program when a specified condition occurs.

When BASIC terminates a program prematurely with an error message and a segmented address, you can either use the BC compiler's /a option to generate a source listing, or use CodeView to see where the error occurred. CodeView can also be used to step and trace through a program at the assembly language source level, and to determine the number of bytes of stack memory a non-static procedure requires.

In Chapter 5 you will learn about compiling and linking BASIC programs. I will present a complete overview of the many BC and LINK options that are available, and discuss the relative merits of each.

5

Compiling and Linking

The final step in the creation of any program is compiling and linking, to produce a stand-alone .EXE file. Although you can run a program in the BASIC editing environment, it cannot be used by others unless they also have their own copy of BASIC. In preceding chapters I explained the fundamental role of the BASIC compiler, and how it translates BASIC source statements to assembly language. However, that is only an intermediate action. Before a final executable program can be created, the compiled code in the object file must be joined to routines in the BASIC language library. This process is called linking, and it is performed by the LINK program that comes with BASIC.

In this chapter you will learn about the many options and features available with the BASIC compiler and LINK. By thoroughly understanding all of the capabilities these programs offer, you will be able to create applications that are as small and fast as possible. Many programmers are content to let the BASIC editor create the final program using the pull-down menu selections. And indeed, it is possible to create a program without invoking BC and LINK manually—many programmers never advance beyond BASIC's "Make .EXE" menu. But only by understanding fully the many options that are available will you achieve the highest performance possible from your programs.

I'll begin with a brief summary of the compiling and linking process, and explain how the two processes interact. I will then move on to more advanced aspects of compiling and linking. BC and LINK are very complex programs which possess many features and capabilities, and all of their many options will be described throughout this chapter. You may also refer back to Chapter 1, which describes compiling in more detail.

An Overview of Compiling and Linking

When you run the BC.EXE compiler, it reads your BASIC source code and translates some statements directly into the equivalent assembly language commands. In particular, integer math and comparisons are converted directly, as well as integer-controlled DO, WHILE, and FOR loops. Floating point arithmetic and comparisons, and string operations and comparisons are instead translated to calls to existing routines written by the programmers at Microsoft. These routines are in the BCOM and BRUN libraries that come with BASIC.

As BC compiles your program, it creates an object file (having an .OBJ extension) that contains both the translated code as well as header information that LINK needs to create a final executable program. Some examples of the information in an object file header are the name of the original source file, copyright notices, offsets within the file that specify external procedures whose addresses are not known at compile time, and code and data segment names. In truth, most of this header information is

of little or no relevance to the BASIC programmer; however, it is useful to know that it exists. All Microsoft-compatible object files use the same header structure, regardless of the original source language they were written in.

The LINK program is responsible for combining the object code that BC produces with the routines in the BASIC libraries. A *library* (any file with a .LIB extension) is merely a collection of individual object files, combined one after the other in an organized manner. A header portion of the .LIB file holds the name of each object file and the procedure names contained therein, as well as the offset within the library where each object module is located. Therefore, LINK identifies which routines are being accessed by the BASIC program, and searches the library file for the procedures with those names. Once found, a copy of that portion of the library is then appended to the .EXE file being created.

LINK can also join multiple object files compiled by BC to create a single executable program, and it can produce a Quick Library comprised of one or more object files. Quick Libraries are used only in the editing environment, primarily to let BASIC access non-BASIC procedures. Because the BASIC editor is really an interpreter and not a true compiler, Quick Libraries were devised as a way to let you call compiled (or assembled) subroutines during the development of a program.

When LINK is invoked it reads the header information in each object file compiled by BC, and uses that to know which routines in the specified library or libraries must be added to your program. Since every external routine is listed by name, LINK simply examines the library header for the same name. It is worth mentioning that BASIC places the name of the default library in the object file, so you don't have to specify it when linking. For example, when you compile a stand-alone program (with the /o switch) using BC version 4.5, it places the name BCOM45.LIB in the header.

BASIC is not responsible for determining where external routines are located. If your program uses a PRINT statement, the compiler generates the instruction `CALL 0000:0000`, and identifies where in the object file that instruction is located. BASIC knows that the print routine will be located in another segment, and so leaves room for both a segment and address in the Call instruction. But it doesn't know where in the final executable file the print routine will end up. The absolute address depends on how many other modules will be linked with the current object file, and the size of the main program.

In fact, LINK does not even know in which segment a given routine will ultimately reside. While it can resolve all of the code and data addresses among modules, the absolute segment in which the program will be loaded depends on whether there are TSR programs in memory, the version of DOS (and thus its size), and the number of buffers specified in the host PC's CONFIG.SYS file, among other factors. Therefore, all .EXE files also have a header portion to identify segment references. DOS actually modifies the program, assigning the final segment values as it loads the program into memory. Figure 5.1 shows how DOS, file buffers, and device drivers are loaded in memory, before any executable programs.

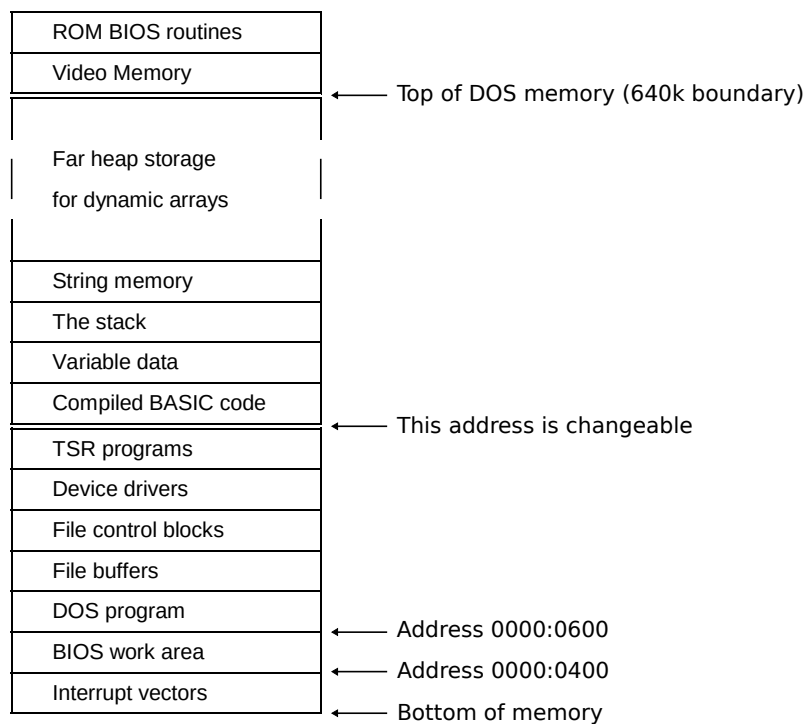


Figure 5-1: DOS and BASIC memory organization.

It is important to understand that library routines are added to your program only once, regardless of how many times they are called. Even if you use PRINT three hundred times in a program, only one instance of the PRINT routine is included in the final .EXE file. LINK simply modifies each use of PRINT to call the same memory address. Further, LINK is generally smart enough to not add all of the routines in the library. Rather, it just includes those that are actually called.

However, LINK can extract only entire object files from a library. If a single object module contains, say, four routines, all of them will be added, even if only one is called. For BASIC modules that you write, you can control which procedures are in which object files, and thus how they are combined. But you have no control over how the object modules provided with BASIC were written. If the routines that handle POS(0), CSRLIN, and SCREEN are contained in a single assembly language source file (and they are), all of them are added to your program even if you use only one of those BASIC statements.

Now that you understand what compiling and linking are all about, you may wonder why it is necessary to know this, or why you would ever want to compile manually from the DOS command line. The most important reason is to control fully the many available compile and link options. For example, when you let the BASIC editor compile for you, there is no way to override BC's default size for the communications receive buffer. Likewise, the QuickBASIC editor does not let you specify the /s (string) option that in many cases will reduce the size of your programs.

LINK offers many powerful options as well, such as the ability to combine code segments to achieve faster performance during procedure calls. Another important LINK option lets you create an .EXE file that can be run under CodeView. Again, these options are not selectable from within the QuickBASIC environment [but PDS and VB/DOS Pro Edition let you select more options than QuickBASIC], and they can be specified only by compiling and linking manually. All of these options are established via command line switches, and each will be discussed in turn momentarily.

Finally, BASIC PDS includes a number of *stub files* which reduce the size of your programs, although at the expense of decreased functionality. For example, if your program does not use the SCREEN statement to enable graphics mode, a stub file is provided to eliminate graphics support for the PRINT statement. BASIC PDS, and the VB/DOS Pro Edition, also support program overlays, and to use those requires linking manually from DOS.

Compiling

To compile a program you run BC.EXE specifying the name of the BASIC program source file. BC accepts several optional parameters, as well as many optional command line switches. The general syntax for BC is as follows, with brackets used to indicate optional information.

```
bc program [/options] [, object] [, listfile] [;]
```

In most cases you will simply give the name of the BASIC source file, any option switches, and a terminating semicolon. A typical BC command is as follows:

```
bc program /o;
```

Here, a BASIC source file named PROGRAM.BAS is being compiled, and the output object file will be called PROGRAM.OBJ. The /o option indicates that the program will be a stand-alone .EXE file that does not require the BRUN library to be present at runtime. If the semicolon is omitted, the compiler will prompt for each of the file name parameters it needs. For example, entering `bc program /o` invokes the compiler, which then prompts you for the output and listing file names. Pressing Enter in response to any prompt tells BC to use the source file's first name. You may also start BC with no source file name, and let it prompt for that as well.

In most cases the default file names are acceptable; however, it is not uncommon to want the output file placed into a different directory. This is done as follows:

```
bc program, \objdir\ /o;
```

```
Note that if the trailing backslash were omitted from \objdir\ above, BC would create an output file named OBJDIR.OBJ in the root directory. Of course, that is not what is intended. Therefore, a trailing backslash is added to tell BC to use the default name of PROGRAM.OBJ, and to place that file in the directory named \OBJDIR.
```

If you are letting BC prompt you for the file names, you would enter the output path name at that prompt position. You may also include a drive letter as part of the path, or a drive letter only to use the default directory on the specified drive. The listing that follows shows a typical BC session that uses prompting.

```
C>bc program /o

Microsoft (R) QuickBASIC Compiler Version 4.50
(C) Copyright Microsoft Corporation 1982-1988.
All rights reserved.
Simultaneously published in the U.S. and Canada.
Object Filename [PROGRAM.OBJ]: d:\objects\ <Enter>
Source Listing [NUL.LST]: <Enter>

43965 Bytes Free
43751 Bytes Available

    0 Warning Error(s)
    0 Severe Error(s)
C>
```

Although you can override the default file extensions, this is not common and you shouldn't do that unless you have a good reason to. For example, the command `BC source.txt , output.out;` will compile a BASIC source file named `SOURCE.TXT` and create an object module named `OUTPUT.OUT`. Since there are already standard default file extension conventions, I recommend against using any others you devise.

The optional list file contains a source listing of the BASIC program showing the addresses of each program statement, and uses a `.LST` extension by default. There are a number of undocumented options you can specify to control how the list file is formatted, and these are described later in this chapter in the section `Compiler Metacommands`. A list file may also include the compiler-generated assembly language instructions, and you specify that with the `/a` option switch. All of the various command options will be discussed in the section following.

Notice that the positioning of the file name delimiting commas must be maintained when the object file name is omitted. If you plan to accept the default file name but also want to specify a listing file, you must use two commas like this:

```
bc source , , listfile;
```

The Bytes Available and Bytes Free messages indicate how much working memory the compiler has at its disposal, and how much of it remained free while compiling your program. BC must keep track of many different kind of information as it processes your source code, and it uses its own internal DGROUP memory for that. For example, every variable that you use must be remembered, as well as its address.

When BASIC sees a statement such as `X = 100`, it must look in its *symbol table* to see if it has already encountered that variable. If so, it creates an assembly language instruction to store the value 100 at the corresponding address. Otherwise, it adds the variable X to the table, assigns a new address for it, and then adds code to assign the value 100 to that address. When you use `PRINT X` later on, BASIC will again search its table, find the address, and use that when it creates the code that calls the PRINT routine.

Other data that BASIC must remember as it works includes the number and type of arguments for each SUB or FUNCTION that is declared, line label names and their corresponding addresses, and quoted string constants. As you may recall, in Chapter 2 I explained that BC maintains a table of string constants, and stores each in the final program only once. Even when the same quoted string is used in different places in a program, BC remembers that they are the same and stores only a single copy. Therefore, an array is used by BC to store these strings while your program is being compiled.

In most cases you can simply ignore the Bytes Available and Bytes Free messages, since how much memory BASIC used or had available is of no consequence. The only exception, of course, is when your program is so large that BC needed more than was available. But again, you will receive an error message when that occurs. However, if you notice that the Bytes Free value is approaching zero, you should consider splitting your program into separate modules.

The error message display indicates any errors that occurred during compilation, and if so how many. This display is mostly a throw-back to the earlier versions of the BASIC compiler, because they had no development environment. These days, most people get their program working correctly in the BASIC editor, before attempting to compile it. Of course, there must still be a facility for reporting errors.

In most cases, any errors that BC reports will be severe errors. These include a mismatched number of parentheses, using a reserved word as a variable name (for example, `PRINT = 12`), and so forth. One example of a warning error is referencing an array that has not been dimensioned. When this happens, BASIC creates the array with a default 11 elements (0 through 10), and then reports that it did this as a warning.

One interesting quirk worth mentioning is that BASIC will not let you compile a program named USER.BAS. If you enter `BC USER`, BC assumes that you intend to enter the entire program manually, statement by statement! This too must be a holdover from earlier versions of the compiler; however, when USER.BAS is specified it will appear that the compiler has crashed, because nothing happens and no prompt is displayed. In my testing with BASIC 7.1, any statements I entered were also ignored, and no object file was created.

Compiler Options

All of the options available for use with the BASIC compiler are described in this section in alphabetical order. Some options pertain only to BASIC 7 PDS, and these are noted in the accompanying discussion. Each option is specified by listing it on the BC command line, along with a preceding forward slash (/). Also, these options apply to the BC compiler only, and not necessarily to the QB and QBX editing environments.

/A

The /a (assembly) switch tells BC to include the assembly language source code it creates in the listing file. The format of the file was described in detail in Chapter 4, so I won't belabor that here. Note, however, that a file name must be given in the list file position of the BC command line. Otherwise, a list file will not be written.

/Ah

Using /ah (array huge) tells BASIC that you plan to create dynamic arrays that may exceed 64K in total data size. This option affects numeric, TYPE, and fixed-length string arrays only, and not conventional string arrays. Normally, BASIC calculates the element addresses for array references directly, based on the segment and other information in the array descriptor. This is the most direct method, and thus provides the fastest performance and smallest code.

When /ah is used, all access to non-string dynamic arrays is instead made through a called routine. This called routine calculates the segment and address of a single array element, and because it must also manipulate segment values, increases the size of your programs. Therefore, /ah should be avoided unless you truly need the ability to create huge arrays. Even if a particular array does not currently exceed the 64K segment limit, BASIC has no way to know that when it compiles your program.

To minimize the size and speed penalty /ah imposes, it may be used selectively on only some of the source modules in a program. If you have one subprogram that needs to manipulate huge arrays but the rest of program does not, you should create a separate file containing only that subprogram and compile it using /ah. When the program is linked, only that module's array accesses will be slower.

Note that the /ah switch is also needed if you plan to create huge arrays when running programs in the BASIC editor. However, with the BASIC editor, using /ah does not impinge on available memory or make the program run slower. Rather, it merely tells BASIC not to display an error message when an array is dimensioned to a size greater than 64K.

```
The BASIC editor always uses the slower code that
checks for illegal array elements anyway, so it can
report an error rather than lock up your computer.
```


One limitation that /ah will not overcome is BASIC's limit of 32,767 elements in a single dimension. That is, the statement REDIM Array%(1 to 32768) will fail, regardless of whether /ah is used. There are two ways to exceed this limit: one is to create a TYPE array in which each element is comprised of two or more variables. The other is to create an array that has more than one dimension. The brief program below shows how to access a 2-dimensional array as if it had only a single dimension.

```
DEFINT A-Z

'----- pick an arbitrary group size, and number of groups (in this
'         case 100,000 elements)
GroupSize = 1000: NumGroups = 100

'----- dimension the array
REDIM Array(1 TO GroupSize, 1 TO NumGroups)

'----- pick an element number to assign (note use of a long integer) Element& =
50000

'----- calculate the first and second subscripts
First = ((Element& - 1) MOD GroupSize) + 1
Second = (Element& - 1) \ GroupSize + 1

'----- assign the appropriate array element
Array(First, Second) = 123

'----- show how to derive the original element based on First and
'         Second (CLNG is needed to prevent an Overflow error) CalcEl& = First +
(Second - 1) * CLNG(GroupSize)
```

/C

The /c (communications) option lets you specify the size of the receive buffer when writing programs that open the COM port. The value specified represents the total buffer size in bytes, and is shared when two ports are open at once. For example, if two ports are open and the total buffer size is 4096 bytes, then each port has 2048 bytes available for itself.

A receive buffer is needed when performing communications, and it accumulates the incoming characters as they are received. Each time a character is accepted by the serial port, it is placed into the receive buffer automatically. When your program subsequently uses INPUT or INPUT\$ or GET to read the data, it is actually reading the characters from the buffer and not from the hardware port. Without this buffering, your program would have to wait in a loop constantly looking for each character, which would preclude it from doing anything else!

Communications data is received in a continuous stream, and each byte must be processed before the next one arrives, otherwise the data will be lost. The communications port hardware generates an interrupt as each character is received, and the communications routines within BASIC act on that interrupt. The byte is retrieved from the hardware port using an assembly language IN instruction, which is equivalent to BASIC's INP function. This allows the characters to accumulate in the background, without any additional effort on your part.

As each byte is received it is placed into the buffer, and a pointer is updated showing the current ending address within the buffer. As your program reads those bytes, another pointer is updated to show the new starting address within the buffer. This type of buffer is called a *circular buffer*, because the starting and ending buffer addresses are constantly changing. That is, the buffer's end point wraps around to the beginning when it becomes full.

The receive buffer whose size is specified with `/c` is located in far memory. However, BASIC also maintains a second buffer in near memory, and its size is dictated by the optional `LEN=` argument used with the `OPEN` statement. Because near memory can be accessed more quickly than far memory, it is sensible for BASIC to copy a group of characters from the far receive buffer to the near buffer all at once, rather than individually each time you use `GET` or `INPUT$`.

When `/c` is not specified, the buffer size defaults to 512 bytes. This means that up to 512 characters can be received with no intervention on your part. If more than 512 bytes arrive and your program still hasn't removed them using `INPUT$` or `GET`, new characters that come later will be lost. It is also possible to stipulate hardware handshaking when you open the communications port. This means that the sender and receiver use physical control wires to indicate when the buffer is full, and when it is okay to resume transmitting.

In many programming situations, the 512 byte default will be more than adequate. However, if many characters are being received at a high baud rate (9600 or greater) and your program is unable to accept and process those characters quickly enough, you should consider using a larger buffer. Fortunately, the buffer is located in far memory, so increasing its size will not impinge on available string and data stored in `DGROUP`.

`/D`

The `/d` (debug) option switch is intended solely to help you find problems in a program while it is being developed. Because `/d` causes BC to generate additional code and thus bloat your executable program, it should be used only during development.

When `/d` is specified, four different types of tests are added to your program. The first is a call to a routine that checks if Ctrl-Break has been pressed. One call is added for every BASIC source statement, and each adds five bytes of code to your final executable program. The second addition is a one-byte assembly language `INTO` instruction following each integer and long integer math operation, to detect overflow errors.

The third is a call to a routine that calculates array element addresses, to ensure that the element number is in fact legal. Normally, element addresses are computed directly without checking the upper and lower bounds, unless you are using huge (greater than 64K) arrays. Without `/d`, it is therefore possible to corrupt memory by assigning an element that doesn't exist.

The final code addition implements GOSUB and RETURN statements using a library routine, rather than calling and returning from the target line directly. Normally, a GOSUB statement is translated into a three-byte assembly language *near call* instruction, and a RETURN is implemented using a one-byte *near return*. But when /d is used, the library routines ensure that each RETURN did in fact result from a corresponding GOSUB, to detect RETURN without GOSUB errors. This is accomplished by incrementing an internal variable each time GOSUB is used, and decrementing it at each RETURN. If that variable is decremented below 0 during a RETURN statement, then BASIC knows that there was no corresponding GOSUB. These library routines are added to your program only once by LINK, and comprise only a few bytes of code. However, a separate five-byte call is generated for each GOSUB and RETURN statement.

Many aspects of the /d option were described in detail in Chapters 1 and 4, and there is no need to repeat that information here. But it is important to remember that /d always makes your programs larger and run more slowly. Therefore, it should be avoided once a program is running correctly.

/E

The /e (error) option is necessary for any program that uses ON ERROR or RESUME with a line label or number. In most cases using /e adds little or no extra code to your final .EXE program, unless ON ERROR and RESUME are actually used, or unless you are using line numbers. For each line number, four bytes are added to remember the number itself as well as its position in the file (two bytes each). As with /d, every GOSUB and RETURN statement is implemented through a far call to a library routine, rather than by calling the target line directly. Without this added protection it would not be possible to trap "RETURN without GOSUB" errors correctly, or recover from them in an ON ERROR handler.

Also see the /x option which is needed when RESUME is used alone, or with a 0 or NEXT argument. The /x switch is closely related to /e, and is described separately below.

/Fpa and /Fpi (BASIC PDS and later)

When Microsoft introduced their BASIC compiler version 6.0, they included an alternate method for performing floating point math. This Floating Point Alternate library (hence the /fpa) offered a meaningful speed improvement over the IEEE standard, though at a cost of slightly reduced accuracy. This optional math library has been continued with BASIC 7 PDS, and is specified using the /fpa command switch.

By default, two parallel sets of floating point math routines are added to every program. When the program runs, code in BASIC's runtime startup module detects the presence of a math coprocessor chip, and selects which set of math routines will be used. The coprocessor version is called the Inline Library, and it merely serves as an interface to the 80x87 math coprocessor that does the real work in its hardware. (Note that inline is really a misnomer, because that term implies that the compiler generates coprocessor instructions directly. It doesn't.) The second version is called the Emulator Library, because it imitates the behavior of the coprocessor using assembly language subroutines.

Although the ability to take advantage of a coprocessor automatically is certainly beneficial, there are two problems with this dual approach: code size and execution speed. The coprocessor version is much smaller than the routines that perform the calculations manually, since it serves only as an interface to the coprocessor chip itself. When a coprocessor is in fact present, the entire emulator library is still loaded into memory. And when a coprocessor is not installed in the host PC, the library code to support it is still loaded. The real issue, however, is that each BASIC math operation requires additional time to route execution to the appropriate routines.

Since BC has no way to know if a coprocessor will be present when the program eventually runs, it cannot know which routine names to call. Therefore, BASIC uses a system of software interrupts that route execution to one library or the other. That is, instead of using, say, `CALL MultSingle`, it instead creates code such as `INT 39h`. The Interrupt 39h vector is set when the program starts to point to the correct library routine. Unfortunately, the extra level of indirection to first read the interrupt address and then call that address impacts the program's speed.

Recall that Chapter 1 explained how the library routines in a BRUN-style program modify the caller's code the first time they are invoked. The compiler creates code that uses an interrupt to access the library routines, and those routines actually rewrite that code to produce a direct call. Although this code modification increases the time needed to call a library routine initially, subsequent calls will be noticeably faster. BASIC statements executed many times within a FOR or DO loop will show the greatest improvement, but statements executed only once will be much slower than usual.

In a similar fashion, the coprocessor routines that are in BASIC's runtime library alter the caller's code, replacing the interrupt commands with equivalent coprocessor instructions. Each floating point interrupt that BC generates includes the necessary variable addresses and other arguments within the caller's code. These arguments are in the same format as a coprocessor instruction. The first time an interrupt is invoked, it subtracts the magic value `&H5C32` from the bytes that comprise the interrupt instruction, thus converting the instruction into a coprocessor command. This will be covered in Chapter 12.

Since the alternate floating point math routines do not use a coprocessor even if one is present, the interrupt method is not necessary. BC simply hardcodes the library subroutine names into the generated code, and the program is linked with the alternate math library. Besides the speed improvement achieved by avoiding the indirection of interrupts, the alternate math library is also inherently faster than the emulator library when a coprocessor is not present.

The `/fpi` switch tells BASIC to use its normal method of including both the coprocessor and emulator math libraries in the program, and determining which to use at runtime. (See the discussion of `/fpa` above.) Using `/fpi` is actually redundant and unnecessary, because this is the default that is used if no math option is specified.

`/Fs` (BASIC PDS only)

BASIC PDS offers an option to use far strings, and this is specified with the /fs (far strings) switch. Without /fs, all conventional (not fixed-length) string variables and string arrays are stored in the same 64K DGROUP memory that holds numeric variables, DATA items, file buffers, and static numeric and TYPE arrays. Using the /fs option tells BASIC to instead store strings and file buffers in a separate segment in far memory.

Although a program using far strings can subsequently hold more data, the capability comes at the expense of speed and code size. Obviously, more code is required to access strings that are stored in a separate data segment. Furthermore, the string descriptors are more complex than when near strings are used, and the code that acts on those descriptors requires more steps. Therefore, you should use /fs only when truly necessary, for example when BASIC reports an Out of string space error.

Far versus near strings were discussed in depth in Chapter 2, and you should refer to that chapter for additional information.

```
One very unfortunate limitation of VB/DOS is that only far strings are supported. The decision makers at Microsoft apparently decided it was too much work to also write a near-strings version of the forms library. So users of VB/DOS are stuck with the additional size and speed overhead of far strings, even for small programs that would have been better served with near strings.
```

/G2 (BASIC PDS and later)

The /g2 option tells BASIC to create code that takes advantage of an 80286 or later CPU. Each new generation of Intel microprocessors has offered additional instructions, as well as performance optimizations to the internal microcode that interprets and executes the original instructions. When an existing instruction is recoded and improved within the CPU, anyone who owns a PC using the newer CPU will benefit from the performance increase. For example, the original 8086/8088 had several instructions that performed poorly. These include Push and Pop, and Mul and Div. When Intel released the 80186, they rewrote the microcode that performs those instructions, increasing their speed noticeably. The 80286 is an offshoot of the 80186, and of course includes the same optimizations. The 80386 and 80486 offer even more improvements and additions to the original 8086 instruction set.

Besides the enhancements to existing instructions, newer CPU types also include additional instructions not present in the original 8086. For example, the 80286 offers the Enter and Leave commands, each of which can replace a lengthy sequence of instructions on the earlier microprocessors. Another useful enhancement offered in the 80286 is the ability to push numbers directly onto the stack. Where the 8086 can use only registers as arguments to Push, the instructions Push 1234 and Push Offset Variable are legal with 80186 and later CPUs. Likewise, the 80386 offers several new commands to directly perform long integer operations. For example, adding two long integer values using the 8086 instruction set requires a number of separate steps. The 80386 and later CPUs can do this using only one instruction.

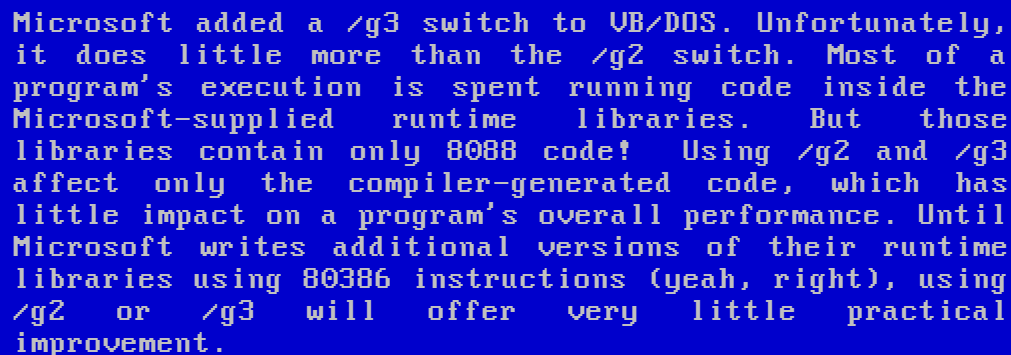
If you are absolutely certain that your program will be run only on PCs with an 80286 or later microprocessor, the /g2 option can provide a modest improvement in code size and performance. In particular, programs that use /g2 can save one byte each time a variable address is passed to a routine. When /g2 is not used, the command PRINT Work\$ results in the code shown below.

```
PRINT Work$
  Mov  AX,Offset Work$      'this requires 3 bytes
  Push AX                   'this requires 1 byte
  Call B$PESD              'a far call is 5 bytes
```

When /g2 is used, the address is pushed directly rather than first being loaded into AX, as shown following.

```
PRINT Work$
  Push Offset Work$        'this requires 3 bytes
  Call B$PESD              'this call is 5 bytes
```

With the rapid proliferation of 80386, 80486 and Pentium computers, Microsoft should certainly consider adding a /g3 switch. Taking advantage of 80386 instructions could provide substantially more improvement over 80286 instructions than the 80286 provides beyond the 8086.



Microsoft added a /g3 switch to VB/DOS. Unfortunately, it does little more than the /g2 switch. Most of a program's execution is spent running code inside the Microsoft-supplied runtime libraries. But those libraries contain only 8088 code! Using /g2 and /g3 affect only the compiler-generated code, which has little impact on a program's overall performance. Until Microsoft writes additional versions of their runtime libraries using 80386 instructions (yeah, right), using /g2 or /g3 will offer very little practical improvement.

/Ix (BASIC PDS and later)

Another important addition to BASIC 7 PDS is its integral ISAM data file handler. Microsoft's ISAM (Indexed Sequential Access Method) offers three key features: The first is indexing, which lets you search a data file very quickly. A simple sequential search reads each record from the disk in order until the desired information is found. That is, to find the record for customer David Eagle you would start at the beginning of the file, and read each record until you found the one containing that name. An index system, on the other hand, keeps as many names in memory as will fit, and searches memory instead of the disk. This is many time faster than reading the disk repeatedly. If Mr. Eagle is found in, say, the 1200th position, the index manager can go directly to the corresponding record on disk and return the data it contains.

The second ISAM feature is its ability to maintain the data file in sorted order. In most situations, records are stored in a data file in the order they were originally entered. For example, with a sales database, each time a customer purchases a product a new record is added holding the item and price for the item. When you subsequently step through the data file, the entries will most likely be ordered by the date and time they were entered. ISAM lets you access records in sorted order—for example, alphabetically by the customer's last name—regardless of the order in which the data was actually entered.

The last important ISAM feature is its ability to establish relationships between files, based on the information they contain. Many business applications require at least two data files: one to hold names and addresses of each customer which rarely changes, and another to hold the products or other items that are ordered periodically. It would be impractical and wasteful to duplicate the name and address information repeatedly in each product detail record. Instead, many database programs store a unique customer number in each record. Then, it is possible to determine which sales record goes with which customer based on the matching numbers in both files. A program that uses this technique is called a *relational database*.

To help the BASIC ISAM routines operate efficiently, you are required to provide some information when compiling your program. Each of the /i switches requires a letter indicating which option is being specified, and a numeric value. For each field in the file that requires fast (indexed) access, ISAM must reserve a block of memory for file buffers. This is the purpose of the /ii: switch. Notice that /ii: is needed only if more than 30 indexes will be active at one time.

The /ie: option tells ISAM how much EMS memory to reserve for buffers, and is specified in kilobytes. This allows other applications to use the remaining EMS for their own use.

The /ib: option switch tells ISAM how many 2K (2048-byte) *page buffers* to create in memory. In general, the more memory that is reserved for buffers, the faster the ISAM program can work. Of course, each buffer that you specify reduces the amount of memory that is available for other uses in your program.

An entire chapter in the BASIC PDS manual is devoted to explaining the ISAM file system, and there is little point in duplicating that information here. Please refer to your BASIC documentation for more examples and tutorial information on using ISAM. In particular, advice and formulas are given that show how to calculate the numeric values these options require.

In Chapter 6 I will cover file handling and indexing techniques in detail, with accompanying code examples showing how you can create your own indexing methods.

/Lp And /Lr (BASIC PDS only)

BASIC 7 PDS includes an option to write programs that operate under OS/2, as well as MS-DOS. Although OS/2 has yet to be accepted by most PC users, many programmers agree that it offers a number of interesting and powerful capabilities. By default, BC compiles a program for the operating

system that is currently running. If you are using DOS when the program is compiled and linked, the resultant program will also be for use with DOS. Similarly, if you are currently running OS/2, then the program will be compiled and linked for use with that operating system.

The `/lp` (protected) switch lets you override the assumption that BC makes, and tell it to create OS/2 instructions that will run in protected mode. The `/lr` (real) option tells BC that even though you are currently running under OS/2, the program will really be run with DOS. Again, these switches are needed only when you need to compile for the operating system that is not currently in use.

`/Mbf`

With the introduction of QuickBASIC 4.0, Microsoft standardized on the IEEE format for floating point data storage. Earlier versions of QuickBASIC and GW-BASIC used a faster, but non-standard proprietary numeric format that is incompatible with other compilers and languages. In many cases, the internal numeric format a compiler uses is of little consequence to the programmer. After all, the whole point of a high-level language is to shield the programmer from machine-specific details.

One important exception is when numeric data is stored in a disk file. While it is certainly possible to store numbers as a string of ASCII characters, this is not efficient. As I described in Chapter 2, converting between binary and decimal formats is time consuming, and also wastes disk space. Therefore, BASIC (and most other languages) write numeric data to a file using its native fixed-length format. That is, integers are stored in two bytes, and double-precision data in eight.

Although QuickBASIC 4 and later compilers use the IEEE format for numeric data storage, earlier version of the compiler do not. This means that values written to disk by programs compiled using earlier version of QuickBASIC or even GW-BASIC cannot be read correctly by programs built using the newer compilers. The `/mbf` option tells BASIC that it is to convert to the original Microsoft Binary Format (hence the MBF) prior to writing those values to disk. Likewise, floating point numbers read from disk will be converted from MBF to IEEE before being stored in memory.

```
Even when /mbf is used, all floating point numbers are
still stored in memory and manipulated using the IEEE
method. It is only when numbers are read from or
written to disk that a conversion between MBF and IEEE
format is performed.
```

Notice that current versions of Microsoft BASIC also include functions to convert between the MBF and IEEE formats manually. For example, the statement `Value# = CVDMBF(Fielded$)` converts the MBF-format number held in `Fielded$`, and assigns an IEEE-format result to `Value#`. When `/mbf` is used, however, you do not have to perform this conversion explicitly, and using `Value# = CVD(Fielded$)` provides the identical result.

Also see the data format discussion in Chapter 2, that compares the IEEE and MBF storage methods in detail.

/O

BASIC can create two fundamentally different types of .EXE programs: One type is a stand-alone program that is completely self-contained. The other type requires the presence of a special runtime .EXE library file when it runs, which contains the routines that handle all of BASIC's commands. By default, BASIC creates a program that requires the runtime .EXE library, which produces smaller program files. However, the runtime library is also needed, and is loaded along with the program into memory. The differences between the BRUN and BCOM programs were described in detail in Chapter 1.

The /o switch tells BASIC to create a stand-alone program that does not require the BRUN library to be present. Notice that when /o is used, the CHAIN command is treated as if you had used RUN, and COMMON variables may not be passed to a subsequently executed program.

/Ot (BASIC PDS and later)

Each time you invoke a BASIC subprogram, function, or DEF FN function, code BC adds to the subprogram or function creates a stack frame that remembers the caller's segment and address. Normally, Call and Return statements in assembly language are handled directly by the microprocessor. DEF FN functions and GOSUB statements are translated by the compiler into near calls, which means that the target address is located in the same segment. Invoking a formal function or subprogram is instead treated as a far call, to support multiple segments and thus larger programs. Therefore, a RETURN or EXIT DEF statement assumes that a single address word is on the stack, where EXIT SUB or EXIT FUNCTION expect both a segment and address to be present (two words).

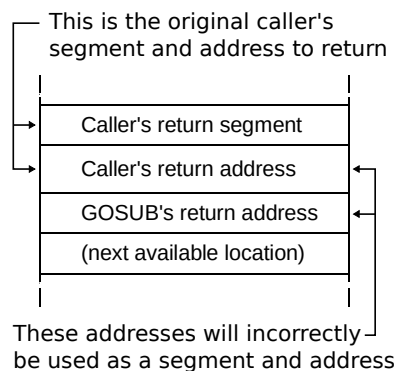


Figure 5-2: The stack frame within a procedure while a GOSUB is pending.

A problem can arise if you invoke a GOSUB routine within a SUB or FUNCTION procedure, and then attempt to exit the procedure from inside that subroutine with EXIT SUB or EXIT FUNCTION. If a GOSUB is active, EXIT SUB will incorrectly return to the segment and address that are currently on

the stack. Unfortunately, the address is that of the statement following the GOSUB, and the *segment* is in fact the address portion of the original caller's return location. This is shown in Figure 5-2.

To avoid this potential problem, the original caller's segment and address are saved when a subprogram or function is first invoked. The current stack pointer is also saved, so it can be restored to the correct value, no matter how deeply nested GOSUB calls may become. Then when the procedure is exited, another library routine is called that forces the originally saved segment and address to be on the stack in the correct position

Because this process reduces the speed of procedure calls and adds to the resultant code size, the /ot option was introduced with BASIC 7 PDS. Using /ot tells BASIC not to employ the larger and slower method, unless you are in fact using a GOSUB statement within a procedure. Since this optimization is disabled automatically anyway in that case, it is curious that Microsoft requires a switch at all. That is, BC should simply optimize procedure calls where it can, and use the older method only when it has to.

/R

The /r switch tells BASIC to store multi-dimensioned arrays in row, rather than column order. All arrays, regardless of their type, are stored in a contiguous block of memory. Even though string data can be scattered in different places, the table of descriptors that comprise a string array is contiguous. When you dimension an array using two or more subscripts, each group of rows and columns is placed immediately after the preceding one. By default, BASIC stores multi-dimensioned arrays in column order, as shown in Figure 5-3.

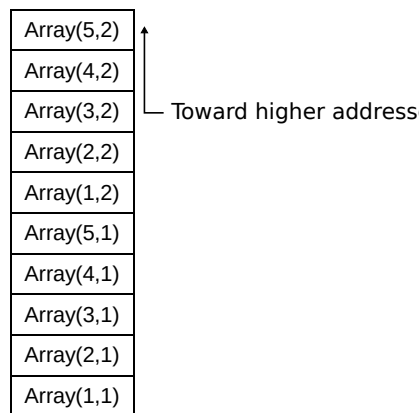


Figure 5-3: How BASIC stores a 2-dimensional array dimensioned created using DIM Array(1 TO 5, 1 TO 2).

As you can see, each of the elements in the first subscript are stored in successive memory locations, followed by each of the elements in the second subscript. In some situations it may be necessary to maintain arrays in row order, for example when interfacing with another language that expects array

data to be organized that way—notably FORTRAN. When an array is stored in row order, the elements are arranged such that Array(1, 1) is followed by Array(1, 2), which is then followed by Array(2, 1), Array(2, 2), Array(3, 1), and so forth.

Although many of the BC option switches described here are also available for use with the QB editing environment, /r is not one of them.

/S

The /s switch has been included with BASIC since the first BASCOM 1.0 compiler, and it remains perhaps the least understood of all the BC options. Using /s affects your programs in two ways. The first is partially described in the BASIC manuals, which is to tell BC not to combine like string constants as it compiles your program. As you learned in Chapter 2, BASIC makes available as much string memory as possible in your programs, by consolidating identical constant string data. For example, if you have the statement `PRINT "Insert disk in drive A"` seven times in your program, the message is stored only once, and used for each instance of `PRINT`.

In order to combine like data the BC compiler examines each string as it is encountered, and then searches its own memory to see if that string is already present. Having to store all of the strings your program uses just to check for duplicates impinges on BC's own working memory. At some point it will run out of memory, since it also has to remember variable and procedure names, line labels and their corresponding addresses, and so on. When this happens, BC has no recourse but to give up and display an "Out of memory" error message.

The /s switch is intended to overcome this problem, because it tells the compiler not to store your program's string constants. Instead of retaining the strings in memory for comparison, each is simply added to the object file as it is encountered. However, strings four characters long or shorter are always combined, since short strings are very common and doing that does not require much of BC's memory.

The second [undocumented] thing /s does is to add two short (eight bytes each) assembly language subroutines to the very beginning of your program. Two of the most common string operations are assignments and concatenations, which are handled by routines in the runtime library. Normally, a call to either of these routines generates thirteen bytes of code, including the statements that pass the appropriate string addresses.

The subroutines that /s adds are accessed using a near rather than a far call, and they receive the string addresses in CPU registers rather than through the stack. Therefore, they can be called using between three and nine bytes, depending on whether the necessary addresses are already in the correct registers at the time. The inevitable trade-off, however, is that calling one subroutine that in turn calls another reduces the speed of your programs slightly.

In many cases—especially when there are few or no duplicated string constants—using /s will reduce the size of your programs. This is contrary to the Microsoft documentation which implies that /s will make your programs larger because the duplicate strings are not combined. I would like to see

Microsoft include this second feature of /s as a separate option, perhaps using /ss (string subroutine) as a designator.

/T

The /t (terse) switch tells BC not to display its copyright notice or any warning (non-fatal) error messages. This option was not documented until BASIC PDS, even though it has been available since at least QuickBASIC 4.0. The only practical use I can see for /t is to reduce screen clutter, which is probably why QB and QBX use it when they shell to DOS to create an .EXE program.

/V and /W

Any programs that use event handling such as ON KEY, ON COM, ON PLAY, or the like—but not ON GOTO or ON GOSUB—require that you compile using either the /v or /w option switches. These options do similar things, adding extra code to call a central handler that determines if action is needed to process an event. However, the /v switch checks for events at every program statement while /w checks only at numbered or labelled lines.

In Chapter 1 I described how event handling works in BASIC, using polling rather than true interrupt handling. There you saw how a five-byte call is required each time BASIC needs to see if an event has occurred. Because of this added overhead, many programmers prefer to avoid BASIC's event trapping statements in favor of manually polling when needed. However, it is important to point out that by using line numbers and labels sparingly in conjunction with /w, you can reduce the amount of extra code BASIC creates thus controlling where such checking is performed.

/X

Like the /e switch, /x is used with ON ERROR and RESUME; however, /x increases substantially the size of your final .EXE program file. When RESUME, RESUME 0, or RESUME NEXT are used, BASIC needs a way to find where execution is to resume in your program. Unfortunately, this is not a simple task. Since a single BASIC source statement can create a long series of assembly language commands, there is no direct correlation between the two. When an error occurs and you use RESUME with no argument telling BASIC to execute the same statement again, it can't know directly how many bytes earlier that statement begins.

Therefore, when /x is specified, a numbered line marker is added in the object code to identify the start of every BASIC source statement. These markers comprise a linked list of statement addresses, and the RESUME statement walks through this list looking for the address that most closely precedes the offending BASIC statement. Because of the overhead to store these addresses—four bytes for each BASIC source statement—many professional programmers avoid using /x unless absolutely necessary. However, the table of addresses is stored within the code segment, and does not take away from DGROUP memory.

/Z (BASIC PDS and later)

The `/z` switch is meant to be used in conjunction with the Microsoft editor. This editor is included with BASIC PDS, and allows editing programs that are too large to be contained within the QB and QBX editing environments. When a program is compiled with `/z`, BASIC includes line number information in the object file. The Microsoft editor can then read these numbers after an unsuccessful compile, to help you identify which lines were in error. Because the addition of these line number identifiers increases a program's size, `/z` should be used only for debugging and not in a final production.

In general, the Microsoft editor has not been widely accepted by BASIC programmers, primarily because it is large, slow, and complicated to use. Microsoft also includes a newer editing environment called the Programmer's Workbench with BASIC PDS; however, that too is generally shunned by serious developers for the same reasons.

`/zd`

Like `/z`, the `/zd` switch tells BC to include line number information in the object file it creates. Unlike `/zi` which works with CodeView (see the `/zi` switch below), `/zd` is intended for use with the earlier SYMDEB debugger included with MASM 4.0. It is extremely unlikely that you will ever need to use `/zd` in your programming.

`/zi`

The `/zi` option is used when you will execute your program in the Microsoft CodeView debugger. CodeView was described in Chapter 4, and there is no reason to repeat that information here. Like `/z` and `/zd`, `/zi` tells BC to include additional information about your program in the object file. Besides indicating which assembler statements correspond to which BASIC source lines, `/zi` also adds variable and procedure names and addresses to the file. This allows CodeView to display meaningful names as you step through the assembly language compiled code, instead of addresses only.

In order to create a CodeView-compatible program, you must also link with the `/co` LINK option. All of the options that LINK supports are listed elsewhere in this chapter, along with a complete explanation of what each does.

Note that CodeView cannot process a BASIC source file that has been saved in the Fast Load format. This type of file is created by default in QuickBASIC, when you save a newly created program. Therefore, you must be sure to select the ASCII option button manually from the Save File dialog box. In fact, there are so many bugs in the Fast Load method that you should never use it. Problems range from QuickBASIC hanging during the loading process to completely destroying your source file!

If a program that has been saved as ASCII is accidentally damaged, it is at least possible to reconstruct it or salvage most of it using a DOS tool such as the Norton Utilities. But a Fast Load file is compressed and encrypted; if even a single byte is corrupted, QB will refuse to load it. Since a Fast Load file doesn't really load that much faster than a plain ASCII file anyway, there is no compelling reason to use it.

```
Rather than fix the Fast Load bug, which Microsoft
claims they cannot reproduce, beginning with PDS
version 7 BASIC now defaults to storing programs as
plain ASCII files.
```

Compiler Metacommands

There are a number of compiler metacommands that you can use to control how your program is formatted in the listing file that BC optionally creates. Although these list file formatting options have been available since the original IBM BASCOM 1.0 compiler, which Microsoft wrote, they are not documented in the current versions. As with '\$INCLUDE and '\$DYNAMIC and the other documented metacommands, each list formatting option is preceded by a REM or apostrophe, and a dollar sign. The requirement to embed metacommands within remarks was originally to let programs run under the GW-BASIC interpreter without error.

Each of the available options is listed below, along with an explanation and range of acceptable values. Many options require a numeric parameter as well; in those cases the number is preceded by a colon. For example, a line width of 132 columns is specified using '\$LINESIZE: 132. Other options such as '\$PAGE do not require or accept parameters. Notice that variables may not be used for metacommand parameters, and you must use numbers. CONST values are also not allowed.

Understand that the list file that BASIC creates is of dubious value, except when debugging a program to determine the address at which a runtime error occurred. While a list file could be considered as part of the documentation for a finished program, it conveys no useful information. These formatting options are given here in the interest of completeness, and because they are not documented anywhere else. In order to use any of these list options you must specify a list file name when compiling.

'\$LINESIZE

The '\$LINESIZE option lets you control the width of the list file, to prevent or force line wrapping at a given column. The default list width is 80 columns, and any text that would have extended beyond that is instead continued on the next line. Many printers offer a 132-column mode, which you can take advantage of by using '\$LINESIZE: 132. Of course, it's up to you to send the correct codes to your printer before printing such a wide listing. Note that the minimum legal width is 40, and the maximum is 255.

'\$LIST

The '\$LIST metacommand accepts either a minus (-) or plus (+) argument, to indicate that the listing should be turned off and on respectively. That is, using '\$LIST suspends the listing at that point in the program, and '\$LIST + turns it back on. This option is useful to reduce the size of the list file and to save paper when a listing is not needed for the entire program.

'\$PAGE

To afford control over the list file format, the '\$PAGE metacommand forces subsequent printing to begin on the next page. Typically '\$PAGE would be used prior to the start of a new section of code; for example, just before each new SUB or FUNCTION procedure. This tells BC to begin the procedure listing on a new page, to avoid starting it near the bottom of a page.

'PAGEIF

'\$PAGEIF is related to '\$PAGE, except it lets you specify that a new page is to be started only if a certain minimum number of lines remain on the current page. For example, '\$PAGEIF : 6 tells BC to advance to the next page only if there are six or less printable lines remaining.

'\$PAGESIZE

You can specify the length of each page with the '\$PAGESIZE metacommand, to override the 66-line default. This would be useful with laser printers, if you are using a small font that supports more than that many lines on each page. Notice that a 6-line bottom margin is added automatically, so specifying a page size of 66 results in only 60 actual lines of text on each page. The largest value that can be used with '\$PAGESIZE is 255, and the smallest is 15. To set the page length to 100 lines you would use '\$PAGESIZE : 100. There is no way to disable the page numbering altogether, and using values outside this range result in a warning error message.

'\$OCODE

Using '\$OCODE (object code) allows you to turn the assembly language source listing on or off, using "+" or "-" arguments. Normally, the /a switch is needed to tell BC to include the assembly language code in the list file. But you can optionally begin a listing at any place in the program source with '\$OCODE+, and then turn it off again using '\$OCODE -.

'\$SKIP

Like '\$PAGE and '\$PAGEIF, the '\$SKIP option lets you control the appearance of the source listing. '\$SKIP accepts a colon and a numeric argument that tells BC to print that many blank lines in the list file or skip to the end of the page, whichever comes first.

'\$TITLE and '\$SUBTITLE

By default, each page of the list file has a header that shows the current page number, and date and time of compilation. The '\$TITLE and '\$SUBTITLE metacommands let you also specify one or two additional strings, which are listed at the start of each page. Using '\$TITLE : 'My program' tells BASIC to print the text between the single quotes on the first line of each page. If a subtitle is also

specified, it will be printed on the second line. Note that the title will be printed on the first page of the list file only if the '\$TITLE metacommand is the very first line in the BASIC source file.

Linking

Once a program has been compiled to an object file, it must be linked with the routines in the BASIC library before it can be run. LINK combines one or more object files with routines in a library, and produces an executable program file having an .EXE extension. LINK is also used to create Quick Libraries for use in the QB editing environment, and that is discussed later in this chapter.

LINK can combine multiple BASIC object files, as well as object files created with other Microsoft-compatible languages. In the section that follows you will learn how the LINK command line is structured, what each parameter is for, and how the many available options may be used. Using the various LINK options can reduce the size of your programs, and help them run faster as well.

I should mention here it is imperative that you use the correct version of LINK. DOS comes with an old version of LINK.EXE that is not suitable for use with QuickBASIC or BASIC PDS. Therefore, you should always use the LINK.EXE program that came with your compiler. I also suggest that you remove or rename the copy of LINK that came with DOS if it is still on your hard disk. More than once I have seen programmers receive inexplicable LINK error messages because their PATH setting included the \DOS directory. In particular, many of the switches that current versions of LINK support cause an "Unrecognized option" message from older versions. If the correct version of LINK is not in the current directory, then DOS will use its PATH variable to see where else to look, possibly running an older version.

The LINK command line is structured as follows, using brackets to indicate optional information. The example below is intended to be entered all on one line:

```
link [/options] objfile [objfile] [libfile.lib], [exefile], [mapfile],  
[libfile] [libfile] [;]
```

As with the BC compiler, you may either enter all of the information on a single command, let LINK prompt you for the file names, or use a combination of the two. That is, you could enter LINK [filename] and let LINK prompt you for the remaining information. Default choices are displayed by LINK, and these are used if Enter alone is pressed. Typing a semicolon on a prompt line by itself or after a file name tells LINK to assume the default responses for the remaining fields. LINK also lets you use a *response file* to hold the file names and options. When there are dozens or even hundreds of files being specified, this is the only practical method. Response files are described later in this section

Also like BC, the separating commas are required as place holders when successive fields are omitted. For example, the command:

```
link program , , mapfile;
```


links PROGRAM.OBJ to produce PROGRAM.EXE, and creates a map file with the name MAPFILE.MAP. If the second comma had not been included, the output file would be named MAPFILE.EXE and a map file would not be written at all.

The first LINK argument is one or more optional command switches, which let you control some of the ways in which link works. For example, the /co switch tells LINK to add line number and other information needed when debugging the resultant EXE program with CodeView. Another option, /ex, tells LINK to reduce the size of the program using a primitive form of data compression. Each LINK option will be discussed in the section that follows.

The second argument is the name of the main program object module, which contains the code that will be executed when the program is run from the DOS command line. Many programs use only a single object file; however, in a multi-module program you must list the main module first. That is then followed by the other modules that contain additional subprograms and functions. Of course, you can precede any file name with a drive letter and/or directory name as necessary.

You may also specify that all of the object modules in an entire library be included in the executable program by entering the library name where the object name would be given. Since LINK assumes an .OBJ file extension, you must explicitly include the .LIB extension when linking an entire library. For example, this command creates a program named MAINPROG.EXE which is comprised of the code in MAINPROG.OBJ and all of the routines in SUBS.LIB:

```
link mainprog subs.lib;
```

Normally, a library is specified at the end of the LINK command line. However, in that case only the routines that are actually called will be added to the program. Placing a library name in the object name field tells LINK to add all of the routines it contains, regardless of whether they are actually needed. Normally you do not want LINK to include unused routines, but that is often needed when creating Quick Libraries which will be discussed in a moment.

Notice that when more than one object file is given, the first listed is the one that is run initially. Its name is also used for the executable file name if an output file name is not otherwise given. Like the BC compiler, LINK assumes that you are using certain file naming conventions but lets you override those assumptions with explicit extensions. I recommend that you use the standard extensions, and avoid any unnecessary heartache and confusion. In particular, using non-standard names is a poor practice when more than one programmer is working on a project. Also notice that either spaces or plus signs (+) may be used to separate each object and library file name. Which you use is a matter of personal preference.

The third LINK field is the optional executable output file name. If omitted, the program will use the base name of the first object file listed. Otherwise, the specified name will be used, and given an .EXE extension. Again, you can override the .EXE extension, but this is not recommended.

Following the output file name field is the map file entry. A map file contains information about the executable program, such as segment names and sizes, the size of the stack, and so forth. The /map

option, which is described later, tells LINK to include additional information in the map file. In general, a map file is not useful in high-level language programming.

One interesting LINK quirk is that it will create a map file if empty commas are used, but not if a semicolon is used prior to that field. You can specify the reserved DOS device name *nul* to avoid creating a map file. For example, this command links PROGRAM.OBJ to create PROGRAM.EXE, but not does not create the file PROGRAM.MAP:

```
link program, , nul, library;
```

I use a similar line in the batch files I use for compiling and linking, to avoid cluttering my hard disk with these useless files.

The last field specifies one or more libraries that hold additional routines needed for the program. In purely BASIC programming you do not need to specify a library name, because the compiler specifies a default library in the object file header. If you are linking with assembly or other language subroutines that are in a library, you would list the library names here. You can list any number of library names, and LINK will search each of them in turn looking for any routines it does not find in the object files.

The version of LINK that comes with BASIC 7 also accepts a definitions file as an optional last argument. But that is used only for OS/2 and Windows programming, and is not otherwise needed with BASIC.

Link Options

All of the available LINK options that are useful with BASIC running under DOS are shown following in alphabetical order. As with the switches supported by BC, each is specified on the LINK command line by preceding it forward slash (/). Many of the options may be abbreviated by entering just the first few letters of their name. For example, what I refer to as the */co* option is actually named */codeview*; however, the first two letters are sufficient for LINK to know what you mean.

Each option is described using only enough letters to understand the meaning of its name. You can see the full name for those options in the section headers below, or run LINK with the */help* switch. Any switch may be specified using only as many characters as needed to distinguish it from other options. That is, */e* is sufficient to indicate */exepack* because it is the only one that starts with that letter. But you must use at least the first three characters of the */nologo* switch, since */no* could mean either */nologo* or */nodefaultlibrary*. The details section for each option shows the minimum letters that are actually needed.

/BATCH

Using */ba* tells LINK that you are running it from a batch file, and that it is not to pause and prompt for library names it is unable to find. When */ba* is used and external routines are not found, a warning

message is issued rather than the usual prompt. The /ba option is not generally very useful—even if you are linking with a batch file—since it offers no chance to fix an incorrect file or directory name.

One interesting LINK quirk worth noting is when it is unable to find a library you must include a trailing backslash (\) after the path name when reentering it manually. If LINK displays the prompt "Enter new file spec:" and you type \pathname, you are telling LINK to use the library named PATHNAME.LIB and look for it in the root directory. What is really needed is to enter \pathname\, which tells it to look in that directory for the library. Furthermore, if you initially enter the directory incorrectly, you must then specify both the directory and library name. If you are not sure of the default library name it is often easier to simply press Ctrl-C and start again.

/CODEVIEW

The /co switch is necessary when preparing a program for debugging with CodeView. Because of the extra information that LINK adds to the resultant executable file, /co should be used only for debugging purposes. However, the added data is stored at the end of the file, and is not actually loaded into memory if the program is run from the DOS command line. The program will therefore have the same amount of memory available to it as if /co had not been used.

/EXEPACK

When /e is used, LINK compresses repeated character strings to reduce the executable file size. Because variables and static arrays are initialized to zero by the compiler, they are normally stored in the file as a group of CHR\$(0) zero bytes. The /e switch tells LINK to replace these groups of zero bytes with a group count. Then when the program is run, the first code that actually executes is the unpacking code that LINK adds to your program. This is not unlike the various self-extracting archive utilities that are available commercially and as shareware.

Notice that the compression algorithm LINK employs is not particularly sophisticated. For example, SLR System's OptLink is an alternate linker that reduces a program to a much smaller file size than Microsoft's LINK. PKWare and SEA Associates are two other third-party companies that produce utilities to create smaller executable files that unpack and run themselves automatically.

/FARCALLTRANSLATE

By default, all calls from BASIC to its runtime library routines are far calls, which means that both a segment and address are needed to specify the location of the routine being accessed. Assembly language and C routines meant to be used with BASIC are also designed as far calls, as are BASIC subprograms and functions. This affords the most flexibility, and also lets you create programs larger than could fit into a single 64K segment.

Within the BASIC runtime library there are both near and far calls to other library routines. Which is used depends on the routines involved, and how the various segments were named by the programmers at Microsoft. Because a far call is a five-byte instruction compared to a near call which is only three, a

near call requires less code and can execute more quickly. In many cases, separate code segments that are less than 64K in size can be combined by LINK to form a single segment. The routines in those segments could then be accessed using near calls. However, BASIC always generates far calls as it compiles your programs.

The /f option tells LINK to replace the far calls it encounters with near calls, if the target address is indeed close enough to be accessed with a near call. The improvement /f affords is further increased by also using the /packcode switch (see below). Although the far call is replaced with a near call, LINK can't actually reduce the size of the original instruction. Instead it inserts a Nop (no operation) assembly language command where part of the far call had been. But since a near call does not require segment relocation information in the .EXE file header, the file size may be reduced slightly. See the text that accompanies Figure 5-1 earlier in this chapter for an explanation of DOS' loading and relocation process.

There is one condition under which the /f option can cause your program to fail. The machine code for a far call is a byte with the value of &H9A, which is what LINK searches for as it converts the far calls to near ones. Most high-level languages store all data in a separate segment, which is ignored by LINK when servicing /f. BASIC, however, stores line label addresses in the program's code segment when ON GOTO and the other ON commands are used. If one of those addresses happens to be &H9A, then LINK may incorrectly change it. In my personal experience, I have never seen this happen. I recommend that you try /f in conjunction with /packc, and then test your program thoroughly. You could also examine any ON statements with CodeView if you are using them, to determine if an address happens to contain the byte &H9A.

/HELP

Starting LINK with the /he option tells it to display a list of all the command options it recognizes. This is useful both as a reminder, and to see what new features may have been added when upgrading to a newer compiler. In many cases, new compilers also include a new version of LINK.

/INFO

The /inf switch tells LINK to display a log of its activity on the screen as it processes your file. The name of each object file being linked is displayed, as are the routines being read from the libraries. It is extremely unlikely that you will find /inf very informative.

/LINENUM

If you have compiled with the /zd switch to create SYMDEB information, you will also need to specify the /li LINK switch. This tells LINK to read the line number information in the object file, and include it in the resultant executable program. SYMDEB is an awkward predecessor to CodeView that is also hard to use, and you are not likely to find /li useful.

/MAP

If you give a map file name when linking, LINK creates a file showing the names of every segment in your program. The /m switch tells LINK to also include all of the public symbol names. A *public symbol* is any procedure or data in the object file whose address must be determined by LINK. This information is not particularly useful in purely BASIC programming, but it is occasionally helpful when writing subroutines in assembly language. Segment naming and grouping will be discussed in Chapter 12.

/NODEFAULTLIB

When BC compiles your program, it places the default runtime library name into the created object file's header. This way you can simply run LINK, without having to specify the correct library manually. Before BASIC PDS there were only two runtime library names you had to deal with—QuickBASIC 4.5 uses BCOM45.LIB and BRUN45.LIB. But PDS version 7.1 comes with 16 different libraries, each intended for a different use.

For example, there are BRUN and BCOM libraries for every combination of near and far strings, IEEE and /fpa (alternate) math, and DOS and OS/2. That is, BRT71EFR.LIB stands for BASIC Runtime 7.1 Emulator Far strings Real mode. Likewise, BCL71ANP is for use with a BCOM stand-alone program using Alternate math and Near strings under OS/2 Protected mode.

Using /nod tells LINK not to use the library name embedded within the object file, which of course means that you must specify a library name manually. The /nod switch also accepts an optional colon and explicit library name to exclude. That is, /nod:libname means use all of the default libraries listed in the object file except libname.

In general, /nod is not useful with BASIC, unless you are using an alternate library such as Crescent Software's P.D.Q. Another possible use for /nod is if you have renamed the BASIC libraries.

/NOEXTDICT

As LINK combines the various object files that comprise your program with routines in the runtime library, it maintains a table of all the procedure and data names it encounters. Some of these names are in the object modules, such as the names of your BASIC subprograms and functions. Other procedure names are those in the library.

In some situations the same procedure or data name may be encountered more than once. For example, when you are linking with a stub file it will contain a routine with the same name as the one it replaces in BASIC's library. Usually, LINK will issue an error message when it finds more than one occurrence of a public name. If you use /noe (No Extended Dictionary) LINK knows to use the routine or data item it finds first, and not to issue an error message.

The /noe option should be used only when necessary, because it causes LINK to run more slowly. Linking with stub files is described separately later in this chapter.

/NOFARCALL

The `/nof` switch is usually not needed, since by default LINK does not translate far calls to near ones (see `/farcalltranslate` earlier in this section). But since you can set an environment variable to tell LINK to assume `/far` automatically, `/nof` would be used to override that behavior. Setting LINK options through the use of environment variables is described later in this chapter.

/NOLOGO

The `/nol` switch tells LINK not to display its copyright notice, and, like the `/t BC` switch may be used to minimize screen clutter.

/NOPACKCODE

As with the `/nof` switch, `/nop` is not necessary unless you have established `/packc` as the default behavior using an environment variable.

/OVERLAYINT

When you have written a program that uses overlays, BASIC uses an *overlay manager* to handle loading subprograms and functions in pieces as they are needed. Instead of simply calling the overlay manager directly, it uses an interrupt. This is similar to how the routines in a BRUN library are accessed.

BASIC by default uses Interrupt &H3F, which normally will not conflict with the interrupts used by DOS, the BIOS, or network adapter cards. If an interrupt conflict is occurring, you can use the `/o` switch to specify that a different interrupt number be used to invoke the overlay manager. This might be necessary in certain situations, perhaps when data acquisition or other special hardware is installed in the host PC.

/PACKCODE

The `/packc` switch is meant to be used with `/far`, and it combines multiple adjacent code segments into as few larger ones as possible. This enable the routines within those segments to call each other using near, rather than far calls. When combined with `/f`, `/packc` will make your programs slightly faster and possibly reduce their size.

/PAUSE

Using `/pau` tells link to pause after reading and processing the object and library files, but before writing the final executable program to disk. This is useful only when no hard drive is available, and all of the files will not fit onto a single floppy disk.

/QUICKLIB

The /q switch tells LINK that you are creating a Quick Library having a .QLB extension, rather than an .EXE program file. A Quick Library is a special file comprised of one or more object modules, that is loaded into the QB editing environment. Although BASIC can call routines written in non-BASIC languages, they must already be compiled or assembled. Since the BASIC editor can interpret only BASIC source code, Quick Libraries provide a way to access routines written in other languages. Creating and using Quick Libraries is discussed separately later in this chapter.

/SEGMENTS

The /seg: switch tells LINK to reserve memory for the specified number of segment names. When LINK begins, it allocates enough memory to hold 128 different segment names. This is not unlike using DIM in a BASIC program you might write to create a 128-element string array. If LINK encounters more than 128 names as it processes your program, it will terminate with a "Too many segments" error. When that happens, you must start LINK again using the /seg switch.

All of the segments in an object module that contain code or data are named according to a convention developed by Microsoft. Segment naming allows routines in separate files to ultimately reside in the same memory segment. Routines in the same segment can access each other using near calls instead of far calls, which results in smaller and faster programs. Also, all data in a BASIC program is combined into a single segment, even when the data is brought in from different modules. LINK knows which segments are to be combined by looking for identical names.

The routines in BASIC's runtime library use only a few different names, and it is not likely that you will need to use /seg in most situations. But when writing a large program that also incorporates many non-BASIC routines, it is possible to exceed the 128-name limit. It is also possible to exceed 128 segments when creating a very large Quick Library comprised of many individual routines.

The /seg switch requires a trailing colon, followed by a number that indicates the number of segment names to reserve memory for. For example, to specify 250 segments you would use this command line:

```
link /seg:250 program, , nul, library;
```

In most cases, there is no harm in specifying a number that is too large, unless that takes memory LINK needs for other purposes. Besides the segment names, LINK must also remember object file names, procedure names, data variables that are shared among programs, and so forth. But if LINK runs out of memory while it is processing your program, it simply creates a temporary work file to hold the additional information.

/STACK

The `/stack:` option lets you control the size of BASIC's stack. One situation where you might need to do this is if your program has deeply nested calls to non-static procedures. Likewise, calling a recursive subprogram or function that requires many levels of invocation will quickly consume stack space.

You can increase the stack size in a QuickBASIC program by using the `CLEAR` command where `stacksize` specifies the number of bytes needed:

```
CLEAR , , stacksize
```

However, `CLEAR` also clears all of your variables, closes all open files, and erases any arrays. Therefore, `CLEAR` is suitable only when used at the very beginning of a program. Unfortunately, this precludes you from using it in a chained-to program, since any variables being passed are destroyed. Using `/stack:` avoids this by letting you specify how much memory is to be set aside for the stack when you link the chained-to program.

The `/stack:` option accepts a numeric argument, and can be used to specify the stack size selectively for each program module. For example, `/stack:4096` specifies that a 4K block be set aside in `DGROUP` for use as a stack. Furthermore, you do not need to use the same value for each module. Since setting aside more stack memory than necessary impinges on available string space, you can override BASIC's default for only those modules that actually need it.

Note that this switch is not needed or recommended if you have BASIC PDS, since that version includes the `STACK` statement for this purpose.

Stub Files (PDS and later)

A stub file is an object module that contains an alternate version of a BASIC language statement. A stub file could also be an alternate library containing multiple object files. The primary purpose of a stub file is to let you replace one or more BASIC statements with an alternate version having reduced capability and hence smaller code. Some stub files completely remove a particular feature or language statement. Others offer increased functionality at the expense of additional code.

Several stub files are included with BASIC PDS, to reduce the size of your programs. For example, `NOCOM.OBJ` removes the routines that handle serial communications, replacing them with code that prints the message "Feature stubbed out" in case you attempt to open a communications port.

When BASIC compiles your program and sees a statement such as `OPEN Some$ FOR OUTPUT AS #1`, it has no way to know what the contents of `Some$` will be when the program runs. That is, `Some$` could hold a file name, a device name such as "CON" or "LPT1:", or a communications argument like "COM1:2400,N,8,1,RS,DS". Therefore, BASIC instructs `LINK` to include code to support all of those possibilities. It does this by placing all of the library routine names in the object file header. When the

program runs, the code that handles OPEN examines Some\$ and determines which routine to actually call.

Within BASIC's runtime library are a number of individual object modules, each of which contains code to handle one or more BASIC statements. In Chapter 1 you learned that how finely LINK can extract individual routines from BASIC's libraries depends on how the routines were combined in the original assembly language source files. In BASIC 7.1, using the SCREEN function in a program also causes LINK to add the routines that handle CSRLIN and POS(0), even if those statements are not used. This is because all three routines are in the same object module. The manner in which these routines are combined is called *granularity*, and a library's granularity dictates which routines can be replaced by a stub file. That is, a stub file that eliminated the code to support SCREEN would also remove CSRLIN and POS(0).

Some of the stub files included with BASIC 7 PDS are NOGRAPH.OBJ, NOLPT.OBJ, and SMALLERR.OBJ. NOGRAPH.OBJ removes all support for graphics, NOLPT.OBJ eliminates the code needed to send data to a printer, and SMALLERR.OBJ contains a small subset of the many runtime error messages that a BASIC program normally contains. Other stub files selectively eliminate VGA or CGA graphics support, and another, OVLDOS21.OBJ, adds the extra code necessary for the BASIC overlay manager to operate with DOS 2.1.

When linking with a stub file, it is essential that you use the /noe LINK switch, so LINK will not be confused by the presence of two routines with the same name. The general syntax for linking with a stub file is as follows:

```
link /noe basfile stubfile;
```

Of course, you could add other LINK options, such as /ex and /packc, and specify other object and library files that are needed as well.

You can also create your own BASIC stub files, perhaps to produce a demo version of a program that has all features except the ability to save data to disk. In order for this to work, you must organize your subprograms and functions such that all of the routines that are to be stubbed out are in separate source files, or combined together in one file.

In the example above, you would place the routines that save the data in a separate file. Then, simply create an empty subprogram that has the same name and the same number and type of parameters, and compile that separately. Finally, you would link the BASIC stub file with the rest of the program. Note that such a replacement file is not technically a stub, unless the BASIC routines being replaced have been compiled and placed into a library. But the idea is generally the same.

Quick Libraries

For many programmers, one of the most confusing aspects of Microsoft BASIC is creating and managing Quick Libraries. The concept is quite simple, however, and there are only a few rules you must follow.

The primary purpose of a Quick Library is to let you access non-BASIC procedures from within the BASIC editor. For example, BASIC comes with a Quick Library that contains the Interrupt routine, to let you call DOS and BIOS system services. A Quick Library can contain routines written in any language, including BASIC.

Although the BASIC editor provides a menu option to create a Quick Library, that will not be addressed here. Rather, I will show the steps necessary to invoke LINK manually from the DOS command line. There are several problems and limitations imposed by BASIC's automated menus, which can be overcome only by creating the library manually.

One limitation is that the automated method adds all of the programs currently loaded into memory into the Quick Library, including the main program. Unfortunately, only subprograms and functions should be included. Code in the main module will never be executed, and its presence merely wastes the memory it occupies. Another, more serious problem is there's no way to specify a /seg parameter, which is needed when many routines are to be included in the library.

```
Actually, you can set a DOS environment variable that tells LINK to default to a given number of segments. But that too has problems when using VB/DOS, because the VB/DOS editor specifies a /seg: value manually, and incorrectly. Unfortunately, LINK honors the value passed to it by VB/DOS, rather than the value you assigned to the environment variable.
```

Quick Libraries are built from one or more object files using LINK with the /q switch, and once created may not be altered. Unlike the LIB.EXE library manager that lets you add and remove object files from an existing .LIB library, there is no way to modify a Quick Library.

When LINK combines the various components of an executable file, it resolves the data and procedure addresses in each object module header. The header contains relocation information that shows the names of all external routines being called, as well as where in the object file the final address is to be placed. Since the address of an external routine is not known when the source file is compiled or assembled, the actual CALL instruction is left blank. This was described earlier in this chapter in the section An Overview of Compiling and Linking.

Resolving these data and procedure addresses is one of the jobs that LINK performs. Because the external names that had been in each object file are removed by LINK and replaced with numeric addresses, there is no way to reconstruct them later. Similarly, when LINK creates a Quick Library it resolves all incomplete addresses, and removes the information that shows where in the object module

they were located. Thus, it is impossible to extract an object module from a Quick Library, or to modify it by adding or removing modules.

Understand that the names of the procedures within the Quick Library are still present, so QuickBASIC can find them and know the addresses to call. But if a routine in a Quick Library in turn calls another routine in the library, the name of the called routine is lost.

Creating a Quick Library

Quick Libraries are created using the version of LINK that came with your compiler, and the general syntax is as follows:

```
link /q obj1 [obj2] [library.lib] , , nul , support;
```

The support library file shown above is included with BASIC, and its name will vary depending on your compiler version. The library that comes with QuickBASIC version 4.5 is named BQLB45.LIB; BASIC 7 instead includes QBXQLB.LIB for the same purpose. You must specify the appropriate support library name when creating a Quick Library.

Notice that LINK also lets you include all of the routines in one or more conventional (.LIB) libraries. Simply list the library names where the object file names would go. The .LIB extension must be given, because .OBJ is the default extension that LINK assumes. You can also combine object files and multiple libraries in the same Quick Library like this:

```
link /q obj1 obj2 lib1.lib lib2.lib , , nul , support;
```

Although Quick Libraries are necessary for accessing non-BASIC subroutines, you can include compiled BASIC object files. In general, I recommend against doing that; however, there are some advantages. One advantage is that a compiled subprogram or function will usually require less memory, because comments are not included in the compiled code and long variable names are replaced with equivalent 2-byte addresses. Another advantage is that compiled code in a Quick Library can be loaded very quickly, thus avoiding the loading and parsing process needed when BASIC source code is loaded.

But there are several disadvantages to storing BASIC procedures in a Quick Library. One problem is that you cannot trace into them to determine the cause of an error. Another is that all of the routines in a Quick Library must be loaded together. If the files are retained in their original BASIC source form, you can selectively load and unload them as necessary. The last disadvantage affects BASIC 7, and VB/DOS users, only.

The QBX and VB/DOS editors places certain subprogram and function procedures into expanded memory if any is available. Understand that all procedures are not placed there; only those whose BASIC source code size is between 1K and 16K. But Quick Libraries are always stored in conventional DOS memory. Therefore, more memory will be available to your programs if the procedures are still in source form, because they can be placed into EMS memory.

Note that when compiling BASIC PDS programs for placement in a Quick Library, it is essential that you compile using the /fs (far strings) option. Near strings are not supported within the QBX editor, and failing to use /fs will cause your program to fail spectacularly.

Response Files

A response file contains information that LINK requires, and it can completely or partially replace the commands that would normally be given from the DOS command line. The most common use for a LINK response file is to specify a large number of object files. If you are creating a Quick Library that contains dozens or even hundreds of separate object files, it is far easier to maintain the names in a file than to enter them each time manually.

To tell LINK that it is to read its input from a response file enter an at sign (@) followed by the response file name, as shown below.

```
link /q @quicklib.rsp
```

Since the /q switch was already given, the response file need only contain the remaining information. A typical response is shown in the listing below:

```
object1 +
object2 +
object3 +
object4 +
object5
qlbname
nul
support
```

Even though this example lists only five object files, there could be as many as necessary. Each object file name except the last one is followed by a plus sign (+), so LINK will know that another object file name input line follows. The qlbname line indicates the output file name. If it is omitted and replaced with a blank line, the library will assume the name of the first object file but with a .QLB extension. In this case, the name would be OBJECT1.QLB. The nul entry could also be replaced with a blank line, in which case LINK would create a map file named OBJECT1.MAP. As shown in the earlier examples, the support library will actually be named BQLB45 or QBXQLB, depending on which version of BASIC you are using.

LINK recognizes several variations on the structure of a response file. For example, several object names could be placed on each line, up to the 126-character line length limit imposed by DOS. That is, you could have a response file like this:

```
object1 object2 object3 +
object4 object5 object6 +
...
```

I have found that placing only one name on each line makes it easier to maintain a large response file. That also lends itself to keeping the names in alphabetical order.

You may also place the various option switches in a response file, by listing them on the first line with the object files:

```
/ex /seg:250 object1 +  
object2 +  
...
```

Response files can be used for conventional linking, and not just for creating Quick Libraries. This is useful when you are developing a very large project comprised of many different modules. Regardless of what you are linking, however, understanding how response files are used is a valuable skill.

Linking With Batch Files

Because so many options are needed to fully control the compiling and linking process, many programmers use a batch file to create their programs. The C.BAT batch file below compiles and links a single BASIC program module, and exploits DOS' replaceable batch parameter feature.

```
bc /o /s /t %1;  
link /e /packc /far /seg:250 %1, , nul, mylib;
```

Like many programs, a batch file can also accept command line arguments. The first argument is known within the batch file as %1, the second is %2, and so forth, up to the ninth parameter. Therefore, when this file is started using this command:

```
c myprog
```

the compiler is actually invoked with the command

```
bc /o /s /t myprog;
```

The second line becomes

```
link /e /far /packc /seg:250 myprog, , nul, mylib;
```

That is, every occurrence of the replaceable parameter %1 is replaced by the first (and in this case only) argument: myprog.

I often create a separate batch file for each new project I begin, to avoid having to type even the file name. I generally use the name C.BAT because its purpose is obvious, and it requires typing only one letter! Once the project is complete, I rename the batch file to have the same first name as the main BASIC program. This lets me see exactly how the program was created if I have to come back to it again months later. An example of a batch file that compiles and links three BASIC source files is shown below.

```
bc /o /s /t mainprog;  
bc /o /s /t module1;  
bc /o /s /t module2;  
link /e /packc /far mainprog module1 module2, , nul, mylib;
```

Of course, you'd use the compiler and link switches that are appropriate to your particular project. You could also specify a LINK response file within a batch file. In the example above you would replace the last line with a command such as this:

```
link @mainprog.rsp;
```

Linking With Overlays (PDS and VB/DOS Pro Only)

At one time or another, most programmers face the problem of having an executable program become too large to fit into memory when run. With QuickBASIC your only recourse is to divide the program into separate .EXE files, and use CHAIN to go back and forth between them. This method requires a lot of planning, and doesn't lend itself to structured programming methods. Each program is a stand-alone main module, rather than a subprogram or function.

Worse, chaining often requires the same subroutine code to be duplicated in each program, since only one program can be loaded into memory at a time. If both PROGRAM1.EXE and PROGRAM2.EXE make calls to the same subprogram, that subprogram will have to be added to each program. Obviously, this wastes disk space. BASIC 6.0 included the BUILDRTM program to create custom runtime program files that combines common subroutine code with the BASIC runtime library. But that program is complicated to use and often buggy in operation.

Therefore, one of the most useful features introduced with BASIC 7 is support for program overlays. An *overlay* is a module that contains one or more subprograms or functions that is loaded into memory only when needed. All overlaid modules are contained in a single .EXE file along with the main program, as opposed to the separate files needed when programs use CHAIN. The loading and unloading of modules is handled for you automatically by the overlay manager contained in the BASIC runtime library.

Consider, as an example, a large accounting program comprised of three modules. The main module would consist of a menu that controls the remaining modules, and perhaps also contains some ancillary subprograms and functions. The second module would handle data entry, and the third would print all of the reports. In this case, the data entry and reporting modules are not both required at the same time; only the module currently selected from the menu is necessary. Therefore, you would link those modules as overlays, and let BASIC's overlay manager load and unload them automatically when they are called.

The overall structure of an overlaid program is shown in Figure 5-4.

```

*** MAINPROG.BAS
CALL Menu(Choice)
IF Choice = 1 THEN
    CALL EnterData
ELSEIF Choice = 2 THEN
    CALL DoReports
END IF

SUB Menu(Choice)
    ...
    CALL GetChoice(Choice)
    ...
END SUB

SUB GetChoice(ChoiceNum)
    ...
    ...
END SUB

*** ENTERDAT.BAS
SUB EnterData
    ...
    CALL GetChoice(Choice)
    ...
END SUB

*** REPORTS.BAS
SUB DoReports
    PRINT "Which report? ";
    CALL GetChoice(Choice)
    ...
    ...
END SUB

```

Figure 5-4: The structure of a program that uses overlays.

Here, the main program is loaded into memory when the program is first run. Since the main program also contains the Menu and GetChoice subprograms, they too are initially loaded into memory. Understand that the main program is always present in memory, and only the overlaid modules are swapped in and out. Thus, EnterData and DoReports can both freely call the GetChoice subprogram which is always in memory, without incurring any delay to load it into memory from disk.

If the host computer has expanded memory, BASIC will use that to hold the overlaid modules. Since EMS can be accessed much more quickly than a disk, this reduces the load time to virtually instantaneous. You should be aware, however, that BASIC PDS contains a bug in the EMS portion of

its overlay manager. If EMS is present but less than 64K is available, your program will terminate with the error message "Insufficient EMS to load overlay."

If no expanded memory is available, BASIC simply reads the overlaid modules from the original disk file each time they are called. It should also use the disk if it determines that there isn't enough EMS to handle the overlay requirements, but it doesn't. Therefore, it is up to your users to determine how much expanded memory is present, and disable the EMS driver in their PC if there isn't at least 64K.

To specify that a module is to be overlaid, simply surround its name with parentheses when linking. Using the earlier example shown in Figure 5-4, you would link MAINPROG.OBJ with ENTERDAT.OBJ and REPORTS.OBJ as follows:

```
link mainprog (enterdat) (reports);
```

Of course, you may include any link switches that are needed, and also include any non-overlaid object files. Any object file names that are not surrounded by parentheses will be kept in memory at all times. Therefore, you should organize your programs such that subprograms and functions that are common to the entire application are always loaded. Otherwise, the program could become very slow if those procedures are swapped in and out of memory each time they are called.

Other LINK Details

The BASIC PDS documentation lists no less than 143 different LINK error messages, and at one time or another you are bound to see at least some of those. LINK errors are divided into two general categories: warning errors and fatal errors. Warning errors can sometimes be ignored. For example, failing to use the /noe switch when linking with a stub file produces the message "Symbol multiply defined", because LINK encountered the same procedure name in the stub file and in the runtime library. In this case LINK simply uses the first procedure it encountered. In general, however, you should not run a program whose linking resulted in any error messages.

Fatal errors are exactly that—an indication that LINK was unable to create the program successfully. Even if an .EXE file is produced, running it is almost certain to cause your PC to lock up. One example of a fatal error is "Unresolved external." This means that your program made a call to a procedure, but LINK wasn't able to find its name in the list of object and library files you gave it. Another fatal error is "Too many segments." You might think that LINK would be smart enough to finish reading the files, count the number of segment names it needs, and then restart itself again reserving enough memory. Unfortunately, it isn't.

Regardless of the type of error messages you receive, it is impossible to read all of them if there are so many that they scroll off the screen. Although you can press Ctrl-P to tell DOS to echo the messages to your printer, there is an even better method. You can use the DOS redirection feature to send the message to a disk file. This lets you load the file into a text editor for later perusal. To send all of LINK's output to a file, simply use the greater than symbol ">" and specify a file name as follows:


```
link [/options] [object files]; > error.log
```

Instead of displaying the messages on the screen, DOS intercepts and routes them to the ERROR.LOG file. It is important to understand that this is a DOS issue, and has nothing to do with LINK. Therefore, you can use this same general technique to redirect the output of most programs to a file. Note that using redirection causes all of the program's output to go to the file, not just the error messages. Therefore, nothing will appear to happen on the screen, since the copyright and sign-on notices are also redirected.

Another LINK detail you should be aware of is that numeric arguments may be given in either decimal or hexadecimal form. Any LINK option that expects a number—for example, the /seg: switch—may be given as a Hexadecimal value by preceding the digits with 0x. That is, /seg:0x100 is equivalent to /seg:256. The use of 0x is a C notation convention, and the "x" character is used because it sounds like "hex".

Finally, if you are using QuickBASIC 4.0 there is a nasty bug you should be aware of. All versions of QuickBASIC let you create an executable program from within the editing environment. And if a Quick Library is currently loaded, QB knows to link your program with a parallel .LIB library having the same name. But instead of specifying that library in the proper LINK field, QB 4.0 puts its name in the object file position. This causes LINK to add every routine in the library to your program, rather than only those routines that are actually called. There is no way to avoid this bug, and QB 4.0 users must compile and link manually from DOS.

Maintaining Libraries

As you already know, multiple object files may be stored in a single library. A library has a .LIB extension, and LINK can extract from it only those object modules actually needed as it creates an executable file. All current versions of Microsoft compiled BASIC include the LIB.EXE program, which lets you manage a library file. With LIB.EXE you can add and remove objects, extract a copy of a single object without actually deleting it from the library, and create a cross-referenced list of all the procedures contained therein.

It is important to understand that a .LIB library is very different from a Quick Library. A .LIB library is simply a collection of individual object files, with a header portion that tells which objects are present, and where in the library they are located. A Quick Library, on the other hand, contains the raw code and data only. The routines in a Quick Library do not contain any of the relocation and address information that was present in the original object module.

The runtime libraries that Microsoft includes with BASIC are .LIB libraries, as are third-party support libraries you might purchase. You can also create your own libraries from both compiled BASIC code and assembly language subroutines. The primary purpose of using a library is to avoid having to list every object file needed manually. Another important use is to let LINK add only those routines actually necessary to your final .EXE program.

Like BC and LINK, you can invoke LIB giving all of the necessary parameters on a single command line, or wait for it to prompt you for the information. LIB can also read file names and options from a response file, which avoids having to enter many object names manually. A LIB response file is similar, but not identical to, a LINK response file. Using LIB response files will be described later in this section.

The general syntax of the LIB command line is shown below, with brackets indicating optional information:

```
lib [/options] libname [commands] , [listfile] , [newlib] [;]
```

After any optional switches, the first parameter is the name of the library being manipulated, and that is followed by one or more commands that tell LIB what you want to do. A list file can also be created, and it contains the names of every object file in the library along with the procedure names each object contains. The last argument indicates an optional new library; if present LIB will leave the original library intact, and copy it to a new one applying the changes you have asked for.

There are three commands that can be used with LIB, and each is represented using a punctuation character. However, LIB lets you combine some of these commands, for a total of five separate actions. This is shown in Table 5-1.

Command	Action
+	Adds an object module for entire library
-	Removes an object module from the library
*	Extracts a copy of an object module
- +	Replaces an object module with a new one
-*	Extracts and then removes an object module

Table 5-1: The LIB commands for managing libraries.

To add the file NEWOBJ.OBJ to the existing library MYLIB.LIB you would use the plus sign "+" as follows:

```
lib mylib +newobj;
```

And to update the library using a newer version of an object already present in the library you would instead use this:

```
lib mylib -+d:\newstuff\anyobj;
```

As you can see, the combination operators use a sensible syntax. Here, you are instructing LIB to first remove ANYOBJ.OBJ from MYLIB.LIB, and then add a newer version in its place. A drive and directory are given just to show that it is possible, and how that would be specified.

To extract a copy of an object file from a library, use the asterisk "*" command. Again, you can specify a directory in which the extracted file is to be placed, as follows:

```
lib mylib *\objdir\thisobj;
```

You should understand that LIB never actually modifies an existing library. Rather, it first renames the original library to have a .BAK extension, and then creates and modifies a new file using the original name. It is up to you to delete the backup copy once you are certain that the new library is correct.

But this backup is made only if you do not specify a new output library name—NEWLIB in the earlier syntax example.

If the named library does not exist, LIB asks if you want to create it. This gives you a chance to abort the process if you accidentally typed the wrong name. If you really do want to create a new library, simply answer Y (Yes) at the prompt. Of course, the only thing you can do to a non-existent library is add new objects to it with the plus "+" command.

One important LIB feature is its ability to create a list file showing what routines are present in the library. This is particularly valuable if you are managing a library you did not create, such as a library purchased from a third-party vendor. Many vendors use the same name for the object file as the routine it contains when possible, but there are exceptions. For example, an object file name is limited to eight characters, even though procedure names can be as long as 40. If you want to know which object file contains the procedure ReadDirectories, you will need to create a list file. Also, one object file can hold multiple procedures, and it is not always obvious which procedure is in which file. Individual procedures cannot necessarily be extracted from a library—only entire object files.

To create a library list file you will run LIB giving the name of the library, as well as the name of a list file to create. The example below creates a list file named MYLIST.LST for the library named MYLIB.LIB:

```
lib mylib , mylist.lst;
```

The list file that is created contains two cross-referenced tables; one shows each object name and the procedures it contains, and the other shows the procedure names and which object they are in. A typical list file is shown in the Figure 5-5, using the QB.LIB file that comes with QuickBASIC 4.5 as an example.

ABSOLUTE	absolute	INT86OLD	int86old
INT86XOLD	int86old	INTERRUPT	intrpt
INTERRUPTX	intrpt		
absolute	Offset: 00000010H	Code and data size: cH	
ABSOLUTE			
intrpt	Offset: 000000e0H	Code and data size: 107H	
INTERRUPT	INTERRUPTX		
int86old	Offset: 000002a0H	Code and data size: 11eH	
INT86OLD	INT86XOLD		

Figure 5-5: The format of a LIB list file.

In this list file, each object module contains only one procedure. The first section shows each procedure name in upper case, followed by the object name in lower case. The second section shows each object file name, its offset within the library and size in bytes, and the routine names within that object file.

Just for fun, you should create a list file from one of the libraries that came with your compiler. Besides showing how a large listing is structured, you will also be able to see which statements are combined with others in the same object file. Thus, you can determine the granularity of these libraries. In many cases the names of the procedures are similar to the corresponding BASIC keywords.

For example, if you create a list file for the BCOM45.LIB library that comes with QuickBASIC 4.5, you will see an object file named STRFCN.OBJ (string function) that contains the procedures B\$FASC, B\$FLEN, B\$FMID, B\$INS2, B\$INS3, B\$LCAS, B\$LEFT, and several other string functions. Most of the library routines start with the characters B\$, which ensures that the names will not conflict with procedure names you are using. A dollar sign is illegal in a BASIC procedure name. Other procedures and data items use an embedded underscore "_" which is also illegal in BASIC.

FASC stands for Function ASC, FLEN is for Function LEN, and so forth. INS2 and INS3 contain the code to handle BASIC's INSTR function, with the first being the two-argument version and the second the three-argument version. That is, using INSTR(Work\$, Substring\$) calls B\$INS2, and INSTR(Start, Work\$, Substring\$) instead calls B\$INS3. As you can see, most of the internal procedure names are sensible, albeit somewhat abbreviated.

LIB Options

Many LIB options are frankly not that useful to purely BASIC programming. However, I will list them here in the interest of completeness. Note that none of these option switches are available in versions of LIB prior to the one that comes with BASIC 7.0.

/HELP

As with the LINK switch of the same name, using /help (or /?) tells LIB to display its command syntax, and a list of all the available options.

/I

Using /i means that LIB should ignore capitalization when searching the library for procedure names. This is the default for LIB, and is not necessary unless you are manipulating an existing library that was created with /noi (see below).

/NOE

The /noe option has a similar meaning as its LINK counterpart, and should be used if LIB reports an Out of memory error. Creating an extended dictionary requires memory, and using /noe will avoid that.

/NOI

The /noi switch tells LIB not to ignore capitalization, and it should not be used with BASIC programs.

/NOLOGO

Like the LINK option, /nologo reduces screen clutter by eliminating the sign-on logo and copyright display.

/PA

The /pa: option lets you change the default library page size of 16 bytes. Larger values waste memory, because each object file will always occupy the next higher multiple number of bytes. For example, with a page size of 200 bytes, a 50 byte object file will require an entire 200-byte page. Since a library can hold no more than 65,536 pages, a larger page size is useful only when you need to create a library larger than 1 megabyte. The /pa: switch requires a colon, followed by an integer value between 16 and 32768. For example, using /pa:256 sets a page size of 256 bytes.

Using Response Files With LIB.EXE

A LIB response file is similar to a LINK response file, in that it lets you specify a large number of operations by entering them on separate lines of a text file. The syntax is similar to a LINK response file, but it is not identical. Since the plus sign continuation character that LINK uses serves as a command character to LIB, an ampersand (&) is used instead. A typical LIB response file is shown below.

```
+ object1 &  
+ \subdir\object2 &  
+ c:\subdir2\object3 &
```

```
+ object4 ;
```

As with LINK, you will use an at sign "@" to tell LIB to look in the file for its input, as opposed to reading the names from the command line:

```
lib @filename.rsp
```

Useful BC, LINK, and LIB Environment Parameters

Most programmers are familiar with the DOS environment as a way to establish PATH and PROMPT variables. The PATH environment variable tells DOS where to search for executable program files it doesn't find in the current directory. The PROMPT variable specifies a new prompt that DOS displays at the command line. For example, many people use the command `SET PROMPT=PG` to show the current drive and directory. However, the DOS environment can be used to hold other, more general information as well.

The environment is simply an area of memory that DOS maintains to hold variables you have assigned. Some of these variables are used by DOS, such as the PATH and PROMPT settings. Other variables may be defined by you or your programs, to hold any type of information. For example, you could enter `SET USERNAME=TAMI` in the AUTOEXEC.BAT file, and a program could read that to know the name of the person who is using it. The contents of this variable (TAMI) could then be used as a file or directory name, or for any other purpose.

LINK looks at the DOS environment to see if you have specified LINK= or LIB= or TMP= variables. The first is used to specify default option switches. For example, if you set `LINK=/SEG:450` from the DOS command line or a batch file, you do not need to use that option each time LINK is run. Multiple options may be included in a single SET statement, by listing each in succession. The command `SET LINK=/NOE/NOD/EX` establishes those three options shown as the default. Additional separating spaces may also be included; however, that is unnecessary and wastes environment memory.

Likewise, setting `LIB=D:\LIBDIR\` tells LINK to look in the LIBDIR directory of drive D: for any libraries it cannot find in the current directory. In this case, LIB= acts as a sort of PATH command. Like PATH, the LIB= variable accepts multiple path names with or without drive letters, and each is separated by a semicolon. The command `SET LIB=C:\LIBS\;D:\WORKDIR\` sets a library path to both C:\LIBS and D:\WORKDIR, and you could add even more directories if needed.

To remove an environment variable simply assign it to a null value; in this case you would use `SET LIB=`.

The TMP= variable also specifies a path that tells LINK where to write any temporary files. When a very large program or Quick Library is being created, it is possible for LINK to run out of memory. Rather than abort with an error message, LINK will open a temporary disk file and spool the excess

data to that file. If no `TMP=` variable has been defined, that file is created in the current directory. However, if you have a RAM disk you can specify that as the `TMP` parameter, to speed up the linking process. For example, `SET TMP=F:\` establishes the root directory of drive F as the temporary directory.

The `INCLUDE=` variable is recognized by both `BC` and `MASM` (the Microsoft Macro Assembler program), to specify where they should look for Include files. In my own programming, I prefer to give an explicit directory name as part of the `$INCLUDE` metacommand. This avoids unpleasant surprises when an obsolete version of a file is accidentally included. But you may also store all `$INCLUDE` files in a single directory, and then set the `INCLUDE` variable to show where that directory is. Like `LIB` and `PATH`, the `INCLUDE` variable accepts one or more directory names separated by semicolons.

Summary

In this chapter you have learned about compiling and linking manually from the DOS command line, to avoid the limitations imposed by the automated menus in the `BASIC` editor. You have also learned how to create and maintain both Quick Libraries and conventional `.LIB` libraries. Besides accepting information you enter at the DOS command line, `LINK` and `LIB` can also process instructions and file names contained in a response file.

All of the commands and option switches available with `BC`, `LINK`, and `LIB` were described in detail, along with a listing of the undocumented `BC` metacommands for controlling the format of a compiler list file. Library list files were also discussed, and a sample printout was given showing how `LIB` shows all the procedure and object names in a library cross-referenced alphabetically.

The discussion about stub files explained what they are and how to use them, to reduce the size of your programs. Overlays were also covered, accompanied by some reasons you will find them useful along with specific linking instructions.

Finally, I explained some of the details of the linking process. Information in each object file header tells `LINK` the names of external procedures being called, and where in the object file the incomplete addresses are located. Besides the segment and address fix-ups that `LINK` performs, `DOS` also makes some last-minute patches to your program as it is loaded into memory.

In the next chapter I will cover file handling in similar detail, explaining how files are manipulated at a low level, and also offering numerous tips for achieving high performance and small program size.

6

File and Device Handling

At some point, all but the most trivial computer programs will need to store and retrieve data using a disk file. Data files are used for two primary purposes: to hold information when there is more than can fit into the computer's memory all at once, and to provide a permanent, non-volatile means of storage. Files are also used to allow data from one computer to be used on another. Such data sharing can be as simple as a *sneaker net* system, whereby a floppy disk is manually carried from one PC to another, or as complex as a multi-user network where disk data can be accessed simultaneously by several users.

Although there are two fundamentally different types of disk drives, floppy and fixed—not counting CD-ROM drives which are removable—they are accessed identically using the same BASIC statements. BASIC's file commands may also be used to communicate with devices such as a printer or modem, and even the screen and keyboard. There are many ways to manipulate files and devices, and some are substantially faster than others. By understanding fully how BASIC interacts with DOS, file access in your programs can often be sped up by a factor of five or even more.

In this chapter I will address the fundamental aspects of file and device handling, and provide specific examples of how to achieve the highest performance possible. I will begin with an overview of how DOS organizes information on a disk, and then continue with practical examples. Unlike earlier chapters in which only short program fragments were shown, several complete programs and subprograms will be presented to illustrate the most important of these techniques in context. I will also describe the underlying theory of how disks are organized, and explain why this is important for the BASIC programmer to know.

In Chapter 7 the subject of files will be continued; there you will learn how to write programs for use with a network, and also how relational databases are constructed. In particular, coverage of these two very important subjects is severely lacking in the documentation that comes with Microsoft BASIC. As personal computers continue to permeate the office environment, networks and databases are becoming ever more common. Many programmers find themselves in the awkward position of having to write programs that run on a network, but with no adequate source of information.

Disk File Fundamentals

All disks used with MS-DOS are organized into groups of bytes called *sectors*, and these sectors are further combined into *clusters*. DOS keeps track of every file on a disk, but with this organization DOS needs to remember only the cluster number at which each file begins. The minimum amount of disk space that is allocated by DOS is one cluster. Therefore, if you create a very small file, say, ten bytes, an entire cluster is allocated to that file, and then marked as unavailable for other use.

In most cases, each disk sector holds 512 bytes; however, one exception is when you use a RAM disk to simulate a disk drive in memory. Many RAM disk programs lets you specify a smaller sector size, to minimize waste when there are many small files. The number of sectors that are stored in each cluster depends on the type of disk and its size. For example, a 360K floppy disk stores two sectors in each cluster, and a 32 MB hard disk formatted using DOS 3.3 stores four sectors in each cluster. Therefore, the minimum unit of storage allocation for these disks is 1K (1024 bytes), and 2K (2048 bytes) respectively. DOS 2.x offers less room to store cluster numbers, and must combine more sectors into each cluster. A 20MB hard disk formatted with DOS 2.1 allocates 8K for even a one-line batch file!

As files are created and appended, DOS allocates new space to hold the file contents. By allocating disk space in units, DOS is also able to minimize disk fragmentation. As you learned in Chapter 2, BASIC manages variable-length strings by claiming new memory as necessary. When available memory is exhausted BASIC compacts its string space, overwriting abandoned string data with strings that are still active.

This method is not practical with disk files, because copying data from one part of the disk to another for the purpose of compaction would take an unacceptable amount of time. Therefore, DOS initially allocates an entire cluster for each file, to provide space for subsequent data. When the ten-byte file mentioned earlier is added to, space on the disk has already been set aside for all or part of the new data that will be written. And when the first cluster's capacity is exceeded, DOS allocates an entire second cluster to hold the additional data.

Even though it is common for a disk to become fragmented, allocating clusters that are comprised of groups of contiguous sectors greatly reduces the number of individual fragments that must be accessed. The track, sector, and cluster makeup of a 360k 5-1/4 inch floppy disk is shown in Figure 6-1.

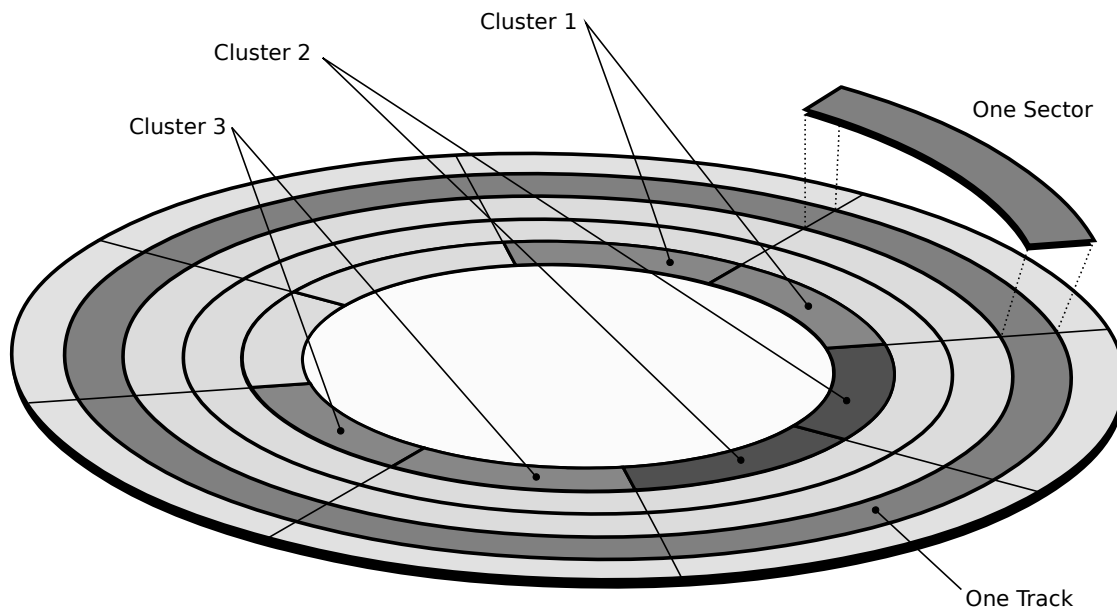


Figure 6-1: Sector and cluster organization for a 360k floppy disk.

This disk is divided into 40 circular tracks, and each track is further divided into nine sectors. One sector holds 512 bytes, and each pair of tracks is combined to form a single cluster. For a 360k disk, no file fragment will ever be smaller than two clusters, since this is the minimum amount of space that DOS allocates. Likewise, a hard disk that combines four sectors into each cluster will never be divided into pieces smaller than four sectors.

Please understand that tracks and sectors are physical entities that are magnetically encoded onto the disk when it is formatted—it is DOS that treats each pair of sectors as a single cluster. Note that since a 360k disk stores nine sectors on each track, some clusters will in fact span two tracks.

Using the disk in Figure 6-1 as an example, the first short file that is written to it will be placed in cluster 1 (sectors 1 and 2), even if the file does not fill both sectors. The second file written to this disk will then be stored starting at cluster 2 (sectors 3 and 4). If the first file is later extended beyond the 1,024 bytes that can fit into cluster 1, the excess will be added beginning at cluster 3 (sectors 5 and 6). Thus, when DOS reads the first file sequentially, it must read cluster 1, skip over cluster 2, and then continue reading at cluster 3.

Of course, this takes longer than reading a file that is contiguous, because the disk drive must wait until the second file's intervening sectors have passed beneath it. This problem is compounded by additional head movement when the fragmentation extends across more than one track, as well as by other timing issues.

There are also three special areas on every disk: the boot sector, the Disk Directory, and the File Allocation Table (FAT). DOS uses the directory and FAT to know the name of each file, and where on

the disk its first cluster is located. For simplicity, these are not shown in Figure 6-1, and indeed, they are in fact stored before any files on a disk.

When a 360K floppy disk is formatted, DOS sets aside room for 112 directory entries. Each entry is 32 bytes long, and holds the name of each file on the disk, its current size, the date and time it was last written to, its attribute (hidden, read-only, and so forth), and starting cluster number. When you open a file, DOS searches each directory entry for the file name you specified, and once found, goes to the first cluster that holds the file's data.

The disk's FAT contains one entry for every cluster in the data area, to show which clusters are in use and by which file. The FAT is organized as a linked list, with each entry pointing to the next. The last cluster in the file is identified with a special value. The FAT also holds other special values to identify unused, reserved, and defective clusters.

Because there are a fixed number of directory entries on a disk, it is possible to receive a "Disk full" message when attempting to open a new file, even when there is sufficient data space. The root directory of a 360K floppy disk is limited to 112 entries, and a 1.2MB disk can hold up to 224 file names. Notice that a volume label takes one directory entry, although no data space is allocated to it. Unlike the root directory on a disk, subdirectories that you create are not limited to an arbitrary number of file name entries. Rather, a subdirectory is in fact a file, and it can be extended indefinitely until there is no more room on the disk.

Fortunately, most programmers do not have to deal with disk access at this level. When you ask BASIC to open a file and then read from or write to it, DOS handles all the low-level details for you. However, I think it is important to have at least a rudimentary understanding of how disks are organized.

If you are interested in learning more about the structure of disks and data files, I recommend Peter Norton's *Programmer's Guide to the IBM PC & PS/2*. This excellent reference is published by Microsoft Press, and can be found at most major book stores.

Disk-Like Devices

A device is related to a file in that you can open it using BASIC's OPEN command, and then access it with GET # and PRINT # and the other file-related BASIC statements. There are a number of devices commonly used with personal computers, and these include printers, modems, tape backup units, and the console (the PC's keyboard and display screen). Some of these devices are maintained by DOS, and others are also controlled by BASIC.

For example, when you open "SCRN:" for Output mode in a BASIC program, BASIC takes responsibility for displaying the characters that you print. However, if you instead open "CON", BASIC merely sends the data to DOS, which in turn sends it to the display screen. Any device whose

name is followed by a colon is considered to be a BASIC device; the absence of a trailing colon indicates a DOS device. This is important to understand, because there may be situations when you want to route your program's output directly through DOS, and not have it be intercepted by BASIC.

One such situation would be when printing the special control characters that the ANSI.SYS device driver recognizes. Normally, BASIC processes data in a PRINT statement by writing directly to screen memory. This provides the fastest response, which is of course desirable in most programs. But ANSI.SYS operates by intercepting the stream of characters sent through DOS. Since BASIC normally bypasses DOS for screen operations, ANSI.SYS never gets a chance to see those characters.

Another reason for printing through DOS is to activate TSR (Terminate and Stay Resident) programs that intercept the BIOS video routines. When data is sent through DOS for display, DOS merely passes it on to the BIOS routines which do the real work. For example, some early screen design utilities use this method, to accommodate multiple programming languages by avoiding the differences in calling and linking. Therefore, to activate, say, a pop-up help screen, you are required to print a special control string. One such utility uses two CHR\$(255) bytes followed by the name of the screen to be displayed.

Although this method is very clumsy when compared to newer products that provide BASIC-linkable object files, it is simpler for the vendor than providing different objects for each supported language. This also allows screens to be displayed from within a batch file using the ECHO command. Therefore, if you need to send data through DOS or the BIOS for whatever reason, you would open and print to the "CON" device, instead of using normal PRINT statements or printing to the "SCRN:" device.

One final point worth mentioning is the value of using the same syntax for both files and devices. Many programs let the user specify where a report is to be sent—either to a disk file, a printer, or the screen. Rather than duplicate similar code three times in a program, you can simply assign a string variable to the appropriate device or file name. This is shown in the listing below.

```
PRINT "Printer, Screen, or File? (P/S/F): ";
DO
  Choice$ = UCASE$(INKEY$)
LOOP UNTIL INSTR(" PSF", Choice$) > 1

IF Choice$ = "P" THEN
  Report$ = "LPT1:"
ELSEIF Choice$ = "S" THEN
  Report$ = "SCRN:"
ELSE
  PRINT
  LINE INPUT "Enter a file name: ", Report$
END IF

OPEN Report$ FOR OUTPUT AS #1
PRINT #1, Header$
PRINT #1, SomeStuff$
PRINT #1, MoreStuff$
...
...
CLOSE #1
END
```

Here, the same block of code can be used regardless of where the report is to be sent. The only alternative is to duplicate similar code three times using PRINT statements if the screen was specified, LPRINT if they want the printer, or PRINT # if the report is being sent to a file. Of course, this example could be further expanded to prompt for a printer number (1, 2, or 3) if a printer is specified.

Exploring Data Files

All data is stored on disk as a continuous stream of binary information, regardless of how the file was opened. Even though BASIC and other languages offer a number of different file access methods, all disk files merely contain a series of individual bytes. When you open a file for random access, you are telling BASIC that it is to treat those bytes in a particular manner. In this case, the file is comprised of one or more fixed-length records. Thus, BASIC can perform many of the low level details that help you to organize and maintain that data.

Likewise, opening a file for INPUT tells BASIC that you plan to read variable-length string data. Rather than reading or writing a single block of a given length, BASIC instead knows to continue to read bytes from the file until a terminating comma or carriage return is encountered. However, in both of these cases the disk file is still comprised of a series of bytes, and the access method you specify merely tells BASIC how it is to treat those bytes.

The short program below illustrates this in context, and you can verify that all three files are identical using the DOS COMP utility program.

```
OPEN "File1" FOR OUTPUT AS #1
  PRINT #1, "Testing"; SPC(13);
CLOSE

OPEN "File2" FOR BINARY AS #1
  Work$ = "Testing" + SPACE$(13)
  PUT #1, , Work$
CLOSE

OPEN "File3" FOR RANDOM AS #1 LEN = 20
  FIELD #1, 20 AS Temp$
  LSET Temp$ = "Testing"
  PUT #1
CLOSE
END
```

In fact, even executable program files are indistinguishable from data files, other than by their file name extension. Again, it is how you choose to view the file contents that determines the actual form of the data.

File Buffers

Before I explain the various file access methods that BASIC provides, there is one additional low-level detail that needs to be addressed: file buffers. A *file buffer* is a portion of memory that holds data on its way to and from a disk file, and it is used to speed up file reads and writes.

As you undoubtedly know, accessing a disk drive is one of the slowest operations that occurs on a PC. Because disk drives are mechanical, data being read or written requires a motor that spins the actual disk, as well as a mechanism to move the drive head to the appropriate location on the disk surface. Even if a file is located in contiguous disk clusters, a substantial amount of mechanical activity is required during the course of accessing a large file.

When you open a file for reading, DOS uses a section of memory that it allocated on boot-up as a disk buffer. The first time the file is accessed, DOS reads an entire sector into memory, even if your program requests only a few bytes. This way, when your program makes a subsequent read request, DOS can retrieve that data from memory instead of from the disk. This provides an enormous performance boost, since memory can be accessed many times faster than any mechanical disk drive. Even if the next portion of data being read is located in the same sector, the disk drive must wait for the disk to spin until that sector arrives at the magnetic read/write head.

When using a floppy disk the time delays are even worse. Once a second or two have passed after accessing a floppy disk, the motor is turned off automatically. Having to then restart it again imposes yet another one or two second delay.

Similarly, when you write data to a file DOS simply stores the data in the buffer, instead of writing it to the disk. When the buffer becomes full (or when you close the file—whichever comes first), DOS writes the entire buffer contents to the disk all at once. Again, this is many times faster than accessing the physical drive every time data is written.

You can control the amount of memory that DOS sets aside for its buffers with a `BUFFERS=` statement in the PC's `CONFIG.SYS` file. For each buffer you specify, 512 bytes of memory is taken and made unavailable for other uses. Even though you might think that more buffers will always be faster than fewer, this is not necessarily the case. For each buffer, DOS also maintains a table that shows which disk sectors the buffer currently holds. At some point it can actually take longer for DOS to search through this table than to read the sector from disk. Of course, this time depends on the type of disk (floppy or hard), and the disk's access speed.

Although DOS' use of disk buffers greatly improves file access speed, there is still room for improvement. Each call to DOS to read or write a file takes a finite amount of time, because most DOS services are handled by the same interrupt service routine. Which particular service a program wants is specified in one of the processor's registers, and determining which of the many possible services has been requested takes time.

To further improve disk access performance, BASIC performs additional file buffering using its own routines. Since BASIC's buffers are usually located in near memory, they can also be accessed very quickly, because additional steps are needed to access data outside of `DGROUP`. However, BASIC PDS and VB/DOS store file buffers in the same segment used for string variables, so there is slightly less

improvement when far strings are being used. When you open a random access file, a block of memory large enough to hold one entire record is set aside in string memory. If a record length is given as part of the OPEN command with LEN =, BASIC uses that for the buffer size. Otherwise, it uses the default size of 128 bytes.

When you open a file for sequential access, BASIC also allocates string memory for a buffer. 512 bytes are used by default, though you can override that with the optional LEN = argument. Specifying a buffer size with non-random files will be discussed later in this chapter.

Note that BASIC PDS does not create a buffer when a file is opened for random access and you are using far strings. If a subsequent FIELD statement is then used, the fielded strings themselves comprise the buffer. Otherwise, BASIC assumes you will be reading the data into a TYPE variable, and avoids the extra buffering altogether. Also, file buffers in a BASIC PDS program are always stored in string memory, which is not necessarily DGROUP. If you are in the QBX environment or have compiled with the /fs far strings option, all file buffers will be stored in the far string data segment.

Although BASIC's additional file buffering does improve your program's speed, it also comes at a cost: the buffers take away from string memory, and the only way to release their memory is to flush their contents to disk by closing the file. DOS offers a service to purge a file's buffers, to ensure that the data will be intact even if the program is terminated abnormally or the power is turned off. Therefore, it is considered good practice to periodically close a file during long data entry sessions. But closing the file and then reopening it after writing each record takes a long time, and more than negates any advantage offered by BASIC's added buffering.

```
Also, the DOS service that flushes a file's buffers
does not flush BASIC's buffers. Any data you have
written to disk that is still pending in a BASIC buffer
will not be written to the file by this service.
```

It is interesting to note that BASIC always closes all open files when a program ends, so it is not strictly necessary to do that manually. I mention this only because you can save a few bytes by eliminating the CLOSE command. Also, DOS flushes its buffers and closes all open files when a program ends, so a few bytes can be saved this way even with non-BASIC programs. Again, I am not necessarily recommending that you do this, and some programmers would no doubt disagree with such advice. But the fact is that an explicit CLOSE is not truly needed.

File Access Methods

BASIC offers three fundamental methods for accessing files, and these are specified when the file is opened. There are also several variations and options available with each method, and these will be discussed in more detail in the sections that describe each method.

The first access method is called Sequential, because it requires you to read from or write to the file in a continuous stream. That is, to read the last item in a sequential file you must read all of the items that precede it. There are three different forms of OPEN for accessing sequential files.

OPEN FOR OUTPUT creates the named file if it does not yet exist, or truncates it to a length of zero if it does. Once a file has been opened for output, you may only write data to it.

OPEN FOR APPEND is related to OPEN FOR OUTPUT, and it also tells BASIC to open the file for writing. Unlike OPEN FOR OUTPUT, however, OPEN FOR APPEND does not truncate a file if it already exists. Rather, it opens the file and then seeks to the place just past the last byte. This way, data that is subsequently written will be appended to the end of the file. Note that OPEN FOR APPEND will also create a file if it does not already exist.

OPEN FOR INPUT requires that the named file be present; otherwise, a "File not found" error will result. Once a file has been opened for input, you may only read from it.

BASIC also offers the SEEK command to skip to any arbitrary position in the file, and SEEK can in fact be used with sequential files. However, sequential files are generally written using a comma or a carriage return/line feed pair, to indicate the end of each data item. Since each item can be of a varying length, it is difficult if not impossible to determine where in the file a given item begins. That is, if you wanted to read, say, the 200th line in a README file, how could you know where to seek to?

OPEN FOR RANDOM allows you to read from and write to the file. When you use OPEN FOR RANDOM, BASIC knows that you will be accessing fixed-length blocks of data called *records*. The advantage of random access is that any record can be accessed by a record number, instead of having to read through the entire file to get to a particular location. That is, you can read or write any record randomly, without regard to where it is in the file. Because each record has the same physical length as every other record, it is easy for BASIC to calculate the location in the file to seek to, based on the desired record number and the fixed record length.

Using random access is ideal for data that is already organized as fixed-length records such as you would find in a name and address database. Since each record contains the same amount of information, there is a natural one-to-one correspondence between the data and the record number in which it resides. For example, the data for customer number 1 would be stored in record number 1, customer 2 is stored in record 2, and so forth.

Random access can also be used for text and other document files; however, that is much less common. Although this would let you quickly access any arbitrary line of text in the file, the trade-off is a considerable waste of disk resources. For each line, space equal to the longest one must be set aside for all of them. In a typical document file line lengths will vary greatly, and it is wasteful to set aside, say, 80 bytes for each line.

OPEN FOR BINARY is a hybrid access method of sequential and random access. A binary file is opened using the command OPEN FOR BINARY, and like random, BASIC lets you both read and

write the file. Binary access is most commonly used when the data in the file is neither fixed-length in nature, nor delimited by commas or carriage returns. One example of a binary file is a Lotus 1-2-3 worksheet file. Each cell's contents follows a well-defined format, but varying types of information are interspersed throughout the file.

For example, an 8-byte double-precision number may be followed by a variable length text field, which is in turn followed by the current column width represented as a 2-byte integer. Another example of binary information is the header portion of a dBASE data file. Although the data itself is of a fixed length, a block of data is stored at the beginning of every dBASE data file to indicate the number of fields in each file and their type. Naturally, the length of this header will vary depending on the number of fields in each record. An example program to read Lotus worksheet files is given later in this chapter, and a program to read and process dBASE files is shown in Chapter 7.

Note that BASIC imposes its own rules on what you may and may not do with each file access method. This is unfortunate, because DOS itself has no such restrictions. That is, DOS allows you to open a file for output, and then freely read from the same file. To do this with BASIC you must first close the file, and then open it again for input. You can bypass BASIC entirely if you want, to open files and then read and write them. This requires using CALL Interrupt, and examples of doing this will be shown in Chapter 11.

BASIC offers two different forms of the OPEN command. The more common method—and the one I prefer—is as follows:

```
OPEN FileName$ FOR OUTPUT AS #FileNum [LEN = Length]
```

Of course, OUTPUT could be replaced with RANDOM, BINARY, INPUT, or APPEND. The other syntax is more cryptic, and it uses a string to specify the file mode. To open a file for output using the second method you'd use this:

```
OPEN "O", #FileNum, FileName$, [Length]
```

The first syntax is available only in QuickBASIC and the other current versions of the BASIC compiler. The second is a holdover from GW-BASIC, and according to Microsoft is maintained solely for compatibility with old programs. The available single-letter mode designators are "O" for output, "I" for input, "R" for random, "A" for append, and "B" for binary. Note that "B" is not supported in GW-BASIC, and was added beginning with QuickBASIC version 4.0.

Besides being more obscure and harder to read, the older syntax does not let you specify the various access and sharing options available in the newer syntax. One advantage of the older method is that you can defer the open mode until the program runs. That is, a string variable can be used to determine how the file will be opened. However, there are few situations I can envision where that would be useful. Of course, the choice is yours, and some programmers continue to use the original version.

Files Manipulation Statements

BASIC offers a number of different statements for opening and manipulating files. In a few cases, the same command may have different meanings, depending on how the file is opened. For example `LEN =` mentioned earlier assumes a different default value when a file is opened for random access compared to when it is opened for output. Similarly, `GET #` may or may not accept or require a variable name and optional seek offset, depending on the file mode. Therefore, pay close attention to each statement as it is described in the sections that follow. Specific differences will be listed as they relate to each of the various file access methods.

Opening and Closing Files

Before any file or device may be accessed, it must first be opened with BASIC's `OPEN` statement. When you use `OPEN`, it is up to you to make up a file number that will be used when you reference the file later. If you use `OPEN "MYDATA" FOR OUTPUT AS #1`, then you will also use the same file number (1) when you subsequently print to the file. For example, you might use `PRINT #1, Any$`. Initially, it might appear that letting the programmer determine his or her own file numbers is a feature. After all, you are allowed to make up your own variable names, so why not file numbers too? Indeed, BASIC is rare among the popular languages in this regard; both C and Pascal require that the programmer remember a file number that is given to them.

There are several problems with BASIC's use of file numbers, and in fact DOS does not use this method either. Instead, DOS returns a *file handle* when a file has been successfully opened. When an assembly language program, or BASIC itself, calls DOS to open a file, it is DOS who issues the number and not the program. BASIC must therefore maintain a translation table to relate the numbers you give to the actual handles that DOS returns. This table requires memory, and that memory is taken from `DGROUP`.

But there is another, more severe problem with BASIC's use of file numbers instead of DOS handles, because it is possible that you could accidentally try to open more than one file using the same number. In a small program that opens only one or two files, it is not difficult to remember which file number goes with which file. But when designing reusable subroutines that will be added to more than one program, it is impossible to know ahead of time what file numbers will be in use.

To solve this problem, Microsoft introduced the `FREEFILE` function with QuickBASIC 4.0. `FREEFILE` was described in Chapter 4, but it certainly bears a brief mention again here. Each time you use `FREEFILE` it returns the next available file number, based on which numbers are already taken. Therefore, any subroutine that needs to open a file can use the number `FREEFILE` returns, confident that the number is not already in use.

Unless you specify otherwise, a file that has been opened for `RANDOM` or `BINARY` can be both read from and written to. The `ACCESS` option of the `OPEN` statement lets you indicate that a random or

binary file may be read or written only. Even though you may ask for both READ and WRITE access when the file is opened, read/write permission is the default. In some cases you may need to open a file for binary access, and also prevent your program from later writing to it. In that case you would use the ACCESS READ option.

Likewise, specifying ACCESS WRITE tells BASIC to let your program write to the file, but prevent it from reading. This may seem nonsensical, but one situation in which write-only access might be desirable is when designing a network mail system. In that case it is quite likely that a program would be permitted to send mail to another user's electronic "mailbox", but not be allowed to read the mail contained in that file. The various ACCESS options are intended for use with any version of DOS higher than 2.0.

Frankly, these ACCESS options are pointless, because if you wrote the program then you can control whether the file is read from or written to. If you are writing the Send Mail portion of a network application, then you would disallow reading someone else's mail as part of the program logic. And if you do open a file for ACCESS WRITE, BASIC will generate an error if you later try to read from it. So I personally don't see any real value in using these ACCESS arguments.

The remaining two OPEN options are LOCK and SHARED, and these are meant for use with shared files under DOS 3.0 or later. Shared access is primarily employed on a network, though it is possible to share files on a single computer. This could be the case when a file needs to be accessed by more than one program when running under a task-switching program such as Microsoft Windows.

You can specify that a file is to be shared by simply adding the SHARED clause to the OPEN statement. Thus, another program could both read and write the file, even while it is open in your program. To specify shared access but prevent other programs from writing to the file you would use LOCK WRITE. Similarly, using LOCK READ lets another program write to the file but not read from it, and LOCK READ WRITE prevents both.

The LOCK statement can optionally be used on a shared file that is already open to prohibit another program from accessing it only at certain times. The LOCK statement allows all or just a portion of a file to be locked, and the UNLOCK statement releases the locks that were applied earlier. Please understand that these network operations are described here just as a way to introduce what is possible. Network and database programming will be described in depth in Chapter 7.

Finally, you close an open file using BASIC's CLOSE command. CLOSE accepts one or more file numbers separated by commas, or no numbers at all which means that every open file is to be closed. You can also use the RESET command to close all currently open files. When a file that has been opened for one of the output modes is closed, its file buffer is flushed to disk and DOS updates the directory entry for that file to indicate the current date and time and new file size. Closing any type of file releases the buffer memory back to BASIC's string memory pool for other uses.

Reading and Writing Data

Once a file has been opened you can read from it, write to it, or both, depending on what form of OPEN was used. Any file that has been opened for input may be read from only. Unlike the BASIC-related limitations I mentioned earlier, DOS imposes this restriction, and for obvious reasons. However, when you open a file for output or append, it is BASIC that prevents you from reading back what you wrote. BASIC imposes several other unfortunate limitations regarding what you can and cannot do with an open file, as you will see momentarily.

Sequential access is commonly used with devices as well as with files. Although it is possible to open a printer for random access, there is little point since data is always printed sequentially. Similarly, reading from the keyboard or writing to the screen must be sequential. In the discussions that follow, you can assume that what is said about accessing files also applies to devices, unless otherwise noted.

Sequential Output

Data is written to a sequential file using the PRINT # statement, using the same syntax as the normal PRINT statement when printing to the display screen. That is, PRINT # accepts an optional semicolon to suppress a carriage return and line feed from being written to the file, or a comma to indicate that one or more blank spaces is to be written after the data. The number of blanks sent to the file depends on the current print position, just like when printing to the screen.

You can also use the WRITE # statement to print data to a sequential file, but I recommend against using WRITE in most situations. Unlike PRINT that merely sends the data you give it, WRITE adds surrounding quotes to all string data, which takes time and also additional disk space. Since a subsequent INPUT from the file will just have to remove those quotes which takes even more time, what's the point? Further, WRITE does not let you specify a trailing semicolon or comma. Although a comma may be used as a delimiter between items written to disk, the comma is stored in the file literally when WRITE is used.

The only time I can see WRITE being useful is for printing data that will be read by a non-BASIC application that explicitly requires this format. Many database and spreadsheet programs let you import comma-delimited data with quoted strings such as WRITE uses. These programs treat each complete line ending with a carriage return as an entire record, and each comma-delimited item within the line as a field in that record. But you should avoid WRITE unless your program really needs to communicate with other such applications, because it results in larger data files and slower performance.

Another use for WRITE is to protect strings that contain commas from being read incorrectly by a subsequent INPUT statement. INPUT uses commas to delimit individual strings, and the quotes allow you to input an entire string with a single INPUT command. But BASIC's LINE INPUT does this anyway, since it reads an entire line of text up to a terminating carriage return. You could also add the quotes manually when needed:

```
IF INSTR(Work$, ",") THEN
  PRINT #1, CHR$(34); Work$; CHR$(34)
ELSE
  PRINT #1, Work$
END IF
```

You may also use TAB and SPC to format the output you print to a file or device. For the most part, TAB and SPC operate like their non-file counterparts, including the need to add an extra empty PRINT to force a carriage return at the end of a line. That is, when you use

```
PRINT Any$; TAB(20)
```

or

```
PRINT #1, SomeVar; SPC(13)
```

BASIC adds a trailing semicolon whether you want it or not. To force a new line at that point in the printing process requires an additional PRINT or PRINT # statement. This isn't really as much of a nuisance as yet another code bloater, since an empty PRINT adds 9 bytes of compiler-generated code and an empty PRINT # adds 18 bytes.

One important difference between the screen and file versions of TAB and SPC is the way long strings are handled. If you use TAB or SPC in a PRINT statement that is then followed by a string too long to fit on the current line, the screen version will advance to the next row, and print the string at the left edge. This is probably not what you expected or wanted. When printing to a file, however, the string is simply written without regard to the current column. Column 80 is the default width for the screen and printer when they have been opened as devices, though you may change that using WIDTH.

The WIDTH statement lets you specify at which column BASIC is to automatically add a carriage return/line feed pair. The default for a printer is at column 80. In most programming situations this behavior is a nuisance, since many printers can accommodate 132 columns. After all, why shouldn't you be allowed to print what you want when you want, without BASIC intervening to add unexpected and often unwanted extra characters? Most programmers disable this automatic line wrapping by using WIDTH # FileNum, 255 if the printer was opened as a device, or WIDTH LPRINT, 255 if using LRPINT statements.

Curiously, this special value is not mentioned anywhere in the otherwise very complete documentation that comes with BASIC PDS. In fact, using a width value of 255 is mandatory if you intend to send binary data to a printer. Most modern printers accept both graphics commands and downloadable fonts. Since either of these will no doubt result in strings longer than 80 or even 255 characters, it is essential that you have a way to disable the favor that BASIC does for you. Undoubtedly, the automatic addition of a carriage return and line feed goes back to the early days of primitive printers that required this. The only reason Microsoft continues this behavior is to assure compatibility with programs written using earlier versions of BASIC.

Related to the WIDTH anomaly is BASIC's insistence on adding a CHR\$(10) line feed whenever you print a CHR\$(13) carriage return to a device. Again, this dubious feature is provided on the assumption that you would always want a line feed after every carriage return. But there are many cases where you wouldn't, such as the font and graphics examples mentioned earlier. If you add the "BIN" (binary) option when opening a printer, you can prevent BASIC from forcing a new line every 80 columns, and also suppress the addition of a line feed following each carriage return. For example, `OPEN "LPT1:BIN" FOR OUTPUT AS #1` tells BASIC to open the first parallel printer in binary mode.

The `PRINT # USING` statement lets you send formatted numeric data to a file, in the same way you would use the regular `PRINT USING` to format numbers on the screen. `PRINT # USING` accepts the same set of formatting commands as `PRINT USING`, allowing you to mix text and formatted numbers in a single `PRINT` operation. If your program will be printing formatted reports from the disk file later, I recommend using `PRINT USING` at that time, instead of when writing the data to disk. Otherwise, the extra spaces and other formatting information are added to the file increasing its size. In fact, `PRINT # USING` is really most appropriate when printing to a device such as a printer.

Finally, it is important to point out the importance of selecting a suitable buffer size. As I described earlier, BASIC and DOS employ an area of memory as a buffer to hold information on its way to and from disk. This way information can often be written to or read from memory, instead of having to access the physical disk each time. Besides the buffers that DOS maintains, BASIC provides additional buffering when your program is using sequential input or output.

BASIC lets you control the size of this buffer, using the `LEN =` option of the `OPEN` statement. In general, the larger you make the buffer, the faster your programs will read and write files. The trade-off, however, is that BASIC's buffers are stored in string memory. With QuickBASIC and near strings in BASIC PDS, the buffer is located in `DGROUP`. When BASIC PDS far strings are used, the buffer is in the same segment that the current module uses for string storage.

Conversely, you can actually reduce the default buffer size when string space is at a premium, but at the expense of disk access speed. When using `OPEN FOR INPUT` and `OPEN FOR OUTPUT`, BASIC sets aside 512 bytes of string memory for the buffer, unless you specify otherwise. If you have many sequential files open at once you could reduce the buffer sizes to 128 bytes, for a net savings of 384 bytes for each file. The legal range of values for `LEN =` is between 1 and 32767 bytes.

Notice that the best buffer values will be a multiple of a power of two, and when increasing the buffer size, a multiple of 512. Since a disk sector is almost always 512 bytes, DOS will fill the buffer with an entire sector. In fact, DOS always reads and writes entire sectors anyway. If you use a buffer size of, say, 600 bytes, DOS will have to read 1024 bytes just to get the first portion of the second sector. But when more data is needed later, BASIC will then have to go back and ask DOS for the same information again. By reading entire sectors or evenly divisible portions of a sector, you can avoid having BASIC and DOS read the same information more than once.

Even though larger buffers usually translate to better performance, you will eventually reach the point of diminishing returns, beyond which little performance improvement will result. Table 6-1 shows the

timing results with various buffer sizes when reading a 104K BASIC source file using LINE INPUT. Understand that this test is informal, and merely shows the results obtained using only one PC. In particular, the hard disk results are for a fairly fast (17 millisecond) 150 MB ESDI drive and a PC equipped with a 25 MHz. 386. Therefore, the improvement from a larger buffer is less than you would get on a slower computer with a slower hard disk or with a floppy disk. Many older XT and AT compatible PCs will probably fall somewhere between the results shown here for the hard and floppy disks. Notice that while the improvement actually seems somewhat worse for some increases, this can be attributed to the lack of resolution in the PC's system timer.

Buffer Size (bytes)	Fast ESDI HD (seconds)	360K floppy disc (seconds)
64	2.699	45.260
128	2.420	45.141
256	2.410	45.148
512	2.420	45.150
1024	2.311	27.180
2048	2.139	18.180
4096	2.201	13.570
8192	2.080	11.650
16384	2.039	11.371

Table 6-1: Timing Results For Sequential Reading Versus Buffer Size.

It is important to point out that a buffer is created only for sequential input and output, and also for random files with QuickBASIC. Opening a file for random access with BASIC PDS (and I'll presume VB/DOS) does not create a buffer, nor does opening a file for binary with either version. Further, with random access files a buffer is created by QuickBASIC only when FIELD is used, and the buffer is located within the actual fielded strings. Therefore, the LEN = argument in an OPEN FOR RANDOM statement merely tells BASIC how to calculate record offsets when SEEK and GET are used.

Sequential Input

Sequential data is read using INPUT #, LINE INPUT #, or INPUT\$ #. Like the console form of INPUT, INPUT # can be used to read one or more variables of any type and in any order with a single statement. When reading a file, INPUT # recognizes both the comma and the carriage return as a valid delimiter, to indicate the end of one variable. This is in contrast to the regular keyboard version of INPUT, which issues a "Redo from start" error if the wrong number of comma-delimited variables are entered. Instead, INPUT # simply moves on to the next line for the remaining variables.

LINE INPUT # avoids this entirely, and simply reads an entire string without regard to commas until a carriage return is encountered. This precludes LINE INPUT # from being used with anything but string variables. However, LINE INPUT # can be used with fixed as well as variable-length strings, without the overhead of copying from one type to the other that BASIC usually adds. This copying was described in Chapter 2. As with INPUT #, LINE INPUT # strips leading and trailing quotes from the line if they are present in the file.

The last method for reading a sequential file or device is with the INPUT\$ # function. INPUT\$ # is used to read a specified number of characters, without regard to their meaning. Where commas and carriage returns are normally used to delimit each line of text, INPUT\$ returns them as part of the string. INPUT\$ # accepts two arguments—the number of characters to read and the file number—and assigns them to the specified string. To read, say, 20 bytes from a sequential file that has been opened as #3, you would use Any\$ = INPUT\$(20, #3). Although the pound sign (#) is optional, I prefer to include it to avoid confusion as to which parameter is the file number and which is the number of bytes.

As with sequential output, specifying a larger buffer size than the default 512 bytes can greatly improve the speed of INPUT # and LINE INPUT # statements, but at the expense of string memory.

Random Access

Unlike sequential files that are almost always read starting at the beginning, data in a random access file can be accessed literally in any arbitrary order. Random access files are comprised of fixed-length *records*, and each record contains one or more *fields*. The most common application of random access techniques is in database programs, where each record holds the same type of information as the next. For example, a customer name and address database is comprised of a first name, a last name, a street address, city, state, and Zip code. Even though different names and addresses will be stored in different records, the format and length of the information in each record is identical.

BASIC provides two different ways to handle random access files: the FIELD statement and TYPE variables. Before QuickBASIC version 4.0, the FIELD method was the only way to define the structure of a random access data file. Although Microsoft has publicly stated that FIELD is provided in current versions of BASIC only for compatibility with older programs, it has several important properties that cannot be duplicated in any other way. FIELD also lets you perform some interesting unobvious tricks that have nothing to do with reading or writing files. These are described later in this chapter in the section Advanced File Techniques.

Once a file has been opened for RANDOM you may use the FIELD statement by specifying one or more string variables to hold each field, along with their length. A typical example showing the syntax for the FIELD statement is as follows:

```
OPEN FileName$ FOR RANDOM AS #1 LEN = 97
FIELD #1, 17 AS LastName$, 14 AS FirstName$, 32 AS Address$, 15 AS City$, _
2 AS State$, 9 AS Zip$, 8 AS BalanceDue$
```


Here, the file is opened for random access, and the record length is established as being 97 characters. This allows room for each of the fields in the FIELD statement. In this case 17 characters are set aside for the last name, 14 for the first name, 32 for the street address, 15 for the city, 2 for the state, 9 for the Zip code, and 8 for the double precision balance due value. I often use a field length of 32 characters for name and address data, because that's how many can fit comfortably on a standard 3-1/2 by 15/16 inch mailing label. The first and last names above add up to 32 characters, including a separating blank space.

Note that the underscore shown above is used here as line continuation character, and you'd actually type the entire statement as one long line. In fact, in most cases a FIELD statement must be able to fit entirely on a single line, and there is no direct way to continue the list of variables. Although the BC compiler recognizes an underscore to continue a line as shown here, the BASIC environment does not. Underscores in a source file are removed by the BASIC editor when the file is loaded, and the lines are then combined.

If a second FIELD statement for the same file number is given on a separate line, the additional strings specified are placed starting at the beginning of the same buffer. While it is possible to coerce a new FIELD statement to begin farther into the buffer, that requires an additional dummy string variable:

```
FIELD #1, 17 AS LastName$, 14 AS FirstName$  
FIELD #1, 31 AS Dummy$, 32 AS Address$, 15 AS City$  
FIELD #1, 78 AS Dummy2$, 2 AS State$, 9 AS Zip$
```

Here, the dummy strings are used as placeholders to force the Address\$ and State\$ variables farther into the buffer, and you would not refer to the dummy strings in your program.

Once a field buffer has been defined, special precautions are needed when assigning and reading the fielded string variables. As you know, BASIC often moves strings around in memory when they are assigned. However, that would be fatal if those strings are in a field buffer. A field buffer is written to disk all at once when you use PUT, and it is essential that all of the strings therein be contiguous. If you simply assign a variable that is part of a field buffer, BASIC may move the string data to a new location outside of the buffer and your program will fail.

To avoid this problem you must assign fielded string using either LSET, RSET, or the statement form of MID\$. These BASIC commands let you insert characters into a string, so BASIC will not have to claim new string memory. This further contributes to FIELD's complexity, and it also adds slightly to the amount of code needed for each assignment. For example, the statement One\$ = Two\$ generates 13 bytes of compiled code, and the statement LSET One\$ = Two\$ creates 17. Although LSET is generally faster than a direct assignment, it is important to understand that it also creates more code. But the situation gets even worse.

Because all of the variables in a field buffer must be strings, additional steps are needed to assign numeric variables such as integer and double precision. The CVI and MKS\$ family of BASIC functions are needed to convert numeric data to their equivalent in string form and back. There are

eight of these functions in QuickBASIC with two each for integer, long integer, single precision, and double precision variables. BASIC PDS adds two more to support the Currency data type. All of the various conversion functions have names that start with the letters MK or CV, and a complete list can be found in your BASIC manual.

To convert a double precision variable to equivalent data in an 8-byte string you would use MKD\$, and to convert a 2-byte string that holds an integer to an actual integer value you would use CVI. MKD\$ stands for *Make Double into a string* and it has a dollar sign to show that it returns a string. CVI stands for *Convert to Integer* and the absence of a dollar sign shows that it returns a numeric value. Combined with the requisite LSET, a complete assignment prior to writing a record to disk with PUT would be something like this: `LSET BalanceDue$ = MKD$(BalDue#)`. And if a record has just been read using GET, an integer value in the field buffer could be retrieved using code such as `MyInt% = CVI(IntVar$)`.

The need for LSET, RSET, CVI, and MKS\$ and so forth has historically made learning random access file techniques one of the most difficult and messy aspects of BASIC programming. Besides having to learn all of the statements and how they are used, you also need to understand how many bytes each numeric data type occupies to set aside the correct amount of space in the field buffer. Further, a lot of compiled code is created to convert large amounts of data between numeric and string form. For these and other reasons, Microsoft introduced the TYPE variable with its release of QuickBASIC 4.0.

The TYPE method allows you to establish a record's structure by defining a custom variable that contains individual components for each field in the record. In general, using TYPE is a much clearer way to define a record, and it also avoids the added library code to handle the FIELD, LSET, CVI, and MKS\$ statements. When you use AS INTEGER and AS DOUBLE and so forth to define each portion of the TYPE, the correct number of bytes are allocated to store the value in its native fixed-length format. This avoids having to convert the data to and from ASCII digits.

Using the earlier example, here's how you would define and assign the same record using a TYPE variable:

```
TYPE Record
  LastName AS STRING * 17
  FirstName AS STRING * 14
  Address AS STRING * 32
  State AS STRING * 2
  Zip AS STRING 9
  BalanceDue AS DOUBLE
END TYPE
DIM MyRecord AS Record

MyRecord.LastName = LastName$
MyRecord.FirstName = FirstName$
MyRecord.Address = Address$
MyRecord.State = State$
MyRecord.Zip = Zip$
MyRecord.BalanceDue = BalanceDue#
```

Even though the same names are used for both the TYPE variable members and the strings they are being assigned from, you may of course use any names you want. You could also assign the portions of a TYPE variable from constants using `MyRecord.Zip = "06896"` or `MyRecord.BalanceDue = 4029.80`. Further, one entire TYPE variable may be assigned to another in a single operation using `ThisType = ThatType`. Dissimilar TYPE variables may be assigned using LSET like this: `LSET MyType = YourType`.

As you can see, using TYPE variables instead of FIELD yields an enormous improvement in a program's clarity. However, there are still some programming problems that only FIELD can solve. One limitation of using TYPE variables is that the file structure must be known when the program is compiled, and you cannot defer this until runtime. Therefore, it is impossible to design a general purpose database program, in which a single program can manipulate any number of differently structured files. The compiler needs to know the length and type of data within a TYPE variable, in order to access the data it contains. So while you can use a variable as the `LEN =` argument with OPEN, the record structure itself must remain fixed.

FIELD avoids that limitation because it accepts a variable number of arguments, and varying lengths within each field component. Therefore, by dimensioning a string array to the number of elements needed for a given record, the entire process of opening, fielding, reading, and writing can be handled using variables whose contents and type are determined at runtime. Some amount of IF testing will of course be required when the program runs, but at least it's possible to process a file using variable information.

The following complete program first creates a random access file with five slightly different records using a TYPE variable. It then reads the file independently of the TYPE structure using the FIELD method. Although the second portion of the program uses DATA statements to define the file's structure, in practice this information would be read from disk. In fact, this is the method used by dBASE and Clipper files, based on the field information that is stored in a header portion of the data file.

```
'----- create a data file containing five records
DEFINT A-Z

TYPE MyType
  FirstName AS STRING * 17
  LastName AS STRING * 14
  DblValue AS DOUBLE
  IntValue AS INTEGER
  MiscStuff AS STRING * 20
  SngValue AS SINGLE
END TYPE
DIM MyVar AS MyType

OPEN "MYFILE.DAT" FOR RANDOM AS #1 LEN = 65
MyVar.FirstName = "Jonathan"
MyVar.LastName = "Smith"
MyVar.DblValue = 123456.7
MyVar.IntValue = 10
MyVar.MiscStuff = "Miscellaneous stuff"
MyVar.SngValue = 14.29
```

```

FOR X = 1 TO 5
  PUT #1, , MyVar
  MyVar.DblValue = MyVar.DblValue * 2
  MyVar.IntValue = MyVar.IntValue * 2
  MyVar.SngValue = MyVar.SngValue * 2
NEXT
CLOSE #1

'----- read the data without regard to the TYPE above
READ FileName$, NumFields
REDIM Buffer$(1 TO NumFields) 'holds the FIELD strings
REDIM FieldType(1 TO NumFields) 'the array of data types

RecLength = 0
FOR X = 1 TO NumFields
  READ ThisType
  FieldType(X) = ThisType
  RecLength = RecLength + ABS(ThisType)
NEXT

OPEN FileName$ FOR RANDOM AS #1 LEN = RecLength

PadLength = 0
FOR X = 1 TO NumFields
  ThisLength = ABS(FieldType(X))
  FIELD #1, PadLength AS Pad$, ThisLength AS Buffer$(X)
  PadLength = PadLength + ThisLength
NEXT

NumRecs = LOF(1) \ RecLength 'calc number of records
FOR X = 1 TO NumRecs 'read each in sequence
  GET #1 'get the current record
  CLS
  FOR Y = 1 TO NumFields 'walk through each field
    PRINT "Field"; Y; TAB(15); 'display each field
    SELECT CASE FieldType(Y) 'see what type of data
      CASE -8 'double precision
        PRINT CVD(Buffer$(Y)) 'so use CVD
      CASE -4 'single precision
        PRINT CVS(Buffer$(Y)) 'as above
      CASE -2 'integer
        PRINT CVI(Buffer$(Y))
      CASE ELSE 'string
        PRINT Buffer$(Y)
    END SELECT
  NEXT
  LOCATE 20, 1
  PRINT "Press a key to view the next record ";
  WHILE LEN(INKEY$) = 0: WEND
NEXT
CLOSE #1
END

DATA MYFILE.DAT, 6
DATA 17, 14, -8, -2, 20, -4

```

There are several issues that need elaboration in this program. First is the use of arrays to hold the fielded string data and also each field's type. When the field buffer is defined with an array, the same variable name can be used repeatedly in a loop. A parallel array that holds the field data types permits

the program to relate the field data to its corresponding type of data. That is, `Buffer$(3)` holds the data for field 3, and `FieldType(3)` indicates what type of data it is.

Second, the `FieldType` array uses a simple coding method that combines both the data type and its length into a single value. That is, positive values are used to indicate string data, and the value itself is the field length. Negative values reflect the data type as well as the length, using a negative version of that data type's length. Specifically, -8 is used to indicate a double precision field type, -4 a single precision type, and -2 an integer. If you need to handle long integers or the BASIC PDS Currency data type, you'll need to devise a slightly different method. I chose this one because it is simple and effective.

The final point worth mentioning when comparing `FIELD` to `TYPE` is that the field buffer is relinquished back to BASIC's string pool when the file is closed. But when a `TYPE` variable is dimensioned, the near memory it occupies is allocated by the compiler, and is never available for other uses. Although there is a solution, it requires some slight trickery. The statement `REDIM TypeVar (1 TO 1) AS TypeName` will create a 1-element `TYPE` array in far memory that can then be used as if it were a single `TYPE` variable. That is, any place you would have used the `TYPE` variable, simply substitute the sole element in the array.

Understand that more code is required to access data in a dynamic array than in a static variable. For example, an integer assignment to a member of a dynamic `TYPE` array generates 17 bytes of code, compared to only 6 bytes for the same operation on a static `TYPE`. But when string space is more important than .EXE file size, this trick can make the difference between a program that runs and one that doesn't.

Regardless of which method you use—`TYPE` or `FIELD`—there are several additional points to be aware of. First, the `PUT #` and `GET #` statements are used to write and read a random access file respectively. `PUT #` and `GET #` accept two different forms, depending on whether you are using `TYPE` or `FIELD` to define the record structure.

When `FIELD` is used, `PUT #` and `GET #` may be used with either no argument to access the current record, or with an optional record number argument. That is, `PUT #1` writes the current field buffer contents to disk at the current DOS `SEEK` position, and `GET #1, RecNum` reads record number `RecNum` into the buffer for subsequent access by your program.

As with sequential files, each time a record is read or written, DOS advances its internal seek location to the next successive position in the file. Therefore, to read a group of records in forward order does not require a record number, nor does writing them in that order. In fact, slightly more time is required to access a record when a record number is given but not needed, because BASIC makes a separate call to perform an explicit `Seek` to that location in the file.

When the `TYPE` method is used to access random access data, the record number is also optional, but you must provide the name of a `TYPE` variable or `TYPE` array element. In this case, the record number is still used as the first argument, and the `TYPE` variable is the second argument. If you omit the record

number you must include an empty comma placeholder. For example, `PUT #1, RecNum, TypeVar` writes the contents of `TypeVar` to the file at record number `RecNum`, and `GET #1, , TypeArray(X)` reads the current record into `TYPE` array element `X`.

It is not essential that the `TYPE` variable be as long as the record length specified when `LEN =` was used with `OPEN`, but it generally should be. When a record number is given with `PUT #` or `GET #`, `BASIC` uses the original `LEN =` value to know where to seek to in the file. If a record number is omitted, `BASIC` will still advance to the next complete record even if the `TYPE` variable being read or written is shorter than the stated record length. In most cases, however, you should use a `TYPE` whose length corresponds to the `LEN =` argument unless you have a good reason not to.

Notice that when `LEN =` is omitted, `BASIC` defaults to a record length of 128 bytes. Indeed, forgetting to include the length can lead to some interesting surprises. One clever trick that avoids having to calculate the record length manually is to use `BASIC`'s `LEN` function. Although earlier versions of `BASIC` allowed `LEN` only in conjunction with string variables, `QuickBASIC 4.0` and later versions recognize `LEN` for any type of data.

For example, `LEN(IntVar%)` is always 2, and `LEN(AnyDouble#)` is always equal to 8. When `LEN` is used this way the compiler merely substitutes the appropriate numeric constant when it builds your program. Since `LEN` can also be used with `TYPE` variables and `TYPE` array elements, you can let `BASIC` do the byte counting for you. The brief program fragment below shows this in context.

```
TYPE Something
  X AS INTEGER
  Y AS DOUBLE
  Z AS STRING * 100
END TYPE
DIM Anything AS Something
OPEN MyData$ FOR RANDOM AS #1 LEN = LEN(Anything)
```

In particular, this method is useful if you later modify the `TYPE` definition, since the program will be self-accommodating. Changing `Z` to `STRING * 102` will also change the value used as the `LEN =` argument to `OPEN`. Be careful to use the actual variable name with `LEN`, and not the `TYPE` name itself. That is, `LEN(Anything)` will equal 110, but `LEN(Something)` will be 2 if `DEFINT` is in effect. When `BASIC` sees `LEN(Something)` it assumes you are referring to a variable with that name, not the `TYPE` definition.

The only time this use of `LEN` will be detrimental is when it is used as a passed parameter many times in a program. Since `LEN` is treated in this case as a numeric constant, it is subject to the same copying issues that `CONST` values and literal numbers are. Therefore, you would probably want to assign a variable once from the value that `LEN` returns, and use that variable repeatedly later as described in Chapter 2.

Binary Access

Binary file access lets you read or write any portion of a file, and manipulate any type of information. Reading a sequential file requires that the end of each data item be identified by a comma, or a carriage return line feed pair. Random access files do not require special delimiters, and instead rely on a fixed record length to know where each record's data starts and ends. A binary file may be organized in any arbitrary manner; however, it is up to the programmer to devise a method for determining what goes where in the file.

The overwhelming advantage of binary over sequential access is the enormous space and speed savings. A file that requires extra carriage returns or commas will be larger than one that does not. Moreover, numeric data in a binary file is stored in its native fixed-length format, instead of as a string of ASCII digits. Therefore, the integer value -32700 will occupy only two bytes, as opposed to the seven needed for the digits plus either a comma or carriage return and line feed.

Furthermore, converting between numbers and their ASCII representation is one of the slowest operations in BASIC. Because the STR\$ and VAL functions must be able to operate on floating point numbers and perform rounding, they are extremely slow. For example, VAL must examine the digits in a string for many special characters such as "e", "d", "&H", and so forth. And with the statement `IntVar% = VAL("1234.56")`, VAL must also round the value to 1235 before assigning the result to IntVar%. Even if you don't use STR\$ or VAL explicitly when reading or writing a file, BASIC does internally. That is, the statement `PRINT #1, D#` is compiled as if you used `PRINT #1, STR$(D#)`. Likewise, `INPUT #1, IntVar%` is compiled the same as `INPUT #1, Temp$: IntVar% = VAL(Temp$)`.

When a file has been opened for binary access you may not use `PRINT #`, `WRITE #`, or `PRINT # USING`. The only statement that can write data to a binary file is `PUT #`. `PUT #` may be used with any type of variable, but not constants or expressions. That is, you can use `PUT #1, , AnyVar`, but not `PUT #1, , 13` or `PUT #1, SeekLoc, X + Y!` or `PUT #1, , LEFT$(Work$, 10)`. This is yet another unnecessary BASIC limitation, which means that to write a constant you must first assign it to a temporary variable, and then use `PUT` specifying that variable.

Reading from a binary file requires `GET #`, which is the complement of `PUT #`. Like `PUT #`, `GET #` may be used with any kind of variable, including `TYPE` variables. When a string variable is written to disk with `PUT #`, the entire string is sent. However, when a string variable is used with `GET #`, BASIC reads only as many bytes as will fit into the target string. So to read, say, 20 bytes into a string from a binary file you would use this:

```
Temp$ = SPACE$(20)           'make room for 20 bytes
GET #FileNum, , Temp$       'read all 20 bytes
```

Although fixed-length strings cannot be cleared to relinquish the memory they occupied, they are equally valid for reading data from a binary file:

```
DIM FLen AS STRING * 20
GET #FileNum, , FLen
```

You can also use `INPUT$` to read a specified number of bytes from a binary file. Therefore you can replace both examples above with the statement `Temp$ = INPUT$(20, #FileNum)`. Contrary to some versions of Microsoft BASIC documentation, `PUT #` does not store the length of the string in a binary file prior to writing the data as it does with files opened for `RANDOM`.

As you've seen, data is written to a binary file using the `PUT #` command, and read using `GET #`. These work much like their random access counterparts in that a seek offset is optional, and if omitted must be replaced with an empty comma placeholder. But where the seek argument in a random `GET #` or `PUT #` specifies a record number, a binary `GET #` treats it as a byte offset into the file.

The first byte in a binary file is considered by BASIC to be byte number 1. This is important to point out now, because DOS considers the first byte to be numbered 0. When we discuss using `CALL Interrupt` to access files in Chapter 11, you will need to take this difference into account.

When reading and writing binary files, BASIC always uses the length of the specified variable to know how many bytes to read or write. The statement `GET #1, , IntVar%` reads two bytes at the current DOS seek location into the integer variable `IntVar%`, and `PUT #1, 1000, LongVar#` writes the contents of `LongVar#` (eight bytes) to the file starting at the 1000th byte. Let's now take a look at a practical application of binary file techniques.

Rather than invent a binary file format as an example, I will instead use the Lotus 1-2-3 file structure to illustrate the effective use of binary access. Although it is possible to skip around in a binary file and read its data in any arbitrary order, a Lotus worksheet file is intended to be read sequentially. Each data item is preceded by an integer code that indicates the type and length of the data that follows. Note that the same format is used by Lotus 1-2-3 versions 1 and 2, and also Lotus Symphony. Newer versions of 1-2-3 that support three-dimensional work sheets use a different format that this program will not accommodate.

A Lotus spreadsheet can contain as many as 63 different kinds of data. However, we will concern ourselves with only those that are of general interest such as cell contents and simple formatting commands. These are Beginning of File, End of File, Integer values, Floating point values, Text labels and their format, and the double precision values embedded within a Formula record. The format used by the actual formulas is quite complex, and will not be addressed. Other records that will not be covered here are those that pertain to the structure of the worksheet itself. For example, range names, printer setup strings, macro definitions, and so forth. You can get complete information on the Lotus file structure as well as other standard formats in Jeff Walden's excellent book, *File Formats for Popular PC Software*, Wiley Press, ISBN 0-471-83671-0, which is available as a digital download at Archive.org:

<https://archive.org/details/FileFormats-ForPopularPCSoftwareAProgrammersReferenceJeffWaldenOCR>

A Lotus file is comprised of individual records, and each record may have a varying length. The length of a record depends on its type and contents, and most records contain a fixed-length header which

describes the information that follows. Regardless of the type of record being considered, each follows the same format: an operation code (opcode), the data length, and the data itself.

The opcode is always a two-byte integer which identifies the type of data that will follow. For example, an opcode of 15 indicates that the data in the record will be treated by 1-2-3 as a text label. The length is also an integer, and it holds the number of bytes in the Data section (the actual text) that follows.

All of the records that pertain to a spreadsheet cell contain a five-byte header at the beginning of the data section. These five bytes are included as part of the data's length word. The first header byte contains the formatting information, such as the number of decimal positions to display. The next two bytes together contain the cell's row as an integer, and the following two bytes hold the cell's column.

Again, this header is present only in records that refer to a cell's contents. For example, the Beginning of File and End of File records do not contain a header, nor do those records that describe the worksheet. Some records such as labels and formulas will have a varying length, while those that contain numbers will be fixed, depending on the type of number. Floating point values are always eight bytes long, and are in the same IEEE format used by BASIC. Likewise, an integer value will always have a length of two bytes. Because the length word includes the five-byte header size, the total length for these double precision and integer examples is 13 and 7 respectively.

It is important to understand that in a Lotus worksheet file, rows and columns are based at zero. Even though 1-2-3 considers the leftmost row to be number 1, it is stored in the file as a zero. Likewise, the first column as displayed by 1-2-3 is labelled "A", but is identified in the file as column 0. Thus, it is up to your program to take that into account as translates the columns to the alphabetic format, if you intend to display them as Lotus does.

In the Read portion of the program that follows, the same steps are performed for each record. That is, binary GET # statements read the record's type, length, and data. If the record type indicates that it pertains to a worksheet cell, then the five-byte header is also read using the GetFormat subprogram. Opcodes that are not supported by this program are simply displayed, so you will see that they were encountered.

The Write portion of the program performs simple formatting, and also ensures that a column-width record is written only once. Figure 6-2 shows the makeup of the numeric formatting byte used in all Lotus files.

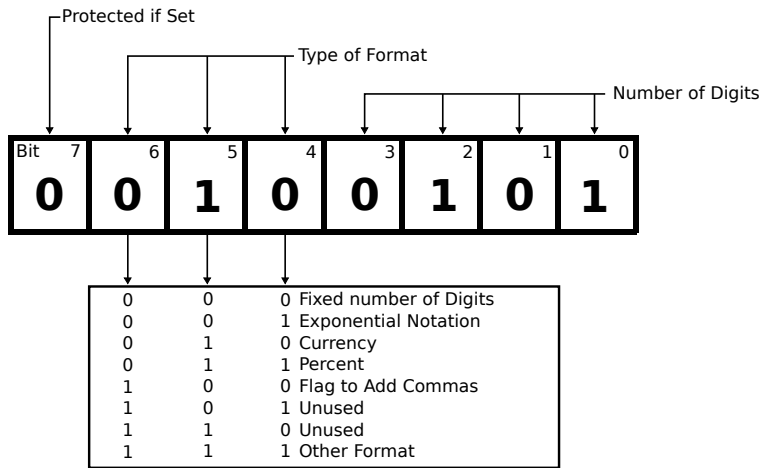


Figure 6-2: The Structure of a Lotus 1-2-3 Format Byte.

The program example below can either read or write a Lotus 1-2-3 worksheet file. If you select Create when this program is run, it will write a worksheet file named SAMPLE.WKS suitable for reading into any version of Lotus 123. This sample file contains an assortment of labels and values. If you select Read, the program will prompt for the name of a worksheet file which it then reads and displays.

```

DEFINT A-Z
DECLARE SUB GetFormat (Format, Row, Column)
DECLARE SUB WriteColWidth (Column, ColWidth)
DECLARE SUB WriteInteger (Row, Column, ColWidth, Temp)
DECLARE SUB WriteLabel (Row, Column, ColWidth, Msg$)
DECLARE SUB WriteNumber (Row, Col, ColWidth, Fmt$, Num#)

DIM SHARED CellFmt AS STRING * 1      'to read one byte
DIM SHARED ColNum(40)                 'max columns to write
DIM SHARED FileNum                    'the file number to use
CLS
PRINT "Read an existing 123 file or ";
PRINT "Create a sample file (R/C)? "
LOCATE , , 1
DO
    X$ = UCASE$(INKEY$)
LOOP UNTIL X$ = "R" OR X$ = "C"
LOCATE , , 0
PRINT X$

IF X$ = "R" THEN

    '----- read an existing file
    INPUT "Lotus file to read: ", FileName$
    IF INSTR(FileName$, ".") = 0 THEN
        FileName$ = FileName$ + ".WKS"
    END IF
    PRINT

    '----- get the next file number and open the file
    FileNum = FREEFILE

```

```

OPEN FileName$ FOR BINARY AS #FileNum

DO UNTIL Opcode = 1      'until End of File code

    GET FileNum, , Opcode 'get the next opcode
    GET FileNum, , Length 'and the data length

    SELECT CASE Opcode   'filter the Opcodes

CASE 0                   'Beginning of File record
    PRINT "Beginning of file, Lotus ";
    GET FileNum, , Temp

    SELECT CASE Temp
        CASE 1028
            PRINT "1-2-3 version 1.0 or 1A"
        CASE 1029
            PRINT "Symphony version 1.0"
        CASE 1030
            PRINT "123 version 2.x"
        CASE ELSE
            PRINT "NOT a Lotus File!"
    END SELECT

CASE 1                   'End of File
    PRINT "End of File"

CASE 12                  'Blank cell
    'Note that Lotus saves blank cells only if
    'they are formatted or protected.
    CALL GetFormat(Format, Row, Column)
    PRINT "Blank:      Format ="; Format,
    PRINT "Row ="; Row,
    PRINT "Col ="; Column

CASE 13                  'Integer
    CALL GetFormat(Format, Row, Column)
    GET FileNum, , Temp
    PRINT "Integer:   Format ="; Format,
    PRINT "Row ="; Row,
    PRINT "Col ="; Column,
    PRINT "Value ="; Temp

CASE 14                  'Floating point
    CALL GetFormat(Format, Row, Column)
    GET FileNum, , Number#
    PRINT "Number:   Format ="; Format,
    PRINT "Row ="; Row,
    PRINT "Col ="; Column,
    PRINT "Value ="; Number#

CASE 15                  'Label
    CALL GetFormat(Format, Row, Column)
    'Create a string to hold the label. 6 is
    'subtracted to exclude the Format, Column,
    'and Row information.

    Info$ = SPACE$(Length - 6)
    GET FileNum, , Info$      'read the label
    GET FileNum, , CellFmt$   'eat the CHR$(0)
    PRINT "Label:      Format ="; Format,

```

```

PRINT "Row ="; Row,
PRINT "Col ="; Column, Info$

CASE 16                                'Formula
CALL GetFormat(Format, Row, Column)
GET FileNum, , Number#                 'read cell value
GET FileNum, , Length                  'and formula length
SEEK FileNum, SEEK(FileNum) + Length  'skip formula
PRINT "Formula:   Format ="; Format,
PRINT "Row ="; Row,
PRINT "Col ="; Column,
PRINT "Value ="; Number#

CASE ELSE
Dummy$ = SPACE$(Length)                'skip the record
GET FileNum, , Dummy$                  'read it in
PRINT "Opcode: "; Opcode               'show its Opcode

END SELECT

'----- pause when the screen fills
IF CSRLIN > 21 THEN
PRINT
PRINT "Press <ESC> to end or ";
PRINT "any other key for more"
DO
K$ = INKEY$
LOOP UNTIL LEN(K$)
IF K$ = CHR$(27) THEN EXIT DO
CLS
END IF

NumRecs = NumRecs + 1                  'count the records

LOOP
PRINT "Number of Records Processed ="; NumRecs
CLOSE

ELSE

'----- write a sample file
FileNum = FREEFILE                    'as above
OPEN "SAMPLE.WKS" FOR BINARY AS #FileNum

Temp = 0                              'OpCode for Start of File
PUT FileNum, , Temp                    'write that
Temp = 2                               'its data length is 2
PUT FileNum, , Temp                    'since it's an integer
Temp = 1030                            'Lotus version 2.x
PUT FileNum, , Temp

Row = 0                                'write this in Row 1
DO
CALL WriteLabel(Row, 0, 16, "This is a Label")
CALL WriteLabel(Row, 1, 12, "So is this")
CALL WriteInteger(Row, 2, 7, 12345)
CALL WriteNumber(Row, 3, 9, "C2", 57.23#)
CALL WriteNumber(Row, 4, 9, "F5", 12.3456789#)
CALL WriteInteger(Row, 6, 9, 99)       'skip a column for fun
Row = Row + 1                          'go on to the next row
LOOP WHILE Row < 6

```

```

'----- Write the End of File record and close the file
Temp = 1 'Opcode for End of File
PUT FileNum, , Temp
Temp = 0 'the data length is zero
PUT FileNum, , Temp
CLOSE

END IF
END

SUB GetFormat (Format, Row, Column) STATIC
GET FileNum, , CellFmt$: Format = ASC(CellFmt$)
GET FileNum, , Column
GET FileNum, , Row
END SUB

SUB WriteColWidth (Column, ColWidth) STATIC

'----- allow a column width only once for each column
IF NOT ColNum(Column) THEN
Temp = 8
PUT FileNum, , Temp
Temp = 3
PUT FileNum, , Temp
PUT FileNum, , Column
Temp$ = CHR$(ColWidth)
PUT FileNum, , Temp$
'----- show we wrote this column's width
ColNum(Column) = -1
END IF
END SUB

SUB WriteInteger (Row, Column, ColWidth, Integ) STATIC
Temp = 13 'OpCode for an integer
PUT FileNum, , Temp
Temp = 7 'Length + 5 byte header
PUT FileNum, , Temp
Temp$ = CHR$(127) 'the format portion
PUT FileNum, , Temp$
PUT FileNum, , Column
PUT FileNum, , Row
PUT FileNum, , Integ
CALL WriteColWidth(Column, ColWidth)
END SUB

SUB WriteLabel (Row, Column, ColWidth, Msg$)
IF LEN(Msg$) > 240 THEN '240 is the maximum length
Msg$ = LEFT$(Msg$, 240)
END IF

Temp = 15 'OpCode for a label
PUT FileNum, , Temp
Temp = LEN(Msg$) + 7 'Length plus 5-byte header
'plus "" plus CHR$(0)

PUT FileNum, , Temp
Temp$ = CHR$(127) '127 is the default format
PUT FileNum, , Temp$
PUT FileNum, , Column
PUT FileNum, , Row

```

```

Temp$ = "" + Msg$ + CHR$(0) 'a "" left-aligns a label
                                'use "^" instead to center
PUT FileNum, , Temp$
CALL WriteColWidth(Column, ColWidth)
END SUB

SUB WriteNumber (Row, Col, ColWidth, Fmt$, Num#) STATIC
IF LEFT$(Fmt$, 1) = "F" THEN 'fixed
  '----- specify the number of decimal places
  Format$ = CHR$(0 + VAL(RIGHT$(Fmt$, 1)))
ELSEIF LEFT$(Fmt$, 1) = "C" THEN 'currency
  Format$ = CHR$(32 + VAL(RIGHT$(Fmt$, 1)))
ELSEIF LEFT$(Fmt$, 1) = "P" THEN 'percent
  Format$ = CHR$(48 + VAL(RIGHT$(Fmt$, 1)))
ELSE 'default
  Format$ = CHR$(127) 'use CHR$(255) for protected
END IF

Temp = 14 'Opcode for a number
PUT FileNum, , Temp
Temp = 13 'Length (8) + 5 = 13
PUT FileNum, , Temp

PUT FileNum, , Format$
PUT FileNum, , Col
PUT FileNum, , Row
PUT FileNum, , Num#

CALL WriteColWidth(Column, ColWidth)
END SUB

```

There are several points worth noting about this program. First, Lotus label strings are always terminated with a CHR\$(0) zero byte, which is the same method used by DOS and the C language. Therefore, the WriteLabel subprogram adds this byte, which is also included as part of the length word that follows the Opcode.

In the WriteNumber subprogram, the 1-byte format code is either 127 to default to unformatted, or bit-coded to indicate fixed, currency, or percent formatting. WriteNumber expects a format string such as "F3" which indicates fixed-point with three decimal positions, or "P1" for percent formatting using one decimal place. If you instead use "C", WriteNumber will use a fixed 2-decimal point currency format.

Earlier I pointed out the extra work is needed to write a constant value to a binary file, because only variables may be used with PUT #. This is painfully clear in each of the Write subprograms, where the integer variable Temp is repeatedly assigned to new values. We can only hope that Microsoft will see fit to remove this arbitrary limitation in a later version of BASIC.

Finally, note the use of the fixed-length string CellFmt\$. Although some language support a one-byte numeric variable type, BASIC does not. Therefore, to read and write these values you must use a fixed-length string. To determine the value after reading a file you will use ASC, and to assign a value prior to writing it you instead use CHR\$. For example, to assign CellFmt\$ to the byte value 123 use CellFmt\$ = CHR\$(123).

Navigating Your Files

BASIC offers a number of file-related functions to determine how long a file is, the current DOS seek location where the next read or write will take place, and also if that location is at the end of the file. These are LOF, LOC and SEEK, and EOF respectively. LOF stands for Length Of File, LOC means current Location, and EOF is End Of File. The SEEK statement is also available to force the next file access to occur at a specified place within the file. All of these require a file number argument to indicate which file is being referred to.

The EOF Function

The EOF function is most useful when reading sequential text files, and it avoids BASIC's "Input past end" error that would otherwise result from trying to read past the end of the available data. The following short complete program reads a text file and displays its contents, and shows how EOF is used for this purpose.

```
OPEN FileName$ FOR INPUT AS #1
  WHILE NOT EOF(1)
    LINE INPUT #1, This$
    PRINT This$
  WEND
CLOSE
```

Notice the use of the NOT operator in this example. The EOF function returns an integer value of either -1 or 0, to indicate true (at the end of the file) or false. Therefore, NOT -1 is equal to 0 (False), and NOT 0 is equal to -1 (True). This use of bit manipulation was described earlier in Chapter 2.

EOF can also be used with binary and random access files for the same purpose. In fact, EOF may be even more useful in those cases, because BASIC does not create an error when you attempt to read past the end as it does for sequential files. Indeed, once you go past the end of a binary or random access file, BASIC simply fills the variables being read with zero bytes. Without EOF there is no way to distinguish between zeros returned by BASIC because you went past the end of the file and zeros that were read as legitimate data.

The EOF function was originally needed with DOS 1.0 for a program to determine when the end of the file was reached. That version of DOS always wrote all data in multiples of 128 bytes, and all file directory entries also were listed with lengths being a multiple of 128. That is, a file which contains only ten bytes of data will be reported by DIR as being 128 bytes long. To indicate the true end of the file, a CHR\$(26) end of file marker was placed just past the last byte of valid data. Thus, EOF was originally written to search for a byte with that value, and return True when it was found.

Most modern applications do not use an EOF character, and instead rely on the file length that is stored in the file's directory entry. However, some older programs still write a CHR\$(26) at the end of the data, and DOS' COPY CON command does this as well. Therefore, BASIC's EOF will return a True value when this character is encountered, even if there is still more data to be read in the file. In fact,

you can provide a minimal amount of data security by intentionally writing a CHR\$(26) at or near the beginning of a sequential file. If someone then uses the DOS TYPE command to view the file, only what precedes the EOF marker will be displayed.

Another implication of EOF characters in BASIC surfaces when you open a sequential file for append mode. BASIC makes a minimal attempt to locate an EOF character, and if one exists it begins appending on top of it. After all, if writing started just past the EOF byte, a subsequent LINE INPUT would fail when it reached that point. Likewise, an EOF test would return true and the program would stop reading at that location in the file. Therefore, BASIC checks the last few bytes in the file when you open for append, to see if an EOF marker is present. However, if the marker is much earlier in a large file, BASIC will not see it.

When EOF is used with serial communications, it returns 0 until a CHR\$(26) byte is received, at which point it continues to return -1 until the communications port is closed.

The LOF Function

The LOF function simply returns the current length of the file, and that too can be used as a way to tell when you have reached the end. In the random access FIELD example program shown earlier, LOF was used in conjunction with the record length to determine the number of records in the file. Since the length of most random access files is directly related to, and evenly divisible by, the number of records in the file, simple division can be used to determine how many records there are. The formula is:

```
NumRecords = LOF(FileNum) \ RecLength
```

Understand that when used with sequential and binary files, LOF returns the length of the file in bytes. But with a random access file, LOF instead provides the number of records.

LOF can also be used as a crude way to see if a file exists. Even though this is done much more effectively and elegantly with assembly language or CALL Interrupt, the short example below shows how LOF can be used for this purpose.

```
FUNCTION Exist% (FileName$) STATIC
  FileNum = FREEFILE
  OPEN FileName$ FOR BINARY AS #FileNum
  Length = LOF(FileNum)
  CLOSE #FileNum
  IF Length = 0 THEN      'it probably wasn't there
    Exist% = 0           'return False to show that
    KILL FileName$       'and delete what we created
  ELSE
    Exist% = -1          'otherwise return True
  END IF
END FUNCTION
```

Besides being clunky, this program also has a serious flaw: If the file does exist but has a perfectly legal length of zero, this function will say it doesn't exist and then delete it. As I said, this method is crude, but a lot of programmers have used it.

The LOC and SEEK Functions

LOC and SEEK are closely related, in that they return information about where you are in the file. However, LOC reports the position of the last read or write, and SEEK tells where the next one will occur. As with LOF, LOC and SEEK return byte values for files that were opened for sequential or binary access, and record numbers when used with random access files.

In practice, LOC is of little value, especially when you are manipulating sequential files. For reasons that only Microsoft knows, LOC returns the number of the last byte read or written, but divided by 128. Since no program I know of treats sequential files as containing 128-byte records, I cannot imagine how this could be useful. Further, since LOC returns the location of the last read or write, it never reflects the true position in the file.

When used with communications, LOC reports the number of characters in the receive buffer that are currently waiting to be read, which is useful. When used with INPUT\$ #, LOC provides a handy way to retrieve all of the characters present in the buffer at one time. This is shown in context below, and the example assumes that the communications port has already been opened.

```
NumChars = LOC(1)
IF NumChars THEN
  This$ = INPUT$(NumChars, #1)
END IF
```

The SEEK function always returns the current file position, which is the point at which the next read or write will take place. One good use for SEEK is to read the current location in a sequential file, to allow a program to walk backwards through the file later. For example, if you need to create a text file browsing program, there is no other way to know where the previous line of a file is located. A short program that shows this in context follows in the section that describes the SEEK statement.

The SEEK Statement

Where the SEEK function lets you determine where you are currently in a file, the SEEK statement lets you move to any arbitrary position. As you might imagine, SEEK as a statement is similar to the function version in that it assumes a byte value when used with sequential and binary files, and a record number with random access files.

SEEK can be very useful in a variety of situations, and in particular when indexing random access files. When an indexing system is employed, selected portions of a data file are loaded into memory where they can be searched very quickly. Since the location of the index information being searched corresponds to the record number of the complete data record, the record can be accessed with a single GET #. This was described briefly in the discussion of the BASIC PDS ISAM options in Chapter 5. Thus, once the record number for a given entry has been identified, the SEEK statement (or the SEEK argument in the GET # command) is used to access that particular record.

For this example, though, I will instead show how SEEK can be used with a sequential file. The following complete program provides the rudiments of a text file browser, but this version displays only one line at a time. It would be fairly easy to expand this program to display entire screenfuls of text, and I leave that as an exercise for you.

The program begins by prompting for a file name, and then opens that file for sequential input. The maximum number of lines that can be accommodated is set arbitrarily at 5000, though you will not be able to specify more than 16384 unless you compile with the /ah option. The long integer Offset&() array is used to remember where each line encountered so far in the file begins, and 16384 is the maximum number of elements that can fit into a single 64K array. For a typical text file with line lengths that average 60 characters, 16384 lines is nearly 1MB of text.

When you run the program, it expects only the up and down arrow keys to advance and go backwards through the file, the Home key to jump to the beginning, or the Escape key to end the program. Notice that the words "blank line" are printed when a blank line is encountered, just so you can see that something has happened.

```

DEFINT A-Z
CONST MaxLines% = 5000
REDIM Offset&(1 TO MaxLines%)

CLS
PRINT "Enter the name of file to browse: ";
LINE INPUT "", FileName$

OPEN FileName$ FOR INPUT AS #1

Offset&(1) = 1           'initialize to offset 1
CurLine = 1            'and start with line 1

WHILE Action$ <> CHR$(27) 'until they press Escape
  SEEK #1, Offset&(CurLine) 'seek to the current line
  LINE INPUT #1, Text$      'read that line
  Offset&(CurLine + 1) = SEEK(1) 'save where the next
                              ' line starts

  CLS
  IF LEN(Text$) THEN        'if it's not blank
    PRINT Text$            'print the line
  ELSE                      'otherwise
    PRINT "(blank line)"   'show that it's blank
  END IF

  DO                        'wait for a key
    Action$ = INKEY$
  LOOP UNTIL LEN(Action$)

  SELECT CASE ASC(RIGHT$(Action$, 1))
    CASE 71                 'Home
      CurLine = 1

    CASE 72                 'Up arrow
      IF CurLine > 1 THEN
        CurLine = CurLine - 1
      END IF

```

```

CASE 80                                'Down arrow
  IF (NOT EOF(1)) AND CurLine < MaxLines% THEN
    CurLine = CurLine + 1
  END IF

CASE ELSE
END SELECT
WEND
CLOSE
END

```

You should be aware that BASIC does not prevent you from using SEEK to go past the end of a file that has been opened for Binary access. If you do this and then write any data, DOS will actually extend the file to include the data that was just written. Therefore, it is important to understand that any data that lies between the previous end of the file and the newly added data will be undefined. When a file is deleted DOS simply abandons the sectors that held its data, and makes them available for later use. But whatever data those sectors contained remains intact. When you later expand a file this way using SEEK, the old abandoned sector contents are incorporated into the file. Even if the sectors that are allocated were never written to previously, they will contain the &HF6 bytes that DOS' FORMAT.COM uses to initialize a disk.

You can turn this behavior into an important feature, and in some cases recreate a file that was accidentally truncated. If you erase a file by mistake, it is possible to recover it using the Norton Utilities or a similar disk utility program. But when an existing file is opened for output, DOS truncates it to a length of zero. The following program shows the steps necessary to reconstruct a file that has been destroyed this way.

```

OPEN FileName$ FOR BINARY AS #1
SEEK #1, 30000
PUT #1, , X%
CLOSE #1

```

In this case, the file is restored to a length of 30000, and you can use larger or smaller values as appropriate. Understand that there is no guarantee that DOS will reassign the same sectors to the file that it originally used. But I have seen this trick work more than once, and it is at least worth a try.

In a similar fashion, you can reduce the size of a file by seeking to a given location and then writing zero bytes there. Since BASIC provides no way to write zero bytes to a file, some additional trickery is needed. This will be described in Chapter 11 in the section that discusses using CALL Interrupt to access DOS and BIOS services.

Advanced File Techniques

There are a number of clever file-related tricks that can be performed using only BASIC programming. Some of these tricks help you to improve on BASIC's speed, and others let you do things that are not possible using the normal and obvious methods. BASIC is no slower than other languages when reading and writing large amounts of data, and indeed, the bottleneck is frequently DOS itself. Further,

if you can reduce the amount of data that is written, your files will be smaller as well. With that in mind, let's look at some ways to further improve your programs.

Speeding Up File Access

The single most important way to speed up your programs is to read and write large amounts of data in one operation. The normal method for saving a numeric or TYPE array is to write each element to disk in a loop. But when there are many thousands of elements, a substantial amount of overhead is incurred just from BASIC's repeated calls to DOS. There are several solutions you can consider, each with increasing levels of complexity.

BLOAD and BSAVE

The simplest way to read and write a large amount of contiguous data is with BLOAD and BSAVE. BSAVE takes a "snapshot" of any contiguous area of memory up to 64K in size, and saves it to disk in a single operation. When an application calls DOS to read or write a file, it furnishes DOS with the segment and address where the data is to be loaded or saved from, and also the number of bytes. BLOAD and BSAVE provide a simple interface to the DOS read and write services, and they can be used to load and save numeric arrays up to 64K in size, as well as screen images.

```
I have seen a number of messages in the MS-BASIC forum
on CompuServe stating that BSAVE and BLOAD do not work
with compressed disks. Many of those messages have come
from Microsoft technical support, and I have no reason
to doubt them. It may be that only VB/DOS has this
problem, but I have no way to test QB and PDS because I
don't use disk compression.
```

A file that has been written using BSAVE includes a 7-byte header that identifies it as a BSAVE file, and also shows where it was saved from and how many bytes it contains. BLOAD requires this header, and thus cannot be used with any arbitrary type of file. But when used together, these commands can be as much as ten times faster than a FOR/NEXT loop.

The example below creates and then saves a single precision array, and then loads it again to prove the process worked.

```
DEFINT A-Z
CONST NumEls% = 20000
REDIM Array(1 TO NumEls%)           'create the array

FOR X = 1 TO NumEls%               'file it with values
  Array(X) = X
NEXT

DEF SEG = VARSEG(Array(1))         'set the BSAVE segment
BSAVE "ARRAY.DAT", VARPTR(Array(1)), NumEls% * LEN(Array(1))
```

```

REDIM Array(1 TO NumEls%)           'recreate the array
DEF SEG = VARSEG(Array(1))         'the array may have moved
BLOAD "ARRAY.DAT", VARPTR(Array(1))

FOR X = 1 TO NumEls%               'prove the data is valid
IF Array(X) <> X THEN
    PRINT "Error in element"; X
END IF
NEXT
END

```

Because BSAVE and BLOAD use the current DEF SEG setting to know the segment the data is in, VARSEG is used with the first element of the array. Once the correct segment has been established, BSAVE is given the name of the file to save, the starting address, and the number of bytes of data. As with the TYPE variable example shown earlier, LEN is ideal here as well to help calculate the number of bytes that must be saved. In this case, each integer array element is two bytes long, and BASIC multiplies the constants NumEls% and LEN(Array(1)) when the program is compiled. Therefore, no additional code is added to the program to calculate this value at runtime.

Once the array has been saved it is redimensioned, which effectively clears it to all zero values prior to reloading. Notice that DEF SEG is used again before the BLOAD statement. This is an important point, because there is no guarantee that BASIC will necessarily allocate the same block of memory the second time. If a file is loaded into the wrong area of memory, your program is sure to crash or at least not work correctly.

Also note that BLOAD always loads the entire file, and a length argument is not needed or expected. This brings up an important issue: how can you determine how large to dimension an array prior to loading it? The answer, as you may have surmised, is to open the file for binary access and read the length stored in the BSAVE header. All that's needed is to know how the header is organized, as the following program reveals.

```

DEFINT A-Z
TYPE BHeader
    Header AS STRING * 1
    Segment AS INTEGER
    Address AS INTEGER
    Length AS INTEGER
END TYPE
DIM BLHeader AS BHeader

OPEN "ARRAY.DAT" FOR BINARY AS #1
GET #1, , BLHeader
CLOSE

IF ASC(BLHeader.Header) <> &HFD THEN
    PRINT "Not a valid BSAVE file"
END
END IF

LongLength& = BLHeader.Length
IF LongLength& < 0 THEN
    LongLength& = LongLength& + 65536

```

```

END IF

NumElements = LongLength & \ 2
REDIM Array(1 TO NumElements)

DEF SEG = VARSEG(Array(1))
BLOAD "ARRAY.DAT", VARPTR(Array(1))
END

```

Even though the original segment and address from which the file was saved is in the BSAVE header, that information is not used here. In most situations you will always provide BLOAD with an address to load the file to. However, if the address is omitted, BASIC uses the segment and address stored in the file, and ignores the current DEF SEG setting. This would be useful when handling text and graphics images which are always loaded to the same segment from which they were originally saved. But in general I recommend that you always define an explicit segment and address.

There are a few other points worth elaborating on as well. First, the program examines the first byte in the file to be sure it is the special value &HFD which identifies a BSAVE file. The ASC function is required for that, since the only way to define a TYPE component one byte long is as a string.

Second, the length is stored as an unsigned integer, which cannot be manipulated directly in a BASIC program if its value exceeds 32767. As you learned in Chapter 2, integer values larger than 32767 are treated by BASIC as signed, and in this case they are considered negative. Therefore, the value is first assigned to a long integer, which is then tested for a value less than zero. If it is indeed negative, 65536 is added to the variable to convert it to an equivalent positive number. Note that the length in a BSAVE header does not include the header length; only the data itself is considered.

If you single-step through this program after running the earlier one that created the file, you will see that the code that adds 65536 is executed, because the header shows that the file contains 40000 bytes.

There are two limitations to using BSAVE and BLOAD this way. One problem is that you may not want the header to be attached to the file. The other, more important problem is that BASIC allows arrays to exceed 64K. Saving a single huge array in multiple files is clumsy, and contributes to the clutter on your disks. The header issue is less important, because you can always access the file with normal binary statements after using a SEEK to skip over the header. But the huge array problem requires some heavy ammunition.

One final point worth mentioning is that BSAVE and BLOAD assume a .BAS file name extension if none is given. This is incredibly stupid, since the contents of a BSAVE file have no relationship to a BASIC source file. Therefore, to save a file with no extension at all you must append a period to the name: BSAVE "MYFILE.", Address, Length.

Beyond BSAVE

The program that follows includes both a demonstration and a pair of subprograms that let you save any data regardless of its size or location. These routines are primarily intended for saving huge numeric and TYPE arrays, but there is no reason they couldn't be used for other purposes. However,

they cannot be used with conventional variable-length string arrays, because the data in those arrays is not contiguous. The file is processed in 16K blocks using multiple passes, and the actual saving and loading is performed by calling BASIC's internal PUT # and GET # routines.

```

DEFINT A-Z
'NOTE: This program must be compiled with the /ah option.

DECLARE SUB BigLoad (FileName$, Segment, Address, Bytes&)
DECLARE SUB BigSave (FileName$, Segment, Address, Bytes&)
DECLARE SUB BCGet ALIAS "B$GET3" (BYVAL FileNum, BYVAL Segment, _
    BYVAL Address, BYVAL NumBytes)
DECLARE SUB BCPut ALIAS "B$PUT3" (BYVAL FileNum, BYVAL Segment, _
    BYVAL Address, BYVAL NumBytes)

CONST NumEls% = 20000
REDIM Array&(1 TO NumEls%)
NumBytes& = LEN(Array&(1)) * CLNG(NumEls%)

FOR X = 1 TO NumEls%           'fill the array
    Array&(X) = X
NEXT

Segment = VARSEG(Array&(1))   'save the array
Address = VARPTR(Array&(1))
CALL BigSave("ARRAY.DAT", Segment, Address, NumBytes&)

REDIM Array&(1 TO NumEls%)   'clear the array
Segment = VARSEG(Array&(1))   'reload the array
Address = VARPTR(Array&(1))
CALL BigLoad("ARRAY.DAT", Segment, Address, NumBytes&)

FOR X = 1 TO NumEls%         'prove this all worked
    IF Array&(X) <> X THEN
        PRINT "Error in element"; X
    END IF
NEXT
END

SUB BigLoad (FileName$, DataSeg, Address, Bytes&) STATIC
    FileNum = FREEFILE
    OPEN FileName$ FOR BINARY AS #FileNum
    NumBytes& = Bytes&
    Segment = DataSeg
    'work with copies to
    'protect the parameters

    DO
        IF NumBytes& > 16384 THEN
            CurrentBytes = 16384
        ELSE
            CurrentBytes = NumBytes&
        END IF
        CALL BCGet(FileNum, Segment, Address, CurrentBytes)
        NumBytes& = NumBytes& - CurrentBytes
        Segment = Segment + &H400
    LOOP WHILE NumBytes&

    CLOSE #FileNum
END SUB

SUB BigSave (FileName$, DataSeg, Address, Bytes&) STATIC
    FileNum = FREEFILE

```

```

OPEN FileName$ FOR BINARY AS #FileNum
NumBytes& = Bytes&           'work with copies to
Segment = DataSeg           'protect the parameters

DO
  IF NumBytes& > 16384 THEN
    CurrentBytes = 16384
  ELSE
    CurrentBytes = NumBytes&
  END IF
  CALL BCPut (FileNum, Segment, Address, CurrentBytes)
  NumBytes& = NumBytes& - CurrentBytes
  Segment = Segment + &H400
LOOP WHILE NumBytes&
CLOSE #FileNum
END SUB

```

Although BASIC lets you save and load only single variables or array elements, its internal library routines can work with data of nearly any size. And since TYPE variables can be as large as 64K, these routines must be able to accommodate data at least that big. Therefore, BASIC's usual restriction on what you can and cannot read or write to disk with GET # and PUT # is an arbitrary one.

Accessing BASIC's internal routines requires that you declare them using ALIAS, since it is illegal to call a routine that has a dollar sign in its name. As you can see, these routines expect their parameters to be passed by value, and this is handled by the DECLARE statements. Normally, you cannot call these routines from within the QB editing environment. But if you separate the two subprograms and place them into a different module, that module can be compiled and added to a Quick Library. That is, the subprograms can be together in one file, but not with the demo that calls them. Be sure to add the two DECLARE statements that define B\$PUT3 and B\$GET3 to that module as well.

The long integer array this program creates exceeds the normal 64K limit, so the /ah compiler switch must be used. Notice in the BigLoad and BigSave subprograms that copies are made of two of the incoming parameters. If this were not done, the subprograms would change the passed values, which is a bad practice in this case. Also, notice how the segment value that is used for saving and loading is adjusted through each pass of the DO loop. Since the data is saved in 16K blocks, the segment must be increased by $16384 \div 16 = 1024$ for each pass. The use of an equivalent &H value here is arbitrary; I translated this program from another version written in assembly language that used Hex for that number.

Processing Large Files

Although the solutions shown so far are valuable when saving or loading large amounts of data, that is as far as they go. In many cases you will also need to process an entire existing file. Some examples are a program that copies or encrypts files, or a routine that searches an entire file for a string of text. As with saving and loading files, processing a file or portion of a file in large blocks is always faster and more effective than processing it line by line.

The file copying subprogram below accepts source and destination file names, and copies the data in 4K blocks. The 4K size is significant, because it is large enough to avoid many repeated calls to DOS, and small enough to allow a conventional string to be used as a file buffer. As with the BigLoad and BigSave routines, the file is processed in pieces. Also, for simplicity a complete file name and path is required. Although the DOS COPY command lets you use a source file name and a destination drive or path only, the CopyFile subprogram requires that entire file names be given for both.

```

DEFINT A-Z
DECLARE SUB CopyFile (InFile$, OutFile$)

SUB CopyFile (InFile$, OutFile$) STATIC
  File1 = FREEFILE
  OPEN InFile$ FOR BINARY AS #File1

  File2 = FREEFILE
  OPEN OutFile$ FOR BINARY AS #File2

  Remaining& = LOF(File1)
  DO
    IF Remaining& > 4096 THEN
      ThisPass = 4096
    ELSE
      ThisPass = Remaining&
    END IF
    Buffer$ = SPACE$(ThisPass)
    GET #File1, , Buffer$
    PUT #File2, , Buffer$
    Remaining& = Remaining& - ThisPass
  LOOP WHILE Remaining&
  CLOSE File1, File2
END SUB

```

Once the basic structure of a routine that processes an entire file has been established, it can be easily modified for other purposes. For example, CopyFile can be altered to encrypt an entire file, search a file for a text string, and so forth. A few of these will be shown here. Note that for simplicity and clarity, CopyFile creates a new buffer with each pass through the loop. You could avoid that by preceding the assignment with `IF LEN(Buffer$) <> ThisPass THEN` or similar logic, to avoid creating the buffer when it already exists and is the correct length. The BufIn function and example below serves as a very fast LINE INPUT replacement. Even though BASIC's own file input routines provide buffering for increased speed, they are not as effective as this function. In my measurements I have found BufIn to be consistently four to five times faster than BASIC's LINE INPUT routine when reading large (greater than 50K) files. With smaller files the improvement is less, but still substantial.

```

DEFINT A-Z
DECLARE FUNCTION BufIn$ (FileName$, Done)

LINE INPUT "Enter a file name: ", FileName$

'---- Show how fast BufIn$ reads the file.
Start! = TIMER
DO
  This$ = BufIn$(FileName$, Done)
  IF Done THEN EXIT DO

```

```

LOOP
Done! = TIMER
PRINT "Buffered input: "; Done! - Start!

'----- Now show how long BASIC's LINE INPUT takes.
Start! = TIMER
OPEN FileName$ FOR INPUT AS #1
DO
  LINE INPUT #1, This$
LOOP UNTIL EOF(1)
Done! = TIMER
PRINT " BASIC's INPUT: "; Done! - Start!
CLOSE
END

FUNCTION BufIn$ (FileName$, Done) STATIC
IF Reading GOTO Process      'now reading, jump in

'----- initialization
Reading = -1                 'not reading so start now
Done = 0                     'clear Done just in case
CR$ = CHR$(13)               'define for speed later

FileNum = FREEFILE          'open the file
OPEN FileName$ FOR BINARY AS #FileNum

Remaining& = LOF(FileNum)    'byte count to be read
IF Remaining& = 0 GOTO ExitFn 'empty or nonexistent file

BufSize = 4096               'bytes to read each pass
Buffer$ = SPACE$(BufSize)    'assume BufSize bytes

DO                            'the main outer loop
  IF Remaining& < BufSize THEN 'read only what remains
    BufSize = Remaining&      'resize the buffer
    IF BufSize < 1 GOTO ExitFn 'possible only if EOF byte
    Buffer$ = SPACE$(BufSize)  'create the file buffer
  END IF
  GET #FileNum, , Buffer$      'read a block

  BufPos = 1                  'start at the beginning
  DO                          'walk through buffer
    CR = INSTR(BufPos, Buffer$, CR$) 'look for a Return
    IF CR THEN                 'we found one
      SaveCR = CR              'save where
      BufIn$ = MID$(Buffer$, BufPos, CR - BufPos)
      BufPos = CR + 2          'skip inevitable LF
      EXIT FUNCTION           'all done for now
    ELSE                       'back up in the file
      '----- if at the end and no CHR$(13) was found
      ' return what remains in the string
      IF SEEK(FileNum) >= LOF(FileNum) THEN
        Output$ = MID$(Buffer$, SaveCR + 2)
        '----- trap a trailing EOF marker
        IF RIGHT$(Output$, 1) = CHR$(26) THEN
          Output$ = LEFT$(Output$, LEN(Output$) - 1)
        END IF
        BufIn$ = Output$      'assign the function
        GOTO ExitFn          'and exit now
      END IF
    END IF
  END IF
END IF

```

```

        Slop = BufSize - SaveCR - 1      'calc buffer excess
        Remaining& = Remaining& + Slop 'calc file excess
        SEEK #FileNum, SEEK(FileNum) - Slop
    END IF

Process:
    LOOP WHILE CR                      'while more in buffer
        Remaining& = Remaining& - BufSize

    LOOP WHILE Remaining&              'while more in the file

ExitFn:
    Reading = 0                        'we're not reading anymore
    Done = -1                          'show that we're all done
    CLOSE #FileNum                     'final clean-up
END FUNCTION

```

As you can see, the BufIn function opens the file, reads each line of text, and then closes the file and sets a flag when it has exhausted the text. Even though this example shows BufIn being invoked in a DO loop, it can be used in any situation where LINE INPUT would normally be used. As long as you declare the function, it may be added to programs of your own and used when sequential line-oriented data must be read as quickly as possible.

I don't think each statement in the BufIn function warrants a complete explanation, but some of the less obvious aspects do. BufIn operates by reading the file in 4K blocks in an outer loop, and each block is then examined for a CHR\$(13) line terminator in an inner loop that uses INSTR. INSTR happens to be extremely fast, and it is ideal when used this way to search a string for a single character.

The only real complication is when a portion of a string is in the buffer, because that requires seeking backwards in the file to the start of the string. Other, less important complications that also must be handled arise from the presence of a CHR\$(26) EOF marker, and a final string that has no terminating carriage return.

I have made every effort to make this function as bullet-proof as possible; however, it is mandatory that every carriage return in the file be followed by a corresponding line feed. Some word processors eliminate the line feed to indicate a *soft return* at the end of a line, as opposed to the *hard return* that signifies the end of a paragraph. Most word processor files use a non-standard format anyway, so that should not be much of a problem.

The last complete program I'll present here is called TEXTFIND.BAS, and it searches a group of files for a specified string. TEXTFIND is particularly useful when you need to find a document, and cannot remember its name. If you can think of a snippet of text the file might contain, TEXTFIND will identify which files contain that text, and then display it in context.

```

'----- TEXTFIND.BAS

'Copyright (c) 1991 by Ethan Winer

DEFINT A-Z

TYPE RegTypeX                      'used by CALL Interrupt

```

```

AX      AS INTEGER
BX      AS INTEGER
CX      AS INTEGER
DX      AS INTEGER
BP      AS INTEGER
SI      AS INTEGER
DI      AS INTEGER
Flags  AS INTEGER
DS      AS INTEGER
ES      AS INTEGER
END TYPE
DIM Registers AS RegTypeX      'holds the CPU registers

TYPE DTA                        'used by DOS services
  Reserved AS STRING * 21      'reserved for use by DOS
  Attribute AS STRING * 1      'the file's attribute
  FileTime AS STRING * 2       'the file's time
  FileDate AS STRING * 2       'the file's date
  FileSize AS LONG             'the file's size
  FileName AS STRING * 13      'the file's name
END TYPE
DIM DTADData AS DTA

DECLARE SUB InterruptX (IntNumber, InRegs AS RegTypeX, OutRegs AS RegTypeX)
CONST MaxFiles% = 1000
CONST BufMax% = 4096

REDIM Array$(1 TO MaxFiles%)   'holds the file names
Zero$ = CHR$(0)                'do this once for speed

'----- This function returns the larger of two integers.
DEF FNMax% (Value1, Value2)
  FNMax% = Value1
  IF Value2 > Value1 THEN FNMax% = Value2
END DEF

'----- This function loads a group of file names.
DEF FNLoadNames%
  STATIC Count

  '---- define a new Data Transfer Area for DOS
  Registers.DX = VARPTR(DTADData)
  Registers.DS = VARSEG(DTADData)
  Registers.AX = &H1A00
  CALL InterruptX(&H21, Registers, Registers)

  Count = 0                      'zero the file counter
  Spec$ = Spec$ + Zero$          'DOS needs an ASCIIIZ string
  Registers.DX = SADD(Spec$)      'show where the spec is
  Registers.DS = SSEG(Spec$)      'use this with PDS
  'Registers.DS = VARSEG(Spec$)  'use this with QB
  Registers.CX = 39               'the attribute for any file
  Registers.AX = &H4E00           'find file name service

  '---- Read the file names that match the search specification. The Flags
  ' registers indicates when no more matching files are found. Copy
  ' each file name to the string array. Service &H4F is used to
  ' continue the search started with service &H4E using the same file
  ' specification.
  DO
    CALL InterruptX(&H21, Registers, Registers)

```

```

    IF Registers.Flags AND 1 THEN EXIT DO
    Count = Count + 1
    Array$(Count) = DTADData.FileName
    Registers.AX = &H4F00
LOOP WHILE Count < MaxFiles%

FNLoadNames% = Count      'return the number of files
END DEF

'----- The main body of the program begins here.
PRINT "TEXTFIND Copyright (c) 1991, Ziff-Davis Press."
PRINT

'---- Get the file specification, or prompt for one if it wasn't given.
Spec$ = COMMAND$
IF LEN(Spec$) = 0 THEN
    PRINT "Enter a file specification: ";
    INPUT "", Spec$
END IF

'----- Ask for the search string to find.
PRINT "    Enter the text to find: ";
INPUT Find$
PRINT

Find$ = UCASE$(Find$)      'ignore capitalization
FindLength = LEN(Find$)   'see how long Find$ is
IF FindLength = 0 THEN END

Count = FNLoadNames%      'load the file names
IF Count = 0 THEN
    PRINT "No matching files"
    END
END IF

'----- Isolate the drive and path if given.
FOR X = LEN(Spec$) TO 1 STEP -1
    Char = ASC(MID$(Spec$, X))
    IF Char = 58 OR Char = 92 THEN    ":" or "\"
        Path$ = LEFT$(UCASE$(Spec$), X)
        EXIT FOR
    END IF
NEXT

FOR X = 1 TO Count        'for each matching file
    Array$(X) = LEFT$(Array$(X), INSTR(Array$(X), Zero$) - 1)
    PRINT "Reading "; Path$; Array$(X)
    OPEN Path$ + Array$(X) FOR BINARY AS #1
    Length& = LOF(1)      'get and save its length
    IF Length& < FindLength GOTO NextFile

    BufSize = BufMax%    'assume a 4K text buffer
    IF BufSize > Length& THEN BufSize = Length&
    Buffer$ = SPACE$(BufSize) 'create the file buffer

    LastSeek& = 1        'seed the SEEK location
    BaseAddr& = 1        'and the starting offset
    Bytes = 0            'how many bytes to search

    DO                    'the file read loop
        BaseAddr& = BaseAddr& + Bytes 'track block start

```

```

IF Length& - LastSeek& + 1 >= BufSize THEN
  Bytes = BufSize      'at least BufSize bytes left
ELSE
  Bytes = Length& - LastSeek& + 1
  Buffer$ = SPACE$(Bytes) 'adjust the buffer size
END IF

SEEK #1, LastSeek&      'seek back in the file
GET #1, , Buffer$       'read a chunk of the file

Start = 1               'this is the INSTR loop for
DO                      'searching within the buffer
  Found = INSTR(Start, UCASE$(Buffer$), Find$)
  IF Found THEN        'print it in context
    Start = Found + 1  'to resume using INSTR later
    PRINT              'add a blank line for clarity
    PRINT MID$(Buffer$, FNMax%(1, Found - 20), FindLength + 40)
    PRINT

    PRINT "Continue searching "; Array$(X);
    PRINT "? (Yes/No/Skip): ";
    WHILE INKEY$ <> "": WEND 'clear kbd buffer
    DO
      KeyHit$ = UCASE$(INKEY$) 'then get a response
      LOOP UNTIL KeyHit$ = "Y" OR KeyHit$ = "N" OR KeyHit$ = "S"
PRINT KeyHit$          'echo the letter
    PRINT

    IF KeyHit$ = "N" THEN      '"No"
      END                    'end the program
    ELSEIF KeyHit$ = "S" THEN '"Skip"
      GOTO NextFile          'go to the next file
    END IF

  END IF

LOOP WHILE Found        'search for multiple hits
                        'within the file buffer
IF Bytes = BufSize THEN 'still more file to examine
  '---- Back up a bit in case Find$ is there but straddling the buffer
  '      boundary. Then update the internal SEEK pointer.
  BaseAddr& = BaseAddr& - FindLength
  LastSeek& = BaseAddr& + Bytes
END IF

LOOP WHILE Bytes = BufSize AND BufSize = BufMax%

NextFile:
CLOSE #1
Buffer$ = ""           'clear the buffer for later

NEXT
END

```

TEXTFIND may be run either in the BASIC editor or compiled to an executable file and then run. If you are using QuickBASIC you will need either QB.QLB or QB.LIB because the program relies on CALL Interrupt to interface with DOS. To start QB and load the QB.QLB library simply enter qb /1. If you are compiling the program, specify the QB.LIB file when it is linked:

```
link textfind , , nul , qb;
```

For BASIC 7 users the appropriate library names are QBX.QLB and QBX.LIB respectively, and for VB/DOS the libraries are VBDOS.QLB and VBDOS.LIB.

When you run TEXTFIND you may either enter a file specification such as *.BAS or LET*.TXT or the like as a command line argument, or enter nothing and let the program prompt you. In either case, you will then be asked to enter the text string you're searching for. TEXTFIND will search through every file that matches the file specification, and display the string in context if it is found.

As written, TEXTFIND shows the 20 characters before and after the string. You may of course modify that to any reasonable number of characters. Simply change the 20 and 40 values in the corresponding PRINT statement. The first value is the number of characters on either side to display, and the second must be twice that to accommodate the length of the search string itself. Note the use of FNMax% which ensures that the program will not try to print characters before the start of the buffer. If the text were found at the very start of the file, attempting to print the 20 characters that precede it will create an "Illegal function call" error at the MID\$ function.

Each time the string is found and displayed you are offered the opportunity to continue searching the same file, ending the program, or skipping to the next file.

Although CALL Interrupt will be discussed in depth in Chapter 11, there are several aspects of the program's operation that require elaboration here. First, any program that uses the DOS Find First and Find Next services to read a list of file names must establish a small block of memory as a Disk Transfer Area (DTA). The DTA holds pertinent information about each file that is found, such as its date, time, size, and attribute. In this case, though, we are merely interested in each file's name. DOS service &H1A is used to assign the DTA to a TYPE variable that is designed to facilitate extracting this information. BASIC PDS, and VB/DOS, include the DIR\$ function which lets you read file names, but I have used CALL Interrupt here so the program will also work with QuickBASIC.

Second, DEF FN-style functions are used instead of formal functions because they are smaller and slightly faster. The FNLoadNames function is responsible for loading all of the file names into the string array, and it returns the number of files that were found. After each call to DOS to find the next matching name, the Carry flag is tested. DOS often uses the carry flag to indicate the success or failure of an operation, and in this case it is set to True when there are no more files.

Note how a CHR\$(0) is appended to the file specification when calling DOS, to indicate the end of the string. Similarly, DOS returns each file name terminated with a zero byte, and INSTR is used to find that byte. Then, only those characters to the left of the zero are kept using LEFT\$.

Third, the block of code that isolates the drive and path name if given is needed because the DOS Find services return only a file name. If you enter D:\ANYDIR*. * as a file specification, that is then passed to DOS. But DOS returns only the names it finds that match the specification. Therefore, the drive and path must be added to the beginning of each name, to create a complete file name for the subsequent OPEN command.

Finally, as with the BufIn function, the files are read in 4K (4096-byte) blocks, except for the last block which of course may be smaller. A smaller block is also used when the file is less than 4K in length. Within each outer read loop, an inner loop is employed to search for the text, and again INSTR is used because of its speed. As written, TEXTFIND looks for the specified string without regard to capitalization. You can remove that feature by eliminating the UCASE\$ function in both the INSTR loop, and at the point in the program where Find\$ is capitalized.

Minimizing Disk Usage

While improving your program's performance is certainly a desirable pursuit, equally important is minimizing the amount of space needed to store data. Besides the obvious savings in disk space, the less data there is, the faster it can be loaded and saved. There are a number of simple tricks you can use to reduce the size of your data files, and some types of data lend themselves quite nicely to compaction techniques.

Date information is particularly easy to reduce. At the minimum, you should remove the separating slashes or dashes—perhaps with a dedicated function. For example, you would convert "06-22-91" to "062291". Even better, however, is to convert each digit pair to an equivalent single CHR\$() byte, and also swap the order of the digits. That is, the date above would be packed to CHR\$(91) + CHR\$(6) + CHR\$(22). By placing the year first followed by the month and then the day, dates may also be compared. Otherwise, a normal string comparison would show the date "01-01-91" as being less (earlier) than "12-31-90" even though it is in fact greater (later). A complementary function would then extract the ASCII values into a date string suitable for display. These are shown below.

```
DEFINT A-Z
DECLARE FUNCTION PackDate$ (D$)
DECLARE FUNCTION UnPackDate$ (D$)

D$ = "03-22-91"
Packed$ = PackDate$(D$)
UnPacked$ = UnPackDate$(Packed$)

PRINT D$
PRINT Packed$
PRINT UnPacked$
END

FUNCTION PackDate$ (D$) STATIC
    Year = VAL(RIGHT$(D$, 2))
    Month = VAL(LEFT$(D$, 2))
    Day = VAL(MID$(D$, 4, 2))
    PackDate$ = CHR$(Year) + CHR$(Month) + CHR$(Day)
END FUNCTION

FUNCTION UnPackDate$ (D$) STATIC
    Month$ = LTRIM$(STR$(ASC(MID$(D$, 2, 1))))
    Day$ = LTRIM$(STR$(ASC(RIGHT$(D$, 1))))
    Year$ = LTRIM$(STR$(ASC(LEFT$(D$, 1))))
    UnPackDate$ = RIGHT$("0" + Month$, 2) + "-" + RIGHT$("0" + Day$, 2) + _
```



```
"-" + RIGHT$("0" + Year$, 2)
END FUNCTION
```

Because the compacted dates will likely contain a CHR\$(26) byte which is used by DOS and BASIC as an EOF marker, this method is useful only with random access and binary data files. But since it is usually large database files that need the most help anyway, these functions are ideal.

Another useful database compaction technique is to replace selected strings with an equivalent integer or byte value. The commercial database program *DataEase* uses a very clever trick to implement multiple choice fields. It is not uncommon to have a string field that contains, say, an income or expense category. For example, most businesses are required to indicate the purpose of each check that is written. Instead of using a string field and requiring the operator to type Entertainment, Payroll, or whatever, a menu can be popped up showing a list of possible choices.

Assuming there are no more than 256 possibilities, the choice number that was entered can be stored on disk in a single byte. You would use something like `FileType.Choice = CHR$(MenuChoice)`, where the Choice portion of the file type was defined as `STRING * 1`. Then to extract the choice after a record was read you would use `MenuChoice = ASC(FileType.Choice)`.

Some database programs support Memo Fields, whereby the user can enter a varying amount of memo information. Since database files almost always use a fixed length for each record, this presents a programming dilemma: How much space do you set aside for the memo field? If you set aside too little, the user won't be very pleased. But setting aside enough to accommodate the longest possible string is very wasteful of disk space.

One good solution is to store a long integer pointer in each record, and keep the memos themselves in a separate file. A long integer requires only four bytes of storage, yet it can hold a seek location for memo data kept in a separate file whose size can be greater than 2000 MB! As each new memo is entered, the current length—derived using LOF—of the memo file is written in the current record of the data file. The memo string is then appended to the memo file. When you want to retrieve the memo, simply seek to the long integer offset held in the main data record and use LINE INPUT to read the string from the memo file.

The only real complication with this method is when a memo field must be edited. There's no reasonable way to lengthen or shorten data in the middle of a file, and no reasonable program would even try. Instead, you would simply overwrite the existing data with special values—perhaps with CHR\$(255) bytes—and then append the new memo to the end of the file. Periodically you would have to run a utility program that copied only the valid memo fields to a new file, and then delete the old file. Be aware that you will also have to update the long integer pointers in the main data file, to reflect the new offsets of their corresponding memo fields.

The last data size reduction technique is probably the simplest of all, and that is to use the appropriate type of data and file access method. If you can get by with a single precision variable, don't use a double precision. And if the range of integer values is sufficient, use those. Many programmers

automatically use single precision variables without even thinking about it, when a smaller data type would suffice.

Finally, avoid using sequential files to store numeric data. As I already pointed out, an integer can be stored in a binary file in only two bytes—no matter what its value—compared to as many as eight bytes needed to store the equivalent digits, possible minus sign, and a terminating carriage return and line feed. Be creative, and don't be afraid to invent a method that is suited to your particular application. The Lotus format is a good one for many other applications, whereby a size and type code precedes each piece of information. If your needs are modest you can probably get away with a single byte as a type code, further reducing the amount of storage that is needed.

Avoiding BASIC's Limitations

So far I have focused on improving what BASIC already does. I showed techniques for speeding up file accesses, and reducing the size of your data. I even showed how to overcome BASIC's unwillingness to directly write binary data larger than a single variable. But there are other BASIC limitations that can be overcome as well.

One important limitation is that BASIC lets you run only .EXE files with the RUN statement. If you need to execute a .COM program or a batch file, BASIC will not let you. However you can trick DOS into believing a .COM program or batch file's name was entered at the DOS prompt. The StuffBuffer subprogram shown below inserts a string of up to 15 characters directly into the keyboard buffer. It works by poking each character one by one into the buffer address in low memory. Thus, when your program ends the characters are there as if someone had typed them manually.

```
DEFINT A-Z
DECLARE SUB StuffBuffer (Cmd$)

SUB StuffBuffer (Cmd$) STATIC
'----- Limit the string to 14 characters plus Enter and save the length.
Work$ = LEFT$(Cmd$, 14) + CHR$(13)
Length = LEN(Work$)

'----- Set the segment for poking, define the buffer head and tail, and
'         then poke each character.
DEF SEG = 0
POKE 1050, 30
POKE 1052, 30 + Length * 2
FOR X = 1 TO Length
  POKE 1052 + X * 2, ASC(MID$(Work$, X))
NEXT
END SUB
```

To run a .COM program or batch file simply call StuffBuffer and end the program:

```
CALL StuffBuffer("PROGRAM"): END
```

A terminating carriage return is added to the command, to include a final Enter keypress. Because the keyboard buffer holds only 15 characters, you cannot specify long path names when using StuffBuffer. However, you can easily open and write a short batch file with the complete path and file name, and run the batch file instead.

Notice that this technique will not work if the original BASIC program itself has been run from a batch file, because that batch file gains control when the program ends. Also, when creating and running a batch file that will be run by StuffBuffer, it is imperative that the last line not have a terminating carriage return. The short example below shows the correct way to create and run a batch file for use with StuffBuffer.

```
OPEN "MYBAT.BAT" FOR OUTPUT AS #1
  PRINT #1, "cd \somedir"
  PRINT #1, "someprog";
CLOSE
CALL StuffBuffer("MYBAT")
END
```

You can also have the batch file re-run the BASIC program by entering its name as the last line in the batch file. In that case you would include the semicolon at the end of that line, instead of the line that runs the program. Note that StuffBuffer is an ideal replacement for BASIC's SHELL command, because with SHELL your BASIC program remains in memory while the subsequent program is run. Using StuffBuffer with a batch file removes the BASIC program entirely, thus freeing up all available system memory for the program being run.

Understand that StuffBuffer cannot be used to activate a TSR or other program that monitors keyboard interrupt 9. This limitation also extends to the special key sequences that enable the Turbo mode on some PC compatibles, and simulating Ctrl-Esc to activate the DOS compatibility box of OS/2. Programs that look for these special keys insert themselves into the keyboard chain before the keyboard buffer, and act on them before the BIOS has the chance to store them in the buffer.

Another BASIC limitation is that only 15 files may be open at one time. In truth, this is really a DOS limitation, and indeed, the fix requires a DOS interrupt service. It is also possible to reduce the number of files open at once by combining data. For example, the BASIC PDS ISAM file manager uses this technique to store both the data and its indexes all in the same file. But doing that requires more complication than many programmers are willing to put up with.

The program below shows how to increase the number of files that DOS will let you open. Be aware that the DOS service that performs this magic requires at least version 3.3, and this program tests for that.

```
DEFINT A-Z
DECLARE SUB Interrupt (IntNum, InRegs AS ANY, OutRegs AS ANY) DECLARE SUB
MoreFiles (NumFiles)
DECLARE FUNCTION DOSVer% ()
```

```

TYPE RegType
  AX    AS INTEGER
  BX    AS INTEGER
  CX    AS INTEGER
  DX    AS INTEGER
  BP    AS INTEGER
  SI    AS INTEGER
  DI    AS INTEGER
  Flags AS INTEGER
END TYPE
DIM SHARED InRegs AS RegType, OutRegs AS RegType

ComSpec$ = ENVIRON$("COMSPEC")
BootDrive$ = LEFT$(ComSpec$, 2)
OPEN BootDrive$ + "\CONFIG.SYS" FOR INPUT AS #1
  DO WHILE NOT EOF(1)
    LINE INPUT #1, Work$
    Work$ = UCASE$(Work$)
    IF LEFT$(Work$, 6) = "FILES=" THEN
      FilesVal = VAL(MID$(Work$, 7))
      EXIT DO
    END IF
  LOOP
CLOSE

INPUT "How many files? ", NumFiles
NumFiles = NumFiles + 5
IF NumFiles > FilesVal THEN
  PRINT "Increase the FILES= setting in CONFIG.SYS"
  END
END IF

IF DOSVer% >= 330 THEN
  CALL MoreFiles(NumFiles)
ELSE
  PRINT "Sorry, DOS 3.3 or later is required."
  END
END IF

FOR X = 1 TO NumFiles
  OPEN "FTEST" + LTRIM$(STR$(X)) FOR RANDOM AS #X
NEXT
CLOSE
KILL "FTEST*."
END

FUNCTION DOSVer% STATIC
  InRegs.AX = &H3000
  CALL Interrupt(&H21, InRegs, OutRegs)
  Major = OutRegs.AX AND &HFF
  Minor = OutRegs.AX \ &H100
  DOSVer% = Minor + 100 * Major
END FUNCTION

SUB MoreFiles (NumFiles) STATIC
  InRegs.AX = &H6700
  InRegs.BX = NumFiles
  CALL Interrupt(&H21, InRegs, OutRegs)
END SUB

```

As with the TEXTFIND program, this also uses CALL Interrupt and therefore requires QB.LIB and QB.QLB to compile or run in the QuickBASIC environment respectively. Even though DOS allows you to increase the number of files past the default 15, an appropriate FILES= statement must also be added to the PC's CONFIG.SYS file. In fact, the FILES= value must be five greater than the desired number of files, because DOS reserves the first five for itself. The reserved files devices are PRN, AUX, STDIN, STDOUT, and STDERR. PRN is of course the printer connected to LPT1, AUX is the first COM port, and the remaining devices are all part of the CON console device.

In order to find the CONFIG.SYS file this program uses the ENVIRON\$ function to retrieve the current COMSPEC= setting. Unless someone has changed it on purpose, the COMSPEC environment variable holds the drive and path from which the PC was booted, and the file name COMMAND.COM. Then each line in CONFIG.SYS is examined for the string FILES=, to ensure that enough file entries were specified. This program makes only a minimal attempt to identify the FILES= string, so if there are extra spaces such as FILES = 30 the test will fail.

Next the DOS version is tested to ensure that it is version 3.3 or later. The DOSVer function is designed to return the DOS version as an integer value 100 times higher than the actual version number. That is, DOS 2.14 is returned as 214, and DOS 3.30 is instead 330. This eliminates the floating point math required to return a value such as 2.14 or 3.3, resulting in less code and faster operation.

Assuming the FILES= setting is sufficiently high and the DOS version is at least 3.30, the program creates and then deletes the specified number of files just to show it worked. You should be aware that the BASIC editor must also open files when it saves your program. I mention this because it is possible to be experimenting with a program such as this one, and not be able to save your work because the maximum allowable number of files are already open. In that case BASIC issues a "Too many files" error message, and refuses to let you save. The solution is to press F6 to go to the Immediate window, and then type CLOSE.

A similar situation happens when you try to shell to DOS from the BASIC editor, because shelling requires BASIC to open COMMAND.COM. But an unsuccessful shell results in an "Illegal function call" error. That message is particularly exasperating when BASIC's SHELL fails, because the failure is usually caused by insufficient memory or because COMMAND.COM cannot be located. Why Microsoft chose to return "Illegal function call" rather than "Out of memory", "File not found", or "Too many files" is anyone's guess.

Another important BASIC limitation that can be overcome only with clever trickery is its inability to map multiple variables to the same memory address. This is an important feature of the C language, and it has some important applications. For example, if you are frequently accessing a group of characters in the middle of a string, you must use MID\$ each time you assign or retrieve them. Unfortunately, MID\$ is very slow because it always extracts a copy of the specified characters, even if you are merely printing them. If only BASIC would let you create a new string that always referred to that group of characters in the first string, the access speed could be greatly improved.

The FIELD statement lets you do exactly this, and each time a new FIELD statement is encountered the same area of memory is referred to. The short example below shows the tremendous speed improvement possible only when two variables can occupy the same address. An additional trick used here is to open the DOS reserved "\DEV\NUL" device. This eliminates any disk access, and avoids also having to create an empty file just to implement the FIELD statement.

```
DEFINT A-Z

OPEN "\DEV\NUL" FOR RANDOM AS #1 LEN = 30
FIELD #1, 10 AS First$, 10 AS Middle$, 10 AS Last$
FIELD #1, 30 AS Entire$
LSET Entire$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ1234"
Start! = TIMER
FOR X = 1 TO 20000
    Temp = ASC(Middle$)
NEXT
Done! = TIMER
PRINT USING "##.### seconds for FIELD"; Done! - Start!
CLOSE

Entire$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ1234"
Start! = TIMER
FOR X = 1 TO 20000
    Temp = ASC(MID$(Entire$, 10, 10))
NEXT
Done! = TIMER
PRINT USING "##.### seconds for MID$"; Done! - Start!
```

As you can see, accessing Middle\$ as defined in the FIELD statement is more than three times faster than accessing the middle portion of Entire\$ using MID\$. There are no doubt other situations where it is useful to treat the same area of memory as different variables, perhaps to provide different views—such as numeric and string—of the same data. We can only hope that Microsoft will see fit to add this important capability to a future version of BASIC. PowerBASIC offers this feature via the UNION command.

The NUL device has other important applications in conjunction with FIELD. One common programming problem that comes up frequently is being able to format numbers to a controlled number of decimal places. Although BASIC's PRINT USING will format a number and write it to the screen, there is no way to actually access the formatted value. It is possible to have PRINT USING write the value on the screen—perhaps in the upper left corner with a color setting of black on black—and then read it character by character with SCREEN. But that method is clunky at best, and also very slow.

The short program below uses PRINT USING # to write to a fielded buffer, and then LINE INPUT # to read the number back from the buffer.

```
Value# = 123.45678#

OPEN "\DEV\NUL" FOR RANDOM AS #1 LEN = 15
FIELD #1, 15 AS Format$
PRINT #1, USING "#####.##"; Value#
LINE INPUT #1, Fmt$
```

```
PRINT "    Value: "; Value#  
PRINT "Formatted: "; Fmt$
```

Notice that the field buffer must be long enough to receive the entire formatted string, including the carriage return and line feed that BASIC sends as part of the PRINT # statement. This technique opens up many exciting possibilities, especially when used in conjunction with PRINT # USING's other extensive formatting options.

```
PDS includes the FORMAT$ function externally in Quick  
and regular link libraries, and UB/DOS goes a step  
further by adding FORMAT$ to the language. But FORMAT$  
offers only a subset of what PRINT USING can do.
```

Advanced Device Techniques

As many tricks as there are for reading and writing files, there are just as many for accessing devices. Many devices such as printers and modems are so much slower than BASIC that the techniques for sending large amounts of data in one operation are not needed or useful. But these devices offer a whole new set of problems that just beg for clever programming solutions. With that in mind, let's continue this tour and examine some of the less obvious aspects of BASIC's device handling capabilities.

The Printer Device

All modern printers accept special control codes to enable and disable underlining, boldfacing, italics, and sometimes even font changes. Many printers honor the standard Epson/IBM control codes, and some recognize additional codes to control unique features available only with that brand or model. However, it is possible to print underline and boldface text with most printers, without regard to the particular model. The examples shown below require that you open the printer as a device using "LPT1:BIN". If you are using LPT2, of course, then you will open "LPT2:BIN" instead. As I mentioned earlier, the BIN option tells BASIC not to interfere with any control codes you send, and also not to add automatic line wrapping.

Most programmers assume that every carriage return is always accompanied by a corresponding line feed, and indeed, that is almost always the case. Even if you print a CHR\$(13) carriage return followed by a semicolon, BASIC steps in and appends a line feed for you. But these are separate characters, and each can be used separately to control a printer. The example below prints a short string and a carriage return without a line feed, and then prints a series of underlines beneath the string.

```
OPEN "LPT1:BIN" FOR OUTPUT AS #1  
PRINT #1, "BASIC Techniques and Utilities"; CHR$(13);  
PRINT #1, "    _____"  
CLOSE
```

Similarly, you can also simulate boldfacing by printing the same string at the same place on the paper two or three times. While this won't work with a laser printer, it is very effective on dot matrix printers. Of course, if you do know the correct control codes for the printer, then those can be sent directly. Be sure, however, to always include a trailing semicolon as part of the print statement, to avoid also sending an unwanted return and line feed. For example, to advance a printer to the start of the next page you would use either `PRINT #1, CHR$(12);` or `LPRINT CHR$(12);`. In this case, a normal `LPRINT` will work because you are not sending a `CHR$(13)` or `CHR$(10)`.

Most printers also accept a `CHR$(8)` to indicate a backspace, which may simplify underlining in some cases. That is, instead of printing a `CHR$(13)` to go the start of the line, you would print the string, and simply back up the print head the appropriate number of columns. BASIC's `STRING$` function is ideal for this, using `LPRINT STRING$(Count, 8);` to send `Count` backspace characters to the printer.

You can also send a complete font file to a printer with the `CopyFile` program shown earlier. Simply give the font file's name as the source, and the string "LPT1:BIN" as the destination.

The Screen Device

As with printers, there are a number of ways to manipulate the display screen by printing special control characters. Where a `CHR$(12)` can be used to advance the printer to the top of the next page, this same character will clear the screen and place the cursor at the upper left corner. Printing a `CHR$(11)` will home the cursor only, and printing a `CHR$(7)` beeps the speaker.

Another useful screen control character is `CHR$(9)`, which advances to the next tab stop. Tab stops are located at every eighth column, with the first at column 9, the second at column 17, and so forth. As with a printer that has not been opened using the `BIN` option, printing either a `CHR$(10)` or a `CHR$(13)`—even with a semicolon—always sends the cursor to the beginning of the next line. There is unfortunately no way to separate the actions of a carriage return and line feed.

The last four control characters that are useful with the screen are `CHR$(28)`, `CHR$(29)`, `CHR$(30)`, and `CHR$(31)`. These move the cursor forward, backward, up a line if possible, and down a line if possible. Although `LOCATE` can be used to move the cursor, these commands allow you to do it relative to the current location. To do the same with `LOCATE` would require code like this: `IF POS(0) > 1 THEN LOCATE , POS(0) - 1`. Obviously, the control characters will result in less generated code, because they avoid the `IF` test and repeated calls to BASIC's `POS(0)` function.

BASIC PDS includes a series of stub files named `TSCNIOxx.OBJ` that eliminate support for all graphics statements, and also ignore the control characters listed above. Because each character must be tested individually by BASIC as it looks for these control codes, using these stub files will increase the speed of your program's display output.

All versions of Microsoft BASIC have always included the WIDTH statement for controlling the number of columns on the screen. With the introduction of QuickBASIC 3.0, SCREEN was expanded to also allow setting the number of rows on EGA and VGA monitors. The statement WIDTH , 43 puts the screen into the 43-line text mode, and may be used with an EGA or VGA display. WIDTH , 50 is valid for VGA monitors only, and as you can imagine, it switches the display to the 50-line text mode.

In many cases it is necessary to know if the display screen is color or monochrome, and also if it is capable of supporting the EGA or VGA graphics modes. The simplest way to detect a color monitor is to look at the display adapter's port address in low memory. The short code fragment below shows how this is done.

```
DEF SEG = 0
IF PEEK(&H463) = &HB4 THEN
  '---- it's a monochrome monitor
ELSE
  '---- it's a color monitor
END IF
```

This information is important if you plan to BLOAD a screen image directly into video memory. If the display adapter is reported as monochrome, then you would use DEF SEG to set the segment to &HB000. A color monitor in text mode instead uses segment &HB800. Knowing if a monitor has color capabilities also helps you to choose appropriate color values, and tells you if it can support graphics. But you will need to know which video modes the display adapter is capable of.

Detecting an EGA or VGA is more complex than merely distinguishing between monochrome and color, because it requires calling a video interrupt service routine located on the display adapter card. A Hercules monitor is also difficult to detect, because that requires a timing loop to see if the Hercules video status port changes. All of this is taken into account in the example and function that follows.

```
DEFINT A-Z

DECLARE SUB Interrupt (IntNum, InRegs AS ANY, OutRegs AS ANY) DECLARE FUNCTION
Monitor% (Segment)

TYPE RegType
  AX AS INTEGER
  BX AS INTEGER
  CX AS INTEGER
  DX AS INTEGER
  BP AS INTEGER
  SI AS INTEGER
  DI AS INTEGER
  Flags AS INTEGER
END TYPE
DIM SHARED InRegs AS RegType, OutRegs AS RegType

SELECT CASE Monitor%(Segment)
CASE 1
  PRINT "Monochrome";
CASE 2
  PRINT "Hercules";
```

```

CASE 3
  PRINT "CGA";
CASE 4
  PRINT "EGA";
CASE 5
  PRINT "VGA";
CASE ELSE
  PRINT "Unknown";
END SELECT
PRINT " monitor at segment &H"; HEX$(Segment)

FUNCTION Monitor% (Segment) STATIC
DEF SEG = 0          'first see if it's color or mono
Segment = &HB800     'assume color

IF PEEK(&H463) = &HB4 THEN 'it's monochrome

  Segment = &HB000   'assign the monochrome segment
  Status = INP(&H3BA) 'get the current video status
  FOR X = 1 TO 30000 'test for a Hercules 30000 times
    IF INP(&H3BA) <> Status THEN
      Monitor% = 2   'the port changed, it's a Herc
      EXIT FUNCTION 'all done
    END IF
  NEXT
  Monitor% = 1      'it's a plain monochrome

ELSE                'it's some sort of color monitor
  InRegs.AX = &H1A00 'first test for VGA
  CALL Interrupt(&H10, InRegs, OutRegs)
  IF (OutRegs.AX AND &HFF) = &H1A THEN
    Monitor% = 5     'it's a VGA
    EXIT FUNCTION   'all done
  END IF

  InRegs.AX = &H1200 'now test for EGA
  InRegs.BX = &H10
  CALL Interrupt(&H10, InRegs, OutRegs)
  IF (OutRegs.BX AND &HFF) = &H10 THEN
    Monitor% = 3     'if BL is still &H10 it's a CGA
  ELSE
    Monitor% = 4     'otherwise it's an EGA
  END IF
END IF
END FUNCTION

```

The Monitor function returns both the type of monitor that is active, as well as the video segment that is used when displaying text. EGA and VGA displays use segment &HA000 for graphics, which is a different issue altogether. Monitor is particularly valuable when you need to know what SCREEN modes a given display adapter can support. The only alternative is to use ON ERROR and try each possible SCREEN value in a loop starting from the highest resolution. When SCREEN finally reaches a low enough value to succeed, then you know what modes are legal. Since BASIC knows the type of monitor installed, it seems inconceivable to me that this information is not made available to your program.

```
PowerBASIC uses an internal variable to hold the display type, and that variable is available to the programmer.
```

Notice that the Registers TYPE variable is dimensioned in the example portion of this program, and not in the Monitor function itself. Each time a TYPE or fixed-length string variable is dimensioned in a STATIC subprogram or function, new memory is allocated permanently to hold it. In this short program the Registers TYPE variable is used only once. But in a real program that incorporates many of the routines from this chapter, memory can be saved by using DIM SHARED in the main program. Then, each subroutine can use the same variable for its own use.

Once you know the type of monitor, you will also know what color combinations are valid and readable. A color monitor can of course use any combination of foreground and background colors, but a monochrome is limited to the choices shown in Table 6-2. Combinations not listed will result in text that is unreadable on a many monochrome monitors.

Color as Displayed	COLOR Values
White on Black	COLOR 7,0
Bright White on Black	COLOR 15,0
Black on White	COLOR 0,7
White Underlined on Black	COLOR 1,0
Bright White Underlined on Black	COLOR 9,0

Table 6-2: Valid Color Combinations For Use With a Monochrome Monitor.

It is important to point out that some computers employ a CGA display adapter connected to a monochrome monitor. For example, the original Compaq portable PC used this arrangement. Many laptop computers also have a monochrome display connected to a CGA, EGA, or VGA adapter. Since it is impossible for a program to look beyond the adapter hardware through to the monitor itself, you will need to provide a way for users with that kind of hardware to alert your program.

The BASIC editor recognizes a /b command line switch to indicate black and white operation, and I suggest that you do something similar. Indeed, many commercial programs offer a way for the user to indicate that color operation is not available or desired.

The last video-related issue I want to cover is saving and loading text and graphics images. As you probably know, the memory organization of a display adapter when it is in one of the graphics modes is very different than when it is in text mode. In the text mode, each character and its corresponding color byte are stored in contiguous memory locations in the appropriate video segment. All of the color text

modes store the characters and their colors at segment &HB800, while monochrome displays use segment &HB000.

The character in the upper left corner of the screen is at address 0 in the video segment, and its corresponding color is at address 1. The character currently at screen location (1, 2) is stored at address 2, and its color is at address 3, and so forth. The brief program fragment below illustrates this visually by using POKE to write a string of characters and colors directly to display memory.

```
DEFINT A-Z

CLS
LOCATE 20
PRINT "Keep pressing a key to continue"
DEF SEG = 0
IF PEEK(&H463) = &HB4 THEN
  DEF SEG = &HB000
ELSE
  DEF SEG = &HB800
END IF

Test$ = "Hello!"
Colr = 9
FOR X = 1 TO LEN(Test$)
  Char = ASC(MID$(Test$, X, 1))
  POKE Address, Char
  WHILE LEN(INKEY$) = 0: WEND
  POKE Address + 1, Colr
  Address = Address + 2
  WHILE LEN(INKEY$) = 0: WEND
NEXT
END
```

The initial CLS command stores blank spaces and the current BASIC color settings in every memory address pair. Assuming you have not changed the color previously, a character value of 32 is stored by CLS into every even address, and a color value of 7 in every odd one. Once the correct video segment is known and assigned using DEF SEG, a simple loop pokes each character in the string to the display starting at address 0. Since Address was never assigned initially, it holds a value of zero.

Saving and loading graphics images is of necessity somewhat more complex, because you need to know not only the appropriate segment from which to save, but also how many bytes. The example program below creates a simple graphic image in CGA screen mode 1, saves the image, and then after clearing the screen loads it again.

```
DEFINT A-Z
SCREEN 1

DEF SEG = 0
PageSize = PEEK(&H44C) + 256 * PEEK(&H44D)

FOR X = 1 TO 10
  CIRCLE (140, 95), X * 10, 2
NEXT

DEF SEG = &HB800
```

```

BSAVE "CIRCLES.CGA", 0, PageSize
PRINT "The screen was just saved, press a key."
WHILE LEN(INKEY$) = 0: WEND

CLS
PRINT "Now press a key to load the screen."
WHILE LEN(INKEY$) = 0: WEND
BLOAD "CIRCLES.CGA", 0

```

Notice the use of PEEK to retrieve the current video page size at addresses &H44C and &H44D. This is a handy value that the BIOS maintains in low memory, and it tells you how many bytes are occupied by the screen whatever its current mode. In truth, this value is often slightly higher than the actual screen dimensions would indicate, since it is rounded up to the next even video page boundary. For example, the 320 by 200 screen mode used here occupies 16000 bytes of display memory, yet the page size is reported as 16384. But this value is needed to calculate the appropriate address when saving video pages other than page 0. That is, page 0 begins at address 0 at segment &HB800, and page 1 begins at address 16384.

Note that many early CGA video adapters contain only 16K of memory, and thus do not support multiple screen pages. Also note that there is a small quirk in Hercules adapters that causes the page size to always be reported as 16384, even when the screen is in text mode. I have found this word to be unreliable in the EGA and VGA graphics mode.

Although you might think that the pixels on a CGA graphics screen occupy contiguous memory addresses, they do not. Although each horizontal line is in fact contiguous, the lines are interlaced. Running the short program below shows how the first half of the video addresses contains the even rows (starting at row zero), and the second half holds the odd rows.

```

SCREEN 1
DEF SEG = &HB800
FOR X = 1 TO 15999
    POKE X, 255
NEXT

```

EGA and VGA displays add yet another level of complexity, because they use a separate video memory *plane* to store each color. Four planes are used for EGA and VGA, with one each to hold the red, blue, green, and intensity (brightness) information. Each plane is identified using the same segment and address, and OUT instructions are needed to select which is to be made currently active. This is called *bank switching*, because multiple, parallel banks of memory are switched in and out of the CPU's address space. When the red plane is active, reading and writing those memory locations affects only the red information on the screen. And when the intensity plane is made active, only the brightness for a given pixel on the screen is considered.

Bank switching is needed to accommodate the enormous amount of information that an EGA or VGA screen can contain. For example, in EGA screen mode 9, each plane occupies 28,000 bytes, for a total of 112,000 bytes of memory. This far exceeds the amount of memory the designers of the original IBM PC anticipated would ever be needed for display purposes. There simply aren't enough addresses available in the PC for video use. Therefore, the only way to deal with that much information is to

provide additional memory in the EGA and VGA adapters themselves. When a program needs to access a memory plane, it must do that one bank at a time so it can be read or written by the CPU.

The program below expands slightly on the earlier example, and shows how to save and load EGA and VGA screens by manipulating each video plane individually.

```
DEFINT A-Z
DECLARE SUB EgaBSave (FileName$)
DECLARE SUB EgaBLoad (FileName$)

SCREEN 9
LOCATE 25, 1
PRINT "Press a key to stop, and save the screen.";

'---- clever video effects by Brian Giedt
WHILE LEN(INKEY$) = 0
  T = (T MOD 150) + 1
  C = (C + 1) MOD 16
  LINE (T, T)-(300 - T, 300 - T), C, B
  LINE (300 + T, T)-(600 - T, 300 - T), C, B
WEND

LOCATE 25, 1
PRINT "Thank You!"; TAB(75);
CALL EgaBSave("SCREEN9")

CLS
LOCATE 25, 1
PRINT "Now press a key to read the screen.";
WHILE LEN(INKEY$) = 0: WEND
LOCATE 25, 1
PRINT TAB(75);

CALL EgaBLoad("SCREEN9")

SUB EgaBLoad (FileName$) STATIC
  'UnREM the KILL statements to erase the saved images after they
  ' have been loaded.

  DEF SEG = &HA000
  OUT &H3C4, 2: OUT &H3C5, 1
  BLOAD FileName$ + ".BLU", 0
  'KILL FileName$ + ".BLU"

  OUT &H3C4, 2: OUT &H3C5, 2
  BLOAD FileName$ + ".GRN", 0
  'KILL FileName$ + ".GRN"

  OUT &H3C4, 2: OUT &H3C5, 4
  BLOAD FileName$ + ".RED", 0
  'KILL FileName$ + ".RED"

  OUT &H3C4, 2: OUT &H3C5, 8
  BLOAD FileName$ + ".INT", 0
  'KILL FileName$ + ".INT"
  OUT &H3C4, 2: OUT &H3C5, 15
END SUB

SUB EgaBSave (FileName$) STATIC
```

```

DEF SEG = &HA000
Size& = 28000      'use 38400 for VGA SCREEN 12

OUT &H3CE, 4: OUT &H3CF, 0
BSAVE FileName$ + ".BLU", 0, Size&

OUT &H3CE, 4: OUT &H3CF, 1
BSAVE FileName$ + ".GRN", 0, Size&

OUT &H3CE, 4: OUT &H3CF, 2
BSAVE FileName$ + ".RED", 0, Size&

OUT &H3CE, 4: OUT &H3CF, 3
BSAVE FileName$ + ".INT", 0, Size&

OUT &H3CE, 4: OUT &H3CF, 0
END SUB

```

In the EGABLoad and EGABSave subroutines, two OUT statements are actually needed to switch planes. The first gets the EGA adapter's attention, to tell it that a subsequent byte is coming. That second value then indicates which memory plane to make currently available.

The Keyboard Device

The last device to consider is the keyboard. BASIC offers several commands and functions for accessing the keyboard, and these are INPUT, LINE INPUT, INPUT\$, and INKEY\$. Further, the "KYBD:" device may be opened as a file, and read using the file versions of the first three statements.

As with the file versions, INPUT reads numbers or text up to a terminating comma or Enter character. LINE INPUT is for strings only, and it ignores commas and requires Enter to be pressed to indicate the end of the line. INPUT\$ waits until the specified number of characters have been typed before returning, without regard to what characters are entered. INKEY\$ returns to the program immediately, even if no key was pressed.

Few serious programmers ever use INPUT or LINE INPUT for accepting entire lines of text, unless the program is very primitive or will be used only occasionally. The major problem with INPUT and LINE INPUT is that there's no way to control how many characters the operator enters. Once you use INPUT or LINE INPUT, you have lost control entirely until the user presses Enter. Worse, when INPUT is used to enter numeric variables, an erroneous entry causes BASIC to print its infamous "Redo from start" message. Either of these can spoil the appearance of a carefully designed data entry screen.

Therefore, the only reasonable way to accept user input is to use INKEY\$ to read the keys one by one, and act on them individually. If a character key is pressed, the cursor is advanced and the character is added to the string. If the back space key is detected, the cursor is moved to the left one column and the current character is erased. A series of IF or CASE statements is often used for this purpose, to handle every key that needs to be recognized.

The Editor input routine below provides exactly this service, and also tells you how editing was terminated. Besides being able to control the size of the input editing field, Editor also handles the Insert and Delete keys, and recognizes Home and End to jump the beginning and end of the field. A single COLOR statements lets you control the editing field color independently of the rest of the screen. The first portion of the code shows how Editor is set up and called.

```

DEFINT A-Z
DECLARE SUB Editor (Text$, LeftCol, RightCol, KeyCode)

COLOR 7, 1                                'clear to white on blue
CLS

Text$ = "This is a test"                  'make some sample text
LeftCol = 20                              'set the left column
RightCol = 60                             'and the right column
LOCATE 10                                  'set the line number
COLOR 0, 7                                'set the field color

DO                                         'edit until Enter or Esc
    CALL Editor(Text$, LeftCol, RightCol, KeyCode)
LOOP UNTIL KeyCode = 13 OR KeyCode = 27

SUB Editor (Text$, LeftCol, RightCol, KeyCode)
    '----- Find the cursor's size.
    DEF SEG = 0
    IF PEEK(&H463) = &HB4 THEN
        CsrSize = 12                      'mono uses 13 scan lines
    ELSE
        CsrSize = 7                       'color uses 8
    END IF

    '----- Work with a temporary copy.
    Edit$ = SPACE$(RightCol - LeftCol + 1)
    LSET Edit$ = Text$

    '----- See where to begin editing and print the string.
    TxtPos = POS(0) - LeftCol + 1
    IF TxtPos < 1 THEN TxtPos = 1
    IF TxtPos > LEN(Edit$) THEN TxtPos = LEN(Edit$)

    LOCATE , LeftCol
    PRINT Edit$;

    '----- This is the main loop for handling key presses.
    DO
        LOCATE , LeftCol + TxtPos - 1, 1

        DO
            Ky$ = INKEY$
            LOOP UNTIL LEN(Ky$)            'wait for a keypress

            IF LEN(Ky$) = 1 THEN           'create a key code
                KeyCode = ASC(Ky$)        'regular character key
            ELSE                            'extended key
                KeyCode = -ASC(RIGHT$(Ky$, 1))
            END IF

            '----- Branch according to the key pressed.
            SELECT CASE KeyCode

```



```

'----- Backspace: decrement the pointer and the
'         cursor, but ignore if in the first column.
CASE 8
    TxtPos = TxtPos - 1
    LOCATE , LeftCol + TxtPos - 1, 0
    IF TxtPos > 0 THEN
        IF Insert THEN
            MID$(Edit$, TxtPos) = MID$(Edit$, TxtPos + 1) + " "
        ELSE
            MID$(Edit$, TxtPos) = " "
        END IF
        PRINT MID$(Edit$, TxtPos);
    END IF

'----- Enter or Escape: this block is optional in
'         case you want to handle these separately.
CASE 13, 27
    EXIT DO                'exit the subprogram

'----- Letter keys: turn off the cursor to hide
'         the printing, handle Insert mode as needed.
CASE 32 TO 254
    LOCATE , , 0
    IF Insert THEN        'expand the string
        MID$(Edit$, TxtPos) = Ky$ + MID$(Edit$, TxtPos)
        PRINT MID$(Edit$, TxtPos);
    ELSE                  'else insert character
        MID$(Edit$, TxtPos) = Ky$
        PRINT Ky$;
    END IF
    TxtPos = TxtPos + 1   'update position counter

'----- Left arrow: decrement the position counter.
CASE -75
    TxtPos = TxtPos - 1

'----- Right arrow: increment position counter.
CASE -77
    TxtPos = TxtPos + 1

'----- Home: jump to the first character position.
CASE -71
    TxtPos = 1

'----- End: search for the last non-blank, and
'         make that the current editing position.
CASE -79
    FOR N = LEN(Edit$) TO 1 STEP -1
        IF MID$(Edit$, N, 1) <> " " THEN EXIT FOR
    NEXT
    TxtPos = N + 1
    IF TxtPos > LEN(Edit$) THEN TxtPos = LEN(Edit$)

'----- Insert key: toggle the Insert state and
'         adjust the cursor size.
CASE -82
    Insert = NOT Insert
    IF Insert THEN
        LOCATE , , , CsrSize \ 2, CsrSize
    ELSE

```

```

        LOCATE , , , CsrSize - 1, CsrSize
    END IF

    '----- Delete: delete the current character and
    '          reprint what remains in the string.
CASE -83
    MID$(Edit$, TxtPos) = MID$(Edit$, TxtPos + 1) + " "
    LOCATE , , 0
    PRINT MID$(Edit$, TxtPos);

    '---- All other keys: exit the subprogram
CASE ELSE
    EXIT DO
END SELECT

'----- Loop until the cursor moves out of the field.
LOOP UNTIL TxtPos < 1 OR TxtPos > LEN(Edit$)

Text$ = RTRIM$(Edit$)          'trim the text
END SUB

```

Most of the details in this subprogram do not require much explanation, and the code should prove simple enough to be self-documenting. However, I would like to discuss INKEY\$ as it is used here.

Each time INKEY\$ is used it examines the keyboard buffer, to see if a key is pending. If not, a null string is returned. If a key is present in the buffer INKEY\$ removes it, and returns either a 1-byte or 2-byte string, depending on what type of key it is. Normal character keys and control keys—entered by pressing the Ctrl key in conjunction with a regular key—are returned as a 1-byte string. Some special keys such as Enter and Escape are also returned as a 1-byte string, because they are in fact control keys. For example, Enter is the same as Ctrl-M, and Escape is identical to the Ctrl-[key.

The IBM PC offers additional keys and key combinations that are not defined by the ASCII standard, and these are returned as a 2-byte string so your program can identify them. Extended keys include the function keys, Home and End and the other cursor control keys, and Alt key combinations. When an extended key is returned the first character is always CHR\$(0), and the second character corresponds to the extended key's code using a method defined by IBM. Therefore, you can determine if a key is extended either by looking for a length of two, or by examining the first character to see if it is a CHR\$(0) zero byte.

There are three ways to accomplish this, and which is best depends on the compiler you are using. The brief program fragment below shows each method, and the number of bytes that are generated by both compilers.

```

IF LEN(X$) = 2 THEN          '17 for QB4, 7 for PDS
IF ASC(X$) THEN             '16 for QB4, 13 for PDS
IF LEFT$(X$, 1) = CHR$(0) THEN '33 for QB4, 30 for PDS

```

The references to QB 4 are valid for both QuickBASIC 4.0 and 4.5. The BASIC PDS byte counts reflect that compiler's improved code optimization, however this improvement is available only with

near strings. When far strings are used the LEN test requires the same 13 bytes as the ASC test. I'll presume that VB/DOS, with its support for only far strings, also uses the longer byte count.

As you can see, the test that uses BASIC's ASC function is slightly better than the one that uses LEN if you are using QuickBASIC. But if you have BASIC PDS the LEN test is quite a bit shorter. Comparing the first character in the string is much worse for either compiler, because individual calls must be made to BASIC's LEFT\$, CHR\$, and string comparison routines.

Even though the length and address of a QuickBASIC string is stored in the string's descriptor and is easily available to the compiler, the BC compiler that comes with QuickBASIC still calls a LEN routine. Where the compiler could use `CMP WORD PTR [DescriptorAddress], 2` to see if the string length is 2, it instead passes the address of the string descriptor on the stack, calls the LEN routine, and compares the result LEN returns. Fortunately, this optimization was added in BASIC PDS when near strings are used. Likewise, SADD when used with PDS near strings directly retrieves the string's address from the descriptor as well, instead of calling a library routine as QuickBASIC does.

The Editor subprogram uses the LEN method to determine the type of key that was pressed, which is most efficient if you are using BASIC PDS. Because integer comparisons are faster and generate less code than the equivalent operation with strings, ASC is then used to obtain either the ASCII value of the key, or the value of the extended key code. The result is assigned to the variable KeyCode as either a positive number to indicate a regular ASCII key, or a negative value that corresponds to an extended key's code. This method helps to reduce the size of the subprogram, by eliminating string comparisons in each CASE statement.

One important warning when using ASC is that it will generate an "Illegal function call" error if you pass it a null string. Therefore, in many cases you must include an additional test just for that:

```
IF LEN(Work$) THEN
  IF ASC(Work$) THEN
    ...
    ...
  END IF
END IF
```

One solution is to create your own function, perhaps called ASCII%(), that does this for you. Since calling a BASIC function requires no more code than when BASIC calls its own routines—assuming you are using the same number of arguments, of course—, this can also help to reduce the size of your programs. I like to use a return value of -1 to indicate a null string, as shown below.

```
FUNCTION ASCII%(This$)
  IF LEN(This$) THEN
    ASCII% = ASC(This$)
  ELSE
    ASCII% = -1
  END IF
END FUNCTION
```

Now you can simply use code such as `IF ASCII%(Any$) = Whatever THEN...` confident that no error will occur and the returned value will still be valid.

Redirection

One clever DOS feature that many programmers are not aware of is its ability to redirect a program's normal input and output to a file. When a program is redirected, print statements go to a specified file, keyboard input is read from a file, or both. The actual redirection commands are entered by the user of your program, and your program has no idea that this has happened. This is really more a DOS issue than a BASIC concern, but it's a powerful feature and you should understand how it works.

Redirection is useful for capturing a program's output to a disk file, or feeding keystrokes to a program using a predefined sequence contained in a file. For example, the output of the DOS DIR command can be redirected to a file with this command:

```
dir *.* > anyfile
```

Redirecting a program's input can be equally valuable. If you often format several diskettes at once you might create a file that contains the answer Y followed by an Enter character, and then run format using this:

```
format < yesfile
```

This way the file will provide the response to "Format another (Y/N)?".

To redirect a program's output, start it from the DOS command line and place a greater than > symbol and the output file name at the end of the command line:

```
program > filename
```

Similarly, using a less than < sign tells DOS to replace the program's requests for keyboard input with the contents of the specified file, thus:

```
program < filename
```

You can combine both redirected input and output at the same time, and the order in which they are given does not matter. It is important to understand that redirecting a program's output to a file is similar to opening that file for output. That is, it is created if it didn't yet exist, or truncated to a length of zero if it did. However, DOS also lets you append to a file when redirecting output, using two symbols in a row:

```
program >> filename
```

Please be aware that you can hang a PC completely when redirecting a program's input, if the necessary characters are not present. For example, this would happen when redirecting a program that uses LINE

INPUT from a file that has no terminating CHR\$(13) Enter character. Even pressing Ctrl-Break will have no effect, and your only recourse is to reboot, or close down the DOS session if you are using Windows.

Summary

This chapter has presented an enormous amount of information about both files and devices in BASIC. It began with a brief overview of how DOS allocates disk storage using sectors and clusters, and continued with an explanation of file buffers. By understanding the relationship between BASIC's own buffers and their impact on string memory, you gain greater control over your program's speed and memory requirements.

This then led to a comparison of files and devices, and showed how they can be controlled by similar BASIC statements. In particular, you learned how the same block of code can be used to send information to either, simplifying the design of reports and other programming output chores.

The section that described file access methods compared all of the available options, and explained when each is appropriate and why. You learned that all DOS files are really just a continuous stream of binary data, and the various OPEN methods merely let you indicate to BASIC how that data is to be handled.

You also learned that the best way to improve a program's file access speed is to read and write data in large blocks. Several complete subprograms and functions were shown to illustrate this technique, and most are general enough to be useful when included within your own programs.

Numerous tips and tricks were presented to determine the type of display adapter installed, run .COM programs and .BAT files, obtain formatted numbers by combining PRINT USING # with FIELD and INPUT #, and many more. You were also introduced to the possibility of calling BASIC's internal library routines as a way to circumvent many otherwise arbitrary limitations in the language.

Finally, video memory organization was revealed for all of the popular screen modes, and example programs were provided to show how they may be saved and loaded.

In the next chapter I will continue this discussion of files with detailed explanations of writing database programs. Chapter 7 will also describe how to write programs that operate on a network, as well as how to access data that uses the popular dBASE file format.

7

Network and Database Programming

In Chapter 6 you learned the principles of accessing files with BASIC, and saw the advantages and disadvantages of each of the various methods. This chapter continues the coverage of file handling in BASIC by discussing the concepts of database application programming. In particular, this chapter will cover database file structures—including fixed and variable length records—as well as the difference between code and data-driven applications.

This chapter also provides an in-depth look at the steps needed to write applications that can run on a network. This is an important topic that is fast becoming even more important, and very little information is available for programmers using BASIC. I will discuss the various file access schemes and record locking techniques, and also how to determine if a program is currently running on a network and if so which one.

This chapter examines common database file formats including the one used by dBASE III Plus, and utility programs are provided showing how to access these files. I will explain some of the fundamental issues of database design, including relationships between files. Also presented is a discussion of the common indexing techniques available, and a comparison of the relative advantages and disadvantages of each. You will also learn about the Structured Query Language (SQL) data access method, and understand the advantages it offers in an application programming context. Finally, several third-party add-on products that facilitate database application programming will be described.

Data Files versus Data Management

Almost every application you create will require some sort of file access, if only to store configuration information. Over time, programmers have developed hundreds of methods for storing information including sequential files, random files, and so forth. However, this type of data file management must not be confused with database management in the strict sense. Database management implies repeated data structures and relationships, with less importance given to the actual data itself.

In Chapter 6 you learned two common methods for defining the structure of a random access data file. But whether you use `FIELD` or `TYPE`, those examples focused on defining a record layout that is known in advance. When the data format will not change, defining a file structure within your program as `FIELD` or `TYPE` statements makes the most sense—a single statement can directly read or write any record in the file very quickly. But this precludes writing a general purpose database program such as dBASE, DataEase, or Paradox. In programs such as these, the user must be allowed to define each field and thus the record structure.

The key to the success of these commercial programs is therefore in their flexibility. If you need to write routines for forms processing, expression evaluation, file sorting, reports, and so forth, you should strive to make them reusable. For example, if you intend to print a report from a data file whose records have 100 fields, do you really want to use 100 explicit PRINT statements? The ideal approach is to create a generic report module that uses a loop to print each selected field in each of the selected records. This is where the concept of data-driven programming comes into play.

Data-Driven Programming

Data-driven programming, as its name implies, involves storing your data definitions as files, rather than as explicit statements in the program's BASIC source code. The advantage to this method of database programming lies in its flexibility and reusability. By storing the data definitions on disk, you can use one block of code to perform the same operations on completely different sets of data.

There are two general methods of storing data definitions on a disk—in the same file as the actual data or in a separate file. Storing the record definition in a separate file is the simplest approach, because it allows the main data file to be comprised solely of identical-length records. Keeping both the record layout and the data itself in a single file requires more work on your part, but with the advantage of slightly less disk clutter. In either case, some format must be devised to identify the number of fields in each data record and their type.

The example below shows a typical field layout definition, along with code to determine the number of fields in each record. Please understand that the random access file considered here is a file of field definitions, and not actual record data.

```
TYPE FldRec
  FldName AS STRING * 15
  FldType AS STRING * 1
  FldOff  AS INTEGER
  FldLen  AS INTEGER
END TYPE

OPEN "CUST.FLD" FOR BINARY AS #1
TotalFields% = LOF(1) \ 20
DIM FldStruc(1 TO TotalFields%) AS FldRec

RecLength% = 0
FOR X% = 1 TO TotalFields%
  GET #1, , FldStruc(X%)
  RecLength% = RecLength% + FldStruc(X%).FldLen
NEXT
CLOSE #1
```

In this program fragment, 15 characters are set aside for each field's name, a single byte is used to hold a field type code (1 = string, 2 = currency, or whatever), and integer offset and length values show how far into the record each field is located and how long it is. Once the field definitions file has been opened, the number of fields is easily determined by dividing the file size by the known 20-byte length

of each entry. From the number of you fields you can then dimension an array and read in the parameters of each field as shown here.

Notice that the record length is accumulated as each field description is read from the field definition file. In a real program, two field lengths would probably be required: the length of the field as it appears on the screen and the number of bytes it will actually require in the record. For example, a single precision number is stored on disk in only four bytes, even though as many as seven digits plus a decimal point could be displayed on the data entry screen. Therefore, the method shown in this simple example to accumulate the record lengths would be slightly more involved in practice.

Once the number and size of each field is known, it is a simple matter to assign a string to the correct length to hold a single data record. Any record could then be retrieved from the file, and its contents displayed as shown following.

```
OPEN "CUST.DAT" FOR RANDOM AS #1 LEN = RecLength%
Record$ = SPACE$(RecLength%)
GET #1, 11, Record$
CLOSE #1

FOR X% = 1 TO TotalFields%
  FldText$ = MID$(Record$, FldStruc(X%).FldOff, FldStruc(X%).FldLen)
  PRINT FldStruc(X%).FldName; ": "; FldText$
NEXT
```

Here, the first record in the file is read, and then the function form of MID\$ is used to extract each data field from that record. Assigning individual fields is just as easy, using the complementary statement form of MID\$:

```
MID$(Record$, FldStruc(FldNum).FldOff, FldStruc(FldNum).FldLen) = NewText$
```

Understand that the entire point of this exercise is to show how a generic routine to access files can be written, and without having to establish the record structure when you write the program. Although you could use FIELD instead of MID\$ to assign and retrieve the information from each field, that works only when the field information is kept in a separate file. If the field definitions are in the same file as the data, you will have to use purely binary file access, to account for the fixed header offset at the start of the file.

When you tell BASIC to open a file for random access, it uses the record length to determine where each record begins in the file. But if a header portion is at the beginning of the file, a fixed offset must be added to skip over the header. Since BASIC does not accommodate specifying an offset this way, it is up to you to handle that manually. However, the added complexity is not really that difficult, as you will see shortly in the routines that create and access dBASE files.

dBASE—and indeed, most commercial database products—store the field information in the same file that contains the data. This has the primary advantage of consolidating information for distribution purposes. For example, if your company sells a database of financial information, this minimizes the number of separate files your users will have to deal with. Modern header structures are variable

length, which allows for a greater optimization of disk space. In fact, most header structures mimic the record array shown above, but also store information such as the length of the header and the number of fields. This is needed because the number of fields cannot be determined from the file size alone, when the file also holds the data.

The dBASE III File Structure

The description of the dBASE file structure that follows serves two important purposes: First, it shows you how such a data file is constructed using a real world example. Second, this information allows you to directly access dBASE files in programs of your own. If you presently write commercial software, or if you aspire to, being compatible with the dBASE standard can give your product a definite advantage in the marketplace. Table 7-1 identifies each component of the dBASE file header.

Offset	Contents
1	dBASE version (3, or &H83 if there's a memo file)
2	Year of last update
3	Month of last update
4	Day of last update
5-8	Total number of records in the file (long integer)
9-10	Number of bytes in the header (integer) *
11-12	Length of records in the file (integer)
13-32	Reserved
The remainder of the header holds the field definitions, built from a repeating group of 32-byte blocks structured as follows:	
33-42	Field name, padded with CHR\$(0) null bytes
43	Always zero
44	Field type (C, D, L, M, or N) **
45-48	Reserved
49	Field width
50	Number of decimal places (Numeric fields only)
51-64	Reserved

Notes:

- * The end of the header is marked with a byte value of 13.
- ** The possible field types at byte 44 are Character, Date, Yes/No, Memo, and Numeric.

Table 7-1: The Structure of a dBASE III File Header

To obtain any item of information from the header you will use the binary form of GET #. For example, to read the number of data records in the file you would do this:

```
OPEN "CUST.DBF" FOR BINARY AS #1
GET #1, 5, NumRecords&
CLOSE #1
```

And to determine the length of each data record you will instead use this:

```
OPEN "CUST.DBF" FOR BINARY AS #1
GET #1, 1, RecordLength%
CLOSE #1
PRINT "The length of each record is "; RecordLength%
```

In the first example, GET # is told to seek to the fifth byte in the file and read the four-byte long integer stored there. The second example is similar, except it seeks to the 11th byte in the file and reads the integer record length field. One potential limitation you should be aware of is BASIC does not offer a byte-sized variable type. Therefore, to read a byte value such as the month you must create a one-character string, read the byte with GET #, and finally use the ASC function to obtain its value:

```
Month$ = " "
GET #1, 3, Month$
PRINT "The month is "; ASC(Month$)
```

Likewise, you will use CHR\$ to assign a new byte value prior to writing a one-character string:

```
Month$ = CHR$(NewMonth%)
PUT #1, 3, Month$
```

With this information in hand, it is a simple matter to open a dBASE file, and by reading the header determine everything your program needs to know about the structure of the data in that file. The simplest way to do this is by defining a TYPE variable for the first portion of the header, and a TYPE array to hold the information about each field. Since both the record and field header portions are each 32 bytes in length, you can open the file for Random access. A short program that does this is shown below.

```
TYPE HeadInfo
  Version AS STRING * 1
  Year    AS STRING * 1
  Month   AS STRING * 1
```

```

    Day      AS STRING * 1
    TRecs    AS LONG
    HLen     AS INTEGER
    RecLen   AS INTEGER
    Padded   AS STRING * 20
END TYPE

TYPE FieldInfo
    FName AS STRING * 10
    Junk1 AS STRING * 1
    FType AS STRING * 1
    Junk2 AS STRING * 4
    FLen  AS STRING * 1
    Dec   AS STRING * 1
    Junk3 AS STRING * 14
END TYPE

DIM Header AS HeadInfo

OPEN "CUST.DBF" FOR RANDOM AS #1 LEN = 32
GET #1, 1, Header
TFields% = (Header.HLen - 32) \ 32
REDIM FInfo(1 TO TFields%) AS FieldInfo

FOR X% = 2 TO TFields%
    GET #1, X%, FInfo(X%)
NEXT
CLOSE #1

```

dBASE File Access Tools

The programs that follow are intended as a complete set of toolbox subroutines that you can add to your own programs. The first program contains the core routines that do all of the work, and the remaining programs illustrate their use in context. Routines are provided to create, open, and close dBASE files, as well as read and write data records. Additional functions are provided to read the field information from the header, and also determine if a record has been marked as deleted.

The main file that contains the dBASE access routines is DBACCESS.BAS, and several demonstration programs are included that show the use of these routines in context. In particular, DBEDIT.BAS exercises all of the routines, and you should study that program very carefully.

There are two other example programs that illustrate the use of the dbAccess routines. DBCREATE.BAS creates an empty dBASE file containing a header with field information only, DBEDIT.BAS lets you browse, edit, and add records to a file, and DBSTRUCT.BAS displays the structure of an existing file. There is also a program to pack a database file to remove deleted records named, appropriately enough, DBPACK.BAS.

When you examine these subroutines, you will notice that all of the data—regardless of the field type—is stored as strings. As you learned in earlier chapters, storing data as strings instead of in their native format usually bloats the file size, and always slows down access to the field values. This is but one of

the fundamental limitations of the dBASE file format. Note that using strings alone is not the problem; rather, it is storing the numeric values as ASCII data.

```
'***** DBACCESS.BAS, module for access to DBF files

'Copyright (c) 1991 Ethan Winer

DEFINT A-Z

'$INCLUDE: 'dbf.bi'
'$INCLUDE: 'dbaccess.bi'

SUB CloseDBF (FileNum, TRecs&) STATIC
    Temp$ = PackDate$
    PUT #FileNum, 2, Temp$
    PUT #FileNum, 5, TRecs&
    CLOSE #FileNum
END SUB

SUB CreateDBF (FileName$, FieldArray() AS FieldStruc) STATIC
    TFields = UBOUND(FieldArray)
    HLen = TFields * 32 + 33
    Header$ = SPACE$(HLen + 1)
    Memo = 0

    FldBuf$ = STRING$(32, 0)
    ZeroStuff$ = FldBuf$
    FldOff = 33
    RecLen = 1

    FOR X = 1 TO TFields
        MID$(FldBuf$, 1) = FieldArray(X).FName
        MID$(FldBuf$, 12) = FieldArray(X).FType
        MID$(FldBuf$, 17) = CHR$(FieldArray(X).FLen)
        MID$(FldBuf$, 18) = CHR$(FieldArray(X).Dec)
        MID$(Header$, FldOff) = FldBuf$
        LSET FldBuf$ = ZeroStuff$
        FldOff = FldOff + 32
        IF FieldArray(X).FType = "M" THEN Memo = -1
        RecLen = RecLen + FieldArray(X).FLen
    NEXT

    IF Memo THEN Version = 131 ELSE Version = 3
    MID$(Header$, 1) = CHR$(Version)
    Today$ = DATE$
    Year = VAL(RIGHT$(Today$, 2))
    Day = VAL(MID$(Today$, 4, 2))
    Month = VAL(LEFT$(Today$, 2))

    MID$(Header$, 2) = PackDate$
    MID$(Header$, 5) = MKL$(0)
    MID$(Header$, 9) = MKI$(HLen)
    MID$(Header$, 11, 2) = MKI$(RecLen)
    MID$(Header$, FldOff) = CHR$(13)
    MID$(Header$, FldOff + 1) = CHR$(26)

    OPEN FileName$ FOR BINARY AS #1
    PUT #1, 1, Header$
    CLOSE #1
END SUB
```

```

FUNCTION Deleted% (Record$) STATIC
    Deleted% = 0
    IF LEFT$(Record$, 1) = "*" THEN Deleted% = -1
END FUNCTION

FUNCTION GetField$ (Record$, FldNum, FldArray() AS FieldStruc) STATIC
    GetField$ = MID$(Record$, FldArray(FldNum).FOff, FldArray(FldNum).FLen)
END FUNCTION

FUNCTION GetFldNum% (FieldName$, FldArray() AS FieldStruc) STATIC
FOR X = 1 TO UBOUND(FldArray)
    IF FldArray(X).FName = FieldName$ THEN
        GetFldNum% = X
        EXIT FUNCTION
    END IF
NEXT
END FUNCTION

SUB GetRecord (FileNum, RecNum&, Record$, Header AS DBFHeadStruc) STATIC
    RecOff& = ((RecNum& - 1) * Header.RecLen) + Header.FirstRec
    GET FileNum, RecOff&, Record$
END SUB

SUB OpenDBF (FileNum, FileName$, Header AS DBFHeadStruc, FldArray() AS _
FieldStruc) STATIC
    OPEN FileName$ FOR BINARY AS FileNum
    GET FileNum, 9, HLen
    Header.FirstRec = HLen + 1
    Buffer$ = SPACE$(HLen)

    GET FileNum, 1, Buffer$
    Header.Version = ASC(Buffer$)
    IF Header.Version = 131 THEN
        Header.Version = 3
        Header.Memo = -1
    ELSE
        Header.Memo = 0
    END IF

    Header.Year = ASC(MID$(Buffer$, 2, 1))
    Header.Month = ASC(MID$(Buffer$, 3, 1))
    Header.Day = ASC(MID$(Buffer$, 4, 1))
    Header.TRecs = CVL(MID$(Buffer$, 5, 4))
    Header.RecLen = CVI(MID$(Buffer$, 11, 2))
    Header.TFields = (HLen - 33) \ 32

    REDIM FldArray(1 TO Header.TFields) AS FieldStruc
    OffSet = 2
    BuffOff = 33
    Zero$ = CHR$(0)

    FOR X = 1 TO Header.TFields
        FTerm = INSTR(BuffOff, Buffer$, Zero$)
        FldArray(X).FName = MID$(Buffer$, BuffOff, FTerm - BuffOff)
        FldArray(X).FType = MID$(Buffer$, BuffOff + 11, 1)
        FldArray(X).FOff = OffSet
        FldArray(X).FLen = ASC(MID$(Buffer$, BuffOff + 16, 1))
        FldArray(X).Dec = ASC(MID$(Buffer$, BuffOff + 17, 1))
    NEXT X
END SUB

```

```

        Offset = Offset + FldArray(X).FLen
        BuffOff = BuffOff + 32
    NEXT
END SUB

FUNCTION PackDate$ STATIC
    Today$ = DATE$
    Year = VAL(RIGHT$(Today$, 2))
    Day = VAL(MID$(Today$, 4, 2))
    Month = VAL(LEFT$(Today$, 2))
    PackDate$ = CHR$(Year) + CHR$(Month) + CHR$(Day)
END FUNCTION

FUNCTION Padded$ (Fld$, FLen) STATIC
    Temp$ = SPACE$(FLen)
    LSET Temp$ = Fld$
    Padded$ = Temp$
END FUNCTION

SUB SetField (Record$, FText$, FldNum, FldArray() AS FieldStruc) STATIC
    FText$ = Padded$(FText$, FldArray(FldNum).FLen)
    MID$(Record$, FldArray(FldNum).FOff, FldArray(FldNum).FLen) = Ftext$
END SUB

SUB SetRecord (FileNum, RecNum&, Record$, Header AS DBFHeadStruc) STATIC
    RecOff& = ((RecNum& - 1) * Header.RecLen) + Header.FirstRec
    PUT FileNum, RecOff&, Record$
END SUB

```

Each of the routines listed above performs a different useful service to assist you in accessing dBASE files, and the following section describes the operation and use of each routine. Please understand that these routines are intended to be loaded as a module, along with your own main program. To assist you, a file named DBACCESS.BI is provided, which contains appropriate DECLARE statements for each routine. You should therefore include this file in your programs that use these routines.

A second include file named DBF.BI is also provided, and it contains TYPE definitions for the header and field information. You may notice that these definitions vary slightly from the actual format of a dBASE file. For efficiency, the OpenDBF routine calculates and saves key information about the file to use later. As an example, the offset of the first record's field information is needed by GetRecord and SetRecord. Rather than require those procedures to calculate the information repeatedly each time, OpenDBF does it once and stores the result in the Header TYPE variable.

Similarly, the field definition header used by these routines does not parallel exactly the format of the information in the file. The modified structures defined in DBF.BI are as follows:

```

'***** DBF.BI - Record declarations for the dbAccess routines
TYPE DBFHeadStruc
    Version AS INTEGER
    Memo AS INTEGER
    Year AS INTEGER
    Month AS INTEGER
    Day AS INTEGER
    FirstRec AS INTEGER
    TRecs AS LONG
    RecLen AS INTEGER

```

```
TFields AS INTEGER
END TYPE
```

```
TYPE FieldStruc
  FName AS STRING * 10
  FType AS STRING * 1
  FOff AS INTEGER
  FLen AS INTEGER
  Dec AS INTEGER
END TYPE
```

CreateDBF

CreateDBF accepts the name of the file to create and a field definition array, and then creates the header portion of a dBASE file based on the field information in the array. The file that is created has no data records in it, but all of the header information is in place. The calling program must have dimensioned the field information TYPE array, and filled it with appropriate information that describes the structure of the records in the file. The DBCREATE.BAS program shows an example of how to set up and call CreateDBF.

OpenDBF And CloseDBF

OpenDBF is used to open a DBF file, and to make information about its structure available to the calling program. It fills a TYPE variable with information from the data file header, and also fills the field definition array with information about each field. When you call it you will pass a BASIC file number you want to be used for later access, the full name of the file, a TYPE variable that receives the header information, and a TYPE array. The array is redimensioned within OpenDBF, and then filled with information about each field in the file.

CloseDBF is called when you want to close the file, and it is also responsible for updating the date and number of records information in the file header.

GetRecord And SetRecord

GetRecord and SetRecord retrieve and write individual records respectively. The calling program must specify the file and record numbers, and also pass a string that will receive the actual record data. GetRecord assumes that you have already created the string that is to receive data from the file. A Header variable is also required, so GetRecord and SetRecord will know the length of each record. Both GetRecord and SetRecord require the file to have already been opened using OpenDBF.

GetField, GetFldNum, SetField, and Padded

These routines are used to retrieve and assign the actual field data within a record string. The dbAccess routines cannot use a TYPE variable to define the records, since they must be able to accommodate any type of file. Therefore, the Record\$ variable is created dynamically, and assigned and read as necessary. However, this also means that you may not refer to the fields by name as would be possible with a TYPE variable.

GetField returns the contents of the specified field, based on the field number; the complementary function GetFldName returns the field number based on the field name. SetField is the opposite of GetField, and it assigns a field into the Record\$ variable. Padded\$ serves as an assistant to SetField, and it ensures that the field contents are padded to the correct length with trailing blanks.

Deleted

Deleted is an integer function that returns a value of -1 to indicate that the record string passed to it holds a deleted record, or 0 if the record is not deleted. The very first byte in each dBASE record is reserved just to indicate if the record has been deleted. An asterisk "*" in that position means the record is deleted; otherwise the field is blank. Using a function for this purpose lets you directly test a record using code such as `IF Deleted%(Record$) THEN` or `IF NOT Deleted%(Record$) THEN`.

Marking deleted records is a common technique in database programming, because the amount of overhead needed to actually remove a record from a file is hardly ever justified. The lost space is recovered in one of two ways: the most common is to copy the data from one file to another. Another, more sophisticated method instead keeps track of which records have been deleted. Then as new data is added, it is stored in the space that was marked as abandoned, thus overwriting the old data. The DBPACK.BAS program described later in this chapter uses the copy method, but uses a trick to avoid having to create a second file.

dBASE Utility Programs

Several programs are presented to show the various dbAccess routines in context, and each is described individually below. DBSTRUCT.BAS displays the header structure of any dBASE file, DBCREATE.BAS creates an empty database file with header information only, and DBEDIT.BAS lets you browse, edit, and add records to an existing data file. These programs are simple enough to understand, even without excessive comments. However, highlights of each program's operation is given.

DBSTRUCT.BAS

DBSTRUCT.BAS begins by including the DBF.BI file which defines the Header TYPE variable and the FldStruc() TYPE array. A short DEF FN-style function is used to simplify formatting when the file date is printed later in the program. Once you enter the name of the dBASE file to be displayed, a call is made to OpenDBF. OpenDBF accepts the incoming file number and name, and returns information about the file in Header and FldStruc(). The remainder of the program simply reports that information on the display screen.

DBCREATE.BAS

The DBCREATE.BAS program accepts the name of a data file to create, and then asks how many fields it is to contain. Once the number of fields is known, a TYPE array is dimensioned to hold the information, and you are prompted for each field's characteristics one by one. As you can see by examining the program source listing, the information you enter is validated to prevent errors such as illegal field lengths, more decimal digits than the field can hold, and so forth.

As each field is defined in the main FOR/NEXT loop, the information you enter is stored directly into the FldStruc TYPE array. At the end of the loop, CreateDBF is called to create an empty .DBF data file.

```
'***** DBCREATE.BAS, creates a DBF file

DEFINT A-Z

'$INCLUDE: 'dbf.bi'
'$INCLUDE: 'dbaccess.bi'

CLS
LOCATE , , 1

LINE INPUT "Enter DBF name: "; DBFName$
IF INSTR(DBFName$, ".") = 0 THEN
    DBFName$ = DBFName$ + ".DBF"
END IF

DO
    INPUT "Enter number of fields"; TFields
    IF TFields <= 128 THEN EXIT DO
    PRINT "Only 128 fields are allowed"
LOOP

REDIM FldStruc(1 TO TFields) AS FieldStruc

FOR X = 1 TO TFields
    CLS
    DO
        PRINT "Field #"; X
        LINE INPUT "Enter field name: ", Temp$
        IF LEN(Temp$) <= 10 THEN EXIT DO
        PRINT "Field names are limited to 10 characters"
    LOOP
    FldStruc(X).FName = Temp$

    PRINT "Enter field type (Char, Date, Logical, Memo, ";
    PRINT "Numeric (C,D,L,M,N): ";
    DO
        Temp$ = UCASE$(INKEY$)
        LOOP UNTIL INSTR(" CDLMN", Temp$) > 1
        PRINT
        FldStruc(X).FType = Temp$
        FldType = ASC(Temp$)

        SELECT CASE FldType
            CASE 67
                'character
                DO
                    INPUT "Enter field length: ", FldStruc(X).FLen
                    IF FldStruc(X).FLen <= 255 THEN EXIT DO
                    PRINT "Character field limited to 255 characters"
                LOOP
```

```

CASE 78                                'numeric
DO
  INPUT "Enter field length: ", FldStruc(X).FLen
  IF FldStruc(X).FLen <= 19 THEN EXIT DO
  PRINT "Numeric field limited to 19 characters"
LOOP
DO
  INPUT "Number of decimal places: ", FldStruc(X).Dec
  IF FldStruc(X).Dec < FldStruc(X).FLen THEN EXIT DO
  PRINT "Too many decimal places"
LOOP

CASE 76                                'logical
  FldStruc(X).FLen = 1

CASE 68                                'date
  FldStruc(X).FLen = 8

CASE 77
  FldStruc(X).FLen = 10

END SELECT
NEXT

CALL CreateDBF(DBFName$, FldStruc())
PRINT DBFName$; " created"
END

```

DBEDIT.BAS

DBEDIT.BAS is the main demonstration program for the dbAccess subroutines. It prompts you for the name of the dBASE file to work with, and then calls OpenFile to open it. Once the file has been opened you may view records forward and backward, edit existing records, add new records, and delete and undelete records. Each of these operations is handled by a separate CASE block, making the code easy to understand.

```

'***** DBEDIT.BAS, edits a record in a DBF file

DEFINT A-Z
'$INCLUDE: 'dbf.bi'
'$INCLUDE: 'dbaccess.bi'

DIM Header AS DBFHeadStruc
REDIM FldStruc(1 TO 1) AS FieldStruc

CLS
LINE INPUT "Enter .DBF file name: ", DBFName$
IF INSTR(DBFName$, ".") = 0 THEN
  DBFName$ = DBFName$ + ".DBF"
END IF

CALL OpenDBF(1, DBFName$, Header, FldStruc())

Record$ = SPACE$(Header.RecLen)
RecNum& = 1
RecChanged = 0

```

```

GOSUB GetTheRecord

DO
PRINT "What do you want to do (Next, Prior, Edit, ";
PRINT "Delete, Undelete, Add, Quit)? ";
SELECT CASE UCASE$(INPUT$(1))
CASE "N"
    IF RecChanged THEN
        CALL SetRecord(1, RecNum&, Record$, Header)
    END IF
    RecNum& = RecNum& + 1
    IF RecNum& > Header.TRecs THEN
        RecNum& = 1
    END IF
    GOSUB GetTheRecord

CASE "P"
    IF RecChanged THEN
        CALL SetRecord(1, RecNum&, Record$, Header)
    END IF
    RecNum& = RecNum& - 1
    IF RecNum& < 1 THEN
        RecNum& = Header.TRecs
    END IF
    GOSUB GetTheRecord

CASE "E"
Edit:
    PRINT
    INPUT "Enter the field number: "; Fld
    DO
        PRINT "New "; FldStruc(Fld).FName;
        INPUT Text$
        IF LEN(Text$) <= FldStruc(Fld).FLen THEN EXIT DO
        PRINT "Too long, only "; FldStruc(Fld).FLen
    LOOP
    CALL SetField(Record$, Text$, Fld, FldStruc())
    RecChanged = -1
    GOSUB DisplayRec

CASE "D"
    MID$(Record$, 1) = "*"
    RecChanged = -1
    GOSUB DisplayRec

CASE "U"
    MID$(Record$, 1, 1) = " "
    RecChanged = -1
    GOSUB DisplayRec

CASE "A"
    Header.TRecs = Header.TRecs + 1
    RecNum& = Header.TRecs
    LSET Record$ = ""
    GOTO Edit

CASE ELSE
    EXIT DO
END SELECT
LOOP

```

```

IF RecChanged THEN
  CALL SetRecord(1, RecNum&, Record$, Header)
END IF

CALL CloseDBF(1, Header.TRecs)
END

GetTheRecord:
  CALL GetRecord(1, RecNum&, Record$, Header)

DisplayRec:
  CLS
  PRINT "Record "; RecNum&; " of "; Header.TRecs;
  IF Deleted%(Record$) THEN PRINT " (Deleted)";

  PRINT
  PRINT
  FOR Fld = 1 TO Header.TFields
    FldText$ = GetField$(Record$, Fld, FldStruc())
    PRINT FldStruc(Fld).FName, FldText$
  NEXT
  PRINT

RETURN

```

DBPACK.BAS

DBPACK.BAS is the final dBASE utility, and it shows how to write an optimized packing program. Since there is no reasonable way to actually erase a record from the middle of a file, dBASE and most database programs reserve a byte in each record solely to show if it has been deleted. The DBPACK.BAS utility program is intended to be run periodically, to actually remove the deleted records.

Most programs perform this maintenance by creating a new file, copying only the valid records to that file, and then deleting the original data file. In fact, this is what dBASE does. The approach taken by DBPACK is much more intelligent in that it works through the file copying good records on top of deleted ones. When all that remains at the end of the file is data that has been deleted or abandoned copies of records, the file is truncated to a new, shorter length. The primary advantage of this approach is that it saves disk space. This is superior to the copy method that of course requires you to have enough free space for both the original data and the copy. Because the actual data file is manipulated instead of a copy, be sure to have a recent backup in case a power failure occurs during the packing process.

DBPACK.BAS is fairly quick, but it could be improved if records were processed in groups, rather than one at a time. This would allow more of the swapping to take place in memory, rather than on the disk. However, DBPACK was kept simple on purpose, to make its operation clearer.

There is no BASIC or DOS command that specifically truncates a file, so this program uses a little-known trick. If a program calls DOS telling it to write zero bytes to a file, DOS truncates the file at the current seek location. Since BASIC does not allow you to write zero bytes, CALL Interrupt must

be used to perform the DOS call. Note that you can also use this technique to extend a file beyond its current length. This will be described in more detail in Chapter 11, which describes using CALL Interrupt to access DOS and BIOS services.

```

'***** DBPACK.BAS, removes deleted records from a file

'NOTE: Please make a copy of your DBF file before running this program.
'      Unlike dBASE that works with a copy of the data file, this program
'      packs, swaps records, and then truncates the original data file.
DEFINT A-Z
'$INCLUDE: 'dbf.bi'
'$INCLUDE: 'dbaccess.bi'
'$INCLUDE: 'regtype.bi'

DIM Registers AS RegType
DIM Header AS DBFHeadStruc
REDIM FldStruc(1 TO 1) AS FieldStruc

LINE INPUT "Enter the dBASE file name: ", DBFName$
IF INSTR(DBFName$, ".") = 0 THEN
    DBFName$ = DBFName$ + ".DBF"
END IF

CALL OpenDBF(1, DBFName$, Header, FldStruc())

Record$ = SPACE$(Header.RecLen)
GoodRecs& = 0

FOR Rec& = 1 TO Header.TRecs
    CALL GetRecord(1, Rec&, Record$, Header)
    IF NOT Deleted%(Record$) THEN
        CALL SetRecord(1, GoodRecs& + 1, Record$, Header)
        GoodRecs& = GoodRecs& + 1
    END IF
NEXT

'This trick truncates the file
RecOff& = (GoodRecs& * Header.RecLen) + Header.FirstRec
Eof$ = CHR$(26)
PUT #1, RecOff&, Eof$
SEEK #1, RecOff& + 1

Registers.AX = &H4000          'service to write to a file
Registers.BX = FILEATTR(1, 2) 'get the DOS handle
Registers.CX = 0              'write 0 bytes to truncate
CALL Interrupt(&H21, Registers, Registers)
CALL CloseDBF(1, GoodRecs&)

PRINT "All of the deleted records were removed from ";
PRINT DBFName$
PRINT GoodRecs&; "remaining records"

```

Limitations of the dBASE III Structure

The primary limitation of the DBF file format is it does not allow complex data types. With support for only five basic field types—Character, Date, Logical, Memo, and Numeric—it is very limited when

compared to what BASIC allows. However, you can easily add new data types to the programs you write using extensions to the standard field format. Since a byte is used to store the field type in the dBASE file header, as many as 256 different types are possible (0 through 255). You would simply define additional code numbers for field types such as Money or Time, or perhaps other Logical field types such as M and F (Male and Female).

Another useful enhancement would be to store numeric values in their native fixed-length format, instead of using the much slower ASCII format that dBASE uses. You could also modify the header structure itself, to improve the performance of your programs. Since BASIC does not offer a single byte numeric data type, it would make sense to replace the `STRING * 1` variables with integers. This would eliminate repeated use of `ASC` and `CHR$` when reading and assigning single byte strings. You could also change the date storage method to pack the date fields to three characters—one for the year, one for the month, and another for the day. Of course, if you do change the header or data format, then your files will no longer be compatible with the dBASE standard.

Indexing Techniques

At some point, the number of records in a database file will grow to the point where it takes longer and longer to locate information in the file. This is where indexing can help. Some of the principles of indexed file access were already described in Chapter 5, in the section that listed the BASIC PDS ISAM compiler switches. In this section I will present more details on how indexing works, and also show some simple methods you can create yourself. Although there are nearly as many indexing systems as there are programmers, one of the most common is the sorted list.

Sorted Lists

A sorted list is simply a parallel `TYPE` array that holds the key field and a record number that corresponds to the data in the main file. By maintaining the array in sorted order based on the key field information, the entire database may be accessed in sorted, rather than sequential order. A typical `TYPE` array used as a sorted list for indexing would look like this:

```
TYPE IndexType
  LastName AS STRING * 15
  RecNum   AS LONG
END TYPE
REDIM IArray(1 TO TotalRecords) AS IndexType
```

Assuming each record in the data file has a corresponding element in the `TYPE` array, locating a given record is as simple as searching the array for a match. Since array searches in memory are much faster than reading a disk file, this provides an enormous performance boost when compared to reading each record sequentially. To conserve memory and also further improve searching speed, you might use a shorter string portion for the last name.

The following short program shows how such an index array could be sorted.

```
FOR X% = MaxEls TO 1 STEP -1
  FOR Y% = 1 TO X% - 1
    IF IArray(Y%).LastName > IArray(Y% + 1).LastName THEN
      SWAP IArray(Y%), IArray(Y% + 1)
    END IF
  NEXT
NEXT
```

Here, the sorting is based on the last name portion of the TYPE elements. Once the array is sorted, the data file may be accessed in order by walking through the record numbers contained in the RecNum portion of each element:

```
DIM RecordVar AS IndexType
FOR X% = 1 TO MaxEls
  GET #1, IArray(X%).RecNum, RecordVar
  PRINT RecordVar.LastName
NEXT
```

Likewise, to find a given name you would search the index array based on the last name, and then use the record number from the same element once it is found:

```
Search$ = "Cramer"
FOR X% = 1 TO MaxEls
  IF RTRIM$(IArray(X%).LastName) = Search$ THEN
    Record% = IArray(X%).RecNum
    GET #1, Record%, RecordVar
    PRINT "Found "; Search$; " at record number"; Record%
    EXIT FOR
  END IF
NEXT
```

Chapter 8 will discuss sorting and searching in detail using more sophisticated algorithms than those shown here, and you would certainly want to use those for your program. However, one simple improvement you could make is to reduce the number of characters in each index entry. For example, you could keep only the first four characters of each last name. Although this might seem to cause a problem—searching for Jackson would also find Jack—you would have the same problem if there were two Jacksons. The solution, therefore, is to retrieve the entire record if a partial match is found, and compare the complete information in the record with the search criteria.

Inserting an entry into a sorted list requires searching for the first entry that is greater than or equal to the one you wish to insert, moving the rest of the entries down one notch and inserting the new entry. The code for such a process might look something like this:

```
FOR X% = 2 TO NumRecs%
  IF Item.LastName <= Array(X%).LastName THEN
    IF Item.LastName >= Array(X% - 1).LastName THEN
      FOR Y% = NumRecs% TO X% STEP -1
        SWAP Array(Y%), Array(Y% + 1)
      NEXT
      Array(X%) = Item
    END IF
  END IF
NEXT
```

```
        EXIT FOR
      END IF
    END IF
  NEXT
```

Understand that this code is somewhat simplified. For example, it will not correctly handle inserting an element before the first existing entry or after the last. Equally important, unless you are dealing with less than a few hundred entries, this code will be extremely slow. The loop that inserts an element by swapping all of the elements that lie beyond the insertion point will never be as efficient as a dedicated subroutine written in assembly language. Commercial toolbox products such as Crescent Software's QuickPak Professional include memory moving routines that are much faster than one written using BASIC.

Finally, you must have dimensioned the array to at least one more element than there are records, to accommodate the inserted element. Many programs that use in-memory arrays for indexing dimension the arrays to several hundred extra elements to allow new data to be entered during the course of the session. Since BASIC 7.1 offers the REDIM PRESERVE command, that too could be used to extend an array as new data is added.

Expression Evaluation

Expression evaluation, in the context of data management, is the process of evaluating a record on the basis of some formula. Its uses include the creation of index keys, reports, and selection criteria. This is where the application of independent file structures such as the dBASE example shows a tremendous advantage. For example, if the user wants to be able to view the file sorted first by Zip code and then by last name, some means of performing a multi-key sort is required.

Another example of expression evaluation is when multiple conditions using AND and OR logic are needed. You may want to select only those records where the balance due is greater than \$100 and the date of last payment is more than 30 days prior to the current date. Admittedly, writing an expression parser is not trivial; however, the point is that data-driven programming is much more suitable than code-driven programming in this case.

Without some sort of look-up table in which you can find the field names and byte offsets, you are going to have a huge number of SELECT CASE statements, none of which are reusable in another application. Indeed, one of the most valuable features of AJS Publishing's db/LIB add-on database library is the expression evaluator it includes. This routine lets you maintain the data structure in a file, and the same code can be used to process all file search operations.

Relational Databases

Most programmers are familiar with traditional random access files, where a fixed amount of space is set aside in each record to hold a fixed amount of information. For very simple applications this method

is sensible, and allows for fast access to each record provided you know the record number. As you learned earlier in this chapter, indexing systems can eliminate the need to deal with record numbers, instead letting you locate records based on the information they contain. Relational databases take this concept one step further, and let you locate records in one file based on information contained in another file. As you will see, this lets you create applications that are much more powerful than those created using standard file handling methods.

Imagine you are responsible for creating an order entry program for an auto parts store. At the minimum, three sets of information must be retained in such a system: the name, address, and phone number of each customer; a description of each item that is stocked and its price; and the order detail for each individual sale. A simplistic approach would be to define the records in a single database with fields to hold the customer information and the products purchased, with a new record used for each transaction. A TYPE definition for these records might look like this:

```
TYPE RecordType
  InvoiceNum AS INTEGER
  CustName AS STRING * 32
  CustStreet AS STRING * 32
  CustCity AS STRING * 15
  CustState AS STRING * 2
  CustZip AS STRING * 5
  CustPhone AS STRING * 10
  Item1Desc AS STRING * 15
  Item1Price AS SINGLE
  Quantity1 AS INTEGER
  Item2Desc AS STRING * 15
  Item2Price AS SINGLE
  Quantity2 AS INTEGER
  Item3Desc AS STRING * 15
  Item3Price AS SINGLE
  Quantity3 AS INTEGER
  Item4Desc AS STRING * 15
  Item4Price AS SINGLE
  Quantity4 AS INTEGER
  TaxPercent AS SINGLE
  InvoiceTot AS SINGLE
END TYPE
```

As sensible as this may seem at first glance, there are a number of problems with this record structure. The primary limitation is that each record can hold only four purchase items. How could the sales clerk process an order if someone wanted to buy five items? While room could be set aside for ten or more items, that would waste disk space for sales of fewer items. Worse, that still doesn't solve the inevitable situation when someone needs to buy eleven or more items at one time.

Another important problem is that the customer name and address will be repeated for each sale, further wasting space when the same customer comes back a week later. Yet another problem is that the sales personnel are responsible for knowing all of the current prices for each item. If they have to look up the price in a printout each time, much of the power and appeal of a computerized system is lost. Solving these and similar problems is therefore the purpose of a relational database.

In a relational database, three separate files would be employed. One file will hold only the customer names and addresses, a second will hold just the item information, and a third is used to store the details of each invoice. In order to bind the three files together, a unique number must be assigned in each record. This is shown as a list of field names in Figure 7-1 below.

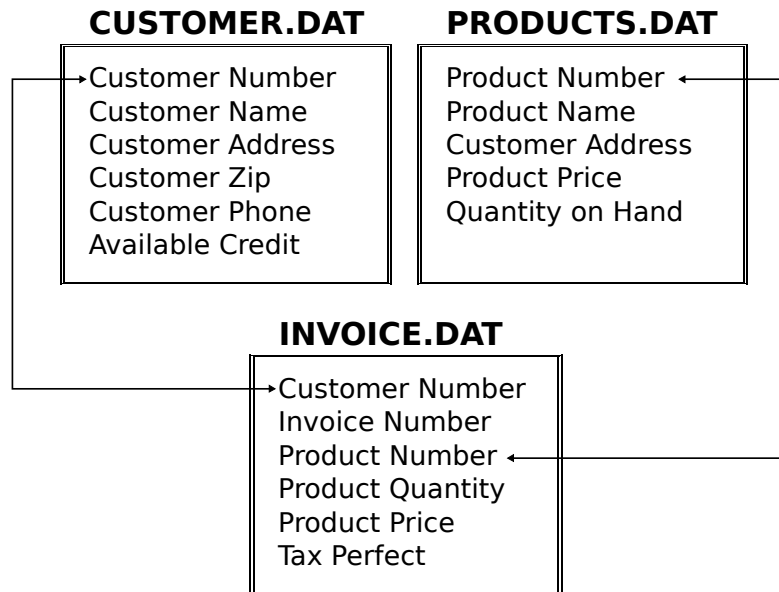


Figure 7-1: How a relational database ties related data in separate files using a unique value in each record.

Now, when Bob Jones goes into the store to buy a radiator cap and a case of motor oil, the clerk can enter the names Jones and see if Bob is already a customer. If so, the order entry program will retrieve Bob's full name and address from the customer file and display it on the screen. Otherwise it would prompt the clerk to enter Bob's name and address. When Bob tells the clerk what he wants to buy, the clerk would enter the part number or name, and the program will automatically look up the price in the products file. A smart program would even subtract the number of radiator caps from the "Quantity on Hand" field, so a report run at the end of each day can identify items that need to be ordered. Once the sale is finalized, two new records will be written to the invoice file—one for the radiator cap and one for the motor oil.

Each invoice record would store Bob's customer number, a program-generated sequential invoice number, the product number, the quantity of this product sold, and the unit price. There's no need to store the subtotal, since that information could be recreated at any time from the quantity and unit price fields. If sales tax is charged, that field could hold just the rate. Again the actual tax amount could be computed at any time. The beauty of this organization is that there is never a need to store duplicated information, and thus there is no wasted disk space.

The relational aspect of this system becomes clear when it is time to produce a report. To print an invoice, the program searches the invoice file for every record with the unique invoice number. From the customer number field the customer's name and address are available, by searching for a match between the customer number in the invoice record and that same unique number in the customer file. And from the part number field the part name can be retrieved, based on finding the same part number in the products file. Thus, the term relational is derived from the ability to relate information in one file to information in a different file, based on unique identifying values. In this case, those values are the invoice number, the customer number, and the part number.

SQL: The Black Box

An important current trend in data processing is the use of Structured Query Language (SQL). The appeal of SQL is that it eliminates explicit coding in a conventional high-level language such as BASIC. Instead, SQL is an even higher-level language that performs most of the low-level details for you. SQL is based on passing SQL commands—called requests—as strings, which are evaluated by the SQL engine. The short example program below shows some typical SQL commands in context.

```
SELECT lastname, firstname, accountcode, phone
FROM customers
WHERE unpaid > credit * .75
      AND today - duedate > 30
ORDER BY accountcode
```

When these commands are sent to the SQL server, the server responds by filling in an array with the resultant data. The beauty of SQL, therefore, is that it eliminates the SELECT CASE statements that you would have to write, and that would be specific to a given data file. In SQL, the data fields are accessed by name instead of by numeric offsets. The SQL program does not have to specify which data is double precision, and which is text, and so forth. Rather, all that is needed is the name of the data being reported on, the selection criteria, and the order in which the data is to be returned.

This program asks to report on the lastname, firstname, accountcode, and phone fields of the data set (file) named customers. It then specifies that only those customers who owe more than 75 percent of their available credit and are more than 30 days overdue should be listed. Finally, the customers are to be listed in order based on their customer account code number.

As a further example of the power of the SQL language, imagine you have written an application to manage a publishing business. In this hypothetical situation, three of the tables in your database are Stores, Titles, and Sales, which hold the names of each retail store, the book titles offered for sale, and the details of each sale.

Now, consider the problem of producing a report showing the total sales in dollars, with individual subtotals for each store. This would first require you generate a list of stores from the Stores table. You would then have to examine each sale in the Sales table, and each entry there would refer to a title which must be looked up in the Titles file to determine the price. You would then multiply this price by

the quantity and add that to a running total being kept for each store, perhaps storing the result in a multi-dimensional array.

As you can see, this is potentially a lot of coding if you attempt to tackle the job using BASIC. While the sequence of SQL commands necessary to retrieve this information is not trivial either, it is certainly less work than writing an equivalent report in BASIC. Here are the SQL commands that perform the store sales report described above:

```
SELECT stores.storename, sum(sales.qty * titles.price)
FROM stores, titles, sales
WHERE stores.store_id = sales.store_id
      AND titles.title_id = sales.title_id
GROUP BY storename
```

As you can see from these short examples, SQL is a simple and intuitive language, and it may well be worth your effort to learn if you specialize in database programming or plan to. One excellent product you may wish to become familiar with is DataEase, a popular PC database product. One of the earliest adopters of SQL-style methods, DataEase lets even the novice user create sophisticated data entry forms and reports in a very short time. Contrast that with procedural languages such as that used by dBASE which require as much effort as programming in BASIC.

There are several good books that go into far greater detail about SQL than can possibly be offered here. One I recommend is *The Practical SQL Handbook: Using Structured Query Language*, by Emerson, Darnovsky, and Bowman; Addison-Wesley Publishing Company; 1989. This book is clearly written, avoids the use of jargon, and contains numerous good explanations of what SQL is all about without getting bogged down in esoteric details.

Programming for a Network

Although network file access has been supported since QuickBASIC version 1.0, many programmers do not fully understand how to use this important feature. However, the concepts are simple once you know the commands. In the earlier auto parts store example, it was assumed that only one computer would be used to enter sales information. But when there are many sales people entering information all at once, some means is needed to let each computer access simultaneously a single group of files from a remote file server.

In this section I will discuss two methods for sharing files—one which is supported by BASIC, and the other supported only indirectly. I will also discuss methods for protecting data across the network and detecting which type of network is being used.

File Sharing and Locking

BASIC offers three commands to allow multiple programs to share files from a central, remote computer: OPEN, LOCK, and UNLOCK. Chapter 6 discussed the OPEN command in great detail, but mentioned the various file sharing options only briefly. OPEN provides four variations that let you specify what other processes have access to the file being opened. For simplicity, the discussions that follow assume the files are being opened for random access; this is the most common access method when writing databases. But only very slight changes are needed to adapt this information for use with binary file access as shown in the earlier dBASE examples.

When you add SHARED to the list of OPEN arguments, you are telling the operating system that any other program may also open the file while you are using it.

```
Without SHARED, another program that tries to open a
file you have opened will receive an "Access denied"
error message.
```

Once the other programs have opened the file they may freely read from it or write to it. If you need to restrict what operations other programs may perform, you would replace SHARED with either LOCK READ, LOCK WRITE, or LOCK READ WRITE.

LOCK READ prevents other program from reading the file while you have it open, although they could write to it. Likewise, LOCK WRITE lets another process read from the file but not write to it. LOCK READ WRITE of course prevents another program from either reading or writing the file.

Because of these complications and limitations, you will most likely use SHARED to allow full file sharing. Then, the details of who writes what and when can be handled by logic in your program, or by locking individual records.

Note that with most networks you cannot open a file for shared access, unless you have previously loaded SHARE.EXE that comes with DOS 3.0 and later versions. SHARE.EXE is a TSR (terminate and stay resident) program that manages *lock tables* for your machine. These tables comprise a list showing which portions of what files are currently locked. A short utility that reports if SHARE.EXE is installed is presented later in this chapter. Some networks, however, require SHARE to be installed only on the computer that is acting as the file server.

Record Locking

The most difficult problem you will encounter when writing a program that runs on a network is arbitrating when each user will be allowed to read and write data. Since more than one operator may call up a given record at the same time, it is possible—even likely—that changes made by one person will be overwritten later by another. Imagine that two operators have just called up the same customer record on their screens. Further, one operator has just changed the customer's address and the other has just changed the phone number. Then the first operator then saves the record with the new address, but

two seconds later the second operator saves the same record with a new phone number. In this case, the second disk write stores the old address on top of the same record that was saved two seconds earlier!

To prevent this from happening requires some type of file locking, whereby the second operator is prevented from even loading the record; the program instead gives them a message saying the record is already in use. There are two primary ways to do this. A *hard lock* is implemented using the BASIC LOCK statement, and it causes the network operating system to deny access to the record if the first program has locked it. A *soft lock* is similar, except it uses program logic that you design to determine if the file is already in use. Let's take a closer at each of these locking methods.

Hard Locks

A hard lock is handled by the network software, and is controlled by the BASIC LOCK and UNLOCK statements. Hard locks may be specified for all or just a part of a file. When a program imposes a hard lock, all other programs are prevented from either reading or writing that portion of the file. You may lock either one record or a range of records: LOCK #1, 3 locks record 3, and UNLOCK #1, 1 TO 10 unlocks records 1 through 10. Files that have been opened for binary access may also be locked, by specifying a range of bytes instead of one or more record numbers.

Because access to the specified record or range of records is denied to all other applications, it is important to unlock the records as soon as you are done with them. A code fragment that shows how to manipulate a record using hard locking would look like this:

```
OPEN "CUST.DAT" SHARED AS #1 LEN = RecordLength%
LOCK #1, RecNum%
GET #1, RecNum%, RecData

'allow the user to edit the record here

PUT #1, RecNum%, RecData
UNLOCK #1, RecNum%
CLOSE #1
```

There are several fundamental problems with hard locks you must be aware of. First, they prevent another application from even looking at the data that is locked. If a record is tied up for a long period of time, this prevents another program from reporting on that data. Another is that all locks must be removed before the file is closed. The BASIC PDS language reference manual warns, "Be sure to remove all locks with an UNLOCK statement before closing a file or terminating your program. Failing to remove locks produces unpredictable results." As in "Yo, get out the Norton disk doctor".

Yet another problem is that each LOCK must have an exactly corresponding UNLOCK statement. It is therefore up to your program to know exactly which record or range of records were locked earlier, and unlock the exact same records later on.

Finally, the last problem with hard locking is that it requires you to use ON ERROR. If someone else has locked a record and you attempt to read it, BASIC will generate a "Permission denied" error that must be trapped. Since there's no way for you to know ahead of time if a record is available or locked

you must be prepared to handle the inevitable errors. Similarly, if you attempt to lock a record when it has already been locked by another program, BASIC will create an error. It is possible to lock and unlock records behind BASIC's back using CALL Interrupt and detect those errors manually; however, soft locks often provide an even better solution.

Soft Locks

A soft lock is implemented using logic you design, which has the decided advantage of letting you customize that logic to your exact needs. Most programs implement a soft lock by reserving a single byte at the beginning of each data record. This is similar to the method dBASE uses to identify deleted records. Understand that the one important limitation of soft locks is that all programs must agree on the method being used. Unless you wrote or at least control all of the other programs that are sharing the file, soft locks will probably not be possible.

One way to implement a soft lock is to use a special character—perhaps the letter "L"—to indicate that a record is in use and may not be written to. Therefore, to lock a record you would first retrieve it, and then check to be sure it isn't already locked. If it is not currently locked you would assign an "L" to the field reserved for that purpose, and finally write the record back to disk. Thereafter, any other program can tell that the record is locked by simply examining that first byte.

If someone tries to access a record that is locked, the program can display the message "Record in use" or something along those lines. A simple enhancement to this would store a user identification number in the lock field, rather than just a locked identifier. This way the program could also report who is using the record, and not just that it is locked. This is shown in context below.

```
GET #1, RecNum%, RecData$
Status$ = LEFT$(RecData$, 1)
SELECT CASE Status$
  CASE " " 'Record is okay to write, lock it now
    MID$(RecData$, 1) = CHR$(UserID)
    PUT #1, RecNum%, RecData$
    GOTO EditRecord
  CASE "*" 'Record is deleted, say so
    PRINT "Record number"; RecNum%; " is deleted."
    GOTO SelectAnotherRecord
  CASE ELSE 'Status$ contains the user number
    PRINT "Record already in use by user: "; Status$
    GOTO ReadOnly
END SELECT
...
...
SaveRecord:
  MID$(RecData$, 1) = " " 'clear the lock status
  PUT #1, RecNum%, RecData$ 'save the new data to disk
```

Additional Network Considerations

Many networks require that SHARE.EXE be installed before a file may be opened for shared access. You can avoid runtime errors by being able to determine ahead of time if this file is loaded. The following short function and example returns either -1 or 0 to indicate if SHARE is currently loaded or not, respectively.

```
DEFINT A-Z
DECLARE FUNCTION ShareThere% ()

'$INCLUDE: 'regtype.bi'

FUNCTION ShareThere% STATIC
  DIM Registers AS RegType
  ShareThere% = -1          'assume Share is loaded
  Registers.AX = &H1000     'service 10h
  CALL Interrupt(&H2F, Registers, Registers)
  AL = Registers.AX AND 255 'isolate the result in AL
  IF AL <> &HFF THEN ShareThere% = 0
END FUNCTION
```

Then, at the start of your program you would invoke ShareThere, and display an error message if SHARE has not been run:

```
IF NOT ShareThere% () THEN
  PRINT "SHARE.EXE is not installed"
END
END IF
```

Operating System Confirmation

Another feature of a well-behaved network application is to determine if the correct network operating system is installed. In most cases, unless you are writing a commercial application for others to use, you'll already know which operating system is expected. However, it is possible to determine with reasonable certainty what network software is currently running. The three functions that follow must be invoked in the order shown, and they help you determine the brand of network your program is running under.

```
'***** NETCHECK.BAS, identifies the network brand

DEFINT A-Z
'$INCLUDE: 'regtype.bi'

DECLARE FUNCTION NWThere% ()
DECLARE FUNCTION BVThere% ()
DECLARE FUNCTION MStThere% ()
DIM SHARED Registers AS RegType

PRINT "I think the network is ";
IF NWThere% THEN
  PRINT "Novell Netware"
ELSEIF BVThere% THEN
  PRINT "Banyon Vines"
ELSEIF MStThere% THEN
  PRINT "Lantastic or other MS compatible"
```



```

ELSE
  PRINT "Something I don't recognize, or no network"
END IF
END

FUNCTION BVThere% STATIC
  BVThere% = -1
  Registers.AX = &HD701
  CALL Interrupt(&H2F, Registers, Registers)
  AL = Registers.AX AND 255
  IF AL <> 0 THEN BVThere% = 0
END FUNCTION

FUNCTION MStThere% STATIC
  MStThere% = -1
  Registers.AX = &HB800
  CALL Interrupt(&H2F, Registers, Registers)
  AL = Registers.AX AND 255
  IF AL = 0 THEN MStThere% = 0
END FUNCTION

FUNCTION NWThere% STATIC
  NWThere% = -1
  Registers.AX = &H7A00
  CALL Interrupt(&H2F, Registers, Registers)
  AL = Registers.AX AND 255
  IF AL <> &HFF THEN NWThere% = 0
END FUNCTION

```

Third-Party Database Tools

There are several tools on the market that can help you to write database applications. Although BASIC includes many of the primitive services necessary for database programming, there are several limitations. Four such products are described briefly below, and all are written in assembly language for fast performance and small code size. You should contact the vendors directly for more information on these products.

AJS Publishing's db/LIB

This is one of the most popular database add-on products for use with BASIC, and rightfully so. db/LIB comes in both single and multi-user versions, and handles all aspects of creating, updating, and indexing relational database files. db/LIB uses the dBASE III+ file format which lets you access files from many different applications. Besides its database handling routines, db/LIB includes a sophisticated expression evaluator that lets you select records based on multiple criteria. Compared to many other database libraries, db/LIB is extremely fast, and is also very easy to use.

db/LIB
AJS Publishing, Inc.
P.O. Box 83220

Los Angeles, CA 90083
213-215-9145

Novell's Btrieve

Btrieve has been around for a very long time, and like db/LIB it lets you easily manipulate all aspects of a relational database. Unlike db/LIB, however, Btrieve can be used with nearly any programming language. The downside is that Btrieve is more complicated to use with BASIC. Also, a special TSR program must be run before your program can call its routines, further complicating matters for your customers. But Btrieve has a large and loyal following, and if you write programs using more than one language it is certainly a product to consider.

Btrieve
Novell, Inc.
122 East 1700 SOutH
Provo, UT 84606
801-429-7000

CDP Consultants' Index Manager

Index Manager is an interesting and unique product, because it handles only the indexing portion of a database program. Where most of the other database add-ons take over all aspects of file creation and updating, Index Manager lets you use any file format you want. Each time a record is to be retrieved based on a key field, a single call obtains the appropriate record number. Index Manager is available in single and multi-user versions, and is designed to work with compiled BASIC only.

Index Manager
CDP Consultants
1700 Circo del Cielo Drive
El Cajon, CA 92020
619-440-6482

Ocelot

Ocelot is unique in that it uses SQL commands instead of the more traditional approach used by the other products mentioned. Ocelot supports both standalone and networked access, and it is both fast and flexible. Although Ocelot is meant for use with several different programming languages, the company provides full support for programmers using BASIC.

Ocelot Computer Services
#1502, 10025-106 Street

Edmonton, Alberta
Canada T5J 1G7
403-421-4187

Summary

In this chapter you learned the principles of data-driven programming, and the advantages this method offers. Unlike the TYPE definition method that Microsoft recommends, storing record and field information as variables allows your programs to access any type of data using the same set of subroutines.

You also learned how to create and access data using the popular dBASE file format, which has the decided advantage of being compatible with a large number of already successful commercial products. A complete set of dBASE file access tools was presented, which may be incorporated directly into your own programs.

This chapter also explained indexing methods, to help you quickly locate information stored in your data files. Besides providing fast access, indexes help to maintain your data in sorted order, facilitating reports on that data. Relational databases were described in detail, using examples to show the importance of maintaining related information in separate files. As long as a unique key value is stored in each record, the information can be joined together at any time for reporting and auditing purposes. SQL was also mentioned, albeit briefly, to provide a glimpse into the future direction that database programming is surely heading.

In the section about programming for a network, a comparison of the various file sharing and locking methods was given. You learned the importance of preventing one program from overwriting data from another, and examined specific code fragments showing two different locking techniques.

Finally, several third-party library products were mentioned. In many situations it is more important to get the job done than to write all of the code yourself. When the absolute fastest performance is necessary, a well written add-on product can often be the best solution to a complex data management problem.

The next chapter discusses searching and sorting data both in memory and on disk, and provides a logical extension to the information presented here. In particular, there are a number of ways that you can speed up index searches using either smarter algorithms, assembly language, or both.

8

Sorting and Searching

Two fundamental operations required of many applications are searching and sorting the data they operate on. Many different types of data are commonly sorted, such as customer names, payment due dates, or even a list of file names displayed in a file selection menu. If you are writing a programmer's cross reference utility, you may need to sort a list of variable names without regard to capitalization. In some cases, you may want to sort several pieces of related information based on the contents of only one of them. One example of that is a list of names and addresses sorted in ascending Zip code order.

Searching is equally important; for example, to locate a customer name in an array or disk file. In some cases you may wish to search for a complete match, while in others a partial match is needed. If you are searching a list of names for, say, Leonard, you probably would want to ignore Leonardo. But when searching a list of Zip codes you may need to locate all that begin with the digits 068. There are many different ways sorting and searching can be accomplished, and the subject is by no means a simple one.

Most programmers are familiar with the Bubble Sort, because it is the simplest to understand. Each adjacent pair of items is compared, and then exchanged if they are out of order. This process is repeated over and over, until the entire list has been examined as many times as there are items. Unfortunately, these repeated comparisons make the Bubble Sort an extremely poor performer. Similarly, code to perform a linear search that simply examines each item in succession for a match is easy to grasp, but it will be painfully slow when there are many items.

In this chapter you will learn how sophisticated algorithms that handle these important programming chores operate. You will also learn how to sort data on more than one key. Often, it is not sufficient to merely sort a list of customers by their last name. For example, you may be expected to sort first by last name, then by first name, and finally by balance due. That is, all of the last names would first be sorted. Then within all of the Smiths you would sort again by first name, and for all of the John Smiths sort that subgroup based on how much money is owed.

For completeness I will start each section by introducing sorting and searching methods that are easy to understand, and then progress to the more complex algorithms that are much more effective. Specifically, I will show the Quick Sort and Binary Search algorithms. When there are many thousands of data items, a good algorithm can make the difference between a sort routine that takes ten minutes to complete, and one that needs only a few seconds.

Finally, I will discuss both BASIC and assembly language sort routines. As important as the right algorithm is for good performance, an assembly language implementation will be even faster. Chapter 12 describes how assembly language routines are written and how they work, and in this chapter I will merely show how to use the routines included with this book.

Sorting Fundamentals

```
Initial array contents:
Element 4  Kathy
Element 3  Barbara
Element 2  Cathy
Element 1  Zorba <

After 1 pass:
Element 4  Kathy
Element 3  Barbara
Element 2  Zorba <
Element 1  Cathy

After 2 passes:
Element 4  Kathy
Element 3  Zorba <
Element 2  Barbara
Element 1  Cathy

After 3 passes:
Element 4  Zorba <
Element 3  Kathy
Element 2  Barbara
Element 1  Cathy
```

Figure 8.1: Data ascending a list during a bubble sort.

Although there are many different ways to sort an array, the simplest sorting algorithm is the Bubble Sort. The name Bubble is used because a FOR/NEXT loop repeatedly examines each adjacent pair of elements in the array, and those that have higher values rise to the top like bubbles in a bathtub. The most common type of sort is ascending, which means that "A" comes before "B", which comes before "C", and so forth. Figure 8-1 shows how the name Zorba ascends to the top of a five-item list of first names.

The Bubble Sort routine that follows uses a FOR/NEXT loop to repeatedly examine an array and exchange elements as necessary, until all of the items are in the correct order.

```
DEFINT A-Z
```

```

DECLARE SUB BubbleSort (Array$())

CONST NumItems% = 20
CONST False% = 0
CONST True% = -1

DIM Array$(1 TO NumItems%)
FOR X = 1 TO NumItems%
    READ Array$(X)
NEXT

CALL BubbleSort (Array$())

CLS
FOR X = 1 TO NumItems%
    PRINT Array$(X)
NEXT

DATA Zorba, Cathy, Barbara, Kathy, Josephine
DATA Joseph, Joe, Peter, Arnold, Glen
DATA Ralph, Elli, Lucky, Rocky, Louis
DATA Paula, Paul, Mary Lou, Marilyn, Keith
END

SUB BubbleSort (Array$()) STATIC
DO
    OutOfOrder = False%           'assume it's sorted
    FOR X = 1 TO UBOUND(Array$) - 1
        IF Array$(X) > Array$(X + 1) THEN
            SWAP Array$(X), Array$(X + 1) 'if we had to swap
            OutOfOrder = True%         'we may not be done
        END IF
    NEXT
LOOP WHILE OutOfOrder
END SUB

```

This routine is simple enough to be self-explanatory, and only a few things warrant discussing. One is the `OutOfOrder` flag variable. When the array is nearly sorted to begin with, fewer passes through the loop are needed. The `OutOfOrder` variable determines when no more passes are necessary. It is cleared at the start of each loop, and set each time two elements are exchanged. If, after examining all of the elements in one pass no exchanges were required, then the sorting is done and there's no need for the `DO` loop to continue.

The other item worth mentioning is that the `FOR/NEXT` loop is set to consider one element less than the array actually holds. This is necessary because each element is compared to the one above it. If the last element were included in the loop, then BASIC would issue a "Subscript out of range" error on the statement that examines `Array$(X + 1)`.

There are a number of features you can add to this Bubble Sort routine. For example, you could sort without regard to capitalization. In that case "adams" would come before "BAKER", even though the lowercase letter "a" has a higher ASCII value than the uppercase letter "B". To add that capability simply use BASIC's `UCASE$` (or `LCASE$`) function as part of the comparisons:

```

IF UCASE$(Array$(X)) > UCASE$(Array$(X + 1)) THEN

```

And to sort based on the eight-character portion that starts six bytes into each string you would use this:

```
IF MID$(Array$(X), 5, 8) > MID$(Array$(X + 1), 5, 8) THEN
```

Although the comparisons in this example are based on just a portion of each string, the SWAP statement must exchange the entire elements. This opens up many possibilities as you will see later in this chapter.

If there is a chance that the strings may contain trailing blanks that should be ignored, you can use RTRIM\$ on each pair of elements:

```
IF RTRIM$(Array$(X)) > RTRIM$(Array$(X + 1)) THEN
```

Of course, you can easily combine these enhancements to consider only the characters in the middle after they have been converted to upper or lower case.

Sorting in reverse (descending) order is equally easy; you'd simply replace the greater-than symbol (>) with a less-than symbol (<).

Finally, you can modify the routine to work with any type of data by changing the array type identifier. That is, for every occurrence of Array\$ you will change that to Array% or Array# or whatever is appropriate. If you are sorting a numeric array, then different modifications may be in order. For example, to sort ignoring whether the numbers are positive or negative you would use BASIC's ABS (absolute value) function:

```
IF ABS(Array!(X)) > ABS(Array!(X + 1)) THEN
```

It is important to point out that all of the simple modifications described here can also be applied to the more sophisticated sort routines we will look at later in this chapter.

Indexed Sorts

Besides the traditional sorting methods—whether a Bubble Sort or Quick Sort or any other type of sort—there is another category of sort routine you should be familiar with. Where a conventional sort exchanges elements in an array until they are in order, an Index Sort instead exchanges elements in a parallel numeric array of *pointers*. The original data is left intact, so it may still be accessed in its natural order. However, the array can also be accessed in sorted order by using the element numbers contained in the index array.

As with a conventional sort, the comparisons in an indexed sort routine examine each element in the primary array, but based on the element numbers in that index array. If it is determined that the data is out of order, the routine exchanges the elements in the index array instead of the primary array. A modification to the Bubble Sort routine to sort using an index is shown below.

```

DEFINT A-Z
DECLARE SUB BubbleISort (Array$(), Index())

CONST NumItems% = 20
CONST False% = 0
CONST True% = -1

DIM Array$(1 TO NumItems%) 'this holds the string data
DIM Ndx(1 TO NumItems%)    'this holds the index

FOR X = 1 TO NumItems%
    READ Array$(X)          'read the string data
    Ndx(X) = X              'initialize the index array
NEXT

CALL BubbleISort(Array$(), Ndx())

CLS
FOR X = 1 TO NumItems%
    PRINT Array$(Ndx(X))    'print based on the index
NEXT

DATA Zorba, Cathy, Barbara, Kathy, Josephine
DATA Joseph, Joe, Peter, Arnold, Glen
DATA Ralph, Elli, Lucky, Rocky, Louis
DATA Paula, Paul, Mary lou, Marilyn, Keith

SUB BubbleISort (Array$(), Index()) STATIC
DO
    OutOfOrder = False%    'assume it's sorted
    FOR X = 1 TO UBOUND(Array$) - 1
        IF Array$(Index(X)) > Array$(Index(X + 1)) THEN
            SWAP Index(X), Index(X + 1)    'if we had to swap
            OutOfOrder% = True%           'we're not done yet
        END IF
    NEXT
LOOP WHILE OutOfOrder%
END SUB

```

In this indexed sort, all references to the data are through the index array. And when a swap is necessary, it is the index array elements that are exchanged. Note that an indexed sort requires that the index array be initialized to increasing values—even if the sort routine is modified to be descending instead of ascending. Therefore, when BubbleISort is called Ndx(1) must hold the value 1, Ndx(2) is set to 2, and so forth.

In this example the index array is initialized by the caller. However, it would be just as easy to put that code into the subprogram itself. Since you can't pass an array that hasn't yet been dimensioned, it makes the most sense to do both steps outside of the subprogram. Either way, the index array must be assigned to these initial values.

As I mentioned earlier, one feature of an indexed sort is that it lets you access the data in both its original and sorted order. But there are other advantages, and a disadvantage as well. The disadvantage is that each comparison takes slightly longer, because of the additional overhead required to first look up the element number in the index array, to determine which elements in the primary array will be

compared. In some cases, though, that can be more than offset by requiring less time to exchange elements.

If you are sorting an array of 230-byte TYPE variables, the time needed for SWAP to exchange the elements can become considerable. Every byte in both elements must be read and written, so the time needed increases linearly as the array elements become longer. Contrast that with the fixed two bytes in the integer index array that are swapped.

Another advantage of an indexed sort is that it lends itself to sorting more data than can fit in memory. As you will see later in the section that shows how to sort files, it is far easier to manipulate an integer index than an entire file. Further, sorting the file data using multiple passes requires twice as much disk space as the file already occupies.

Data Manipulation Techniques

Before I show the Quick Sort algorithm that will be used as a basis for the remaining sort examples in this chapter, you should also be aware of a few simple tricks that can help you maintain and sort your data. One was described in Chapter 6, using a pair of functions that pack and unpack dates such that the year is stored before the month, which in turn is before the day. Thus, date strings are reduced to only three characters each, and they can be sorted directly.

Another useful speed-up trick is to store string data as integers or long integers. If you had a system of four-digit account numbers you could use an integer instead of a string. Besides saving half the memory and disk space, the integer comparisons in a sort routine will be many times faster than a comparison on string equivalents. Zip codes are also suited to this, and could be stored in a long integer. Even though the space savings is only one byte, the time needed to compare the values for sorting will be greatly reduced.

This brings up another important point. As you learned in Chapter 2, all conventional (not fixed-length) strings require more memory than might be immediately apparent. Besides the amount of memory needed to hold the data itself, four additional bytes are used for a string descriptor, and two more beyond those for a back pointer. Therefore, a Zip code stored as a string will actually require eleven bytes rather than the five you might expect. With this in mind, you may be tempted to think that using a fixed-length string to hold the Zip codes will solve the problem. Since fixed-length strings do not use either descriptors or back pointers, they do not need the memory they occupy. And that leads to yet another issue.

Whenever a fixed-length string or the string portion of a TYPE variable is compared, it must first be converted to a regular descriptor string. BASIC has only one string comparison routine, and it expects the addresses for two conventional string descriptors. Every time a fixed-length string is used as an argument for comparison, BASIC must create a temporary copy, call its comparison routine, and then delete the copy. This copying adds code and wastes an enormous amount of time; in many cases the

copying will take longer than the comparison itself. Therefore, using integers and long integers for numeric data where possible will provide more improvement than just the savings in memory use.

In some cases, however, you must use fixed-length string or TYPE arrays. In particular, when sorting information from a random access disk file it is most sensible to load the records into a TYPE array. And as you learned in Chapter 2, the string components of a TYPE variable or array element are handled by BASIC as a fixed-length string. So how can you effectively sort fixed-length string arrays without incurring the penalty BASIC's overhead imposes? With assembly language subroutines, of course!

Rather than ask BASIC to pass the data using its normal methods, assembly language routines can be invoked passing the data segments and addresses directly. When you use SEG, or a combination of VARSEG and VARPTR with fixed-length and TYPE variables, BASIC knows that you want the segmented address of the variable or array element. Thus, you are tricking BASIC into not making a copy as it usually would when passing such data. An assembly language subroutine or function can be designed to accept data addresses in any number of ways. As you will see later when we discuss sorting on multiple keys, extra trickery is needed to do the same thing in a BASIC procedure.

The three short assembly language functions that follow compare two portions of memory, and then return a result that can be tested by your program.

```
;COMPARE.ASM - compares two ranges of memory

.Model Medium, Basic
.Code

Compare Proc Uses DS ES DI SI, SegAdr1:DWord, _
    SegAdr2:DWord, NumBytes:Word

    Cld                ;compare in the forward direction
    Mov  SI,NumBytes   ;get the address for NumBytes%
    Mov  CX,[SI]       ;put it into CX for comparing below
    Les  DI,SegAdr1    ;load ES:DI with the first
                    ; segmented address
    Lds  SI,SegAdr2    ;load DS:SI with the second
                    ; segmented address

    Repe Cmpsb        ;do the compare
    Mov  AX,0          ;assume the bytes didn't match
    Jne  Exit          ;we were right, skip over
    Dec  AX            ;wrong, decrement AX down to -1

Exit:
    Ret                ;return to BASIC

Compare Endp
End

;COMPARE2.ASM - compares memory case-insensitive

.Model Medium, Basic
.Code
```

```

Compare2 Proc Uses DS ES DI SI, SegAdr1:DWord, _
    SegAdr2:DWord, NumBytes:Word

    Cld                ;compare in the forward direction
    Mov  BX,-1         ;assume the ranges are the same

    Mov  SI,NumBytes   ;get the address for NumBytes%
    Mov  CX,[SI]       ;put it into CX for comparing below
    Jcxz Exit         ;if zero bytes were given, they're
                    ; the same
    Les  DI,SegAdr1    ;load ES:DI with the first address
    Lds  SI,SegAdr2    ;load DS:SI with the second address
Do:
    Lodsb              ;load the current character from
                    ; DS:SI into AL
    Call Upper         ;capitalize as necessary
    Mov  AH,AL         ;copy the character to AH

    Mov  AL,ES:[DI]    ;load the other character into AL
    Inc  DI            ;point at the next one for later
    Call Upper         ;capitalize as necessary

    Cmp  AL,AH         ;now, are they the same?
    Jne  False        ;no, exit now and show that
    Loop Do           ;yes, continue
    Jmp  Short Exit    ;if we get this far, the bytes are
                    ; all the same
False:
    Inc  BX            ;increment BX to return zero

Exit:
    Mov  AX,BX        ;assign the function output
    Ret              ;return to BASIC

Upper:
    Cmp  AL,"a"       ;is the character below an "a"?
    Jb   Done         ;yes, so we can skip it
    Cmp  AL,"z"       ;is the character above a "z"?
    Ja   Done         ;yes, so we can skip that too
    Sub  AL,32        ;convert to upper case

Done:
    Retn             ;do a near return to the caller

Compare2 Endp
End

```

;COMPARE3.ASM - case-insensitive, greater/less than

```

.MODEL Medium, Basic
.CODE

```

```

Compare3 Proc Uses DS ES DI SI, SegAdr1:DWord, _
    SegAdr2:DWord, NumBytes:Word

    Cld                ;compare in the forward direction
    Xor  BX,BX         ;assume the ranges are the same

    Mov  SI,NumBytes   ;get the address for NumBytes%
    Mov  CX,[SI]       ;put it into CX for comparing below
    Jcxz Exit         ;if zero bytes were given, they're

```

```

        ; the same
    Les  DI,SegAdr1 ;load ES:DI with the first address
    Lds  SI,SegAdr2 ;load DS:SI with the second address

Do:
    Lodsb          ;load the current character from
                  ; DS:SI into AL
    Call Upper    ;capitalize as necessary, remove for
                  ; case-sensitive
    Mov  AH,AL     ;copy the character to AH

    Mov  AL,ES:[DI] ;load the other character into AL
    Inc  DI        ;point at the next character for later
    Call Upper    ;capitalize as necessary, remove for
                  ; case-sensitive

    Cmp  AL,AH     ;now, are they the same?
    Loope Do      ;yes, continue
    Je   Exit     ;we exhausted the data and they're
                  ; the same
    Mov  BL,1     ;assume block 1 was "greater"
    Ja  Exit     ;we assumed correctly
    Dec  BX      ;wrong, bump BX down to -1
    Dec  BX

Exit:
    Mov  AX,BX    ;assign the function output
    Ret          ;return to BASIC

Upper:
    Cmp  AL,"a"   ;is the character below an "a"?
    Jb  Done     ;yes, so we can skip it
    Cmp  AL,"z"   ;is the character above a "z"?
    Ja  Done     ;yes, so we can skip that too
    Sub  AL,32    ;convert to upper case

Done:
    Retn         ;do a near return to the caller

Compare3 Endp
End

```

The first Compare routine above simply checks if all of the bytes are identical, and returns -1 (True) if they are, or 0 (False) if they are not. By returning -1 or 0 you can use either depending on which logic is clearer for your program:

```

IF Compare%(Type1, Type2, NumBytes%) THEN
or
IF NOT Compare%(Type1, Type2, NumBytes%) THEN

```

Compare2 is similar to Compare, except it ignores capitalization. That is, "SMITH" and "Smith" are considered equal. The Compare3 function also compares memory and ignores capitalization, but it returns either -1, 0, or 1 to indicate if the first data range is less than, equal to, or greater than the second.

The correct declaration and usage for each of these routines is shown below. Note that Compare and Compare2 are declared and used in the same fashion.

Compare and Compare2:

```
DECLARE FUNCTION Compare%(SEG Type1 AS ANY, SEG Type2 AS ANY, NumBytes%)
  Same = Compare%(Type1, Type2, NumBytes%)
OR
DECLARE FUNCTION Compare%(BYVAL Seg1%, BYVAL Adr1%, BYVAL Seg2%, BYVAL Adr2%,
  NumBytes%)
  Same = Compare%(Seg1%, Adr1%, Seg2%, Adr2%, NumBytes%)
```

Here, Same receives -1 if the two TYPE variables or ranges of memory are the same, or 0 if they are not. NumBytes% tells how many bytes to compare.

Compare3:

```
DECLARE FUNCTION Compare3%(SEG Type1 AS ANY, SEG Type2 AS ANY, NumBytes%)
  Result = Compare3%(Type1, Type2, NumBytes%)
OR
DECLARE FUNCTION Compare3%(BYVAL Seg1%, BYVAL Adr1%, BYVAL Seg2%, BYVAL Adr2%,
  NumBytes%)
  Result = Compare3%(Seg1%, Adr1%, Seg2%, Adr2%, NumBytes%)
```

Result receives 0 if the two type variables or ranges of memory are the same, -1 if the first is less when compared as strings, or 1 if the first is greater. NumBytes% tells how many bytes are to be compared. In the context of a sort routine you could invoke Compare3 like this:

```
IF Compare3%(TypeEl(X), TypeEl(X + 1), NumBytes%) = 1 THEN
  SWAP TypeEl(X), TypeEl(X + 1)
END IF
```

As you can see, these routines may be declared in either of two ways. When used with TYPE arrays the first is more appropriate and results in slightly less setup code being generated by the compiler. When comparing fixed-length strings or arbitrary blocks of memory (for example, when one of the ranges is on the display screen) you should use the second method. Since SEG does not work correctly with fixed-length strings, if you want to use that more efficient version you must create a dummy TYPE comprised solely of a single string portion:

```
TYPE FixedLength
  Something AS STRING * 35
END TYPE
```

Then simply use DIM to create a single variable or an array based on this or a similar TYPE, depending on what your program needs. The requirement to create a dummy TYPE was discussed in Chapter 2. These comparison routines will be used extensively in the sort routines presented later in this chapter; however, their value in other, non-sorting situations should also be apparent.

Although these routines are written in assembly language, they are fairly simple to follow. It is important to understand that you do not need to know anything about assembly language to use them. All of the files you need to add these and all of the other routines in this book are contained on the accompanying ZIP file with this text. Chapter 12 discusses assembly language in great detail, and you can refer there for further explanation of the instructions used.

If you plan to run the programs that follow in the QuickBASIC editor, you must load the BASIC.QLB Quick Library as follows:

```
qb program /l basic
```

Later when you compile these or other programs you must link with the parallel BASIC.LIB file:

```
bc program [/o];  
link program , , nul , basic;
```

If you are using BASIC PDS start QBX using the BASIC7.QLB file, and then link with BASIC7.LIB to produce a stand-alone .EXE program. (VB/DOS users will also use the BASIC7 version.)

The Quick Sort Algorithm

It should be obvious to you by now that a routine written in assembly language will always be faster than an equivalent written in BASIC. However, simply translating a procedure to assembly language is not always the best solution. Far more important than which language you use is selecting an appropriate algorithm. The best sorting method I know is the Quick Sort, and a well-written version of Quick Sort using BASIC will be many times faster than an assembly language implementation of the Bubble Sort.

The main problem with the Bubble Sort is that the number of comparisons required grows exponentially as the number of elements increases. Since each pass through the array exchanges only a few elements, many passes are required before the entire array is sorted. The Quick Sort was developed by C.A.R. (Tony) Hoare, and is widely recognized as the fastest algorithm available. In some special cases, such as when the data is already sorted or nearly sorted, the Quick Sort may be slightly slower than other methods. But in most situations, a Quick Sort is many times faster than any other sorting algorithm.

As with the Bubble Sort, there are many different variations on how a Quick Sort may be coded. You may have noticed that the Bubble Sort shown in Chapter 7 used a nested FOR/NEXT loop, while the one shown here uses a FOR/NEXT loop within a DO/WHILE loop. A Quick Sort divides the array into sections—sometimes called partitions—and then sorts each section individually. Many implementations therefore use recursion to invoke the subprogram from within itself, as each new section is about to be sorted. However, recursive procedures in any language are notoriously slow, and also consume stack memory at an alarming rate.

The Quick Sort version presented here avoids recursion, and instead uses a local array as a form of stack. This array stores the upper and lower bounds showing which section of the array is currently being considered. Another refinement I have added is to avoid making a copy of elements in the array. As a Quick Sort progresses, it examines one element selected arbitrarily from the middle of the array, and compares it to the elements that lie above and below it. To avoid assigning a temporary copy this version simply keeps track of the selected element number.

When sorting numeric data, maintaining a copy of the element is reasonable. But when sorting strings—especially strings whose length is not known ahead of time—the time and memory required to keep a copy can become problematic. For clarity, the generic Quick Sort shown below uses the copy method. Although this version is meant for sorting a single precision array, it can easily be adapted to sort any type of data by simply changing all instances of the "!" type declaration character.

```
'***** QSORT.BAS, Quick Sort algorithm demonstration

'Copyright (c) 1991 Ethan Winer

DEFINT A-Z
DECLARE SUB QSort (Array!(), StartEl, NumEls)

RANDOMIZE TIMER          'generate a new series each run

DIM Array!(1 TO 21)     'create an array
FOR X = 1 TO 21         'fill with random numbers
    Array!(X) = RND(1) * 500 'between 0 and 500
NEXT

FirstEl = 6            'sort starting here
NumEls = 10            'sort this many elements

CLS
PRINT "Before Sorting:"; TAB(31); "After sorting:"
PRINT "====="; TAB(31); "======"

FOR X = 1 TO 21        'show them before sorting
    IF X >= FirstEl AND X <= FirstEl + NumEls - 1 THEN
        PRINT "==>";
    END IF
    PRINT TAB(5); USING "###.##"; Array!(X)
NEXT
CALL QSort(Array!(), FirstEl, NumEls)

LOCATE 3
FOR X = 1 TO 21        'print them after sorting
    LOCATE , 30
    IF X >= FirstEl AND X <= FirstEl + NumEls - 1 THEN
        PRINT "==>"; 'point to sorted items
    END IF
    LOCATE , 35
    PRINT USING "###.##"; Array!(X)
NEXT

SUB QSort (Array!(), StartEl, NumEls) STATIC
REDIM QStack(NumEls \ 5 + 10) 'create a stack array

First = StartEl        'initialize work variables
```

```

Last = StartEl + NumEls - 1
StackPtr = 0
DO
  DO
    Temp! = Array!((Last + First) \ 2) 'seek midpoint
    I = First
    J = Last

    DO 'reverse both < and > below to sort descending
      WHILE Array!(I) < Temp!
        I = I + 1
      WEND
      WHILE Array!(J) > Temp!
        J = J - 1
      WEND
      IF I > J THEN EXIT DO
      IF I < J THEN SWAP Array!(I), Array!(J)
      I = I + 1
      J = J - 1
    LOOP WHILE I <= J

    IF I < Last THEN
      QStack(StackPtr) = I 'Push I
      QStack(StackPtr + 1) = Last 'Push Last
      StackPtr = StackPtr + 2
    END IF

    Last = J
  LOOP WHILE First < Last

  IF StackPtr = 0 THEN EXIT DO 'Done
  StackPtr = StackPtr - 2
  First = QStack(StackPtr) 'Pop First
  Last = QStack(StackPtr + 1) 'Pop Last
LOOP

ERASE QStack 'delete the stack array
END SUB

```

Notice that I have designed this routine to allow sorting only a portion of the array. To sort the entire array you'd simply omit the StartEl and NumEls parameters, and assign First and Last from the LBOUND and UBOUND element numbers. That is, you will change these:

```

First = StartEl
and
Last = StartEl + NumEls - 1

```

to these:

```

First = LBOUND(Array!)
and
Last = UBOUND(Array!)

```

As I mentioned earlier, the QStack array serves as a table of element numbers that reflect which range of elements is currently being considered. You will need to dimension this array to one element for every five elements in the primary array being sorted, plus a few extra for good measure. In this

program I added ten elements, because one stack element for every five main array elements is not enough for very small arrays. For data arrays that have a large amount of duplicated items, you will probably need to increase the size of the stack array.

Note that this ratio is not an absolute. The exact size of the stack that is needed depends on how badly out of order the data is to begin with. Although it is possible that one stack element for every five in the main array is insufficient in a given situation, I have never seen this formula fail. Because the stack is a dynamic integer array that is stored in far memory, it will not impinge on near string memory. If this routine were designed using the normal recursive method, BASIC's stack would be used which is in near memory.

Each of the innermost DO loops searches the array for the first element in each section about the midpoint that belongs in the other section. If the elements are indeed out of order (when I is less than J) the elements are exchanged. This incrementing and comparing continues until I and J cross. At that point, assuming the variable I has not exceeded the upper limits of the current partition, the partition bounds are saved and Last is assigned to the top of the next inner partition level. When the entire partition has been processed, the previous bounds are retrieved, but as a new set of First and Last values. This process continues until no more partition boundaries are on the stack. At that point the entire array is sorted.

In the accompanying ZIP you will find a program called SEEQSORT.BAS that contains an enhanced version of the QSort demo and subprogram. This program lets you watch the progress of the comparisons and exchanges as they are made, and actually see this complex algorithm operate. Simply load SEEQSORT.BAS into the BASIC editor and run it. A constant named Delay! is defined at the beginning of the program. Increasing its value makes the program run more slowly; decreasing it causes the program to run faster.

An Assembly Language Quick Sort

As fast as the BASIC QuickSort routine is, we can make it even faster. The listing below shows an assembly language version that is between ten and twenty percent faster, depending on which compiler you are using and if the BASIC PDS /fs (far strings) option is in effect.

```
;SORT.ASM - sorts an entire BASIC string array

.Model Medium, Basic
.Data
    S            DW 0
    F            DW 0
    L            DW 0
    I            DW 0
    J            DW 0
    MidPoint    DW 0

.Code
    Extern B$$WSD:Proc ;this swaps two strings
    Extern B$$CMP:Proc ;this compares two strings
```

Sort Proc Uses SI DI ES, Array:Word, Dir:Word

```
Cld ;all fills and compares are forward
Push DS ;set ES = DS for string compares
Pop ES

Xor CX,CX ;clear CX
Mov AX,7376h ;load AL and AH with the opcodes
; Jae and Jbe in preparation for
; code self-modification

Mov BX,Dir ;get the sorting direction
Cmp [BX],CX ;is it zero (ascending sort)?
Je Ascending ;yes, skip ahead
Xchg AL,AH ;no exchange the opcodes
```

Ascending:

```
Mov CS:[X1],AH ;install correct comparison opcodes
Mov CS:[X2],AL ; based on the sort direction

Mov BX,Array ;load the array descriptor address
Mov AX,[BX+0Eh] ;save the number of elements
Dec AX ;adjust the number to zero-based
Jns L0 ;at least 1 element, continue
Jmp L4 ;0 or less elements, get out now!
```

L0:

```
Mov BX,Array ;reload array descriptor address
Mov BX,[BX] ;Array$(LBOUND) descriptor address
Mov S,SP ;StackPtr = 0 (normalized to SP)
Mov F,CX ;F = 0
Mov L,AX ;L = Size%
```

;----- calculate the value of MidPoint

L1:

```
Mov DI,L ;MidPoint = (L + F) \ 2
Add DI,F
Shr DI,1
Mov MidPoint,DI

Mov AX,F ;I = F
Mov I,AX

Mov AX,L ;J = L
Mov J,AX
```

;----- calculate the offset into the descriptor table for Array\$(MidPoint

L1_2:

```
Shl DI,1 ;multiply MidPoint in DI times 4
Shl DI,1 ;now DI holds how far beyond Array$(Start)
;Array$(MidPoint)'s descriptor is
Add DI,BX ;add the array base address to produce the final
;address for Array$(MidPoint)
```

;----- calculate descriptor offset for Array\$(I)

L2:

```
Mov SI,I ;put I into SI
Shl SI,1 ;as above
Shl SI,1 ;now SI holds how far beyond Array$(Start)
; Array$(I)'s descriptor is
```

```

Add SI,BX          ;add the base to produce the final descriptor
                   ; address

;IF Array$(I) < Array$(MidPoint) THEN I = I + 1: GOTO L2
Push BX           ;save BX because B$SCMP trashes it
Push SI
Push DI
Call B$SCMP      ;do the compare
Pop BX          ;restore BX

X1 Label Byte     ;modify the code below to "Jbe" if descending sort
Jae L2_1         ;Array$(I) isn't less, continue on
Inc Word Ptr I   ;I = I + 1
Jmp Short L2     ;GOTO L2

;----- calculate descriptor offset for Array$(J)
L2_1:
Mov SI,J         ;put J into SI
Shl SI,1        ;as above
Shl SI,1        ;now SI holds how far beyond Array$(Start)
                   ; Array$(J)'s descriptor is
Add SI,BX       ;add the base to produce the final descriptor
                   ; address

;IF Array$(J) > Array$(MidPoint) THEN J = J - 1: GOTO L2.1
Push BX         ;preserve BX
Push SI
Push DI
Call B$SCMP    ;do the compare
Pop BX        ;restore BX

X2 Label Byte   ;modify the code below to "Jae" if descending sort
Jbe L2_2       ;Array$(J) isn't greater, continue on
Dec Word Ptr J ;J = J - 1
Jmp Short L2_1 ;GOTO L2.1

L2_2:
Mov AX,I       ;IF I > J GOTO L3
Cmp AX,J
Jg L3         ;J is greater, go directly to L3
Je L2_3       ;they're the same, skip the swap

;Swap Array$(I), Array$(J)
Mov SI,I       ;put I into SI
Mov DI,J       ;put J into DI

Cmp SI,MidPoint ;IF I = MidPoint THEN MidPoint = J
Jne No_Mid1    ;not equal, skip ahead
Mov MidPoint,DI ;equal, assign MidPoint = J
Jmp Short No_Mid2 ;don't waste time comparing again

No_Mid1:
Cmp DI,MidPoint ;IF J = MidPoint THEN MidPoint = I
Jne No_Mid2    ;not equal, skip ahead
Mov MidPoint,SI ;equal, assign MidPoint = I

No_Mid2:
Mov SI,I       ;put I into SI
Shl SI,1      ;multiply times four for the
Shl SI,1      ; for the descriptors
Add SI,BX     ;add address for first descriptor

```

```

    Mov  DI,J          ;do the same for J in DI
    Shl  DI,1
    Shl  DI,1
    Add  DI,BX

    Push BX          ;save BX because B$SWSD destroys it
    Call B$SWSD      ;and swap 'em good
    Pop  BX

L2_3:
    Inc  Word Ptr I   ;I = I + 1
    Dec  Word Ptr J   ;J = J - 1

    Mov  AX,I         ;IF I <= J GOTO L2
    Cmp  AX,J
    Jg   L3           ;it's greater, skip to L3
    Mov  DI,MidPoint  ;get MidPoint again
    Jmp  L1_2        ;go back to just before L2

L3:
    Mov  AX,I         ;IF I < L THEN PUSH I: PUSH L
    Cmp  AX,L
    Jnl  L3_1        ;it's not less, so skip Pushes

    Push I           ;Push I
    Push L           ;Push L

L3_1:
    Mov  AX,J         ;L = J
    Mov  L,AX

    Mov  AX,F         ;IF F < L GOTO L1
    Cmp  AX,L
    Jnl  L3_2        ;it's not less, jump ahead to L3_2
    Jmp  L1          ;it's less, go to L1

L3_2:
    Cmp  S,SP        ;IF S = 0 GOTO L4
    Je   L4
    Pop  L           ;Pop L
    Pop  F           ;Pop F
    Jmp  L1         ;GOTO L1

L4:
    Ret              ;return to BASIC

Sort Endp
End

```

Besides being faster than the BASIC version, the assembly language Sort routine is half the size. This version also supports sorting either forward or backward, but not just a portion of an array. The general syntax is:

```
CALL Sort(Array$(), Direction)
```

Where Array\$() is any variable-length string array, and Direction is 0 for ascending, or any other value for descending. Note that this routine calls upon BASIC's internal services to perform the actual

comparing and swapping; therefore, the exact same code can be used with either QuickBASIC or BASIC PDS. Again, I refer you forward to Chapter 12 for an explanation of the assembly language commands used in SORT.ASM.

Sorting on Multiple Keys

In many situations, sorting based on one key is sufficient. For example, if you are sorting a mailing list to take advantage of bulk rates you must sort all of the addresses in order by Zip code. When considering complex data such as a TYPE variable, it is easy to sort the array based on one component of each element. The earlier Bubble Sort example showed how MID\$ could be used to consider just a portion of each string, even though the entire elements were exchanged. Had that routine been designed to operate on a TYPE array, the comparisons would have examined just one component, but the SWAP statements would exchange entire elements:

```
IF Array(X).ZipCode > Array(X + 1).ZipCode THEN
    SWAP Array(X), Array(X + 1)
END IF
```

This way, each customer's last name, first name, street address, and so forth remain connected to the Zip codes that are being compared and exchanged.

There are several ways to sort on more than one key, and all are of necessity more complex than simply sorting based on a single key. One example of a multi-key sort first puts all of the last names in order. Then within each group of identical last names the first names are sorted, and within each group of identical last and first names further sorting is performed on yet another key—perhaps Balance Due. As you can see, this requires you to sort based on differing types of data, and also to compare ranges of elements for the subgroups that need further sorting.

The biggest complication with this method is designing a calling syntax that lets you specify all of the information. A table array must be established to hold the number of keys, the type of data in each key (string, double precision, and so forth), and how many bytes into the TYPE element each key portion begins. Worse, you can't simply use the name of a TYPE component in the comparisons inside the sort routine—which would you use: Array(X).LastName, Array(X).FirstName, or Array(X).ZipCode? Therefore, a truly general multi-key sort must be called passing the address where the array begins in memory, and a table of offsets beyond that address where each component being considered is located.

To avoid this added complexity I will instead show a different method that has only a few minor restrictions, but is much easier to design and understand. This method requires you to position each TYPE component into the key order you will sort on. You will also need to store all numbers that will be used for a sort key as ASCII digits. To sort first on last name, then first name, and then on balance due, the TYPE might be structured as follows:

```
TYPE Customer
    LastName AS STRING * 15
    FirstName AS STRING * 15
```

```

BalanceDue AS STRING * 9
Street     AS STRING * 32
City       AS STRING * 15
State      AS STRING * 2
ZipCode    AS STRING * 5
AnyNumber  AS DOUBLE
END TYPE

```

In most cases the order in which each TYPE member is placed has no consequence. When you refer to TypeVar.LastName, BASIC doesn't care if LastName is defined before or after FirstName in the TYPE structure. Either way it translates your reference to LastName into an address. Having to store numeric data as strings is a limitation, but this is needed only for those TYPE fields that will be used as a sort key.

The key to sorting on multiple items simultaneously is by treating the contiguous fields as a single long field. Since assignments to the string portion of a TYPE variable are handled internally by BASIC's LSET routine, the data in each element will be aligned such that subsequent fields can be treated as an extension of the primary field. Figure 8-2 below shows five TYPE array elements in succession, as they would be viewed by a string comparison routine. This data is defined as a subset of the name and address TYPE shown above, using just the first three fields. Notice that the balance due fields must be right-aligned (using RSET) for the numeric values to be considered correctly.

Type.LastName	Type.FirstName	Type.BalanceDue
Munro	Jay	8000.00
Smith	John	122.03
Johnson	Alfred	14637.89
Rasmussen	Peter	100.90
Hudson	Cindy	21.22
↑ Field 1 starts here	↑ Field 2 starts here	↑ Field 3 starts here

Figure 8-2: Multiple contiguous fields in a TYPE can be treated as a single long field.

Thus, the sort routine would be told to start at the first field, and consider the strings to be 15 + 15 + 9 = 39 characters long. This way all three fields are compared at one time, and treated as a single entity. Additional fields can of course follow these, and they may be included in the comparison or not at your option.

The combination demonstration and subroutine below sorts such a TYPE array on any number of keys using this method, and it has a few additional features as well. Besides letting you confine the sorting to just a portion of the array, you may also specify how far into each element the first key is located. As long as the key fields are contiguous, they do not have to begin at the start of each TYPE. Therefore, you could sort just on the first name field, or on any other field or group of fields.

```
'TYPESORT.BAS - performs a multi-key sort on TYPE arrays
```

```

'Copyright (c) 1991 Ethan Winer
DEFINT A-Z
DECLARE FUNCTION Compare3% (BYVAL Seg1, BYVAL Adr1, BYVAL Seg2, _
    BYVAL Adr2, NumBytes)
DECLARE SUB SwapMem (BYVAL Seg1, BYVAL Adr1, BYVAL Seg2, BYVAL Adr2, _
    BYVAL Length)
DECLARE SUB TypeSort (Segment, Address, ElSize, Offset, KeySize, NumEls)
CONST NumEls% = 23          'this keeps it all on the screen
TYPE MyType
    LastName AS STRING * 10
    FirstName AS STRING * 10
    Dollars AS STRING * 6
    Cents AS STRING * 2
END TYPE
REDIM Array(1 TO NumEls%) AS MyType

'---- Disable (REM out) all but one of the following blocks to test
Offset = 27                'start sorting with Cents
ElSize = LEN(Array(1))    'the length of each element
KeySize = 2                'sort on the Cents only

Offset = 21                'start sorting with Dollars
ElSize = LEN(Array(1))    'the length of each element
KeySize = 8                'sort Dollars and Cents only

Offset = 11                'start sorting with FirstName
ElSize = LEN(Array(1))    'the length of each element
KeySize = 18               'sort FirstName through Cents

Offset = 1                 'start sorting with LastName
ElSize = LEN(Array(1))    'the length of each element
KeySize = ElSize          'sort based on all 4 fields

FOR X = 1 TO NumEls%      'build the array from DATA
    READ Array(X).LastName
    READ Array(X).FirstName
    READ Amount$          'format the amount into money
    Dot = INSTR(Amount$, ".")
    IF Dot THEN
        RSET Array(X).Dollars = LEFT$(Amount$, Dot - 1)
        Array(X).Cents = LEFT$(MID$(Amount$, Dot + 1) + "00", 2)
    ELSE
        RSET Array(X).Dollars = Amount$
        Array(X).Cents = "00"
    END IF
NEXT

Segment = VARSEG(Array(1)) 'show where the array is
Address = VARPTR(Array(1)) ' located in memory
CALL TypeSort(Segment, Address, ElSize, Offset, KeySize, NumEls%)
CLS                          'display the results
FOR X = 1 TO NumEls%
    PRINT Array(X).LastName, Array(X).FirstName,
    PRINT Array(X).Dollars; "."; Array(X).Cents
NEXT

DATA Smith, John, 123.45
DATA Cramer, Phil, 11.51
DATA Hogan, Edward, 296.08
DATA Cramer, Phil, 112.01

```

```

DATA Malin, Donald, 13.45
DATA Cramer, Phil, 111.3
DATA Smith, Ralph, 123.22
DATA Smith, John, 112.01
DATA Hogan, Edward, 8999.04
DATA Hogan, Edward, 8999.05
DATA Smith, Bob, 123.45
DATA Cramer, Phil, 11.50
DATA Hogan, Edward, 296.88
DATA Malin, Donald, 13.01
DATA Cramer, Phil, 111.1
DATA Smith, Ralph, 123.07
DATA Smith, John, 112.01
DATA Hogan, Edward, 8999.33
DATA Hogan, Edward, 8999.17
DATA Hogan, Edward, 8999.24
DATA Smith, John, 123.05
DATA Cramer, David, 1908.80
DATA Cramer, Phil, 112
END

```

```

SUB TypeSort (Segment, Address, ElSize, Displace, KeySize, NumEls) STATIC
REDIM QStack(NumEls \ 5 + 10) 'create a stack array

```

```

First = 1 'initialize working variables
Last = NumEls
Offset = Displace - 1 'decrement once now rather than
' repeatedly later

```

```
DO
```

```
DO
```

```

Temp = (Last + First) \ 2 'seek midpoint
I = First
J = Last

```

```
DO
```

```

WHILE Compare3%(Segment, Address + Offset + (I - 1) * ElSize, Segment, _
Address + Offset + (Temp-1) * ElSize, KeySize) = -1 '< 1 for descending
I = I + 1
WEND

```

```
WEND
```

```

WHILE Compare3%(Segment, Address + Offset + (J - 1) * ElSize, Segment, _
Address + Offset + (Temp-1) * ElSize, KeySize) = 1 '< -1 for descending
J = J - 1
WEND

```

```
WEND
```

```
IF I > J THEN EXIT DO
```

```
IF I < J THEN
```

```

CALL SwapMem(Segment, Address + (I - 1) * ElSize, Segment, _
Address + (J - 1) * ElSize, ElSize)

```

```
IF Temp = I THEN
```

```
Temp = J
```

```
ELSEIF Temp = J THEN
```

```
Temp = I
```

```
END IF
```

```
END IF
```

```
I = I + 1
```

```
J = J - 1
```

```
LOOP WHILE I <= J
```

```
IF I < Last THEN
```

```
QStack(StackPtr) = I 'Push I
```

```
QStack(StackPtr + 1) = Last 'Push Last
```

```
StackPtr = StackPtr + 2
```



```

    END IF

    Last = J
    LOOP WHILE First < Last

    IF StackPtr = 0 THEN EXIT DO           'Done
    StackPtr = StackPtr - 2
    First = QStack(StackPtr)             'Pop First
    Last = QStack(StackPtr + 1)         'Pop Last
    LOOP

    ERASE QStack                          'delete the stack array
    END SUB

```

As you can see, this version of the Quick Sort subprogram is derived from the one shown earlier. The important difference is that all of the incoming information is passed as segments, addresses, and bytes, rather than using an explicit array name. But before describing the inner details of the subprogram itself, I'll address the demonstration portion and show how the routine is set up and called.

As with some of the other procedures on the disk that comes with this book, you will extract the TypeSort subprogram and add it to your own programs by loading it as a module, and then using the Move option of BASIC's View Subs menu. You can quickly access this menu by pressing F2, and then use Alt-M to select Move. Once this is done you will unload TYPESORT.BAS using the Alt-F-U menu selection, and answer "No" when asked if you want to save the modified file. You could also copy the TypeSort subprogram into a separate file, and then load that file as a module in each program that needs it.

Although the example TYPE definition here shows only four components, you may of course use any TYPE structure. TypeSort expects six parameters to tell it where in memory the array is located, how far into each element the comparison routines are to begin, the total length of each element, the length of the key fields, and the number of elements to sort.

After defining MyType, the setup portion of TYPESORT.BAS establishes the offset, element size, and key size parameters. As you can see, four different sample setups are provided, and you should add remarking apostrophes to all but one of them. If the program is left as is, the last setup values will take precedence.

The next section reads sample names, addresses and dollar amounts from DATA statements, and formats the dollar amounts as described earlier. The dollar portion of the amounts are right justified into the Dollars field of each element, and the Cents portion is padded with trailing zeros as necessary to provide a dollars and cents format. This way, the value 12.3 will be assigned as 12.30, and 123 will be formatted to 123.00 which gives the expected appearance.

The final setup step is to determine where the array begins in memory. Since you specify the starting segment and address, it is simple to begin sorting at any array element. For example, to sort elements 100 through 200—even if the array is larger than that—you'd use VARSEG(Array(100)) and VARPTR(Array(100)) instead of element 1 as shown in this example.

In addition to the starting segment and address of the array, TypeSort also requires you to tell it how many elements to consider. If you are sorting the entire array and the array starts with element 1, this will simply be the UBOUND of the array. If you are sorting just a portion of the array then you give it only the number of elements to be sorted. So to sort elements 100 through 200, the number of elements will be 101. A general formula you can use for calculating this based on element numbers is $\text{NumElements} = \text{LastElement} - \text{FirstElement} + 1$.

Now let's consider the TypeSort subprogram itself. Since it is more like the earlier QSort program than different, I will cover only the differences here. In fact, the primary difference is in the way comparisons and exchanges are handled. The Compare3 function introduced earlier is used to compare the array elements with the midpoint. Although QSort made a temporary copy of the midpoint element, that would be difficult to do here. Since the routine is designed to work with any type of data—and the size of each element can vary depending on the TYPE structure—it is impractical to make a copy.

While SPACE\$ could be used to claim a block of memory into which the midpoint element is copied, there's a much better way: the Temp variable is used to remember the element number itself. The only complication is that once elements I and J are swapped, Temp must be reassigned if it was equal to either of them. (This happens just below the call to SwapMem.) But the simple integer IF test and assignment required adds far less code and is much faster than making a copy of the element.

TypeSort is designed to sort the array in ascending order, and comments in the code show how to change it to sort descending instead. If you prefer to have one subprogram that can do both, you should add an extra parameter, perhaps called Direction. Near the beginning of the routine before the initial outer DO you would add this:

```
IF Direction = 0 THEN      'sort ascending
  ICompare = -1
  JCompare = 1
ELSE                        'sort descending
  ICompare = 1
  JCompare = -1
END IF
```

Then, where the results from Compare3 are compared to -1 and 1 replace those comparisons (at the end of each WHILE line) to instead use ICompare and JCompare:

```
WHILE Compare3%(...) = ICompare
  I = I + 1
WEND
WHILE Compare3%(...) = JCompare
  J = J - 1
WEND
```

This way, you are using variables to establish the sorting direction, and those variables can be set either way each time TypeSort is called.

The last major difference is that elements are exchanged using the SwapMem routine rather than BASIC's SWAP statement. While it is possible to call SWAP by aliasing its name as shown in Chapter

5, it was frankly simpler to write a new routine for this purpose. Further, BASIC's SWAP is slower than SwapMem because it must be able to handle variables of different lengths, and also exchange fixed-length and conventional strings. SwapMem is extremely simple, and it works very quickly.

As I stated earlier, the only way to write a truly generic sort routine is by passing segments and addresses and bytes, instead of array names. Although it would be great if BASIC could let you declare a subprogram or function using the AS ANY option to allow any type of data, that simply wouldn't work. As BASIC compiles your program, it needs to know the size and type of each parameter. When you reference TypeVar.LastName, BASIC knows where within TypeVar the LastName portion begins, and uses that in its address calculations. It is not possible to avoid this limitation other than by using addresses as is done here.

Indeed, this is the stuff that C and assembly language programs are made of. In these languages—especially assembly language—integer pointer variables are used extensively to show where data is located and how long it is. However, the formulas used within the Compare3 and SwapMem function calls are not at all difficult to understand.

The formula $\text{Address} + \text{Offset} - (I - 1) * \text{ElSize}$ indicates where the key field of element I begins. Address holds the address of the beginning of the first element, and Offset is added to identify the start of the first sort key. $(I - 1)$ is used instead of I because addresses are always zero-based. That is, the first element in the array from TypeSort's perspective is element 0, even though the calling program considers it to be element 1. Finally, the element number is multiplied times the length of each element, to determine the value that must be added to the starting address and offset to obtain the final address for the data in element I. Please understand that calculations such as these are what the compiler must do each time you access an array element.

Note that if you call TypeSort incorrectly or give it illegal element numbers, you will not receive a "Subscript out of range" error from BASIC. Rather, you will surely crash your PC and have to reboot. This is the danger, and fun, of manipulating pointers directly.

As I stated earlier, the SwapMem routine that does the actual exchanging of elements is very simple, and it merely takes a byte from one element and exchanges it with the corresponding byte in the other. This task is greatly simplified by the use of the XCHG assembly language command, which is similar to BASIC's SWAP statement. Although XCHG cannot swap a word in memory with another word in memory, it can exchange memory with a register. SwapMem is shown in the listing below.

```
;SWAPMEM.ASM, swaps two sections of memory

.Model Medium, Basic
.Code

SwapMem Proc Uses SI DI DS ES, Var1:DWord, Var2:DWord, NumBytes:Word
    lds  SI,Var1          ;get the segmented address of the
                        ; first variable
    les  DI,Var2          ;and for the second variable
    mov  CX,NumBytes     ;get the number of bytes to exchange
    jcxz Exit            ;we can't swap zero bytes!
```

```

DoSwap:
    Mov  AL,ES:[DI]    ;get a byte from the second variable
    Xchg AL,[SI]      ;swap it with the first variable
    Stosb              ;complete the swap and increment DI
    Inc  SI            ;point to the next source byte
    Loop DoSwap       ;continue until done

Exit:
    Ret                ;return to BASIC

SwapMem Endp
End

```

Indexed Sorting on Multiple Keys

Earlier I showed how to modify the simple Bubble Sort routine to sort a parallel index array instead of the primary array. One important reason you might want to do that is to allow access to the primary array in both its original and sorted order. Another reason, and one we will get to shortly, is to facilitate sorting disk files. Although a routine to sort the records in a file could swap the actual data, it takes a long time to read and write that much data on disk. Further, each time you wanted to access the data sorted on a different key, the entire file would need to be sorted again.

A much better solution is to create one or more sorted lists of record numbers, and store those on disk each in a separate file. This lets you access the data sorted by name, or by Zip code, or by any other field, without ever changing the actual file. The TypeISort subprogram below is adapted from TypeSort, and it sorts an index array that holds the element numbers of a TYPE array.

```

'TYPISORT.BAS, indexed multi-key sort for TYPE arrays

DEFINT A-Z

DECLARE FUNCTION Compare3% (BYVAL Seg1, BYVAL Adr1, BYVAL Seg2, _
    BYVAL Adr2, NumBytes)
DECLARE SUB SwapMem (BYVAL Seg1, BYVAL Adr1, BYVAL Seg2, _
    BYVAL Adr2, BYVAL Length)
DECLARE SUB TypeISort (Segment, Address, ElSize, Offset, KeySize, _
    NumEls, Index())

CONST NumEls% = 23                'this keeps it all on the screen
TYPE MyType
    LastName AS STRING * 10
    FirstName AS STRING * 10
    Dollars AS STRING * 6
    Cents AS STRING * 2
END TYPE
REDIM Array(1 TO NumEls%) AS MyType
REDIM Index(1 TO NumEls%) 'create the index array

Offset = 1                        'start sorting with LastName
ElSize = LEN(Array(1))           'the length of each element
KeySize = ElSize                  'sort based on all 4 fields

FOR X = 1 TO NumEls%              'build the array from DATA
    READ Array(X).LastName

```

```

    READ Array(X).FirstName
    READ Amount$
    ...
    ...
NEXT
                                'this continues as already
                                ' shown in TypeSort

FOR X = 1 TO NumEls%
    Index(X) = X - 1
NEXT
                                'initialize the index
                                'but starting with 0

Segment = VARSEG(Array(1))
Address = VARPTR(Array(1))
CALL TypeISort(Segment, Address, ElSize, Offset, KeySize, NumEls%, Index())
CLS
FOR X = 1 TO NumEls%
    PRINT Array(Index(X) + 1).LastName,
    PRINT Array(Index(X) + 1).FirstName,
    PRINT Array(Index(X) + 1).Dollars; ". ";
    PRINT Array(Index(X) + 1).Cents
NEXT
                                'show where the array is
                                ' located in memory
                                'display the results
                                '+ 1 adjusts to one-based

DATA Smith, John, 123.45
...
...
                                'this continues as already
                                ' shown in TypeSort

END

SUB TypeISort (Segment, Address, ElSize, Displace, KeySize, NumEls, _
    Index()) STATIC
REDIM QStack(NumEls \ 5 + 10) 'create a stack

First = 1
Offset = Displace - 1
DO
    DO
        Temp = (Last + First) \ 2 'seek midpoint
        I = First
        J = Last

        DO 'change -1 to 1 and 1 to -1 to sort descending
            WHILE Compare3%(Segment, Address + Offset + (Index(I) * ElSize), _
                Segment, Address + Offset + (Index(Temp) * ElSize), KeySize) = -1
                I = I + 1
            WEND
            WHILE Compare3%(Segment, Address + Offset + (Index(J) * ElSize), _
                Segment, Address + Offset + (Index(Temp) * ElSize), KeySize) = 1
                J = J - 1
            WEND
            IF I > J THEN EXIT DO
            IF I < J THEN
                SWAP Index(I), Index(J)
                IF Temp = I THEN
                    Temp = J
                ELSEIF Temp = J THEN
                    Temp = I
                END IF
            END IF
            I = I + 1
            J = J - 1
        LOOP WHILE I <= J

        'initialize working variables Last = NumEls
        'make zero-based now for speed later
    DO

```

```

    IF I < Last THEN
        QStack(StackPtr) = I           'Push I
        QStack(StackPtr + 1) = Last    'Push Last
        StackPtr = StackPtr + 2
    END IF

    Last = J
    LOOP WHILE First < Last

    IF StackPtr = 0 THEN EXIT DO       'Done
    StackPtr = StackPtr - 2
    First = QStack(StackPtr)           'Pop First
    Last = QStack(StackPtr + 1)        'Pop Last
    LOOP

    ERASE QStack                       'delete the stack array
    END SUB

```

As with TypeSort, TypeISort is entirely pointer based so it can be used with any type of data and it can sort multiple contiguous keys. The only real difference is the addition of the Index() array parameter, and the extra level of indirection needed to access the index array each time a comparison is made. Also, when a swap is required, only the integer index elements are exchanged, which simplifies the code and reduces its size. Like TypeSort, you can change the sort direction by reversing the -1 and 1 values used with Compare3, or add a Direction parameter to the list and modify the code to use that.

As with BubbleISort, the index array is initialized to increasing values by the calling program; however, here the first element is set to hold a value of 0 instead of 1. This reduces the calculations needed within the routine each time an address must be obtained. Therefore, when TypeISort returns, the caller must add 1 to the element number held in each index element. This is shown within the FOR/NEXT loop that displays the sorted results.

Sorting Files

With the development of TypeISort complete, we can now use that routine to sort disk files. The sorting strategy will be to determine how many records are in the file, to determine how many separate passes are needed to process the entire file. TypeISort and TypeSort are restricted to working with arrays no larger than 64K (32K in the editing environment), so there is a limit as to how much data may be loaded into memory at one time. These sort routines can accommodate more data when compiled because address calculations that result in values larger than 32767 cause an overflow error in the QB editor. This overflow is in fact harmless, and is ignored in a compiled program unless you use the /d switch.

Although the routines could be modified to perform segment and address arithmetic to accommodate larger arrays, that still wouldn't solve the problem of having more records than can fit in memory at once. Therefore, separate passes must be used to sort the file contents in sections, with each pass writing a temporary index file to disk. A final merge pass then reads each index to determine which pieces fits where, and then writes the final index file. The program FILESORT.BAS below incorporates

all of the sorting techniques shown so far, and includes a few custom BASIC routines to improve its performance.

```
'FILESORT.BAS, indexed multi-key random access file sort

DEFINT A-Z

DECLARE FUNCTION Compare3% (BYVAL Seg1, BYVAL Adr1, BYVAL Seg2, _
    BYVAL Adr2, NumBytes)
DECLARE FUNCTION Exist% (FileSpec$)
DECLARE SUB DOSInt (Registers AS ANY)
DECLARE SUB FileSort (FileName$, NDXName$, RecLength, Offset, KeySize)
DECLARE SUB LoadFile (FileNum, Segment, Address, Bytes&)
DECLARE SUB SaveFile (FileNum, Segment, Address, Bytes&)
DECLARE SUB SwapMem (BYVAL Seg1, BYVAL Adr1, BYVAL Seg2, BYVAL Adr2, _
    BYVAL Length)
DECLARE SUB TypeISort (Segment, Address, ElSize, Offset, KeySize, _
    NumEls, Index())

RANDOMIZE TIMER                                'create new data each run
DEF FnRand% = INT(RND * 10 + 1) 'returns RND from 1 to 10

TYPE RegType                                'used by DOSInt
    AX AS INTEGER
    BX AS INTEGER
    CX AS INTEGER
    DX AS INTEGER
    BP AS INTEGER
    SI AS INTEGER
    DI AS INTEGER
    FL AS INTEGER
    DS AS INTEGER
    ES AS INTEGER
END TYPE

DIM SHARED Registers AS RegType 'share among all subs
REDIM LastNames$(1 TO 10)       'we'll select names at
REDIM FirstNames$(1 TO 10)     ' random from these

NumRecords = 2988                'how many test records to use
FileName$ = "TEST.DAT"         'really original, eh?
NDXName$ = "TEST.NDX"         'this is the index file name
TYPE RecType
    LastName AS STRING * 11
    FirstName AS STRING * 10
    Dollars AS STRING * 6
    Cents AS STRING * 2
    AnyNumber AS LONG          'this shows that only key
    OtherNum AS LONG           ' information must be ASCII
END TYPE

FOR X = 1 TO 10                  'read the possible last names
    READ LastNames$(X)
NEXT

FOR X = 1 TO 10                  'and the possible first names
    READ FirstNames$(X)
NEXT

DIM RecordVar AS RecType        'to create the sample file
```

```

RecLength = LEN(RecordVar) 'the length of each record
CLS
PRINT "Creating a test file..."

IF Exist%(FileName$) THEN 'if there's an existing file
  KILL FileName$          'kill the old data from prior
END IF                    ' runs to start fresh

IF Exist%(NDXName$) THEN  'same for any old index file
  KILL NDXName$
END IF

'----- Create some test data and write it to the file
OPEN FileName$ FOR RANDOM AS #1 LEN = RecLength
  FOR X = 1 TO NumRecords
    RecordVar.LastName = LastNames$(FnRand%)
    RecordVar.FirstName = FirstNames$(FnRand%)
    Amount$ = STR$(RND * 10000)
    Dot = INSTR(Amount$, ".")
    IF Dot THEN
      RSET RecordVar.Dollars = LEFT$(Amount$, Dot - 1)
      RecordVar.Cents = LEFT$(MID$(Amount$, Dot + 1) + "00", 2)
    ELSE
      RSET RecordVar.Dollars = Amount$
      RecordVar.Cents = "00"
    END IF
    RecordVar.AnyNumber = X
    PUT #1, , RecordVar
  NEXT
CLOSE

'----- Created a sorted index based on the main data file
Offset = 1 'start sorting with LastName
KeySize = 29 'sort based on first 4 fields
PRINT "Sorting..."
CALL FileSort(FileName$, NDXName$, RecLength, Offset, KeySize)

'----- Display the results
CLS
VIEW PRINT 1 TO 24
LOCATE 25, 1
COLOR 15
PRINT "Press any key to pause/resume";
COLOR 7
LOCATE 1, 1

OPEN FileName$ FOR RANDOM AS #1 LEN = RecLength
OPEN NDXName$ FOR BINARY AS #2
  FOR X = 1 TO NumRecords
    GET #2, , ThisRecord 'get next rec. number
    GET #1, ThisRecord, RecordVar 'then the actual data

    PRINT RecordVar.LastName; 'print each field
    PRINT RecordVar.FirstName;
    PRINT RecordVar.Dollars; ".";
    PRINT RecordVar.Cents

    IF LEN(INKEY$) THEN 'pause on a keypress
      WHILE LEN(INKEY$) = 0: WEND
    END IF
  NEXT

```


CLOSE

VIEW PRINT 1 TO 24 'restore the screen
END

DATA Smith, Cramer, Malin, Munro, Passarelli
DATA Bly, Osborn, Pagliaro, Garcia, Winer
DATA John, Phil, Paul, Anne, Jacki
DATA Patricia, Ethan, Donald, Tami, Elli
END

```
FUNCTION Exist% (Spec$) STATIC 'reports if a file exists
DIM DTA AS STRING * 44 'the work area for DOS
DIM LocalSpec AS STRING * 60 'guarantee the spec is in
LocalSpec$ = Spec$ + CHR$(0) ' DGROUP for BASIC PDS

Exist% = -1 'assume true for now

Registers.AX = &H1A00 'assign DTA service
Registers.DX = VARPTR(DTA) 'show DOS where to place it
Registers.DS = VARSEG(DTA)
CALL DOSInt(Registers)

Registers.AX = &H4E00 'find first matching file
Registers.CX = 39 'any file attribute okay
Registers.DX = VARPTR(LocalSpec)
Registers.DS = VARSEG(LocalSpec)
CALL DOSInt(Registers) 'see if there's a match

IF Registers.FL AND 1 THEN 'if the Carry flag is set
    Exist% = 0 ' there were no matches
END IF
END FUNCTION
```

```
SUB FileSort (FileName$, NDXName$, RecLength, Displace, KeySize) STATIC
CONST BufSize% = 32767 'holds the data being sorted
Offset = Displace - 1 'make zero-based for speed later

'----- Open the main data file
FileNum = FREEFILE
OPEN FileName$ FOR BINARY AS #FileNum

'----- Calculate the important values we'll need
NumRecords = LOF(FileNum) \ RecLength
RecsPerPass = BufSize% \ RecLength
IF RecsPerPass > NumRecords THEN RecsPerPass = NumRecords

NumPasses = (NumRecords \ RecsPerPass) - ((NumRecords MOD RecsPerPass) <> 0)
IF NumPasses = 1 THEN
    RecsLastPass = RecsPerPass
ELSE
    RecsLastPass = NumRecords MOD RecsPerPass
END IF

'----- Create the buffer and index sorting arrays
REDIM Buffer(1 TO 1) AS STRING * BufSize
REDIM Index(1 TO RecsPerPass)
IndexAdjust = 1
```

```

'----- Process all of the records in manageable groups
FOR X = 1 TO NumPasses

  IF X < NumPasses THEN          'if not the last pass
    RecsThisPass = RecsPerPass   'do the full complement
  ELSE                            'the last pass may have
    RecsThisPass = RecsLastPass ' fewer records to do
  END IF

  FOR Y = 1 TO RecsThisPass      'initialize the index
    Index(Y) = Y - 1            'starting with value of 0
  NEXT

  '----- Load a portion of the main data file
  Segment = VARSEG(Buffer(1))    'show where the buffer is
  CALL LoadFile(FileNum, Segment, Zero, RecsThisPass * CLNG(RecLength))
  CALL TypeISort(Segment, Zero, RecLength, Displace, KeySize, _
    RecsThisPass, Index())

  '----- Adjust the zero-based index to record numbers
  FOR Y = 1 TO RecsThisPass
    Index(Y) = Index(Y) + IndexAdjust
  NEXT

  '----- Save the index file for this pass
  TempNum = FREEFILE
  OPEN "$$PASS." + LTRIM$(STR$(X)) FOR OUTPUT AS #TempNum
  CALL SaveFile(TempNum, VARSEG(Index(1)), Zero, RecsThisPass * 2&)
  CLOSE #TempNum

  '----- The next group of record numbers start this much higher
  IndexAdjust = IndexAdjust + RecsThisPass

NEXT

ERASE Buffer, Index              'free up the memory

'----- Do a final merge pass if necessary
IF NumPasses > 1 THEN

  NDXNumber = FREEFILE
  OPEN NDXName$ FOR BINARY AS #NDXNumber
  REDIM FileNums(NumPasses)      'this holds the file numbers
  REDIM RecordNums(NumPasses)   'this holds record numbers
  REDIM MainRec$(1 TO NumPasses) 'holds main record data
  REDIM Remaining(1 TO NumPasses) 'tracks index files
  '----- Open the files and seed the first round of data
  FOR X = 1 TO NumPasses
    FileNums(X) = FREEFILE
    OPEN "$$PASS." + LTRIM$(STR$(X)) FOR BINARY AS #FileNums(X)
    Remaining(X) = LOF(FileNums(X)) 'this is what remains
    MainRec$(X) = SPACE$(RecLength) 'holds main data file
    GET #FileNums(X), , RecordNums(X) 'get the next record number
    RecOffset& = (RecordNums(X) - 1) * CLNG(RecLength) + 1
    GET #FileNum, RecOffset&, MainRec$(X) 'then get the data
  NEXT

  FOR X = 1 TO NumRecords
    Lowest = 1                    'assume this is the lowest data in the group

```

```

WHILE Remaining(Lowest) = 0 'Lowest can't refer to a dead index
  Lowest = Lowest + 1      'so seek to the next higher active index
WEND

FOR Y = 2 TO NumPasses      'now seek out the truly lowest element
  IF Remaining(Y) THEN     'consider only active indexes
    IF Compare3%(SSEG(MainRec$(Y)), _ '<-- use VARSEG with QB
      SADD(MainRec$(Y)) + Offset, _
      SSEG(MainRec$(Lowest)), _      '<-- use VARSEG with QB
      SADD(MainRec$(Lowest)) + Offset, KeySize) = -1 THEN
      Lowest = Y
    END IF
  END IF
NEXT

PUT #NDXNumber, , RecordNums(Lowest) 'write the main index
Remaining(Lowest) = Remaining(Lowest) - 2
IF Remaining(Lowest) THEN 'if the index is still active
  GET #FileNums(Lowest), , RecordNums(Lowest)
  RecOffset& = (RecordNums(Lowest) - 1) * CLNG(RecLength) + 1
  GET #FileNum, RecOffset&, MainRec$(Lowest)
END IF

NEXT

ELSE
  '----- Only one pass was needed so simply rename the index file
  NAME "$$PASS.1" AS NDXName$
END IF

CLOSE 'close all open files

IF Exist%("$$PASS.*") THEN 'ensure there's a file to kill
  KILL "$$PASS.*" 'kill the work files
END IF

ERASE FileNums, RecordNums 'erase the work arrays
ERASE MainRec$, Remaining
END SUB

SUB LoadFile (FileNum, Segment, Address, Bytes&) STATIC
  IF Bytes& > 32767 THEN Bytes& = Bytes& - 65536
  Registers.AX = &H3F00 'read from file service
  Registers.BX = FILEATTR(FileNum, 2) 'get the DOS handle
  Registers.CX = Bytes& 'how many bytes to load
  Registers.DX = Address 'and at what address
  Registers.DS = Segment 'and at what segment
  CALL DOSInt(Registers)
END SUB

SUB SaveFile (FileNum, Segment, Address, Bytes&) STATIC
  IF Bytes& > 32767 THEN Bytes& = Bytes& - 65536
  Registers.AX = &H4000 'write to file service
  Registers.BX = FILEATTR(FileNum, 2) 'get the DOS handle
  Registers.CX = Bytes& 'how many bytes to load
  Registers.DX = Address 'and at what address
  Registers.DS = Segment 'and at what segment
  CALL DOSInt(Registers)
END SUB

```

```
SUB TypeISort (....) STATIC      'as shown in TYPISORT.BAS
END SUB
```

FILESORT.BAS begins by defining a function that returns a random number between 1 and 10. Although the earlier sort demonstrations simply read the test data from DATA statements, that is impractical when creating thousands of records. Instead, two arrays are filled—one with ten last names and another with ten first names—and these names are drawn from at random.

The Registers TYPE variable that is defined is used by three of the supporting routines in this program. RegType is normally associated with CALL Interrupt and InterruptX, but I have written a small-code replacement to mimic InterruptX that works with DOS Interrupt &H21 only. DOSInt accepts just a single Registers argument, instead of the three parameters that BASIC's Interrupt and InterruptX require. Besides adding less code each time it is used, the routine itself is smaller and simpler than InterruptX.

The remainder of the demonstration program should be easy to follow, so I won't belabor its operation; the real action is in the FileSort subprogram.

Like TypeSort and TypeISort, FileSort is entirely pointer based, to accommodate TYPE elements of any size and structure. You provide the name of the main data file to be sorted, the name of an index file to create, and the length and offset of the keys within the disk records. The Displace parameter tells how far into the TYPE structure the key information is located. When calling TypeISort this value is should be one-based, but in the final merge pass where Compare3 is used, a zero-based number is required. Therefore, a copy is made ($\text{Offset} = \text{Displace} - 1$) near the beginning of the routine. This way, both are available quickly without having to calculate $- 1$ repeatedly slowing its operation.

The initial steps FileSort performs are to determine how many records are in the data file, and from that how many records can fit into memory at one time. Once these are known, the number of passes necessary can be easily calculated. An extra step is needed to ensure that RecsPerPass is not greater than the number of records in the file. Just because 200 records can fit into memory at once doesn't mean there are that many records. In most cases where multiple passes are needed the last pass will process fewer records than the others. If there are, say, 700 records and each pass can sort 300, the last pass will sort only 100 records.

Once the pass information is determined, a block of memory is created to hold each portion of the file for sorting. This is the purpose of the Buffer array. REDIM is used to create a 32K chunk of memory that doesn't impinge on available string space.

For each pass that is needed, the number of records in the current pass is determined and the index array is initialized to increasing values. Then, a portion of the main data file is read using the LoadFile subprogram. BASIC does not allow you to read records from a random access file directly into a buffer specified by its address. And even if it did, we can load data much faster than pure BASIC by reading a number of records all at once.

Once the current block of records has been loaded, TypeISort is called to sort the index array. The index array is also saved very quickly using SaveFile, which is the compliment to LoadFile. A unique name is given to each temporary index file such that the first one is named \$\$PASS.1, the second \$\$PASS.2, and so forth. By using dollar signs in the name it is unlikely that the routine will overwrite an existing file from another application. Of course, you may change the names to anything else if you prefer.

Notice the extra step that manipulates the IndexAdjust variable. This adjustment is needed because each sort pass returns the index array holding record numbers starting at 0. The first time through, 1 must be added to each element to reflect BASIC's use of record numbers that start at 1. If the first pass sorts, say, 250 records, then the index values 1 through 250 are saved to disk. But the second pass is processing records 251 through 500, so an adjustment value of 251 must be added to each element prior to writing it to disk.

If the data file is small and only one pass was needed, the \$\$PASS.1 file is simply renamed to whatever the caller specified. Otherwise, a merge pass is needed to determine which record number is the next in sequence based on the results of each pass. Believe it or not, this is the trickiest portion of the entire program. For the sake of discussion, we'll assume that four passes were required to sort the file.

Each of the four index files contains a sequence of record numbers, and all of the records within that sequence are in sorted order. However, there is no relationship between the data records identified in one index file and those in another. Thus, each index file and corresponding data record must be read in turn. A FOR/NEXT loop then compares each of the four records, to see which is truly next in the final sequence. The complication arises as the merge nears completion, because some of the indexes will have become exhausted. This possibility is handled by the Remaining array.

Elements in the Remaining array are initialized to the length of each index file as the indexes are opened. Then, as each index entry is read from disk, the corresponding element is decremented by two to show that another record number was read. Therefore, the current Remaining element must be checked to see if that index has been exhausted. Otherwise, data that was already processed might be considered in the merge comparisons.

The final steps are to close all the open files, delete the temporary index files, and erase the work arrays to free the memory they occupied.

One important point to observe is the use of SSEG to show Compare3 where the MainRec\$ elements are located. SSEG is for BASIC 7 only; if you are using QuickBASIC you must change SSEG to VARSEG. SSEG can be used with either near or far strings in BASIC 7, but VARSEG works with near strings only. SSEG is used as the default, so an error will be reported if you are using QuickBASIC. The cursor will then be placed near the comment in the program that shows the appropriate correction.

Searching Fundamentals

As with sorting, searching data effectively also requires that you select an appropriate algorithm. There are many ways to search data, and we will look at several methods here. The easiest to understand is a linear search, which simply examines each item in sequence until a match is found:

```

FoundAt = 0                'assume no match

FOR X = 1 TO NumElements  'search all elements
  IF Array$(X) = Sought$ THEN
    FoundAt = X           'remember where it is
    EXIT FOR              'no need to continue
  END IF
NEXT

IF FoundAt THEN           'if it was found
  PRINT "Found at element"; FoundAt
ELSE
  PRINT "Not found"       'otherwise
END IF

```

For small arrays a linear search is effective and usually fast enough. Also, integer and long integer arrays can be searched reasonably quickly even if there are many elements. But with string data, as the number of elements that must be searched increases, the search time can quickly become unacceptable. This is particularly true when additional features are required such as searching without regard to capitalization or comparing only a portion of each element using MID\$. Indeed, many of the same techniques that enhance a sort routine can also be employed when searching.

To search ignoring capitalization you would first capitalize Sought\$ outside of the loop, and then use UCASE\$ with each element in the comparisons. Using UCASE\$(Sought\$) repeatedly within the loop is both wasteful and unnecessary:

```

Sought$ = UCASE$(Sought$)
.
.
IF UCASE$(Array$(X)) = Sought$ THEN

```

Likewise, comparing only a portion of each string will require MID\$ with each comparison, after using MID\$ initially to extract what is needed from Sought\$:

```

Sought$ = MID$(Sought$, 12, 6)
.
.
IF MID$(Array$(X), 12, 6) = Sought$ THEN

```

And again, as with sorting, these changes may be combined in a variety of ways. You could even use INSTR to see if the string being searched for is within the array, when an exact match is not needed:

```

IF INSTR(UCASE$(Array$(X)), Sought$) THEN

```

However, each additional BASIC function you use will make the searching slower and slower. Although BASIC's INSTR is very fast, adding UCASE\$ to each comparison as shown above slows the overall process considerably.

There are three primary ways that searching can be sped up. One is to apply simple improvements based on understanding how BASIC works, and knowing which commands are fastest. The other is to select a better algorithm. The third is to translate selected portions of the search routine into assembly language. I will use all three of these techniques here, starting with enhancements to the linear search, and culminating with a very fast binary search for use with sorted data.

One of the slowest operations that BASIC performs is comparing strings. For each string, its descriptor address must be loaded and passed to the comparison routine. That routine must then obtain the actual data address, and examine each byte in both strings until one of the characters is different, or it determines that both strings are the same. As I mentioned earlier, if one or both of the strings are fixed-length, then copies also must be made before the comparison can be performed.

There is another service that the string comparison routine must perform, which is probably not obvious to most programmers and which also impacts its speed. BASIC frequently creates and then deletes temporary strings without your knowing it. One example is the copy it makes of fixed-length strings before comparing them. But there are other, more subtle situations in which this can happen.

For example, when you use `IF X$ + Y$ > Z$` BASIC must create a temporary string comprised of `X$ + Y$`, and then pass that to the comparison routine. Therefore, that routine is also responsible for determining if the incoming string is a temporary copy, and deleting it if so. In fact, all of BASIC's internal routines that accept string arguments are required to do this.

Therefore, one good way to speed searching of conventional (not fixed-length) string arrays is to first compare the lengths. Since strings whose lengths are different can't possibly be the same, this will quickly weed those out. BASIC's `LEN` function is much faster than its string compare routine, and it offers a simple but effective opportunity to speed things up. `LEN` is made even faster because it requires only a single argument, as opposed to the two required for the comparison routine.

```
SLen = LEN(Sought$)          'do this once outside the loop
FOR X = 1 TO NumElements
  IF LEN(Array$(X)) = SLen THEN  'maybe...
    IF Array$(X) = Sought$ THEN  'found it!
      FoundAt = X
      EXIT FOR
    END IF
  END IF
END IF
NEXT
```

Similarly, if the first characters are not the same then the strings can't match either. Like `LEN`, BASIC's `ASC` is much faster than the full string comparison routine, and it too can improve search time by eliminating elements that can't possibly match. Depending on the type and distribution of the data in the array, using both `LEN` and `ASCII` can result in a very fast linear search:

```
SLen = LEN(Sought$)
SAsc = ASC(Sought$)
FOR X = 1 TO NumElements
  IF LEN(Array$(X)) = SLen THEN
```

```

    IF ASC(Array$(X)) = SAsc THEN
        IF Array$(X) = Sought$ THEN
            . . .
        END IF
    END IF
END IF
NEXT

```

Notice that the LEN test must always be before the ASC test, to avoid an "Illegal function call" error if the array element is a null string. If all or most of the strings are the same length, then LEN will not be helpful, and ASC should be used alone.

As I mentioned before, when comparing fixed-length string arrays BASIC makes a copy of each element into a conventional string, prior to calling its comparison routine. This copying is also performed when using ASC is used, but not LEN. After all, the length of a fixed-length never changes, and BASIC is smart enough to know the length directly. But then, comparing the lengths of these string is pointless anyway.

Because of the added overhead to make these copies, the performance of a conventional linear search for fixed-length data is generally quite poor. This is a shame, because fixed-length strings are often the only choice when as much data as possible must be kept in memory at once. And fixed-length strings lend themselves perfectly to names and addresses. It should be apparent by now that the best solution for quickly comparing fixed-length string arrays—and the string portion of TYPE arrays too—is with the various Compare functions already shown.

If you are searching for an exact match, then either Compare or Compare2 will be ideal, depending on whether you want to ignore capitalization. If you have only a single string element in each array, you should define a dummy TYPE. This avoids the overhead of having to use both VARSEG and VARPTR as separate arguments. The short example program and SearchType functions that follow search a fixed-length string array for a match.

```

DEFINT A-Z
DECLARE FUNCTION Compare% (SEG Type1 AS ANY, SEG Type2 AS ANY, NumBytes)
DECLARE FUNCTION Compare2% (SEG Type1 AS ANY, SEG Type2 AS ANY, NumBytes)
DECLARE FUNCTION SearchType% (Array() AS ANY, Sought AS ANY)
DECLARE FUNCTION SearchType2% (Array() AS ANY, Sought AS ANY)
DECLARE FUNCTION SearchType3% (Array() AS ANY, Sought AS ANY)
CLS
TYPE FLen                                'this lets us use SEG
    LastName AS STRING * 15
END TYPE

REDIM Array(1 TO 4000) AS FLen           '4000 is a lot of names
DIM Search AS FLen                       'best comparing like data

FOR X = 1 TO 4000 STEP 2                 'impart some realism
    Array(X).LastName = "Henderson"
NEXT

Array(4000).LastName = "Henson"         'almost at the end
Search.LastName = "Henson"              'find the same name

```



```

'----- first time how long it takes using Compare
Start! = TIMER          'start timing

FOR X = 1 TO 5          'search five times
    FoundAt = SearchType%(Array(), Search)
NEXT

IF FoundAt >= 0 THEN
    PRINT "Found at element"; FoundAt
ELSE
    PRINT "Not found"
END IF

Done! = TIMER
PRINT USING "##.## seconds with Compare"; Done! - Start!
PRINT

'----- then time how long it takes using Compare2
Start! = TIMER          'start timing

FOR X = 1 TO 5          'as above
    FoundAt = SearchType2%(Array(), Search)
NEXT

IF FoundAt >= 0 THEN
    PRINT "Found at element"; FoundAt
ELSE
    PRINT "Not found"
END IF

Done! = TIMER
PRINT USING "##.## seconds with Compare2"; Done! - Start!
PRINT

'----- finally, time how long it takes using pure BASIC
Start! = TIMER

FOR X = 1 TO 5
    FoundAt = SearchType3%(Array(), Search)
NEXT

IF FoundAt >= 0 THEN
    PRINT "Found at element"; FoundAt
ELSE
    PRINT "Not found"
END IF

Done! = TIMER
PRINT USING "##.## seconds using BASIC"; Done! - Start!
END

FUNCTION SearchType% (Array() AS FLen, Sought AS FLen) STATIC
SearchType% = -1          'assume not found

FOR X = LBOUND(Array) TO UBOUND(Array)
    IF Compare%(Array(X), Sought, LEN(Sought)) THEN
        SearchType% = X          'save where it was found
        EXIT FOR                'and skip what remains
    END IF

```

```

NEXT
END FUNCTION

FUNCTION SearchType2% (Array() AS FLen, Sought AS FLen) STATIC
SearchType2% = -1          'assume not found

FOR X = LBOUND(Array) TO UBOUND(Array)
  IF Compare2%(Array(X), Sought, LEN(Sought)) THEN
    SearchType2% = X      'save where it was found
    EXIT FOR             'and skip what remains
  END IF
NEXT
END FUNCTION

FUNCTION SearchType3% (Array() AS FLen, Searched AS FLen) STATIC
SearchType3% = -1        'assume not found

FOR X = LBOUND(Array) TO UBOUND(Array)
  IF Array(X).LastName = Searched.LastName THEN
    SearchType3% = X     'save where it was found
    EXIT FOR             'and skip what remains
  END IF
NEXT
END FUNCTION

```

When you run this program it will be apparent that the SearchType function is the fastest, because it uses Compare which doesn't perform any case conversions. SearchType2 is only slightly slower with that added overhead, and the purely BASIC function, SearchType3, lags far behind at half the speed. Note that the array is searched five times in succession, to minimize the slight errors TIMER imposes. Longer timings are generally more accurate than short ones, because of the 1/18th second resolution of the PC's system timer.

Binary Searches

This is about as far as we can go using linear searching, and to achieve higher performance requires a better algorithm. The Binary Search is one of the fastest available; however, it requires the data to already be in sorted order. A Binary Search can also be used with a sorted index, and both methods will be described.

Binary searches are very fast, and also very simple to understand. Unlike the Quick Sort algorithm which achieves great efficiency at the expense of being complicated, a Binary Search can be written using only a few lines of code. The strategy is to start the search at the middle of the array. If the value of that element value is less than that of the data being sought, a new halfway point is checked and the process repeated. This way, the routine can quickly zero in on the value being searched for. Figure 8-3 below shows how this works.

If you are searching for Mexico, the first element examined is number 7, which is halfway through the array. Comparing Mexico to Finland shows that Mexico is greater, so the distance is again cut in half.

In this case, a match was found after only two tries—remarkably faster than a linear search that would have required ten comparisons. Even when huge arrays must be searched, data can often be found in a dozen or so tries. One interesting property of a binary search is that it takes no longer to find the last element in the array than the first one.

```

13: Zambia
12: Sweden
11: Peru
10: Mexico ← step 2
09: Holland
08: Germany
07: Finland ← step 1
06: England
05: Denmark
04: China
03: Canada
02: Austria
01: Australia

```

Figure 8-3: How a Binary Search locates data in a sorted array.

The program below shows one way to implement a Binary Search.

```

DEFINT A-Z
DECLARE FUNCTION BinarySearch% (Array$(), Find$)

CLS
PRINT "Creating test data..."

REDIM Array$(1 TO 1000)           'create a "sorted" array
FOR X = 1 TO 1000
    Array$(X) = "String " + RIGHT$("000" + LTRIM$(STR$(X)), 4) NEXT

PRINT "Searching array..."

FoundAt = BinarySearch%(Array$(), "String 0987")
IF FoundAt >= 0 THEN
    PRINT "Found at element"; FoundAt
ELSE
    PRINT "Not found"
END IF

END

FUNCTION BinarySearch% (Array$(), Find$) STATIC
BinarySearch% = -1                'no matching element yet
Min = LBOUND(Array$)             'start at first element
Max = UBOUND(Array$)             'consider through last

DO
    Try = (Max + Min) \ 2         'start testing in middle

```

```

IF Array$(Try) = Find$ THEN      'found it!
  BinarySearch% = Try           'return matching element
  EXIT DO                       'all done
END IF

IF Array$(Try) > Find$ THEN      'too high, cut in half
  Max = Try - 1
ELSE
  Min = Try + 1                 'too low, cut other way
END IF
LOOP WHILE Max >= Min
END FUNCTION

```

The BinarySearch function returns either the element number where a match was found, or -1 if the search string was not found. Not using a value of zero to indicate failure lets you use arrays that start with element number 0. As you can see, the simplicity of this algorithm belies its incredible efficiency. The only real problem is that the data must already be in sorted order. Also notice that two string comparisons must be made—one to see if the strings are equal, and another to see if the current element is too high. Although you could use Compare3 which examines the strings once and tells if the data is the same or which is greater, a Binary Search is so fast that this probably isn't worth the added trouble. As you will see when you run the test program, it takes far longer to create the data than to search it!

Besides the usual enhancements that can be applied to the comparisons using UCASE\$ or MID\$, this function could also be structured to use a parallel index array. Assuming the data is not sorted but the index array is, the modified Binary Search would look like this:

```

FUNCTION BinaryISearch% (Array$(), Index(), Find$) STATIC
BinaryISearch% = -1             'assume not found
Min = LBOUND(Array$)          'start at first element
Max = UBOUND(Array$)          'consider through last

DO
  Try = (Max + Min) \ 2        'start testing in middle

  IF Array$(Index(Try)) = Find$ THEN      'found it!
    BinaryISearch% = Try                 'return matching element
    EXIT DO                              'all done
  END IF

  IF Array$(Index(Try)) > Find$ THEN      'too high, cut
    Max = Try - 1
  ELSE
    Min = Try + 1                       'too low, cut other way
  END IF
LOOP WHILE Max >= Min
END FUNCTION

```

Numeric Arrays

All of the searching techniques considered so far have addressed string data. In most cases, string array searches are the ones that will benefit the most from improved techniques. As you have already seen, BASIC makes copies of fixed-length strings before comparing them, which slows down searching. And the very nature of strings implies that many bytes may have to be compared before determining if they are equal or which string is greater. In most cases, searching a numeric array is fast enough without requiring any added effort, especially when the data is integer or long integer.

However, a few aspects of numeric searching are worth mentioning here. One is avoiding the inevitable rounding errors that are sure to creep into the numbers you are examining. Another is that in many cases, you may not be looking for an exact match. For example, you may need to find the first element that is higher than a given value, or perhaps determine the smallest value in an array.

Unlike strings that are either the same or they aren't, the binary representation of numeric values is not always so precise. Consider the following test which should result in a match, but doesn't.

```
Value! = 1!  
Result! = 2!  
CLS  
  
FOR X = 1 TO 1000  
    Value! = Value! + .001  
NEXT  
  
IF Value! = Result! THEN  
    PRINT "They are equal"  
ELSE  
    PRINT "Value! ="; Value!  
    PRINT "Result! ="; Result!  
END IF
```

After adding .001 to Value! 1000 times Value! should be equal to 2, but instead it is slightly higher. This is because the binary storage method used by computers simply cannot represent every possible value with absolute accuracy. Even changing all of the single precision exclamation points (!) to double precision pound signs (#) will not solve the problem. Therefore, to find a given value in a numeric array can require some extra trickery.

What is really needed is to determine if the numbers are very close to each other, as opposed to exactly the same. One way to accomplish this is to subtract the two, and see if the result is very close to zero. This is shown below.

```
Value! = 1!  
Result! = 2!  
CLS  
  
FOR X = 1 TO 1000  
    Value! = Value! + .001  
NEXT  
  
IF ABS(Value! - Result!) < .0001 THEN  
    PRINT "They are equal"  
ELSE  
    PRINT "Value! ="; Value!
```

```

    PRINT "Result! =" ; Result!
END IF

```

Here, the absolute value of the difference between the numbers is examined, and if that difference is very small the numbers are assumed to be the same. Unfortunately, the added overhead of subtracting before comparing slows the comparison even further. There is no simple cure for this, and an array search must apply this subtraction to each element that is examined.

Another common use for numeric array searches is when determining the largest or smallest value. Many programmers make the common mistake shown below when trying to find the largest value in an array.

```

MaxValue# = 0

FOR X = 1 TO NumElements
    IF Array#(X) > MaxValue# THEN
        MaxValue# = Array#(X)
        Element = X
    END IF
NEXT

PRINT "The largest value found is"; MaxValue#
PRINT "And it was found at element"; Element

```

The problem with this routine is that it doesn't account for arrays where all of the elements are negative numbers! In that case no element will be greater than the initial MaxValue#, and the routine will incorrectly report zero as the result. The correct method is to obtain the lowest element value, and use that as a starting point:

```

MaxValue# = Array#(1)

FOR X = 2 TO NumElements
    IF Array#(X) > MaxValue# THEN
        MaxValue# = Array#(X)
    END IF
NEXT

PRINT "The largest value found is"; MaxValue#

```

Determining the highest value in an array would be handled similarly, except the greater-than symbol (>) would be replaced with a less-than operator (<).

Soundex

The final searching technique I will show is Soundex. It is often useful to search for data based on its sound, for example when you do not know how to spell a person's name. Soundex was invented in the 1920's and has been used since then by, among others, the U.S. Census Bureau. A Soundex code is an alpha-numeric representation of the sound of a word, and it is surprisingly accurate despite its simplicity. The classic implementation of Soundex returns a four-character result code. The first

character is the same as the first letter of the word, and the other three are numeric digits coded as shown in Table 8-1.

Soundex Code	Letters
1	B, F, P, V
2	C, G, J, K, Q, S, X
3	D, T
4	L
5	M, N
6	R

Table 8-1: The Soundex code numbers returned for significant letters of the alphabet.

Letters not shown are simply skipped as being statistically insignificant to the sound of the word. In particular, speaking accents often minimize the importance of vowels, and blur their distinction. If the string is short and there are fewer than four digits, the result is simply padded with trailing zeros. One additional rule is that a code digit is never repeated, unless there is an uncoded letter in between. In the listing that follows, two different implementations of Soundex are shown.

```
'SOUNDEX.BAS, Soundex routines and example

DEFINT A-Z

DECLARE FUNCTION ASoundex$ (Word$)
DECLARE FUNCTION ISoundex% (Word$)

CLS
DO
PRINT "press Enter alone to exit"
INPUT "What is the first word"; FWord$
IF LEN(FWord$) = 0 THEN EXIT DO
INPUT "What is the second word"; SWord$
PRINT

'Test by alpha-numeric soundex
PRINT "Alpha-Numeric Soundex: "; FWord$; " and ";
PRINT SWord$; " do ";
IF ASoundex$(FWord$) <> ASoundex$(SWord$) THEN
PRINT "NOT ";
END IF
PRINT "sound the same."
PRINT

'Test by numeric soundex
PRINT "      Numeric Soundex: "; FWord$; " and ";
PRINT SWord$; " do ";
IF ISoundex%(FWord$) <> ISoundex%(SWord$) THEN
PRINT "NOT ";
END IF
PRINT "sound the same."
PRINT
LOOP
END

FUNCTION ASoundex$ (InWord$) STATIC
```

```

Word$ = UCASE$(InWord$)
Work$ = LEFT$(Word$, 1) + "000"
WkPos = 2
PrevCode = 0

FOR L = 2 TO LEN(Word$)
  Temp = INSTR("BFPVCGJKQSXZDTLMNR", MID$(Word$, L, 1))
  IF Temp THEN
    Temp = ASC(MID$("111122222222334556", Temp, 1))
    IF Temp <> PrevCode THEN
      MID$(Work$, WkPos) = CHR$(Temp)
      PrevCode = Temp
      WkPos = WkPos + 1
      IF WkPos > 4 THEN EXIT FOR
    END IF
  ELSE
    PrevCode = 0
  END IF
NEXT

ASoundex$ = Work$
END FUNCTION

```

```

FUNCTION ISoundex% (InWord$) STATIC
Word$ = UCASE$(InWord$)
Work$ = "0000"
WkPos = 1
PrevCode = 0

FOR L = 1 TO LEN(Word$)
  Temp = INSTR("BFPVCGJKQSXZDTLMNR", MID$(Word$, L, 1))
  IF Temp THEN
    Temp = ASC(MID$("111122222222334556", Temp, 1))
    IF Temp <> PrevCode THEN
      MID$(Work$, WkPos) = CHR$(Temp)
      PrevCode = Temp
      WkPos = WkPos + 1
      IF WkPos > 4 THEN EXIT FOR
    END IF
  ELSE
    PrevCode = 0
  END IF
NEXT

ISoundex% = VAL(Work$)
END FUNCTION

```

The first function, ASoundex, follows the standard Soundex definition and returns the result as a string. The ISoundex version cheats slightly by coding the first letter as a number, but it returns an integer value instead of a string. Because integer searches are many times faster than string searches, this version will be better when thousands—or even hundreds of thousands—of names must be examined.

An additional benefit of the integer-only method is that it allows for variations on the first letter. For example, if you enter Cane and Kane in response to the prompts from SOUNDEX.BAS ASoundex will not recognize the names as sounding alike where ISoundex will.

Linked Data

No discussion of searching and sorting would be complete without a mention of linked lists and other data links. Unlike arrays where all of the elements lie in adjacent memory locations, linked data is useful when data locations may be disjointed. One example is the linked list used by the DOS File Allocation Table (FAT) on every disk. As I described in Chapter 6, the data in each file may be scattered throughout the disk, and only through a linked list can DOS follow the thread from one sector in a file to another.

Another example where linked data is useful—and the one we will focus on here—is to keep track of memo fields in a database. A memo field is a field that can store free-form text such as notes about a sales contact or a patient's medical history. Since these fields typically require varying lengths, it is inefficient to reserve space for the longest one possible in the main database file. Therefore, most programs store memo fields in a separate disk file, and use a *pointer field* in the main data file to show where the corresponding memo starts in the dedicated memo file. Similarly, a back pointer adjacent to each memo identifies the record that points to it. This is shown in Figure 8-4 below.

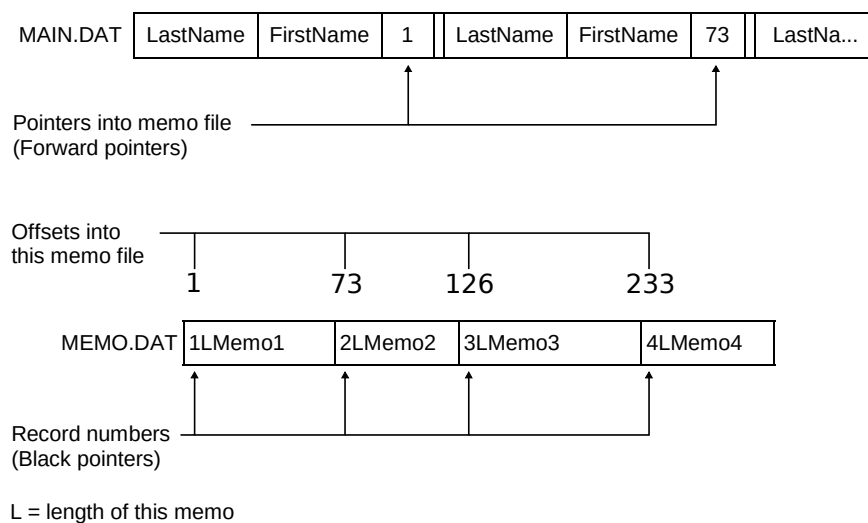


Figure 8-4: Pointers relate record numbers to memo file offsets and vice versa.

Here, the pointer in the main data file record is a long integer that holds the byte offset into the memo file where the corresponding memo text begins. And just before the memo text is an integer record number that shows which record this memo belongs to. If you anticipate more than 65,535 records a long integer must be used instead. Thus, these pointers provide links between the two files, and relate the information they contain.

When a new record is added to the main file, the memo that goes with it is appended to the end of the memo file. BASIC's LOF function can be used to determine the current end of the memo file, which is

then used as the beginning offset for the new memo text. And as the new memo is appended to MEMO.DAT, the first data actually written is the number of the new record in the main data file.

The record number back pointer in the memo file is needed to allow memo data to be edited. Since there's no reasonable way to extend memo text when other memo data follows it, most programs simply abandon the old text, and allocate new space at the end of the file. The abandoned text is then marked as such, perhaps by storing a negative value as the record number. Storing a negative version of the abandoned data's length is ideal, because that both identifies the data as obsolete, and also tells how much farther into the file the next memo is located.

The idea here is that you would periodically run a memo file maintenance program that compacts the file, thus eliminating the wasted space the abandoned memos occupy. This is similar to the DBPACK.BAS utility shown in Chapter 7, and also similar to the way that BASIC compacts string memory when it becomes full. But when an existing memo is relocated in the memo file, the field in the main data file that points to the memo must also be updated. And that's why the record number back pointer is needed: so the compaction program can know which record in the main file must be updated.

The "L" identifier in the memo file in Figure 8-5, shown between the record number and memo text, is a length byte or word that tells how long the text is. If you plan to limit the memo field lengths to 255 or fewer characters, then a single byte is sufficient. Otherwise an integer must be used. An example of code that reads a data record and then its associated memo text is shown below.

```
GET #MainFile, RecNumber, TypeVar
MemoOffset& = TypeVar.MemoOff
GET #MemoFile, MemoOffset& + 2, MemoLength%
Memo$ = SPACE$(MemoLength%)
GET #MemoFile, , Memo$
```

The first step reads a record from the main data file into a TYPE variable, and the second determines where in the memo file the memo text begins. Two is added to that offset in the second GET statement, to skip over the record number back pointer which isn't needed here. Once the length of the memo text is known, a string is assigned to that length, and the actual text is read into it.

If you are using long integer record numbers you would of course use MemoOffset& + 4 in the second GET. And if you're using a single byte to hold the memo length you would define a fixed-length string to receive that byte:

```
DIM Temp AS STRING *1
GET #MemoFile, MemoOffset& + 2, Temp
MemoLength = ASC(Temp)
```

Since BASIC doesn't offer a byte-sized integer data type, ASC and STR\$ can be used to convert between numeric and string formats.

Array Element Insertion and Deletion

The last issue related to array and memory manipulation I want to cover is inserting and deleting elements. If you intend to maintain file indexes or other information in memory and in sorted order, you will need some way to insert a new entry. By the same token, deleting an entry in a database requires that the parallel index entry also be deleted.

The most obvious way to insert or delete elements in an array is with a FOR/NEXT loop. The first example below inserts an element, and the second deletes one.

```
'----- Insert an element:
Element = 200
InsertValue = 999

FOR X = UBOUND(Array) TO Element + 1 STEP -1
    Array(X) = Array(X - 1)
NEXT
Array(Element) = InsertValue

'----- Delete an element:
Element = 200
FOR X = Element TO UBOUND(Array) - 1
    Array(X) = Array(X + 1)
NEXT
Array(UBOUND(Array)) = 0 'optionally clear last element
```

For integer, long integer, and fixed-length arrays this is about as efficient as you can get, short of rewriting the code in assembly language. However, with floating point and string arrays the performance is less than ideal. Unless a numeric coprocessor is installed, floating point values are assigned using interrupts and support code in the emulator library. This adds an unnecessary level of complication that also impacts the speed. When strings are assigned the situation is even worse, because of the memory allocation overhead associated with dynamic string management.

A better solution for floating point and string arrays is a series of SWAP statements. The short program below benchmarks the speed difference of the two methods, as it inserts an element into a single precision array.

```
REDIM Array(1 TO 500)
CLS
Element% = 200
InsertValue = 999

Start = TIMER
FOR A% = 1 TO 500
    FOR X% = UBOUND(Array) TO Element% + 1 STEP -1
        Array(X%) = Array(X% - 1)
    NEXT
    Array(Element%) = InsertValue
NEXT
Done = TIMER
PRINT USING "##.## seconds when assigning"; Done - Start
```

```

Start = TIMER
FOR A% = 1 TO 500
  FOR X% = UBOUND(Array) TO Element% + 1 STEP -1
    SWAP Array(X%), Array(X% - 1)
  NEXT
  Array(Element%) = InsertValue
NEXT
Done = TIMER
PRINT USING "##.## seconds when swapping"; Done - Start

```

If you run this program in the BASIC environment, the differences may not appear that significant. But when the program is compiled to an executable file, the swapping method is more than four times faster. In fact, you should never compare programming methods using the BASIC editor for exactly this reason. In many cases, the slowness of the interpreting process overshadows significant differences between one approach and another.

String arrays also benefit greatly from using SWAP instead of assignments, though the amount of benefit varies depending on the length of the strings. If you modify the previous program to use a string array, also add this loop to initialize the elements:

```

FOR X% = 1 TO 500
  Array$(X%) = "String number" + STR$(X)
NEXT

```

With BASIC PDS far strings the difference is only slightly less at about three to one, due to the added complexity of far data. Also, SWAP will always be worse than assignments when inserting or deleting elements in a fixed-length string or TYPE array. An assignment merely copies the data from one location to another. SWAP, however, must copy the data in both directions.

Understand that when using SWAP with conventional string arrays, the data itself is not exchanged. Rather, the four-byte string descriptors are copied. But because BASIC PDS program modules store string data in different segments, extra work is necessary to determine which descriptor goes with which segment. When near strings are being used, only six bytes are exchanged, regardless of the length of the strings. Four bytes hold the descriptors, and two more store the back pointers.

Summary

This chapter explained many of the finer points of sorting and searching all types of data in BASIC. It began with sorting concepts using the simple Bubble Sort as a model, and then went on to explain indexed and multi-key sorts. One way to implement a multi-key sort is by aligning the key fields into adjacent TYPE components. While there are some restrictions to this method, it is fairly simple to implement and also very fast.

The Quick Sort algorithm was shown, and the SEEQSORT.BAS program on the accompanying disk helps you to understand this complex routine by displaying graphically the progress of the comparisons and exchanges as they are performed. Along the way you saw how a few simple modifications to any

string sort routine can be used to sort regardless of capitalization, or based on only a portion of a string element.

You also learned that writing a truly general sort routine that can handle any type of data requires dealing exclusively with segment and address pointers. Here, assembly language routines are invaluable for assisting you when performing the necessary comparisons and data exchanges. Although the actual operation of the assembly language routines will be deferred until Chapter 12, such routines may easily be added to a BASIC program using .LIB and .QLB libraries.

I mentioned briefly the usefulness of packing and aligning data when possible, as an aid to fast sorting. In particular, dates can be packed to only three bytes in Year/Month/Day order, and other data such as Zip codes can be stored in long integers. Because numbers can be compared much faster than strings, this helps the sorting routines operate more quickly.

Array searching was also discussed in depth, and both linear and binary search algorithms were shown. As with the sorting routines, searching can also employ UCASE\$ and MID\$ to search regardless of capitalization, or on only a portion of each array element. Two versions of the Soundex algorithm were given, to let you easily locate names and other data based on how they sound.

Besides showing the more traditional searching methods, I presented routines to determine the minimum and maximum values in a numeric array. I also discussed some of the ramifications involved when searching floating point data, to avoid the inevitable rounding errors that might cause a legitimate match to be ignored.

Finally, some simple ways to insert and delete elements in both string and numeric arrays were shown. Although making direct assignments in a loop is the most obvious way to do this, BASIC's often-overlooked SWAP command can provide a significant improvement in speed.

The next chapter will conclude this section about hands-on programming by showing a variety of program optimization techniques.



PART 3
Beyond BASIC

9

Program Optimization

Throughout the preceding chapters I have shown a variety of tips and techniques that can help to improve the efficiency of your programs. For example, Chapter 6 explained that processing files in large pieces reduces the time needed to save and load data. Likewise, Chapter 8 discussed the improvement that SWAP often provides over conventional assignments. Some optimizations, however, do not fit into any of the well-defined categories that have been used to organize this book. In this chapter I will share several general optimization techniques you can employ to reduce the size of your programs and make them run faster.

The material in this chapter is organized into three principle categories: programming shortcuts and speed improvements, miscellaneous tips and techniques, and benchmarking. Each section addresses BASIC programming ideas and methods that are not immediately obvious in most cases.

Programming Shortcuts and Speed Improvements

Chapter 3 discussed the use of AND, OR, and the other logical operations that can be used for both logical (IF and CASE) tests and also bit operations. But there are a few other related points that are worth mentioning here. When you need to know if a variable is zero or not, you can omit an explicit test for zero like this:

```
IF Variable THEN...
```

You might be tempted to think that two variables could be tested for non-zero values at one time in the same way, using code such as this:

```
IF Var1 AND Var2 THEN...
```

However, that will very likely fail. The expression `Var1 AND Var2` combines the bits in these variables, which could result in a value of zero even when both variables are non-zero. As an example, if `Var1` currently holds a value of 1, its bits will be set as follows:

```
0000 0000 0000 0001
```

Now, if `Var2` is assigned the value 2, its bits will be set like this:

```
0000 0000 0000 0010
```

Since no two bits are set in the same position in each variable, the result of `Var1 AND Var2` is zero. An effective solution is `IF Var1 * Var2 THEN` to ensure that neither variable is zero. And to test

if either variable is non-zero you'd use OR. Whatever follows the test `IF Var1 OR Var2 THEN` will be executed as long as one (or both) variables are not zero. These are important short cuts to understand, because the improvement in code size and execution speed can be significant.

Each of the AND, OR, and multiplication tests shown here generates only 11 bytes of code. Contrast that to the 28 bytes that BC creates for the alternative: `IF Var1 <> 0 AND Var2 <> 0 THEN`. Because of the improved method of expression evaluation in BASIC PDS, this last example generates only 14 bytes when using that version of BC. None the less, if you can avoid explicit comparisons to zero you will go a long way toward improving the efficiency of your code.

This short cut is equally appropriate with LOOP comparisons as well as IF tests. In the `BufIn` function shown in Chapter 6, `INSTR` was used to see if a `CHR$(13)` carriage return was present in the buffer. In the statement `CR = INSTR(BufPos, Buffer$, CR$)`, `CR` receives either 0 if that character is present, or a non-zero position in the string where it was found. The LOOP statement that surrounded the buffer searching uses `LOOP WHILE CR`, which continues looping as long as `CR` is not zero.

When an integer variable is compared in a LOOP WHILE condition, seven bytes of code are generated whether it is compared to zero or not. But when a long integer is used to control the LOOP WHILE condition, omitting the explicit test for zero results in 11 bytes of compiled code where including it creates 20 bytes. Note that with floating point values identical code is generated in either case, because an explicit comparison to zero is required and added by the compiler.

Predefining Variables

Another important point is illustrated in the same code fragment that uses `INSTR` to search for a carriage return. There, the `CR$` string variable had been assigned earlier to `CHR$(13)`. Although the `BufIn` code could have used `CR = INSTR(BufPos, Buffer$, CHR$(13))` instead of a previously defined string variable to replace the `CHR$(13)`, that would take longer each time the statement is executed. Since `CHR$` is a function, it must be called each time it is used. If `CR$` is defined once ahead of time, only its address needs to be passed to `INSTR`. This can be done with four bytes of assembly language code.

If `CHR$(13)` will be used only once in a program, then the only savings afforded by predefining it will be execution speed. But when it is needed two or more times, several bytes can be saved at each occurrence by using a replacement string variable. Other common `CHR$` values that are used in BASIC programs are the `CHR$(34)` quote character, and `CHR$(0)` which is often used when accessing DOS services with `CALL Interrupt`.

Likewise, you should avoid calling any functions more than is absolutely necessary. I have seen many programmers use code similar to the following, to see if a drive letter has been given as part of a file name.

```
IF INSTR(Path$, ":") THEN
```



```

    Drive$ = LEFT$(INSTR(Path$, ":") - 1)
END IF

```

A much better approach is to invoke INSTR only once, and save the results for subsequent testing:

```

Found% = INSTR(Path$, ":") 'save the result from INSTR
IF Found% THEN
    Drive$ = LEFT$(Path$, Found%) - 1)
END IF

```

The same situation holds true for UCASE\$, MID\$, and all of the other BASIC functions. Rather than this:

```

IF INSTR(UCASE$(MID$(Work$, 3, 22)), "/A") THEN A = True
IF INSTR(UCASE$(MID$(Work$, 3, 22)), "/B") THEN B = True
IF INSTR(UCASE$(MID$(Work$, 3, 22)), "/C") THEN C = True

```

use this instead:

```

Temp$ = UCASE$(MID$(Work$, 3, 22))
IF INSTR(Temp$, "/A") THEN A = True
IF INSTR(Temp$, "/B") THEN B = True
IF INSTR(Temp$, "/C") THEN C = True

```

Where the first example generates 138 bytes of code, the second uses only 111. The time savings will be even more significant, because BASIC's UCASE\$ and MID\$ functions allocate and deallocate memory by making further calls to BASIC's string memory management routines.

Indeed, it is always best to avoid creating new strings whenever possible, precisely because of the overhead needed to assign and erase string data. Each time a string is assigned, memory must be found to hold it; add to that the additional code needed to release the older, abandoned version of the string.

This has further ramifications with simple string tests as well. As Chapter 3 explained, testing for single characters or the first character in a string is always faster if you isolate the ASCII value of the character first, and then use integer comparisons later. In the example below, the first series of IF tests generates 60 bytes of code. This is much less efficient than the second which generates only 46, even though the steps to obtain and assign the ASCII value of Answer\$ comprise 12 of those bytes.

```

PRINT "Abort, Retry, or Fail? (A/R/F) ";
DO
    Answer$ = UCASE$(INKEY$)
LOOP UNTIL LEN(Answer$)

'----- Method 1:
IF Answer$ = "A" THEN
    REM
ELSEIF Answer$ = "R" THEN
    REM
ELSEIF Answer$ = "F" THEN
    REM
END IF

```

```
'----- Method 2:
A% = ASC(Answer$)
IF A% = 65 THEN
    REM
ELSEIF A% = 82 THEN
    REM
ELSEIF A% = 70 THEN
    REM
END IF
```

Another prime candidate for speed enhancement is when you need to create a string from individual characters. The first example below reads the 80 characters in the top row of display memory, and builds a new string from those characters.

```
Scrn$ = ""
FOR X = 1 TO 80
    Scrn$ = Scrn$ + CHR$(SCREEN(1, X))
NEXT
```

Since we already know that 80 characters are to be read, a much better method is to pre-assign the destination string, and insert the characters using the statement form of MID\$, thus:

```
Scrn$ = SPACE$(80)
FOR X% = 1 TO 80
    MID$(Scrn$, X%, 1) = CHR$(SCREEN(1, X%))
NEXT
```

An informal timing test that executed these code fragments 100 times using QuickBASIC 4.5 showed that the second example is nearly twice as fast as the first. Moreover, since BASIC's SCREEN function is notoriously slow, the actual difference between building a new string and inserting characters into an existing string is no doubt much greater.

Integer and Long Integer Assignments

Another facet of compiled BASIC that is probably not immediately obvious is the way that integer and long integer assignments are handled by the compiler. When many variables are to be assigned the same value—perhaps cleared to zero—it is often more efficient to assign one of them from that value, and then assign the rest from the first. To appreciate why this is so requires an understanding of how BASIC compiles such assignments.

Normally, assigning an integer or long integer variable from a numeric constant requires the same amount of code as assigning from another variable. The BASIC statement `X% = 1234` is compiled to the following 6-byte assembly language statement.

```
C7063600D204  MOV WORD PTR [X%],1234
```

Assigning the long integer variable Y& requires two such 6-byte instructions—one for the low word and another for the high word:

```
C7063600D204  MOV  WORD PTR [Y&],1234 ;assign the low word
C70638000000  MOV  WORD PTR [Y&+2],0 ;then the high word
```

The 80x86 family of microprocessors does not have direct instructions for moving the contents of one memory location to another. Therefore, the statement X% = Y% is compiled as follows, with the AX register used as an intermediary.

```
A13800  MOV  AX,WORD PTR [Y%] ;move Y% into AX
A33600  MOV  WORD PTR [X%],AX ;move AX into X%
```

Assigning one long integer from another as in X& = Y& is handled similarly:

```
A13A00  MOV  AX,WORD PTR [Y&] ;move AX from Y& low
8B163C00  MOV  DX,WORD PTR [Y&+2] ;move DX from Y& high
A33600  MOV  WORD PTR [X&],AX ;move X& low from AX
89163800  MOV  WORD PTR [X&+2],DX ;move X& high from DX
```

You may have noticed that instructions that use the AX registers require only three bytes to access a word of memory, while those that use DX (or indeed, any register other than AX) require four. But don't be so quick to assume that BASIC is not optimizing your code. The advantage to using separate registers is that the full value of Y& is preserved. Had AX been used both times, the low word would be lost when the high word was transferred from Y& to X&.

When assigning one variable to many in a row, BASIC is smart enough to remember which values are in which registers, and it reuses those values for subsequent assignments. The combination BASIC and assembly language code shown below was captured from a CodeView session and edited slightly for clarity. It shows the actual assembly language code bytes generated for a series of assignments.

Plain integer assignments:

```
A% = 1234
C7063600D204  MOV  WORD PTR [A%],&H04D2
B% = 1234
C7063800D204  MOV  WORD PTR [B%],&H04D2
C% = 1234
C7063A00D204  MOV  WORD PTR [C%],&H04D2
D% = 1234
C7063C00D204  MOV  WORD PTR [D%],&H04D2
E% = 1234
C7063E00D204  MOV  WORD PTR [E%],&H04D2
```

Plain long integer assignments:

```
V& = 1234
C7064000D204  MOV  WORD PTR [V&],&H04D2
C70642000000  MOV  WORD PTR [V&+2],0
W& = 1234
C7064400D204  MOV  WORD PTR [W&],&H04D2
```

```

C70642000000    MOV    WORD PTR [W&+2], 0
X& = 1234
C7064800D204    MOV    WORD PTR [X&], &H04D2
C70642000000    MOV    WORD PTR [X&+2], 0
Y& = 1234
C7064C00D204    MOV    WORD PTR [Y&], &H04D2
C70642000000    MOV    WORD PTR [Y&+2], 0
Z& = 1234
C7065000D204    MOV    WORD PTR [Z&], &H04D2
C70642000000    MOV    WORD PTR [Z&+2], 0

```

Assigning multiple integers from another:

```

A% = 1234
C7063600D204    MOV    WORD PTR [A%], &H04D2
B% = A%
A13600          MOV    AX, WORD PTR [A%]
A33800          MOV    WORD PTR [B%], AX
C% = A%
A33A00          MOV    WORD PTR [C%], AX
D% = A%
A33C00          MOV    WORD PTR [D%], AX
E% = A%
A33E00          MOV    WORD PTR [E%], AX

```

Assigning multiple long integers from another:

```

V& = 1234
C7064000D204    MOV    WORD PTR [V&], &H04D2
C70642000000    MOV    WORD PTR [V&+2], 0
W& = V&
A14000          MOV    AX, WORD PTR [V&]
8B164200        MOV    DX, WORD PTR [V&+2]
A34400          MOV    WORD PTR [W&], AX
89164600        MOV    WORD PTR [W&+2], DX
X& = V&
A34800          MOV    WORD PTR [X&], AX
89164A00        MOV    WORD PTR [X&+2], DX
Y& = V&
A34C00          MOV    WORD PTR [Y&], AX
89164E00        MOV    WORD PTR [Y&+2], DX
Z& = V&
A35000          MOV    WORD PTR [Z&], AX
89165200        MOV    WORD PTR [Z&+2], DX

```

The first five statements assign the value 1234 (04D2 Hex) to integer variables, and each requires six bytes of code. The next five instructions assign the same value to long integers, taking two such instructions for a total of 12 bytes for each assignment. Note that a zero is assigned to the higher word of each long integer, because the full Hex value being assigned is actually &H000004D2. Simple multiplication shows that the five integer assignments generates five times six bytes, for a total of 30 bytes. The long integer assignments take twice that at 60 bytes total.

But notice the difference in the next two statement blocks. The first integer assignment requires the usual six bytes, and the second does as well. But thereafter, any number of additional integer variables will be assigned with only three bytes apiece. Likewise, all but the first two long integer assignments

are implemented using only seven bytes each. Remembering what values are in each register is yet one more optimization that BASIC performs as it compiles your program.

Short Circuit Expression Evaluation

Many programming situations require more than one test to determine if a series of instructions are to be executed or a branch taken. The short example below tests that a string is not null, and also that the row and column to print at are legal.

```
IF Work$ <> "" AND Row <= 25 AND Column <= 80 THEN
  LOCATE Row, Column
  PRINT Work$
END IF
```

When this program is compiled with QuickBASIC, all three of the tests are first performed in sequence, and the results are then combined to see if the LOCATE and PRINT should be performed. The problem is that time is wasted comparing the row and column even if the string is null. When speed is the primary concern, you should test first for the condition that is most likely to fail, and then use a separate test for the other conditions:

```
IF Work$ <> "" THEN
  IF Row <= 25 AND Column <= 80 THEN
    LOCATE Row, Column
    PRINT Work$
  END IF
END IF
```

This separation of tests is called *short circuit expression evaluation*, because you are bypassing or short circuiting the remaining tests when the first fails. Although it doesn't really take BASIC very long to determine if a string is null, the principle can be applied to other situations such as those that involve file operations like EOF and LOF. Further, as you learned in Chapter 3, a better way to test for a non-null string is `IF LEN(Work$) THEN`. However, the point is to perform those tests that are most likely to fail first, before others that are less likely or will take longer.

Another place where you will find it useful to separate multiple tests is when accessing arrays. If you are testing both for a legal element number and a particular element value, QuickBASIC will give a "Subscript out of range" error if the element number is not valid. This is shown below.

```
IF Element <= MaxEls AND Array(Element) <> 0 THEN
```

Since QuickBASIC always performs both tests, the second will cause an error if Element is not a legal value. In this case, you have to implement the tests using two separate statements:

```
IF Element <= MaxEls THEN
  IF Array(Element) <> 0 THEN
    .
    .
  END IF
```

```
END IF
```

You may have noticed the I have referred to QuickBASIC here exclusively in this discussion. Beginning with BASIC 7.0, Microsoft has added short circuit testing to the compiler as part of its built-in decision making process. Therefore, when you have a statement such as this one:

```
IF X > 1 AND Y = 2 AND Z < 3 THEN
```

BASIC PDS substitutes the following logic automatically:

```
IF X <= 1 THEN GOTO SkipIt
IF Y <> 2 THEN GOTO SkipIt
IF Z >= 3 THEN GOTO SkipIt
.
SkipIt:
```

Speaking of THEN and GOTO, it is worth mentioning that the keyword THEN is not truly necessary when the only thing that follows is a GOTO. That is, `IF X < 1 GOTO Label` is perfectly legal, although the only savings is in the program's source code.

This next and final trick isn't technically a short circuit expression test, but it can reduce the size of your programs in a similar fashion. Chapter 3 compared the relative advantages of GOSUB routines and called subprograms, and showed that a subprogram is superior when passing parameters, while a GOSUB is much faster and smaller. An ideal compromise in some situations is to combine the two methods.

If you have a called subprogram (or function) that requires a large number of parameters and it is called many times, you can use a single call within a GOSUB routine. Since a GOSUB statement generates only three bytes of code each time it is used, this can be an ideal way to minimize the number of times that the full CALL is required. Of course, GOSUB does not accept parameters, but many of them may be the same from call to call. In particular, some third-party add-on libraries require a long series of arguments that are unlikely to change. This is shown below.

```
Row = 10
Column = 20
Message$ = "Slap me five"
GOSUB DisplayMsg
.
.
DisplayMsg:
CALL ManyParams(Row, Column, Message$, MonType, NoSnow, FGColr, BGColr, _
    HighlightFlag, VideoMode, VideoPage)
RETURN
```

In many cases you would have assigned permanent values for the majority of these parameters, and it is wasteful to have BASIC create code to pass them repeatedly. Here, the small added overhead of the three assignments prior to each GOSUB results in less code than passing all ten arguments repeatedly.

Miscellaneous Tips and Techniques

There are many tricks that programmers learn over the years, and the following are some of the more useful ones I have developed myself, or come across in magazines and other sources.

Formatting and Rounding

One frequent requirement in many programs is having control over how numbers are formatted. Of course, BASIC has the PRINT USING statement which is adequate in most cases. And Chapter 6 also showed how to trick BASIC's file handling statements into letting you access a string formatted by PRINT USING. But there are other formatting issues that are not handled by BASIC directly.

One problem for many programmers is that BASIC adds leading and trailing blanks when printing numbers on the screen or to a disk file. The leading blank is a placeholder for a possible minus sign, and is not added when the number is in fact negative. Avoiding the trailing blank is easy; simply use `PRINT STR$(Number)`. And the easiest way to omit the leading blank for positive numbers is to use `LTRIM$: PRINT LTRIM$(STR$(Number))`.

PRINT USING is notoriously slow, because examines each character in a string version of the number, and reformat the digits while interpreting the many possible options specified in a separate formatting string. But in many cases all that is really needed is simple right justification. To right-align an integer value (or series of values) you can use RSET to assign the numbers into a string, and then print that string as shown below.

```
Work$ = SPACE$(10)
RSET Work$ = STR$(Number)
PRINT TAB(15); Work$
```

In this case, Work\$ could also have been dimensioned as a fixed-length string. Adding leading zeros to a number is also quite easy using RIGHT\$ like this:

```
PRINT RIGHT$("00000" + LTRIM$(STR$(Number)), 6)
```

You will need at least as many zeros in the string as the final result requires, less one since STR\$ always returns at least one digit. Trailing digits are handled similarly, except you would use LEFT\$ instead of RIGHT\$.

Rounding numbers is an equally common need, and there are several ways to handle this. Of course, INT and FIX can be used to truncate a floating point value to an integer result, but neither of these perform rounding. For that you should use CINT or CLNG, which do round the number to the closest integer value. For example, `Value = CINT(3.59)` will assign 4 to Value, regardless of whether Value is an integer, single precision, or whatever.

Some BASICs have a CEIL function, which returns the next higher integer result. That is, CEIL(3) is 3, but CEIL(3.01) returns the value 4. This function can be easily simulated using `Ceil = -INT(-Number)`.

Rounding algorithms are not quite so simple to implement, as you can see in the short DEF FN function below.

```
DEF FnRound# (Value#, Digits%)
  Mult% = 10 ^ Digits%
  FnRound# = FIX((Mult% * Value#) + (SGN(Value#)) * .5#) / Mult%
END DEF
```

Another important math optimization is to avoid exponentiation whenever possible. Whether you are using integers or floating point numbers, using `Number ^ 2` and `Number ^ 3` are many times slower than `Number * Number`, and `Number * Number * Number`, respectively.

String Tricks and Minimizing Parameters

There are a few string tricks and issues worth mentioning here too. The fastest and smallest way to clear a string without actually deleting it is with `LSET Work$ = ""`. Another clever and interesting string trick lets you delete a string with only nine bytes of code, instead of the usual 13.

In Chapter 6 you learned that the assembly language routines within BASIC's runtime library are accessible if you know their names. You can exploit that by using the B\$STD L (string delete) routine, which requires less code to set up and call than the more usual `Work$ = ""`. When a string is assigned to a null value, two parameters—the address of the target string and the address of the null—are passed to the string assignment routine. But B\$STD L needs only the address of the string being deleted. You might think that BASIC would be smart enough to see the "" null and call B\$STD L automatically, but it doesn't. Here is how you would declare and call B\$STD L:

```
DECLARE SUB DeleteStr ALIAS "B$STD L" (Work$)
CALL DeleteStr(Any$)
```

As with the examples that let you call GET # and PUT # directly, DeleteStr will not work in the QB environment unless you first create a wrapper subprogram written in BASIC, and include that wrapper in a Quick Library. And this brings up an important point. Why bother to write a BASIC subprogram that in turn calls an internal routine, when the BASIC subprogram could just as easily delete the string itself? Therefore, the best solution—especially because it's also the easiest—is to write DeleteStr in BASIC thus:

```
SUB DeleteStr(Work$)
  Work$ = ""
END SUB
```

This is an important concept to be sure, because it shows how to reduce the number of parameters when a particular service is needed many times. Other similar situations are not hard to envision,

whereby multiple parameters that do not change from call to call can be placed into a subprogram that itself requires only one or two arguments.

This technique can be extended to several BASIC statements that use more parameters than might otherwise be apparent. For example, whenever you use LOCATE, additional hidden parameters are passed to the B\$LOCT routine beyond those you specify. The statement LOCATE X, Y generates 22 bytes of code, even though other called routines that take two parameters need only 13. (Every passed parameter generates four bytes of code, and the actual CALL adds five more. This is the same whether the routine being called is an internal BASIC statement, a BASIC subprogram or function, or an assembly language routine). Therefore, if you use LOCATE with two arguments frequently in a program, you can save nine bytes for each by creating a BASIC subprogram that performs the LOCATE:

```
SUB LocateIt(Row, Column) STATIC
  LOCATE Row, Column
END SUB
```

Similarly, if you frequently turn the cursor on and off, you should create two subprograms—perhaps called CursorOn and CursorOff—that invoke LOCATE. Since no parameters are required, the savings will add up quickly. Calling either of the subprograms below generates only five bytes of code, as opposed to 18 for the statement LOCATE , , 1 and 20 for LOCATE , , 0.

```
SUB CursorOn STATIC
  LOCATE , , 1
END SUB

SUB CursorOff STATIC
  LOCATE , , 0
END SUB
```

The COLOR statement also requires more parameters than the number of arguments you give. Where COLOR FG, BG generates 22 bytes of compiled code, CALL ColorIt(FG, BG) creates only 13. CLOSE is yet another BASIC statement that accepts multiple arguments, and it too requires hidden parameters. Using CLOSE #X compiles to 13 bytes, and CALL CloseIt(X) is only nine.

The reason that BASIC sends more parameters than you specify is because these routines need extra information to know which and how many arguments were given. In the case of LOCATE, each argument is preceded with a flag that tells if the next one was given. CLOSE is similar, except the last parameter tells how many file numbers were specified. Remember, you can use CLOSE alone to close all open files, or CLOSE 1, 3, 4 to close only those files numbers. Therefore, BASIC requires some way to tell the CLOSE statement how many file numbers there are.

Another place where several statements can be consolidated within a single procedure is when peeking and poking memory. BASIC's PEEK and POKE are limited because they can access only one byte in memory at a time. But many useful memory locations are in fact organized as a pair of bytes, as you will see in Chapter 10. Instead of using code to combine or separate the bytes each time memory is accessed, you can use the following short routines that let you peek and poke two bytes at once.

```

DECLARE FUNCTION PeekWord%(Address%)
  PeekWord% = PEEK(Address%) + 256 * PEEK(Address% + 1)
END FUNCTION

DECLARE SUB PokeWord(Address%, Value%)
  POKE Address%, Value% AND 255
  POKE Address% + 1, Value% \ 256
END SUB

```

Because these routines use BASIC's PEEK and POKE, you still need to use DEF SEG separately. Of course, the segment could be added as another parameter, and assigned within the routines:

```

DECLARE FUNCTION PeekWord%(Segment%, Address%)
  DEF SEG = Segment%
  PeekWord% = PEEK(Address%) + 256 * PEEK(Address% + 1)
END FUNCTION

```

Word Wrapping

A string handling technique you will surely find useful is implementing word wrapping. There are a number of ways to do this, and the following code shows one that I have found to be very efficient.

```

DEFINT A-Z
SUB WordWrap (X$, Wide, LeftMargin)
  Length = LEN(X$) 'remember the length
  Pointer = 1 'start at the beginning of the string
  IF LeftMargin = 0 THEN LeftMargin = 1

  'Scan a block of Wide characters backwards, looking for a blank. Stop
  ' at the first blank, or upon reaching the beginning of the string.
  DO
    FOR X = Pointer + Wide TO Pointer STEP -1
      IF MID$(X$, X, 1) = " " OR X = Length + 1 THEN
        LOCATE , LeftMargin
        PRINT MID$(X$, Pointer, X - Pointer);
        Pointer = X + 1
        WHILE MID$(X$, Pointer, 1) = " "
          Pointer = Pointer + 1
        WEND
        IF POS(0) > 1 THEN PRINT
        EXIT FOR
      END IF
    NEXT
  LOOP WHILE Pointer < Length
END SUB

```

The WordWrap subprogram expects the text for display to be in a single long string. You pass it that text, a left margin, and a width. You could certainly add enhancements to this routine such as a color parameter, or the ability to format the text and send it to a printer or disk file.

Unusual Ways to Access Display Memory

If you ever tried to print a character in the lower-right corner of the display screen, you probably discovered that it cannot be done—as with many BASIC versions—without causing the screen to scroll up. The only solution I am aware of is to use POKE to assign the character (and optionally its color) to display memory directly as shown below.

```
DEF SEG = &HB800      'use &HB000 for a monochrome display
POKE 3998, 65        'ASCII code for the letter "A"
POKE 3999, 9         'bright blue on black
```

The second trick also uses display memory in an unconventional manner. All video adapters contain at least 4096 bytes of on-board memory. Even though a 25 line by 80 column text mode screen uses only 4000 bytes (2000 characters plus 2000 colors), memory chips are built in multiples of 1,024 bytes. Therefore, you can use the last 96 bytes on the display adapter in your programs. If the adapter supports multiple video pages, then you can use the last 96 bytes in each 25-line page.

One use for this memory is to provide a way to communicate small amounts of information between separate programs. When you don't want to structure an application to use CHAIN, the only other recourse is to use a disk file to pass information between the programs. But if all that is needed is a file name or drive letter, using a file can be awkward and slow, especially if the program is running from a floppy disk.

One way to access this video memory is with PEEK and POKE. But PEEK and POKE are awkward too, and can access only one byte at a time. A better approach is to use an assembly language routine to copy one contiguous memory block to another location. The MemCopy routine below is designed to do exactly this.

```
;MEMCOPY.ASM, copies a block of memory from here to there

.Model Medium, Basic
.Code

MemCopy Proc Uses DS ES SI DI, FromAdr:DWord, ToAdr:DWord, NumBytes:Word
    Cld                ;copy in the forward direction
    Mov  SI,NumBytes   ;get the address for NumBytes%
    Mov  CX,[SI]       ;put it into CX for copying below

    Les  DI,FromAdr    ;load ES:DI with the source address
    Lds  SI,ToAdr      ;load DS:SI with destination address

    Shr  CX,1          ;copy words instead of bytes for speed
    Rep  Movsw         ;do the copy
    Adc  CX,CX         ;this will set CX to either 0 or 1
    Rep  Movsb         ;copy the odd byte if necessary

    Ret                ;return to BASIC

MemCopy Endp
End
```

MemCopy may be declared and called in two different ways. The first uses SEG and is most appropriate when you are copying data between variables, for example from a group of elements in one array to elements in another. The second lets you specify any arbitrary segment and address, and it requires the BYVAL modifier either in the DECLARE statement, the CALL, or both. Each method is shown below.

```
DECLARE SUB MemCopy(SEG AnyVar1, SEG AnyVar2, Numbytes%)
CALL MemCopy(AnyVar1, AnyVar2, NumBytes%)
```

```
DECLARE SUB MemCopy(BYVAL Seg1%, BYVAL Adr1%, BYVAL Seg2%, BYVAL Adr2%, _
NumBytes%)
CALL MemCopy(SourceSeg%, SourceAdr%, DestSeg%, DestAdr%, NumBytes%)
```

You may also use a combination of these, perhaps with SEG for the source argument and BYVAL for the second. For example, to copy a 20-byte TYPE variable to the area just past the end of video memory on a color display adapter you would do this:

```
CALL MemCopy(SEG TypeVar, BYVAL &HB800, BYVAL 4000, 20)
```

In many cases you may need to use MemCopy in more than one way in the same program. For this reason it is probably better not to declare it at all. Once a subprogram or function has been declared, BASIC will refuse to let you change the number or type of parameters. But if you don't include a declaration at all, you are free to use any combination of SEG and BYVAL, and also any type of variable.

It is important to understand that numeric and TYPE variables should be specified using SEG, so MemCopy will know the full address where the variable resides. You could use a combination of BYVAL VARSEG(Variable) and BYVAL VARPTR(Variable), but that is not quite as efficient as SEG. Copying to or from a conventional string using QuickBASIC requires SADD (string address) instead of VARPTR; far strings in BASIC 7 require SADD, and also SSEG (string segment) instead of VARSEG.

Rebooting a PC

Another simple trick that is not obvious to many programmers is how to reboot a PC. Although most PC technical reference manuals show an interrupt service for rebooting, that simply does not work with most computers. However, every PC has a BIOS routine that is at a fixed address, and which may be called directly like this:

```
DEF SEG = &HFFFF
CALL Absolute(0)
```

The Absolute routine is included in the QB and QBX libraries that come with BASIC. If a cold boot with the full memory test and its attendant delay is acceptable, then the code shown above is all that

you need. Otherwise, you must poke the special value &H1234 in low memory as a flag to the BIOS routine, so it will know that you want a warm boot instead:

```
DEF SEG = 0
POKE &H473, &H12
POKE &H472, &H34
DEF SEG = &HFFFF
CALL Absolute(0)
```

Integer Values Greater Than 32K

As you learned in Chapter 2, an integer variable can hold any value between -32768 and 32767. When this range of numbers is considered, the integer is referred to as being a signed number. But the same range of values can also be treated as unsigned numbers spanning from 0 through 65535. Since BASIC does not support unsigned integers, additional trickery is often needed to pass values between 32768 and 65535 to assembler routines and DOS and BIOS services you invoke with CALL Interrupt. One way to do this is to use a long integer first, and add an explicit test for values higher than 32767:

```
Temp& = NumBytes&
IF Temp& > 32767 THEN
  IntBytes% = Temp& - 65536
ELSE
  IntBytes% = Temp&
END IF
```

To reverse the process you would test for a negative value:

```
IF IntBytes% < 0 THEN
  NumBytes& = IntBytes% + 65536
ELSE
  NumBytes& = IntBytes%
END IF
```

Although this method certainly works, it is inefficient because of the added IF testing. When you merely need to pass a variable to a called routine, you can skip this testing and simply pass the long integer directly. This may appear counter to the rule that you must always pass the same type of variable that a subroutine expects. But as long as the arguments are not being passed by value using BYVAL, this method works and adds no extra code.

When a parameter is passed to a subprogram or function, BASIC sends the address of its first byte as shown in Figure 9-1.

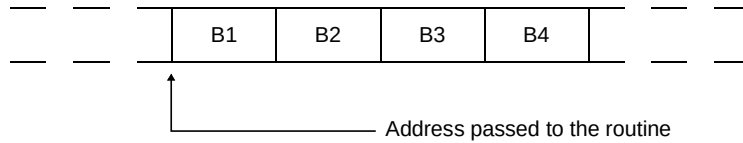


Figure 9-1: Passing a long integer where a regular integer is expected.

Here, B1, B2, and so forth refer to the Bytes 1 through 4 of a long integer variable. Since the assembly language routine is expecting a regular integer, it looks at just the first two bytes of the variable. Thus, a long integer can be used even when a conventional integer is expected. Of course, any excess greater than 65535 will be ignored by the routine, since the bits that hold the excess are in the third and fourth bytes.

Benchmarking

Throughout this book I have emphasized the importance of writing code that is as small and fast as possible. And these goals should be obvious to all but the most novice programmer. But it is not always obvious how to determine for yourself which of several approaches yields code that is the smallest or fastest. One way is to use Microsoft CodeView, which lets you count the bytes of assembler code that are generated. This is how I obtained the byte counts stated throughout this book.

But smaller is not always faster. Further, the code that BASIC generates is not the whole story. In many cases BASIC makes calls to its runtime library routines, and you would have to trace through those as well to know the total byte count for a given statement. It is not impossible to trace through the BASIC runtime using CodeView, but it certainly can be tedious. Many of BASIC's internal routines are very convoluted—especially those that allocate and deallocate string and other memory. Often it is simpler to devise a test that executes a series of statements many times, and then time how long the test took.

As an example for this discussion, I will compare two different ways to print three strings in succession and show how to tell which produces less code, and which is faster. The first statement below prints each string separately, and the second combines the strings and then prints them as one.

```
1: PRINT X$; Y$; Z$
2: PRINT X$ + Y$ + Z$
```

Since the length of each string will certainly influence how long it takes to print them, each of the strings is first initialized to 80 characters as follows:

```
X$ = STRING$(80, "X")
Y$ = STRING$(80, "Y")
Z$ = STRING$(80, "Z")
```

It is important to understand that the PRINT statement itself will be a factor, since it takes a certain amount of time to copy the characters from each string to display memory. Worse, if the screen needs to be scrolled because the text runs past the bottom of the display, that will take additional time. To avoid the overhead of scrolling, the test program uses LOCATE to start each new print statement at the top of the screen. Of course, using LOCATE adds further to the overhead, but in this case much less than scrolling would. To prove this to yourself, disable the line that contains the LOCATE statement. Here's the complete benchmark program:

```
CLS
X$ = STRING$(80, "X")   'create the test string
Y$ = STRING$(80, "Y")
Z$ = STRING$(80, "Z")

Synch! = TIMER          'synchronize to TIMER
DO
  Start! = TIMER
LOOP WHILE Start! = Synch!

FOR X = 1 TO 1000      '1000 times is adequate
  LOCATE 1
  PRINT X$; Y$; Z$
NEXT

Done! = TIMER          'calculate elapsed time
Test1! = Done! - Start!

Synch! = TIMER          'as above
DO
  Start! = TIMER
LOOP WHILE Start! = Synch!

FOR X = 1 TO 1000
  LOCATE 1
  PRINT X$ + Y$ + Z$
NEXT

Done! = TIMER
Test2! = Done! - Start!

PRINT USING "##.## seconds using three strings"; Test1!
PRINT USING "##.## seconds using concatenation"; Test2!
```

Notice the extra step that synchronizes the start of each test to BASIC's TIMER function. As you probably know, the PC's system time is updated approximately 18 times per second. Therefore, it is possible that the test loop could begin just before the timer is about to be incremented. In that case the elapsed time would appear to be 1/18th second longer than the actual time. To avoid this potential inaccuracy, the DO loop waits until a new time period has just begun. There is still a similar accuracy loss at the end of the test when Done! is assigned from TIMER. But by synchronizing the start of the test, the error is limited to 1/18th second instead of twice that.

When you compile and run this program using QuickBASIC 4.5, it will be apparent that the first test is more than three times faster than the second. However, with BASIC 7.1—using either near or far

strings—the second is in fact slightly faster. Therefore, which is better depends on the version of your compiler, and there is no single best answer. Now let's compare code size.

The disassemblies shown below are valid for both QuickBASIC 4.5 and BASIC 7.1. By counting bytes you can see that printing the strings using a semicolon generates 27 bytes, while first concatenating the strings requires 29 bytes.

```
PRINT X$; Y$; Z$
B83600      MOV    AX,X$    ;get the address for X$
50          PUSH   AX      ;pass it on
9AD125FF4A  CALL  B$PSSD   ;print with a semicolon
B83A00      MOV    AX,Y$    ;as above for Y$
50          PUSH   AX
9AD125FF4A  CALL  B$PSSD
B83E00      MOV    AX,Z$
50          PUSH   AX
9AD625FF4A  CALL  B$PESD   ;print with end of line

PRINT X$ + Y$ + Z$
B83600      MOV    AX,X$    ;get the address for X$
50          PUSH   AX      ;pass it on
B83A00      MOV    AX,Y$    ;get the address for Y$
50          PUSH   AX      ;pass that on too
9AD728FF4A  CALL  B$SCAT   ;call String Concatenate
50          PUSH   AX      ;pass the combined result
B83E00      MOV    AX,Z$    ;get the address for Z$
50          PUSH   AX      ;pass it on
9AD728FF4A  CALL  B$SCAT   ;combine that too
50          PUSH   AX      ;pass X$ + Y$ + Z$
9AD625FF4A  CALL  B$PESD   ;print with end of line
```

Even though the first example uses a single PRINT statement, BASIC treats it as three separate commands:

```
PRINT X$;
PRINT Y$;
PRINT Z$
```

The second example that concatenates the strings requires slightly more code because of the repeated calls to the B\$SCAT (string concatenate) routine. Therefore, if you are using QuickBASIC it is clear that printing the strings separately is both smaller and faster. BASIC PDS users must decide between slightly faster performance, or slightly smaller code.

These tests were repeated 1000 times to minimize the inaccuracies introduced by the timer's low resolution. Since this method of timing can be off by as much as 1/18th second (55 milliseconds), for test results to be accurate to 1% the test must take at least 5.5 seconds to complete. In most cases that much precision is not truly necessary, and other factors such as the time to use LOCATE will prevent absolute accuracy anyway.

It is important that any timing tests you perform be done after compiling the program to an .EXE file. The BASIC editor is an interpreter, and is generally slower than a stand-alone program. Further, the

reduction in speed is not consistent; some statements are nearly as fast as in a compiled program, and some are much slower.

To obtain more accurate results than those shown here requires some heavy ammunition. I recommend the Source Profiler from Microsoft. This is a utility program that times procedure calls within a running program to an accuracy of one microsecond. The Source Profiler supports all Microsoft languages including QuickBASIC and BASIC PDS.

To time a program you must compile and link it using the /zi and /co CodeView switches. This tells BASIC and LINK to add symbolic information that shows where variables and procedures are located, and also relates each logical line of source code to addresses in the .EXE file. The Source Profiler then uses this information to know where each source-language statement begins and ends.

You should also understand that there's a certain amount of overhead associated with the timing loop itself. Any FOR/NEXT loop requires a certain amount of time just to increment the counter variable and compare it to the ending value. Fortunately, this overhead can be easily isolated, using an empty loop with the same number of iterations. The short complete program that follows shows this in context.

```
Synch! = TIMER
DO
  Start! = TIMER
LOOP WHILE Start! = Synch!

FOR X& = 1 TO 50000
NEXT

Done! = TIMER
Empty! = Done! - Start!
PRINT USING "##.## seconds for the empty loop"; Empty!

Synch! = TIMER
DO
  Start! = TIMER
LOOP WHILE Start! = Synch!

FOR X& = 1 TO 50000
  X! = -Y!
NEXT

Done! = TIMER
Assign! = Done! - Start!
PRINT USING "##.## seconds for the assignments"; Assign!

Actual! = Assign! - Empty!
PRINT USING "##.## seconds actually required"; Actual!
```

Summary

In this chapter you learned a variety of programming shortcuts and other techniques. You saw firsthand how it is more efficient to avoid using CHR\$ and other BASIC functions repeatedly, in favor a single call ahead of time when possible. In a similar vein, you can reduce the size of your programs by

consolidating multiple instances of UCASE\$, MID\$, LTRIM\$, and other functions once before a series of IF tests, rather than use them each time for each test.

You also learned that assigning multiple variables in succession from another often results in smaller code than assigning from the same numeric constant. Short circuit expression evaluation was described, and examples showed you how that technique can improve the efficiency of a QuickBASIC program. But since BASIC PDS already employs this optimization, multiple AND conditions are not needed when using that version of compiler.

This chapter explained the importance of reducing the number of parameters you pass to a subprogram or function, and showed how you can use GOSUB to invoke a central handler that in turn calls the routine. Likewise, when using BASIC statements such as LOCATE, COLOR, and CLOSE that require additional arguments beyond those you specify, a substantial amount of code can be saved by creating a BASIC subprogram wrapper. Examples for turning the cursor on and off were shown, and these can save 13 and 15 bytes per use respectively.

Several programming techniques were shown, including a word wrap subprogram, a numeric rounding function, and a simple way to reboot the PC. You also learned how small amounts of data can be safely stored in the last 96 bytes of video memory, perhaps for use as a common data area between separately run and non-chained programs.

Finally, this chapter included a brief discussion of some of the issues surrounding benchmarking, and explained how to obtain reasonably accurate statement timings. To determine the size of the compiler-generated code requires disassembling with CodeView.

Chapter 10 continues with a complete list of key addresses in low memory you are sure to find useful, and discusses each in depth along with accompanying examples.

10

Key Memory Areas in the PC

Two very important BASIC keywords that are sadly neglected by many programmers are PEEK and POKE. Most people understand that these let you read from and write to memory locations. But what are they really good for? The whole point of a high-level language like BASIC is to avoid such direct memory access, and to many programmers these commands may seem like an enigma.

In most cases, you don't need to access memory with PEEK and POKE. Unlike C and assembly language that require direct memory operations to process strings and arrays, BASIC includes a full complement of commands for this. However, there is at least one important use for PEEK and POKE that cannot be accomplished in any other way: accessing low memory.

The portion of memory in every PC that begins at Hex address 0000:0400 is called the *BIOS Data Area*, and it contains much useful information. For example, the equipment word at address &H410 tells how many diskette drives are installed, and how many parallel and serial ports there are. The keyboard status flags at address &H417 can be read (and written), to reflect whether the Caps Lock and NumLock states are active.

In this chapter I will describe all of the low memory locations that are relevant to a BASIC program, and present numerous practical examples to show how this data can be utilized. This is by no means a complete list of every BIOS data address that is available in the PC. Rather, I have purposely limited it to those that I have found useful.

Improving PEEK and POKE

One potential limitation that needs to be addressed first is how to access full words of data. BASIC's PEEK and POKE operate on single bytes only, and reading or writing two bytes at a time is a messy proposition at best.

Chapter 9 introduced a pair of routines called PeekWord and PokeWord, that allowed accessing memory a word at a time. In the context those were presented, a fair amount of code could be saved by consolidating the necessary code into a subprogram or function. But in the interest of speed and even further code size reductions, the following assembly language routines are better still.

```
;PEEKPOKE.ASM, simplifies access to full words  
  
.Model Medium, Basic  
.Code  
  
PeekWord Proc Uses ES, SegAddr:DWord
```

```

    Les  BX,SegAddr      ;load the segment and address
    Mov  AX,ES:[BX]     ;read the word into AX
    Ret                ;return to BASIC
PeekWord Endp

```

```

PokeWord Proc Uses ES, SegAddr:DWord, Value:Word
    Les  BX,SegAddr      ;load the segment and address
    Mov  AX,Value        ;and the new value to store there
    Mov  ES:[BX],AX     ;write the value into memory
    Ret                ;return to BASIC
PokeWord Endp
End

```

Both of these routines expect the parameters to be passed by value, for faster speed and smaller code. Therefore, you will declare them as follows:

```

DECLARE FUNCTION PeekWord%(BYVAL Segment%, BYVAL Address%)
DECLARE SUB
    PokeWord(BYVAL Segment%, BYVAL Address%, BYVAL Value%)

```

Then to read a word of memory, say, the address of the LPT1 printer adapter at address &H408, PeekWord would be invoked like this:

```

LPT1Addr% = PeekWord%(0, &H408)

```

And to write the letter "A" in the lower left corner of a color display screen in white on blue you could use PokeWord, thus:

```

CALL PokeWord(&HB800, 3998, &H1741)

```

Notice that PeekWord returns a negative value for numbers greater than 32767. This is normal, as explained in Chapter 2. However, the same negative value that PeekWord returns can be used as an argument to PokeWord with the correct results.

Low Memory Addresses

The sections that follow are organized by category, since this is how low memory is arranged in the PC. That is, one section discusses the RS-232 communications data area, the next shows the portion of memory used by the printer adapters, and so forth. Each address is listed in ascending order; by convention, Hex notation is used exclusively for these addresses. In all of the examples shown here, you will use a segment value of zero.

It is important to understand that besides memory addresses that are accessed with PEEK and POKE (or in this case their full-word equivalents), the IBM PC family also has a series of input and output ports. These ports are accessed using INP and OUT commands instead of PEEK and POKE. I mention this here because ports are referred to in several places in the discussions that follow. In particular, the communications ports that are exchanged in the next section are in fact port numbers, and not memory

addresses. Some useful port numbers are given at the end of this chapter, along with code examples that show how to read from and write to them.

Table 10-1 provides a summary of all the low memory addresses that are described in this chapter.

Address	Meaning
&H400	2 bytes, COM1 port number
&H402	2 bytes, COM2 port number
&H404	2 bytes, COM3 port number
&H406	2 bytes, COM4 port number
&H408	2 bytes, LPT1 port number
&H40A	2 bytes, LPT2 port number
&H40C	2 bytes, LPT3 port number
&H40E	2 bytes, LPT4 port number
&H410	2 bytes, Equipment List
&H413	2 bytes, installed memory (K)
&H417	2 bytes, keyboard status
&H418	2 bytes, enhanced keyboard status
&H41A	2 bytes, keyboard buffer head pointer
&H41C	2 bytes, keyboard buffer tail pointer
&H41E	30 bytes, keyboard buffer
&H43F	1 byte, diskette motor on indicator
&H440	1 byte, diskette motor countdown timer
&H449	1 byte, current video mode

Address	Meaning
&H44A	2 bytes, current screen width (columns)
&H44C	2 bytes, current video page size (bytes)
&H462	1 byte, current video page number
&H463	2 bytes, CRT controller port number
&H46C	4 bytes, long integer system timer count
&H478	4 bytes, LPT1 - LPT4 timeout values
&H484	1 byte, EGA/VGA screen height (rows)
&H485	2 bytes, character height (scan lines)
&H487	1 byte, EGA/VGA Features bits
&H4F0	16 bytes, Inter-Application Area
&H500	1 byte, PrtSc busy flag
&H504	1 byte, active drive for one-diskette PC

Table 10-1: Key low memory addresses in the PC.

Communications Port Addresses

The four words starting at address &H400 hold the port numbers for each installed RS-232 communications adapter. For example, the port number for COM1 is contained in the word at address &H400, and the port number for COM3 is at address &H404. Because these port numbers are words rather than bytes, the COM1 port number is contained in both &H400 and &H401. Thus, COM2 starts at address &H402, and COM3 starts at &H404.

BASIC allows you to open only COM ports 1 and 2; however by exchanging these addresses you can substitute ports 3 and 4 if necessary. The complete program that follows first swaps the port numbers for COM1 and COM3, and then opens COM1 for output. Since the port numbers are swapped, it is actually COM3 that is being opened.

```
DEFINT A-Z
DECLARE FUNCTION PeekWord% (BYVAL Segment, BYVAL Address)
DECLARE SUB PokeWord (BYVAL Segment, BYVAL Address, BYVAL Value)
COM1 = PeekWord%(0, &H400)      'save COM1 port number
COM3 = PeekWord%(0, &H404)      'save COM3 port number
CALL PokeWord(0, &H400, COM3)  'assign COM3 to COM1
CALL PokeWord(0, &H404, COM1)  'and then COM1 to COM3

OPEN "COM1:1200,N,8,1,RS,DS" FOR RANDOM AS #1
PRINT #1, "ATDT 1-555-1212"    'dial information
CLOSE #1

CALL PokeWord(0, &H400, COM1)  'restore the original values
CALL PokeWord(0, &H404, COM3)
```

Printer Port Addresses

The four printer port numbers start at address &H408, and they are similar to those used to hold the communications ports and may also be exchanged if necessary. For example, if you have a program that uses LPRINT commands, all printed output will be sent to LPT1. If at some later time you want to use the same program with LPT2, you can exchange the port numbers instead of having to rewrite the program. A short code fragment that does this is shown following.

```
DEFINT A-Z
DECLARE FUNCTION PeekWord% (BYVAL Segment, BYVAL Address)
DECLARE SUB PokeWord (BYVAL Segment, BYVAL Address, BYVAL Value)
LPT1 = PeekWord%(0, &H408)      'save LPT1 port number
LPT2 = PeekWord%(0, &H40A)      'save LPT2 port number
CALL PokeWord(0, &H408, LPT2)  'assign LPT2 to LPT1
CALL PokeWord(0, &H40A, LPT1)  'and LPT1 to LPT2

LPRINT "This is printed on LPT2"
CALL PokeWord(0, &H408, LPT1)  'restore the original values
CALL PokeWord(0, &H40A, LPT2)
LPRINT "And now we're back to LPT1"    'prove it worked
```

Like the communications port addresses, each printer port address is a full word, so while the first is located at address &H408, the second is at &H40A. You will also find PeekWord useful because it does not require you to change the current DEF SEG setting. Although there is no harm in assigning a new DEF SEG value in most cases, it is not easy to restore it to the original setting. Therefore, when writing reusable subprograms and functions that need to access memory, you don't have to worry about affecting a subsequent PEEK or BLOAD in the main program.

System Data

One of the most valuable data items in low memory is the equipment list in the word starting at address &H410. The information contained here is bit coded, to indicate which and how many peripherals are installed in the host PC. Figure 10-1 shows the organization of this word. Bits not identified are either reserved, or not particularly useful.

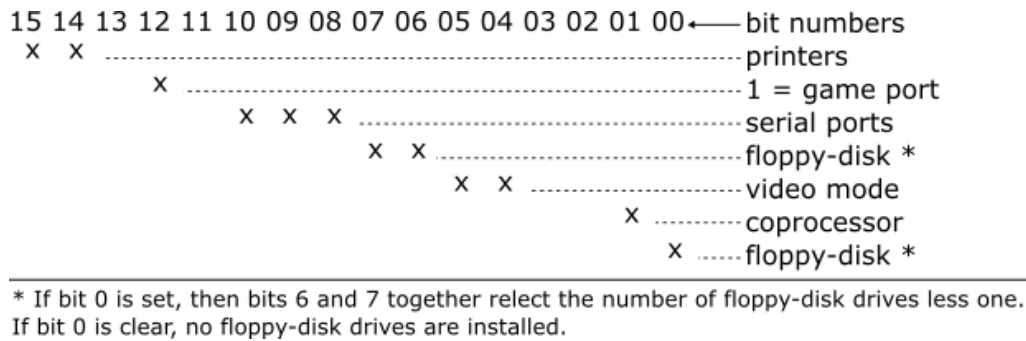


Figure 10-1: The organization of the equipment list word at address &H410.

Because the data in this word is bit coded, you must use AND to extract the necessary information. For example, to see if a math coprocessor is installed you must turn off all but bit 1, and see if the result is zero or not:

```
IF PeekWord%(0, &H410) AND 2 THEN
  PRINT "A coprocessor is installed."
ELSE
  PRINT "Sorry, no coprocessor detected."
END IF
```

This brings up an important point, because it is not immediately obvious what values you should use to isolate the various bits in a word. It would be terrific if Microsoft BASIC offered the ability to handle binary values directly. The Microsoft Macro Assembler allows this, as does PowerBasic. In the absence of &B and a BIN\$ function, the following short function can be used to determine the correct integer value for a given sequence of binary bits.

```
FUNCTION Bin% (Bit$) STATIC
  Temp& = 0
  Length = LEN(Bit$)
  FOR X = 1 TO Length
    IF MID$(Bit$, Length - X + 1, 1) = "1" THEN
      Temp& = Temp& + 2 ^ (X - 1)
    END IF
  NEXT
  IF Temp& > 32767 THEN
    Bin% = Temp& - 65536
  ELSE
    Bin% = Temp&
  END IF
END FUNCTION
```

Given a string of binary digits of the form "01011001", the Bin function returns an equivalent integer value. You could add this function to your programs, or use it to determine constant values ahead of

time. For example, to determine the number of diskette drives that are installed requires isolating bits 6 and 7. This is simple in assembly language, where you can specify an AND mask using 11000000b as a value. The example below obtains the equipment word, and then uses the Bin function to disable all but bits 6 and 7.

```
Equipment = PeekWord%(0, &H410)
Floppies = 1 + (Equipment AND Bin%("11000000")) \ 64
PRINT Floppies; "diskette drive(s) installed"
```

Although the Bin function is used in the code, I recommend that you create a simple test program first, to determine the value of 11000000 (192) once ahead of time. Then, the Bin function can be omitted from the final program and the second line would be changed as follows:

```
Floppies = 1 + (Equipment AND 192) \ 64
```

Notice the use of parentheses to force BASIC to combine Equipment and the number 192 before dividing by 64 with AND. If these are omitted BASIC will instead combine Equipment with the result of 192 divided by 64, which is not correct.

One final technique you should understand is how to shift bits into the correct position to obtain the actual value the bits represent. Treated as bits alone, the number of diskette drives is represented as 00, 01, 10, or 11, and the decimal equivalents for these binary numbers are 0, 1, 2, and 3. But because of their positioning in the equipment word, the bits must be shifted to the right six places. After all, the value 11000000 (192) is certainly not the same as the value 11 (3).

This is handled simply and elegantly using integer division as shown. To shift a number right one position divide it by 2; to shift right 2 places divide by 4, and so forth. Since the diskette bits need to be shifted six places, the equipment variable is divided by 64 after AND is used to mask off the unrelated bits. Likewise, to shift bits left you can multiply by 2, 4, 8, and so forth. The number to use when dividing or multiplying can also be determined by raising 2 to the number of bits power. For example, to shift a number right five places you would divide by $2^5 = 32$.

A problem arises when dealing with the highest order bit, because to BASIC this bit implies a negative number. Therefore, when bit 15 is set, dividing will not produce the expected results. One workaround that is admittedly clumsy is to test that bit explicitly, then mask it off and shift the bits as needed, and finally use an IF test to see if the bit had been set. The only place this is necessary in the equipment list is when reading the number of parallel printers that are present. The first example below reports the number of serial ports, and the second tells how many parallel ports are installed.

```
Equipment = PeekWord%(0, &H410)
Serial = (Equipment AND Bin%("110000000000")) \ 512
PRINT Serial; "serial port(s) installed"

IF Equipment AND Bin%("1000000000000000") THEN
  HiBitSet = -1
END IF
Parallel = (Equipment AND Bin%("0100000000000000")) \ 16384
IF HiBitSet THEN
  Parallel = Parallel + 2
```

```
PRINT Parallel; "parallel port(s) installed"
```

In the interest of completeness I should point out that it is not strictly necessary to manipulate bit 15 when accessing the equipment word. Since none of the information straddles a byte boundary, BASIC's PEEK can in fact be used to read just the high byte. Since a byte value is never higher than 255, the entire issue of saving and then masking that bit can be avoided. But there are other situations you may encounter where an entire word must be processed and the highest bit may be set.

The final useful item in the equipment word is the initial video mode. I've seen many programmers read use information to determine if a color or monochrome monitor is installed like this:

```
DEF SEG = 0
IF (PEEK(&H410) AND &H30) = &H30 THEN
  ' monochrome
ELSE
  ' color
END IF
```

There are two problems with this approach. The most serious is that this reflects the monitor that was active when the PC was first powered up. These days, many people have two monitors connected to their PC, and you usually need to know which is currently active. The other problem is this requires more code than the better method I showed in Chapter 6 which reads the port address of the currently active video adapter:

```
DEF SEG = 0
IF PEEK(&H463) = &HB4 THEN
  ' monochrome
ELSE
  ' color
END IF
```

Besides the equipment word at address &H410, another word at address &H413 holds the amount of memory that is installed in Kilobytes. Note that this word does not reflect any extended or expanded memory that may be present. Also note that a much better indicator of how much memory is actually available to a program is BASIC's FRE(-1) function. The short code fragment below shows how to determine the total DOS-accessible memory that is installed.

```
TotalK = PeekWord%(0, &H413)
PRINT TotalK; "K Bytes present in this PC."
```

Keyboard Data

As with the equipment word, the keyboard data area also maintains bit-coded information. However, this word indicates the setting of the various keyboard shift states. Unlike many of the other addresses in the BIOS data area, some of these bits may be written to as well as read from.

The byte at address &H417 shows the current status of all of the shift keys, and the upper four bits may be either read or written. The remaining bits in this byte should not be written to, nor should you alter any of the bits in the next byte at address &H418. Figure 10-2 shows the meaning of each bit in the byte at address &H417, and Figure 10-3 shows the bits at address &H418 that relate to extended keyboards only.

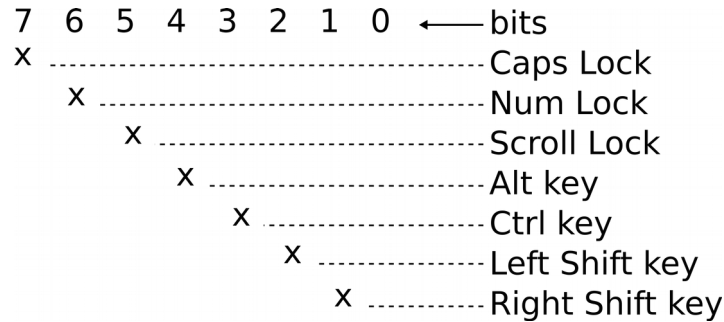


Figure 10-2: The organization of the keyboard data byte at address &H417.

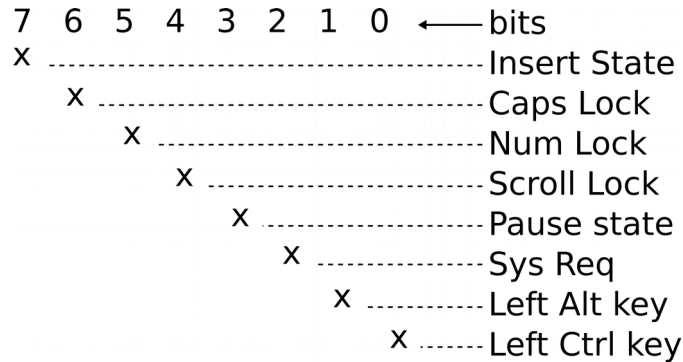


Figure 10-3: The organization of the extended keyboard data byte at address &H418.

The various flags in the upper four bits at address &H417 are toggled on and off by the BIOS each time the corresponding keys are pressed. For example, bit 6 is set while the Caps Lock is active, and bit 5 is clear when Num Lock is not in effect. Note, however, that the Insert flag is of no practical use, and you should not rely on that bit in your programs. If you are writing an input routine, or using the one shown in Chapter 6, you should keep track of the insert status manually.

The lower four bits indicate the current state of the various shift keys, and they are set only while the associated key is actually being pressed. Bits in the next word at address &H418 let you determine which Alt and Ctrl keys are pressed, for keyboards that have more than one of those keys. In most cases you will probably just want to know if these keys are active, and not distinguish between the left and the right key. Therefore, you will usually ignore the extended keyboard information, unless you need to detect the SysReq key.

As with the equipment list, you will use a combination of PeekWord (or PEEK) to read all of the flags, and then use AND to isolate just those bits you care about. Because there is only one bit that corresponds to each keyboard state flag, it is not necessary to divide or multiply to convert multiple bits into a number.

The examples below show how to test each of the bits in the byte at address &H417, without regard to the extra Ctrl and Alt key information contained at address &H418.

```
CLS
PRINT "Press the various Shift and Lock keys, ";
PRINT "then press Escape to end this madness."
COLOR 0, 7
```

```
DO
  Status = PeekWord%(0, &H417)
```

```
  LOCATE 10, 1
  IF Status AND 1 THEN
    PRINT "RightShift"
  ELSE
    GOSUB ClearIt
  END IF
```

```
  LOCATE 10, 11
  IF Status AND 2 THEN
    PRINT "Left Shift"
  ELSE
    GOSUB ClearIt
  END IF
```

```
  LOCATE 10, 21
  IF Status AND 4 THEN
    PRINT "Ctrl key"
  ELSE
    GOSUB ClearIt
  END IF
  LOCATE 10, 31
  IF Status AND 8 THEN
    PRINT "Alt key"
  ELSE
    GOSUB ClearIt
  END IF
```

```
  LOCATE 10, 41
  IF Status AND 16 THEN
    PRINT "ScrollLock"
  ELSE
    GOSUB ClearIt
  END IF
```

```
  LOCATE 10, 51
  IF Status AND 32 THEN
    PRINT "Num Lock"
  ELSE
    GOSUB ClearIt
  END IF
```

```
  LOCATE 10, 61
```

```

IF Status AND 64 THEN
    PRINT "Caps Lock"
ELSE
    GOSUB ClearIt
END IF

LOCATE 10, 71
IF Status AND 128 THEN
    PRINT "Insert"
ELSE
    GOSUB ClearIt
END IF

LOOP UNTIL INKEY$ = CHR$(27)
COLOR 7, 0
END

ClearIt:
    COLOR 7, 0
    PRINT SPACE$(10);
    COLOR 0, 7
    RETURN

```

As you can see, to read a single bit you use AND to isolate it from the rest, and then test if the result is non-zero. Setting a bit requires slightly more work, because it is important not to disturb the other bits in that byte. This requires that you first read the current information, change only the bit or bits of interest, and then write the modified data back to the same location. The next short example shows how to turn the CapsLock state on and then off again.

```

CurStatus = PeekWord%(0, &H417)
NewStatus = CurStatus OR Bin%("1000000")
CALL PokeWord(0, &H417, NewStatus)

PRINT "Press a key to turn off CapsLock"
WHILE INKEY$ = "": WEND

NewStatus = NewStatus AND Bin%("10111111")
CALL PokeWord(0, &H417, NewStatus)

```

Notice the difference between how OR is used in the first example, and how AND is used in the second one. In the first case we want to set a bit, so only that bit is specified in the binary mask. The remaining bits stay the same as they were—if they are already set then OR will leave them that way. But to turn off the CapsLock bit requires that all of the mask bits be set; except the one you wish to force off. Other bits that were already on will remain on after being combined with AND and 1.

The Keyboard Buffer

The next group of low memory keyboard addresses relate to the keyboard buffer. As you undoubtedly know, every PC has a keyboard buffer that can hold up to fifteen keystrokes. When a program is off doing something and is unable to read the keyboard, the BIOS keyboard routines will store keys that

have been typed. Then, when the program finally gets around to reading the keyboard, they are waiting there to be read. The keyboard buffer is therefore also called the *type-ahead* buffer.

A series of 34 bytes are set aside for the keyboard buffer. Two words (four bytes) are used to hold the current head and tail pointers that show where the next key will be read from, and where the next will be stored. The current head address is stored at address &H41A and the tail at address &H41C. Thirty additional bytes are used to store the actual keystrokes, with two bytes used for each. The keyboard buffer is called a *circular buffer*, because the start and end points are constantly revolving.

When a PC is first powered up, the head of the buffer holds the address &H41E, which is the start of the buffer memory area. The tail is also initially set to that same address, until a key is pressed. When that happens, the tail pointer is advanced by 2, and the character and its scan code are placed into the buffer. Each time a new key is pressed the character and scan code are added to the end of the buffer and the tail pointer is advanced by two; each time a key is read by an application the word at the current head is returned and the head pointer is advanced.

Note that the head and tail addresses assume a segment of &H40, rather than zero. Therefore, the actual values stored range from &H1E through &H3A rather than &H41E through &H43A. Of course, address 0000:041E is the same as address 0040:001E, and you can think of the buffer address either way. I usually treat all of low memory as being located in segment 0, because that can often save a byte of code. BASIC (or assembly language, for that matter) can pass the number zero by value using only three bytes, compared to the four bytes needed to pass any other number.

The program below shows how to determine the number of keys that are currently pending in the buffer, and also which one will be returned next.

```
CLS
PRINT "You have two seconds to press a few keys..."
Pause! = TIMER
WHILE Pause! + 2 > TIMER: WEND

BufferHead = PeekWord%(0, &H41A)
BufferTail = PeekWord%(0, &H41C)

NumKeys = (BufferTail - BufferHead) \ 2
IF NumKeys < 0 THEN NumKeys = NumKeys + 16
PRINT "There are"; NumKeys; "keys pending in the buffer."

PRINT "The next key waiting to be read is ";
NextKey = PeekWord%(&H40, BufferHead)
IF NextKey AND &HFF THEN
    PRINT CHR$(34); CHR$(NextKey AND &HFF); CHR$(34)
ELSE
    PRINT "Extended key scan code"; NextKey \ 256
END IF
```

This program starts by waiting two seconds giving you a chance to press a few keys. It then reads the buffer head and tail pointers, and from that calculates the number of keys that are pending in the buffer. With a circular buffer the head address may be higher the tail address, so a separate test is needed to account for that.

Next, the word at the head of the buffer is retrieved, which indicates the next available key. Since the head and tail pointers assume segment &H40, I used that instead of segment 0. `PeekWord%(0, &H41E)` produces less code than `PeekWord%(&H40, &H1E)`; however, `PeekWord%(0, &H400 + BufferHead)` is worse than `PeekWord%(&H40, BufferHead)` because of the addition needed.

Data in the keyboard buffer is always a full word, and it is up to you to determine if it is a normal ASCII key or an extended key's scan code. A normal key is indicated with a non-zero low byte, and the high byte then holds the physical hardware scan code which can usually be ignored. If the low byte instead holds a value of zero, it is an extended key and the scan code in the high byte indicates which one. Therefore, the BASIC statement `NextKey AND &HFF` masks the high byte, to test if the low byte is non-zero.

If the key is extended, then `NextKey \ 256` returns the value in the high byte. This is similar to the earlier examples that shifted bits to the right by dividing. Unlike the earlier tests that examined only some of the bits in the equipment flag, we are interested in all of the bits in the upper byte. Dividing by 256 copies the upper byte to the lower byte, thus discarding the lower byte entirely.

You should also refer back to the `StuffBuffer` program shown in Chapter 6, which accesses the keyboard buffer directly and inserts new keystrokes.

Diskette Data

There are several bytes in low memory that relate to the floppy and fixed disks in your PC, but most of them are best left alone. One exception, however, is the diskette drive motor timeout duration. Whenever a diskette drive is accessed, DOS first turns on the motor, and then waits a second or two until the motor has come up to speed. Once DOS is certain that the disk speed is correct, reading and writing are allowed.

Because of the time it takes the diskette to become ready, DOS also keeps the motor running for two more seconds after a read or write has been completed. This way, if another request comes along within that time, further delays can be avoided because the motor is already running. If you know that the data your program is accessing is on a floppy disk and there may be pauses in the reading or writing, you can force the motor to stay on longer than the normal two seconds.

The byte at address &H440 controls the motor hold time, and its value is decremented at every system timer tick, every 1/18th second. When DOS has finished accessing a diskette, it places a value into this memory location. And when the value is decremented to zero the motor is turned off. The current motor on/off state is reflected by the byte at address &H43F. The program that follows shows how you can modify the timeout value by poking a new, higher value into address &H440 immediately after a command that accesses the disk.

```
PRINT "Place a diskette in drive A and press a key ";
```

```

WHILE INKEY$ = "": WEND
FILES "A:*.*)" 'this starts the motor

DEF SEG = 0
POKE &H440, 91 'force drive motor on for five seconds
DO
  LOCATE 10, 1, 0
  PRINT PEEK(&H43F),
  PRINT PEEK(&H440)
LOOP WHILE PEEK(&H440)

BEEP 'watch the diskette light go out when you hear the beep

```

The value you store at address &H440 is the number of timer ticks that are to elapse before the motor is turned off. Since a new timer tick occurs every 18.2 seconds, you will multiply the number of seconds times this value using `Value% = Seconds * 18.2`.

Display Adapter Data

As with the diskette data area, a lot of information is available that pertains to the video display, and most of it is of little use in an application programming context. Therefore, I will discuss only some of this data.

The byte at address &H449 holds the current video mode. Unfortunately, there is no easy way to relate the information in this byte to the current BASIC SCREEN setting. Table 10-2 shows all of the possible values that might be present.

Video Mode	Description
0	40 by 25 16-color text
1	40 by 25 16-color text, with color burst
2	80 by 25 16-color text
3	80 by 25 16-color text, with color burst
4	320 by 200 pixels 4-color graphics
5	320 by 200 pixels 4-color
6	640 by 200 pixels 2-color
7	80 by 25 monochrome text
13	320 by 200 pixels 16-color graphics
14	640 by 200 pixels 16-color graphics

Video Mode	Description
15	640 by 350 pixels monochrome EGA graphics
16	640 by 350 pixels 16-color graphics
17	640 by 480 pixels 2-color graphics
18	640 by 480 pixels 16-color graphics
19	320 by 200 pixels 256-color graphics

Table 10-2: The video mode value at Hex address 0000:0449

Since you will always have set the video mode yourself with a SCREEN statement, there is little reason to have to read the current mode manually.

The word at address &H44A tells how many columns are on the display, and the word at address &H44C holds the total size of the screen in bytes. In a normal 80 column by 25 line screen mode, the value at address &H44C will be 4096, even though the screen can hold only 4000 characters.

The byte at address &H462 holds the current video page number, starting at page 0. Please understand that BASIC lets you set pages individually for writing to and displaying, and the page reported here is that which is visible on the monitor.

We have already looked at the data at address &H463, which holds the CRT controller port address. Although this address is a full word, only the lower byte needs to be examined to know the type of display that is active. If the byte value at address &H463 is &HB4, then a monochrome monitor is connected and being used. If a color adapter is active the value at this byte will instead be &HD4.

System Timer Data

Every 18th second the BIOS timer generates an interrupt that increments the master system timer count at address &H46C. This counter is stored as a four-byte long integer; the count is initialized to zero at midnight, and increases to a value of just over one 1.5 million at 11:59:59 pm.

In some cases using the BIOS timer count directly can help to reduce the size of your programs, because BASIC's TIMER requires floating point math. Chapter 9 discussed some of the issue involved in benchmarking a program, and the examples there used TIMER to know when a new 1/18th second period has just started and how long a sequence of commands took. The following short program times a long integer assignment within a FOR/NEXT loop, and it uses the PeekWord function to access the BIOS timer count directly.

```
Synch = PeekWord%(0, &H46C)
DO
```

```

    Start = PeekWord%(0, &H46C)
LOOP WHILE Synch = Start

FOR X& = 1 TO 70000
    Y& = X&
NEXT

Done = PeekWord%(0, &H46C)
PRINT Done - Start; "timer ticks have elapsed"

```

Note that it is possible for this program to report an incorrect elapsed time, since it considers only the lower of the two timer words. If the count exceeded 65,535 during the course of the timing, the lower word will have wrapped around to a value of zero. An enhancement to this technique would therefore be to create a PeekLong% function that returns the entire four bytes in one operation. You could write such a function in assembly language, or use BASIC like this:

```

FUNCTION PeekLong& (Segment%, Address%) STATIC
    PeekLong& = PeekWord%(Segment%, Address%) + 65536 * _
        PeekWord%(Segment%, Address% + 2)
END FUNCTION

```

Here, the PeekWord function is used to do most of the work, and the two words are combined into a single long integer. When many timing operations are needed using these functions can increase the speed of your programs, as well as help to avoid the inclusion of the floating point math library routines.

Printer Timeout Data

Whenever data is sent to a parallel printer it is routed through a BIOS service that handles the actual communications with the printer hardware. If the printer is turned off or disconnected, the BIOS can detect that immediately, and report the error to the calling program. But when the printer is turned on but deselected (off-line) or if it has run out of paper, the BIOS waits for a certain period of time before returning with an error condition. This gives the operator a chance to fix the problem.

The amount of time the BIOS waits varies from PC to PC, and even between different models of the same brand. The original IBM PC waited for only a very short time, and would occasionally report an error incorrectly when used with very slow printers. Modern PCs wait as long as two minutes before timing out, which is more than enough time to reload a new ream of paper. Unfortunately, if you want to test if a printer is ready before using it, your program may appear to hang if the printer is disabled.

Although BASIC provides ON ERROR to trap for printer errors, many programmers prefer to avoid ON ERROR because it makes the program larger and run more slowly. Also, ON ERROR cannot avoid the long wait the BIOS imposes. There are several solutions to this problem.

One is to print a flashing message at the bottom of the screen that says something like, "Turn on the printer!" immediately before printing, and then clear the message afterwards:

```

LOCATE 25, 1
COLOR 23
PRINT "Turn on the printer!";
LPRINT Some$
COLOR 7
PRINT SPC(20)

```

If the printer is in fact on line and ready, the message will be displayed and cleared so quickly that it is not likely to be noticed. Otherwise, the operator will see the message and take the appropriate action.

This technique can be enhanced to instead test the printer, before sending any data. The most reliable way I have found to test a printer is to first send it a CHR\$(32) space character, and if that is accepted print a CHR\$(8) backspace to cancel the original space. A further enhancement alters the BIOS printer timeout values stored beginning at address &H478. The combined demonstration and function that follows performs this service using CALL INTERRUPT to circumvent BASIC's normal error handling routine.

```

DEFINT A-Z
DECLARE SUB INTERRUPT (IntNo, InRegs AS ANY, OutRegs AS ANY)
DECLARE FUNCTION LPTReady% (LPTNumber)

'$INCLUDE: 'REGTYPE.BI'

LPTNumber = 1

IF LPTReady%(LPTNumber) THEN
  PRINT "The printer is on-line and ready to go."
ELSE
  PRINT "Sorry, the printer is not available."
END IF
END

FUNCTION LPTReady% (LPTNumber) STATIC
  DIM Regs AS RegType
  LPTReady% = 0
  'for CALL INTERRUPT
  'assume not ready

  Address = &H477 + LPTNumber
  'LPT timeout address
  DEF SEG = 0
  'access segment zero
  OldValue = PEEK(Address)
  'save current setting
  POKE Address, 1
  '1 retry

  Regs.AX = 32
  'first print a space
  Regs.DX = LPTNumber - 1
  'convert to 0-based
  CALL INTERRUPT(&H17, Regs, Regs)
  'print the space

  Result = (Regs.AX \ 256) OR 128
  'get AH, ignore busy
  Result = Result AND 191
  'and acknowledge
  IF Result = 144 THEN
    'it worked!
    Regs.AX = 8
    'print a backspace
    CALL INTERRUPT(&H17, Regs, Regs)
    ' to undo CHR$(32)
    LPTReady% = -1
    'return success
  END IF

  POKE Address, OldValue
  'restore original
  ' timeout value
END FUNCTION

```

There are several important points worth mentioning here. First, you must never use zero for the printer timeout value, or the timeout will be a lot longer than you anticipated. A value of zero tells the BIOS to continue trying indefinitely, and is equivalent to using the DOS MODE LPT1: command with the ",p" argument.

Another point is that you should not use this function many times in a row, without ever printing anything. All modern printers provide a buffer, which accepts characters as fast as the computer can send them. If the buffer fills with spaces and backspaces before any printable characters are sent, it may be impossible to clear the buffer. Therefore, you should perform the printer test only once or twice, just before you actually need to begin printing.

EGA and VGA Data

The seven bytes starting at address &H484 hold information about an installed EGA or VGA display adapter. This data should not be relied upon until you have determined that the adapter is in fact an EGA or VGA. The Monitor function shown in Chapter 6 can be used for this.

The first byte holds the number of rows currently displayed on the screen. The next word at addresses &H485 and &H486 tells how high each character is in scan lines. For a normal 80 by 25 line screen this value will be 16. After using WIDTH , 43 or WIDTH , 50 the height of each character is 8 scan lines. Notice that this value also includes the spacing between each line. Curiously, two bytes are set aside to hold this value, even though it is extremely unlikely that any video mode would ever require a number larger than 255.

The only other information you are likely to find useful in this data area is the amount of installed memory on the EGA or VGA adapter card. Bits 5 and 6 at address &H487 hold the number of 64K banks, and the code that follows shows how to turn this into a meaningful number:

```
DEF SEG = 0           'look in segment zero
Byte = PEEK(&H487)    'get the byte
Byte = Byte AND 96    'keep what we need (96 = 1100000b)
Byte = Byte \ 32      'shift the bits right five places
Byte = (Byte + 1) * 64 'add 1 because 0 means 64K
PRINT "This EGA/VGA adapter has"; Byte; "K memory"
```

After reading the EGA Features byte (listed earlier in Figure 10-1), the statement `Byte = Byte AND 96` masks off all of the bits that are irrelevant. `Byte` is then divided by 32 to slide those bits into the lowest position. The number that results is coded such that 0 means 64K of installed video memory, 1 means 128K, 2 means 192K (which is never really possible), and 3 indicates 256K. Because this value is zero-based, 1 is added to `Byte` before multiplying by 64.

Miscellaneous Data

The 16-byte data area that begins at address &H4F0 is called the *inter-application communications area*, and it is available for any arbitrary use by a program. One possibility is for passing just a few parameters between separate programs, instead of having to use COMMON and CHAIN. Although this data area has been available since the original IBM PC was introduced, there is a risk involved with using it because it is possible that another program or TSR has stored information there. Chapter 9 described using the last 96 bytes in the display adapter's memory, which is both a larger buffer and is probably safer to use.

The byte at address &H500 is used as a flag by the BIOS Print Screen service to detect when it is busy. When you press Shift-PrtSc, the BIOS routine that handles that key sets this byte to a value of 1 before beginning to print the screen. This way if you press Shift-PrtSc again before it has finished printing, the second request can be ignored. When the printing has completed the flag is then reset to zero.

You can set this flag manually to disable the action of the PrtSc key, and then reenable it again later:

```
DEF SEG = 0
POKE &H500, 1
.
.
POKE &H500, 0
```

In fact, you must be sure to re-enable PrtSc before ending your program if you have disabled it. Otherwise, that key will be disabled until the PC is rebooted.

The last low memory address I'll describe is also one of the most potentially useful. For systems that have only one diskette drive, the byte at address &H504 tells which drive (A or B) is currently active. In this case, that drive serves as both A and B. Most PC users are familiar with DOS' infamous "Insert disk for drive B" message. This message is displayed whenever you attempt to access one of the logical drives while the other is currently active.

The problem is that this message will ruin an otherwise attractive screen design, and you have no control over where or if the message is displayed. Fortunately, you can determine if only one drive is available, and also which is currently active. Even better, you can set this byte to reflect either drive, and thus avoid the intervention by DOS.

If the byte at address &H504 is currently zero, then drive A is active; a value of 1 indicates drive B. The short complete program that follows shows how to detect which drive is current.

```
DEF SEG = 0
Floppies% = (PEEK(&H410) AND 192) \ 64 + 1
PRINT "This PC has"; Floppies%; "floppy disk drive(s)."
```

```
IF Floppies% = 1 THEN
  PRINT "The disk is now acting as drive ";
  CurDrive% = PEEK(&H504)
  IF CurDrive% THEN
    PRINT "B"
  ELSE
    PRINT "A"
```

```
END IF
END IF
```

To change from drive A to B simply use `POKE &H504, 1`, assuming that the current DEF SEG value is already zero. Likewise, to change from B to A you will use `POKE &H504, 0`. Of course, you must also prompt the user to change disks as DOS would. But at least you can control how the prompt message is displayed. If you do switch drives behind DOS' back, it is up to you to prompt the user to exchange disks as necessary, and also to ensure that files are updated and closed correctly before each switch.

Input/Output Ports

Besides the low memory addresses that are reserved for BIOS and DOS uses, every PC also has a collection of Input/Output (I/O) ports. Like memory, ports are addressed by number, and data may be read from or to written to them. In truth, some ports are write-only, others may only be read, and still others can be read and written.

Where conventional memory is often used by the operating system to hold flags, status words, and other values, ports are used to actually control the hardware. For example, port number `&H3F2` controls the diskette drive motors, and appropriate OUT commands to that port can turn the motor for any drive on or off.

For the most part, you should not experiment with the ports unless you know what they are for, and which values are appropriate. As an example, it is possible to damage your monitor by sending incorrect values through the display adapter controller ports. Two useful ports I will describe here control the PC's speaker and the keyboard.

Although BASIC offers the SOUND and PLAY statements, using them can quickly increase the size of a program. Both of these commands can operate in the background, thereby continuing to produce sound after they return to your program. As you can imagine, this requires a lot of code to implement. An informal test showed that adding a single SOUND statement increased the program size by more than 11K. Therefore, if you do not need the ability to have tones play in the background, the combination demonstration and subprogram that follows can be used in place of SOUND. Besides avoiding the code to plays tones as a background task, this routine also avoids SOUND's inclusion of floating point math.

```
DEFINT A-Z
DECLARE SUB BSound (Frequency, Duration)

CLS

PRINT "Sweep sound"
FOR X = 1 TO 10
    READ Frequency
    CALL BSound(Frequency, 1)
NEXT
```

```

DATA 100, 200, 300, 400, 600, 900, 1200, 1500, 1800, 2100

PRINT "Press a key for more..."
WHILE INKEY$ = "": WEND

PRINT "Telephone"
FOR X = 1 TO 10
  CALL BSound(600, 1)
  CALL BSound(800, 1)
NEXT

PRINT "Press a key for more..."
WHILE INKEY$ = "": WEND

PRINT "Siren"
FOR X = 1 TO 2
  FOR Y = 600 TO 1000 STEP 15
    CALL BSound(Y, -1)      'negative values leave
  NEXT                    ' the speaker turned on
  FOR Y = 1000 TO 600 STEP -15
    CALL BSound(Y, -1)
  NEXT
NEXT
CALL BSound(600, 1)      'force the speaker off

SUB BSound (Frequency, Duration) STATIC
  IF Frequency < 33 THEN EXIT SUB

  IF NOT BeenHere THEN      'do this only once for a
    BeenHere = -1          ' smoother sound effect
    OUT &H43, 182          'initialize speaker port
  END IF

  Period = 1190000 \ Frequency 'convert to period
  OUT &H42, Period AND &HFF 'send it as two bytes
  OUT &H42, Period \ 256    ' in succession

  Speaker = INP(&H61)      'read Timer port B
  Speaker = Speaker OR 3    'set the speaker bits on
  OUT &H61, Speaker

  DEF SEG = 0
  FOR X = 1 TO ABS(Duration) 'for each tick specified
    ThisTime = PEEK(&H46C) ' count changes again
    DO 'wait until the timer
      LOOP WHILE ThisTime = PEEK(&H46C)
  NEXT

  IF Duration > 0 THEN      'turn off if requested
    Speaker = INP(&H61)    'read Timer port B
    Speaker = Speaker AND &HFC 'set the speaker bits off
    OUT &H61, Speaker
  END IF
END SUB

```

The BSound routine accepts the same frequency and duration arguments as BASIC's SOUND statement. Each time it is called it calculates the appropriate period based on the incoming frequency, which is what the timer ports expect. Period is the reciprocal of frequency. Here, the period is related to

the PC's clock frequency of 1,190,000 Hz. BSound then turns on the speaker, waits in a loop for the specified duration, and finally turns off the speaker before returning.

Two extra steps are required to create a smooth effect when BSound is called rapidly in succession. One is that the speaker port is initialized only once, the very first time BSound is called. The other step lets you optionally leave the speaker turned on when BSound returns, to avoid the choppiness that otherwise results with sounds like the siren effect. To tell BSound to leave the speaker on, use an equivalent negative value for the Duration parameter. Just be sure to call BSound once again with a positive duration value, or use the same set of INP and OUT statements that BSound uses to turn the speaker off. This is shown in the last demonstration that creates a siren sound.

Keyboard Ports

There are several ports associated with the keyboard, and one is of particular interest. The enhanced keyboards that come with AT-class and later computers allow you to control how quickly keystrokes are repeated automatically. There are actually two values: one sets the initial delay before keys begin to repeat, and the other establishes the repeat rate. By sending the correct values through the keyboard port, you can control the keyboard's *typematic* response. The complete program that follows shows how to do this, and Table 10-3 shows how the delay and repeat rate values are determined.

```
OUT &H60, &HF3          'get the keyboard's attention
FOR D& = 1 TO 100: NEXT 'brief delay to give the hardware time to settle
Value = 7              '1/4 second initial delay, 16 CPS
OUT &H60, Value
```

Initial Delay	0.25	0.50	0.75	1.00
30 characters per second	00	20	40	60
16 characters per second	07	27	47	67
8 characters per second	0F	2F	4F	6F
4 characters per second	17	37	57	77
2 characters per second	1F	3F	5F	7F

NOTE: All values are shown in Hexadecimal.

Table 10-3: AT-style keyboard delay and repeat rates

Table 10-3 shows only some of the possible values that can be used. However, you can interpolate additional values for delay times and repeat rates between those shown.

Summary

This chapter explained what the BIOS low memory data area is, and also discussed many of the addresses that are useful to application programs. A number of practical examples were given, including useful PEEK and POKE replacements that operate on data a word, rather than a byte, at a time. A simple binary conversion function was shown, to help you determine the correct values to use with AND and OR.

You learned how to exchange serial and parallel port addresses, and how to access communications ports 3 and 4 which BASIC normally does not allow. Exchanging printer ports lets you access any printer as LPT1, perhaps to avoid having to rewrite a large program that relies on existing LPRINT statements. Other useful printer data that can be accessed is the BIOS timeout value, and a routine was shown for testing the printer status without the usual delay.

The equipment list word was described in detail, showing how to determine the number of diskette drives and other peripherals that are installed. Another useful routine showed how to determine if drive A or B is active on a one-floppy system, and also how to change the current status of that drive. The various keyboard status bits were also described, and code fragments showed how to read and set the current state.

Finally, you learned how the hardware ports are read and written using INP and OUT commands. One example produced sound with much less generated code than BASIC's SOUND, and another showed how to alter the typematic rate on enhanced (AT) keyboards.

The next chapter explores using CALL Interrupt in great detail, using many examples that show how to access DOS and BIOS system services.

11

Accessing DOS and BIOS Services

BASIC is arguably the most capable of all the popular high-level languages available for the PC. However, one area where all PC languages are weak is when accessing DOS and BIOS system interrupts. Previous chapters included subroutines and functions that access DOS interrupt services using `CALL Interrupt`, but in most cases with little explanation. This chapter explains what interrupts are, how they are accessed, and how they return information to your program.

Only assembly language—the native language of the processor in every PC—can directly access interrupts. Assembly language programmers use the `Int` instruction, which transfers control to an *interrupt service routine*. An `Int` instruction is nearly identical to a conventional `CALL` statement, except a slightly different mechanism within the computer's hardware is used to implement it.

BASIC lets you access system interrupts by providing a pair of assembly language interface routines called `Interrupt` and `InterruptX`. These routines accept the interrupt number and other parameters the interrupt requires, and they then perform the actual interrupt call. `InterruptX` is similar to `Interrupt`; the only real difference is that it lets you access two additional CPU registers.

What is an Interrupt?

The IBM PC family of personal computers supports two types of interrupts: hardware and software. A *hardware interrupt* is invoked by an external device or event, such as pressing a key on the keyboard. When this happens, a signal is sent from the keyboard hardware to the PC's microprocessor telling it to stop what it's currently doing and instead call one of the routines in the PC's BIOS.

For example, while your PC is currently copying a group of files you may type `DIR` simultaneously, to display the results when the copying has finished. Even though DOS is reading and writing the files, you interrupt those operations for a few microseconds each time a key is pressed. The BIOS routine that handles the keyboard interrupt is responsible for placing the keystrokes into the PC's 15-character keyboard buffer. Then when DOS has finished copying your files, the `DIR` command will already be there. Because there is a direct physical connection between the keyboard circuitry and the PC's microprocessor, you are able to interrupt whatever else is happening at the time.

A *software interrupt*, on the other hand, doesn't really interrupt anything. Rather, it is a form of `CALL` command that an assembly language program may issue. Just like the `CALL` command in BASIC that transfers control to a subroutine, a software interrupt is used in an assembly language program to access DOS and BIOS services. Although assembly language programs may use a `CALL` statement to invoke a subroutine, an interrupt instruction is needed to access the operating system routines.

When a program issues a subroutine call, the address of that subroutine must be known, so the processor will be able to jump to the code there. With most programs, subroutine addresses are determined and assigned by LINK.EXE when it combines the various portions of your program into a single executable file. But this method can't be used with the DOS and BIOS routines, because their addresses are not known ahead of time. For example, if you compile a BASIC program on an IBM PC, it must also be able to be run on, say, a Tandy 1000 using a different version of DOS. Of course, it is impossible for LINK to know where the DOS and BIOS routines are located on the Tandy computer.

To solve this problem and allow a program to call a routine whose address is not known, a list of addresses is stored in a known place in low memory. This place is called the *interrupt vector table*. The first 1,024 bytes in every PC contains a table of addresses for all 256 possible interrupts. Each table entry requires two words (four bytes): one word is used to hold the routine's segment, and the other holds its address within that segment. Whenever an assembly language program issues an interrupt instruction, the PC's processor automatically fetches the segment and address from this table, and then calls that address. Thus, any program may access any interrupt routine, without having to know where in memory the routine actually resides. The first four bytes in the interrupt vector table hold the address for Interrupt 0, the next four show where Interrupt 1 is, and so forth.

DOS and BIOS services are specified by interrupt number, and most interrupt routines also expect a *service number*. Nearly all of the DOS services you will find useful are accessed through Interrupt &H21, with the desired service number specified in the AH register. In many cases, information is also returned in the CPU registers. For instance, the DOS service that returns the current default disk drive is specified by placing the value &H19 in the AH register. When the interrupt has finished, the current drive number is returned in the AL register. Registers will be described in the section that follows. As with the low memory addresses discussed in Chapter 10, the DOS and BIOS interrupt numbers use Hexadecimal numbering by convention.

There are also several BIOS interrupts you will find useful, and these include video interrupt &H10, printer interrupt &H17, Print Screen interrupt 5, and the two equipment interrupts &H11 and &H12. There are other BIOS and DOS interrupts, but those are mostly useful when accessed from assembly language. For example, there is little need to call keyboard interrupt &H16 to read a key, since INKEY\$ already does this. Likewise, you are unlikely to find disk interrupt &H13 very interesting, although it is used when performing copy protection and other low-level direct disk accesses. But unless you know what you are doing, it is possible—even likely—to trash your hard disk in the process of experimenting with this disk interrupt.

I won't attempt to provide all of the information you need to access every possible DOS and BIOS service here. Indeed, a complete discussion would fill several books. Two excellent books that I recommend are *Peter Norton's Programmer's Guide to the IBM PC* (1988), and *Advanced MS-DOS*, by Ray Duncan (1988). Both of these books are published by Microsoft Press, and can be found in most book stores. These books list every DOS and BIOS interrupt service, and show which registers are used to exchange information with each interrupt service.

Also, once you have read and understood the information in this chapter you should go back to some of the examples presented in earlier chapters. In particular, Chapter 6 shows how to access DOS Interrupt &H21 to read file names, and Chapter 7 includes routines that access Interrupt &H2F to see if a network is running on the host PC and if so which one.

Registers

Microprocessors in the Intel 8086 family contain a set of built-in integer variables called *registers*. Each register can hold a single word (two bytes), which nicely corresponds to the size of a BASIC integer variable. Because these registers are contained within the microprocessor itself, they can be accessed by the CPU very quickly—much faster than variables which are stored in memory.

The 8086 and 8088 microprocessors contain a total of fourteen registers.

Newer CPUs contain more registers, but they are not accessible via CALL Interrupt nor are they useful to a BASIC program.

Some of these registers are intended for a specific use, while others may be used as general purpose variables. For example, the CS and DS registers contain the current code and data segments respectively, while the CX register is often used as a counter in an assembly language FOR/NEXT loop. I'm not going to pursue a lengthy discussion of microprocessor theory here though, because it's not really necessary if you simply want to access a few system interrupts. Rather, I will focus on how to set up and invoke the various interrupt services, and interpret the results they return. Assembly language and CPU registers will be discussed more fully in Chapter 12.

Both Interrupt and InterruptX (Interrupt Extended) require a TYPE variable with components that mirror each of the processor's registers. Table 11-1 lists all of the 8086 registers that are accessible from BASIC, showing which are available with each of the interrupt routines.

InterruptX	Interrupt
AX	AX
BS	BX
CX	CX
DX	DX
BP	BP
SI	SI

DI	DI
Flags	Flags
DS	
ES	

Table 11-1: The registers accessible from BASIC through Interrupt and InterruptX.

When you call the either Interrupt routine, the values in a TYPE variable are copied into the CPU's registers, the interrupt is performed, and then the results returned in each register are copied back into a TYPE variable again. All of the CALL Interrupt examples Microsoft shows use two TYPE variables called InRegs and OutRegs. However, you can also use the same TYPE variable to both send and receive the register values. In fact, using a single TYPE variable will save a few bytes of DGROUP memory. Therefore, the remaining examples that use CALL Interrupt use a single TYPE variable.

One important issue that needs to be addressed before we can proceed is how the CPU registers are accessed. I stated earlier that there are fourteen such registers, and each is the same size as an integer variable: 2 bytes. While this is certainly true, there is more to the story. Four of the registers—AX, BX, CX, and DX—can also be treated as being two separate one-byte registers.

Each register half uses the designator "H" or "L" to mean High or Low. For example, the high-byte portion of AX is called AH, and the low-byte portion of CX is CL. When considered as a composite register, the two halves form a single integer word. Figure 11-1 shows how the AX register is constructed, with each half contributing to the total combined value.

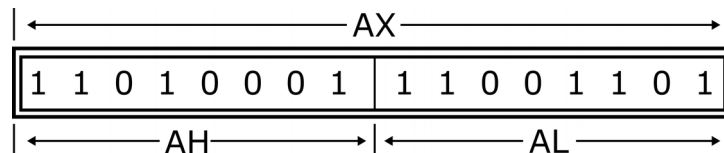


Figure 11-1: How a single word-sized register may also be treated as two byte-sized registers.

In an assembly language program it is simple to access each register half separately. However, BASIC does not offer a byte-sized variable type to use within the TYPE declaration. Therefore, a slight amount of math is required to get at each half separately. Although a fixed-length string with a length of one character could be used, the added overhead BASIC imposes to access a string as a number reduces the usefulness of that approach.

Using Hexadecimal notation and multiplication simplifies access to each register half when it is being assigned, and integer division and BASIC's AND operator lets you separate the two halves when reading them. That is, you can assign the value &H12 to the upper byte in AH and the value &H34 to the lower byte in AL at one time, like this:

```
Registers.AX = &H1234
```

In many cases it is necessary to assign only AH, which can be done like this:

```
Registers.AX = &H0600
```

Here, the value 6 is placed into AH, and 0 is assigned to AL. Since many of the DOS and BIOS services ignore what is in AL, assigning a value of zero is the simplest and most effective solution. Again, using Hexadecimal notation lets you clearly define what is in each register half, because the first two digits represent the upper portion, and the second two represent the lower byte.

When both the upper and lower bytes are important, you can use multiplication to assign them. By definition, any byte value in the high portion of a register is 256 times greater than it would be in the lower part. Thus, to assign the variable Low% to AL and High% to AH is as simple as this:

```
Registers.AX = Low% + (256 * High%)
```

In practice the parentheses are not really necessary because multiplication is always performed before addition. But I included them here for clarity.

When an interrupt routine returns information in one of the combination registers, you may easily isolate the high and low portions as follows:

```
Low% = Registers.DX AND 255  
High% = Registers.DX \ 256
```

Some examples you may have seen use MOD to extract the lower byte, and that will also work:

```
Low% = Registers.DX MOD 256
```

Although MOD and AND cause BASIC to generate the same amount of assembly language code (three bytes), I generally prefer using AND because that instruction is somewhat faster on the older 8088 processors.

Accessing the BIOS

The simplest BIOS interrupt to call is the Print Screen interrupt, Interrupt 5. No parameters are required by this interrupt, and no values are returned when it finishes. But since the Interrupt routine expects the TYPE variable to be present and copies data to it, you must still dimension it in your program.

Because Interrupt and InterruptX are external subroutines as opposed to built-in commands, you will need to load the Quick Library containing these routines. QuickBASIC comes with the file QB.QLB; BASIC PDS provides the same routines in a library named QBX.QLB. [And in VB/DOS this file is

called VBDOS.QLB.] You must of course use whichever is appropriate for your version of BASIC. To start QuickBASIC and load the Quick Library that contains these routines use the /L switch like this:

```
qb /l
```

Normally, the name of a Quick Library must be given after the /L switch. However, QB and QBX know that /L by itself means to load the default QB.QLB or QBX.QLB Quick Library.

The following complete program prints a simple pattern on the screen, and then sends it to the printer designated as LPT1: as if the PrtSc key had been pressed.

```
DEFINT A-Z
TYPE RegType
  AX AS INTEGER
  BX AS INTEGER
  CX AS INTEGER
  DX AS INTEGER
  BP AS INTEGER
  SI AS INTEGER
  DI AS INTEGER
  Flags AS INTEGER
END TYPE
DIM Registers AS RegType

CLS
FOR X% = 1 TO 24
  PRINT STRING$(80, X% + 64);
NEXT
CALL Interrupt(5, Registers, Registers)
```

Although the Registers TYPE definition is shown here, the remaining examples in this chapter will instead specify the REGTYPE.BI include file that contains this code. QuickBASIC includes a similar include file called QB.BI, and BASIC PDS uses the name QBX.BI for the same file.

```
I created REGTYPE.BI so all of the programs in this
book will run as is with any version of BASIC. But the
BASIC-supplied versions also include DECLARE statements
for the Interrupt routines, where my REGTYPE.BI file
does not. Since all of these programs use the CALL
keyword, a declaration is not strictly necessary.
```

The BIOS Video Interrupt

The next example shows how to call BIOS video interrupt &H10 to clear just a portion of the display screen. It is designed as a combination demonstration and subprogram, so you can extract just the subprogram and add it to programs of your own.

```
DEFINT A-Z
DECLARE SUB ClearScreen (ULRow, ULCol, LRRow, LRCol, Colr)
```

```

'$INCLUDE: 'REGTYPE.BI'
DIM SHARED Registers AS RegType

CLS
FG = 7: BG = 1           'set the foreground and background colors COLOR FG, BG

FOR X% = 1 TO 24
  PRINT STRING$(80, X% + 64);
NEXT

Colr = FG + 16 * BG      'use the same colors for clearing CALL ClearScreen(5,
10, 20, 70, Colr)

SUB ClearScreen (ULRow, ULCol, LRRow, LRCol, Colr) STATIC
  Registers.AX = &H600
  Registers.BX = Colr * 256
  Registers.CX = (ULCol - 1) + (256 * (ULRow - 1))
  Registers.DX = (LRCol - 1) + (256 * (LRRow - 1))
  CALL Interrupt(&H10, Registers, Registers)
END SUB

```

There are two important benefits to using the BIOS for a routine such as this. One is of course the reduced amount of code that is needed, when compared to manually looping through memory using POKE to clear each character position. The second is the BIOS is responsible for determining the type of monitor installed, to select the correct video segment.

The demonstration portion of the program first clears the screen, and then creates a simple test pattern using a color of white on blue. Just before the call to ClearScreen, the correct Colr parameter is calculated based on the same foreground and background specified to BASIC. Where BASIC accepts separate foreground and background values, the BIOS requires a single composite color byte.

The simplified formula used in this example will accommodate normal colors, but does not support adding 16 to the foreground to specify a flashing color. This next formula shows how to derive a single color byte while also honouring flashing:

```
Colr = (FG AND 16) * 8 + ((BG AND 7) * 16) + (FG AND 15)
```

ClearScreen is then called telling it to clear a rectangular portion of the screen that lies within the boundary specified by an upper-left corner at location 5, 10 to the lower-right corner at location 20, 70. The color value calculated earlier is also passed, so the white on blue color will be maintained even after the screen is cleared.

Within ClearScreen, four of the CPU's registers are assigned to values needed by the BIOS video interrupt. The first statement specifies service 6 in AH, which tells the BIOS to scroll the screen. The number of rows to scroll is then placed into the AL register, which we've set to zero. This particular BIOS service recognizes zero as a special flag, which tells it to clear the screen rather than scroll it.

Service 6 also expects the color to use for clearing in the BH register. As I explained earlier, multiplying by 256 is equivalent to assigning just the higher portion of an integer, so the statement

`Registers.BX = Colr * 256` is equivalent to placing the one byte that is actually used by the `Colr` variable into `BH`.

The next two instructions take the upper left and lower right corner arguments, and place them into the appropriate registers. In this case, the upper left column is placed into `CL` and the upper left row in `CH`. Similarly, the lower right column goes into `DL` and the lower right row into `DH`. Even though BASIC considers screen rows and columns to be numbered beginning at 1, the BIOS routines assume these to be zero-based. Therefore, 1 is subtracted from the parameters before they are placed into each component of the `Registers` `TYPE` variable. Finally, BASIC's Interrupt routine is called specifying Interrupt number `&H10`.

Note that the same BIOS interrupt service can also be used to scroll a rectangular portion of the screen. Indeed, this is the primary purpose of service 6. To scroll a portion of the screen up a certain number of lines, you will place the number of lines into `AL`:

```
Registers.AX = NumLines + (6 * 256)
```

Scrolling the screen downward is also possible, using service 7 like this:

```
Registers.AX = NumLines + (7 * 256)
```

Also note that the `Registers` `TYPE` variable was dimensioned to be shared. This allows it to be accessed from all of the subprograms in a single program. If `Registers` is dimensioned in many different subprograms and functions, then a new instance will be created, with each stealing 20 bytes of `DGROUP` memory. Beware, however, that this memory savings has the potential drawback of introducing subtle bugs due to the same variable being used by different services. Whatever register values remain after one use of `CALL` Interrupt will still be present the next time, unless new values are explicitly assigned. But that is rarely a problem, since you will generally assign all of the registers that a given interrupt needs just before calling that interrupt.

Although this short example simply clears or scrolls a portion of the display screen, it provides a foundation for nearly anything else you may need to do using `CALL` Interrupt. The DOS interrupt examples that follow will build on this foundation, and show how to access a wealth of useful services that are not otherwise possible using BASIC alone.

Accessing DOS Interrupts

As with the BIOS video interrupt services, DOS interrupt `&H21` expects a service number to be given in the `AH` register. Many DOS services require additional information in other registers as well, including integer values and the segments and addresses of variables.

The DOS services that accept or return a string (such as a file or directory name) require the address of the string, to know where it is located. For example, the DOS service that changes the current directory

is called with AH set to &H3B, and DS:DX holding the address of a string that contains the name of the directory to change to.

Likewise, to obtain the current directory you would load AH with the value &H47, and DS:SI with the address of a string that will receive the current directory's name. It is essential that this string already be initialized to a sufficient length before calling DOS. Otherwise, the returned directory name will likely overwrite other existing data. And if that data happens to be a BASIC string descriptor or back pointer you will likely crash the program and possibly even have to reboot the PC. When a string is sent as a parameter to a DOS routine, it must be terminated with a CHR\$(0), so DOS can tell where it ends. Likewise, when DOS returns a string to your program such as the current directory, it indicates the end with a CHR\$(0). Therefore, it is up to your program to manually append a CHR\$(0) to any file or directory names you pass to DOS. And when receiving a string from DOS, you must use INSTR to locate the CHR\$(0) that marks the end, and keep only what precedes that character.

I will start with some simple examples that access DOS Interrupt &H21, and proceed to more complex routines that pass and receive string data.

Accessing the Default Drive

The first DOS example shows how to determine the current default drive, and it is designed as a DEF FN-style function. A function is a natural way to design a routine that returns information, as opposed to a called subprogram. Further, using a DEF FN-style function reduces the amount of code that BASIC generates, and also reduces the code needed each time the function is invoked.

```
DEFINT A-Z

'$INCLUDE: 'REGTYPE.BI'
DIM Registers AS RegType

DEF FnGetDrive%
  Registers.AX = &H1900
  CALL Interrupt(&H21, Registers, Registers)
  FnGetDrive% = (Registers.AX AND &HFF) + 65
END DEF

PRINT "The current default drive is "; CHR$(FnGetDrive%)
```

Here, service number &H19 is assigned to the AH portion of AX prior to calling Interrupt &H21, and the value that DOS returns in AL indicates the current drive. For this service DOS uses 0 to indicate drive A, 1 for drive B, and so forth. Therefore, you use AND with the value &HFF (255) to keep just the low portion in AX. Once the DOS drive number has been isolated, the program adds 65 to adjust that to the equivalent ASCII character value.

Setting a new default drive is just as easy as obtaining the current drive. Although BASIC PDS provides the CHDRIVE command to set a new drive as the current default, QuickBASIC does not. The ChDrive subprogram that follows affords the same functionality to QuickBASIC users, and it accepts a single letter to indicate which drive is to be made the new current default.

```

DEFINT A-Z
DECLARE SUB ChDrive (Drive$)

'$INCLUDE: 'REGTYPE.BI'
DIM SHARED Registers AS RegType

INPUT "Enter the drive to make current: ", NewDrive$
CALL ChDrive(NewDrive$)

SUB ChDrive (Drive$) STATIC
    Registers.AX = &HE00
    Registers.DX = ASC(UCASE$(Drive$)) - 65
    CALL Interrupt(&H21, Registers, Registers)
END SUB

```

Now that you know how to set and get the current default drive, you can combine the two and create a function that tells if a given drive letter is valid. Many DOS services return the success or failure of an operation using the CPU's Carry flag. However, the service that sets a new drive is a notable exception. Therefore, to determine if a given drive letter is in fact valid requires more than simply trying to set the new drive, and then seeing if an error resulted.

The only way to tell if a request to change the current drive was accepted is to make another call to get the current drive, thereby seeing if the original request took effect. The program that follows accepts a drive letter as a string, and returns True or False (-1 or 0) to indicate whether or not the drive is valid.

```

DEFINT A-Z
DECLARE SUB ChDrive (Drive$)

'$INCLUDE: 'REGTYPE.BI'

DIM SHARED Registers AS RegType

DEF FnGetDrive%
    Registers.AX = &H1900
    CALL Interrupt(&H21, Registers, Registers)
    FnGetDrive% = (Registers.AX AND &HFF) + 65
END DEF

DEF FnDriveValid% (TestDrive$)
    STATIC Current
    Current = FnGetDrive%
    FnDriveValid% = 0
    CALL ChDrive(TestDrive$)
    IF ASC(UCASE$(TestDrive$)) = FnGetDrive% THEN
        FnDriveValid% = -1
    END IF
    CALL ChDrive(CHR$(Current))
END DEF

INPUT "Enter the drive to test for validity: ", Drive$
IF FnDriveValid%(Drive$) THEN
    PRINT Drive$; " is a valid drive."
ELSE
    PRINT "Sorry, drive "; Drive$; " is not valid."
END IF

```

```

SUB ChDrive (Drive$) STATIC
  Registers.AX = &HE00
  Registers.DX = ASC(UCASE$(Drive$)) - 65
  CALL Interrupt(&H21, Registers, Registers)
END SUB

```

The strategy used here is to first save the current default drive, and then set a new drive on a trial basis. If the current drive is the one that was just set, then the specified drive was indeed valid. In either case, the original drive must be restored.

Determining if a File Exists

Both of the DOS services we have considered so far use integer arguments to indicate the new drive, or which drive is the current default. The next example shows how to pass a BASIC string to a DOS service, which is somewhat more complicated. The situation is made worse by the far strings feature available in BASIC PDS. Therefore, be sure to observe the comment that shows how to replace SSEG with VARSEG for use with QuickBASIC.

Chapter 6 showed an admittedly clunky way to determine if a file is present. The example given there attempted to open the specified file for random access, and then used LOF to see if the file had a length of zero. The problem with that method—besides requiring a lot of unnecessary DOS activity—is that it reports a file with a perfectly legal length of zero as not being present, and then deletes it.

The FnFileExist function that follows is intended for use with BASIC PDS, and comments show how to change it for use with QuickBASIC. Please understand that PDS doesn't really need a File Exist function, since DIR\$ can be used for that purpose. The statement IF LEN(DIR\$(FileSpec\$)) THEN will quickly tell if a file is present. However, the point is to show how strings are passed to DOS, and for that purpose this example serves quite nicely.

```

DEFINT A-Z
'$INCLUDE: 'REGTYPE.BI'

DIM Registers AS RegType

TYPE DTA
  Reserved AS STRING * 21      'used by DOS services
  Attribute AS STRING * 1     'reserved for use by DOS
  FileTime AS STRING * 2      'the file's attribute
  FileDate AS STRING * 2      'the file's time
  FileSize AS LONG            'the file's date
  FileName AS STRING * 13     'the file's size
                              'the file's name
END TYPE
DIM DTADData AS DTA

DEF FnFileExist% (Spec$)
  FnFileExist% = -1           'assume the file exists

  Registers.DX = VARPTR(DTADData) 'set a new DOS DTA
  Registers.DS = VARSEG(DTADData)
  Registers.AX = &H1A00
  CALL InterruptX(&H21, Registers, Registers)

```

```

Spec$ = Spec$ + CHR$(0)      'DOS needs an ASCIIZ string
Registers.AX = &H4E00        'find file name service
Registers.CX = 39           'attribute for any file
Registers.DX = SADD(Spec$)   'show where the spec is
Registers.DS = SSEG(Spec$)   'use this with BASIC PDS
'Registers.DS = VARSEG(Spec$) 'use this with QuickBASIC

CALL InterruptX(&H21, Registers, Registers)
IF Registers.Flags AND 1 THEN FnFileExist% = 0
END DEF

INPUT "Enter a file name or specification: ", FileSpec$
IF FnFileExist%(FileSpec$) THEN
  PRINT FileSpec$; " does exist"
ELSE
  PRINT "Sorry, no files match "; FileSpec$
END IF

```

FnFileExist calls upon the DOS Find First service that searches a directory and attempts to locate the first file that matches a given specification template. Therefore, besides being able to see if ACCOUNTS.DAT or F:\UTILS\NU.EXE exist, you can also use the DOS wild cards. For example, given C:\QB45*.BAS, FnFileExist will report if any files with a .BAS extension are in the \QB45 directory of drive C.

As part of its directory searching mechanism, DOS requires a block of memory known as a Disk Transfer Area, or DTA for short. If a matching file name is found, DOS stores important information about the file there, where your program can read it. As you can see by examining the DTAType structure, this includes the file's name and extension, the date and time it was last written to, its current size, and attribute. The 21-byte string at the beginning identified as Reserved holds sector numbers and other information, and is used by DOS for subsequent searches. This function doesn't use any of the information in the DTA; however, it must still be defined for use by DOS.

You will notice that FnFileExist uses the InterruptX routine rather than Interrupt, and this is to provide support for use with BASIC PDS far strings. Two of the CPU's registers are used to hold the DS and ES data segment registers. When Interrupt is called, it simply leaves whatever is currently in DS and ES and then calls the interrupt. InterruptX, on the other hand, loads DS and ES from those components of the Registers TYPE variable, and those are the values the interrupt itself receives. Were FnFileExist limited to working with QuickBASIC—where all strings are in the DS segment—Interrupt would be sufficient and the added complication of using either VARSEG or SSEG could be avoided.

Note that InterruptX can also be told to use the current value of DS for both DS and ES, when the calling program doesn't need or want to change them. This is specified by placing a value of -1 into either or both portions of the Registers TYPE variable. For example, the statement Registers.DS = -1 tells InterruptX not to assign DS before performing the interrupt. Otherwise, if Registers.DS were not assigned, DS would receive the value 0 which is incorrect for DOS services that receive a variable's address. In a similar manner, Registers.ES = -1 tells InterruptX to set ES to the current value of DS.

The Carry Flag

The last item to note in this function is how the Carry flag is tested. As I mentioned earlier, many DOS services indicate the success or failure of an operation by either clearing or setting the CPU's Carry flag. This flag is held in one bit in the Flags register, and its primary purpose is to assist multi-word arithmetic in assembly language programs. But because the 80x86 provides single instructions that easily set and test this flag, the designers of DOS decided to use it as an error indicator.

The Carry flag is stored in the lowest bit of the Flags register, and can therefore be tested using the AND instruction with a value of 1. If that bit is set, the result of the AND test will be one; otherwise it will be zero. Thus, the statement `IF Registers.Flags AND 1 THEN` will be true if the Carry flag is set, which indicates an error. In the case of DOS' Find First function this is not really an error in the strictest sense. But there is no need here to distinguish between, say, an invalid path name and the lack of any matching files. Either a match was found or it wasn't.

Improving On Interrupt

Recall that Chapter 8 introduced the DOSInt routine which serves as a small-code replacement for BASIC's InterruptX routine. Although the reduction in code size gained by using DOSInt versus Interrupt or InterruptX is not dramatic, it can save several hundred bytes in a program that calls it many times. DOSInt is also somewhat easier to set up and use, because it requires only a single Registers argument.

Of course, DOSInt is meant only for use with DOS Interrupt &H21, and it will not work with any other DOS or BIOS interrupt services. Because of the savings that DOSInt affords, the remaining DOS examples in this chapter will use DOSInt instead of Interrupt or InterruptX. Like InterruptX, DOSInt lets you access the DS and ES registers, and it also recognizes an incoming value of -1 to specify the current contents of DS.

Obtaining the Current Directory

Where FnFileExist shows how to pass a BASIC string to a DOS interrupt service, the FnGetDir function following shows how to receive a string from DOS. Again, BASIC PDS users have the CURDIR\$ function which reports the current directory, but most QuickBASIC programmers will find this function invaluable.

```
DEFINT A-Z
'$INCLUDE: 'REGTYPE.BI'

DIM Registers AS RegType

DEF FnGetDir$ (Drive$)
    STATIC Temp$, Drive, Zero      'local variables

    IF LEN(Drive$) THEN           'did they pass a drive?
```

```

    Drive = ASC(UCASE$(Drive$)) - 64
ELSE
    Drive = 0
END IF

Temp$ = SPACE$(65)           'DOS stores the name here

Registers.AX = &H4700        'get directory service
Registers.DX = Drive        'the drive goes in DL
Registers.SI = SADD(Temp$)  'show DOS where Temp$ is
Registers.DS = SSEG(Temp$)  'use this with BASIC PDS
'Registers.DS = -1          'use this with QuickBASIC

CALL DOSInt(Registers)      'call DOS

IF Registers.Flags AND 1 THEN 'must be an invalid drive
    FnGetDir$ = ""
ELSE
    Zero = INSTR(Temp$, CHR$(0)) 'find the zero byte
    FnGetDir$ = "\" + LEFT$(Temp$, Zero)
END IF
END DEF

PRINT "Which drive? ";
DO
    Drive$ = INKEY$
LOOP UNTIL LEN(Drive$)
PRINT

Cur$ = FnGetDir$(Drive$)
IF LEN(Cur$) THEN
    PRINT "The current directory is ";
    PRINT Drive$; ":"; FnGetDir$(Drive$)
ELSE
    PRINT "Invalid drive"
END IF

PRINT "The current directory for the default drive is ";
PRINT FnGetDir$("")

```

The variables Temp\$, Drive, and Zero are declared as STATIC to prevent them from conflicting with variables of the same name in your program. Of course, you could convert this to a formal FUNCTION procedure if you prefer, which considers variables local by default. Converting to a formal function is also needed if you plan to access it from multiple source modules.

Unlike the DOS Get Drive and Set Drive services, service &H47 uses a value of one to indicate drive A, 2 for drive B, and so forth. To request the current directory on the default drive you must use a value of zero. An explicit test for this is made at the beginning of the function. Later, this value is assigned to Registers.DX where DOS expects it. Note that it is really DL that will hold the specified drive number. But assigning DX from Drive as shown does this, and also clears the high (DH) portion in the process. Since the contents of DH are ignored by this DOS service, no harm is done and the extra code that would be needed to assign only DL can be avoided.

As I mentioned earlier, it is essential that you set aside space to hold the returned directory name. Since the longest path name that DOS can accommodate is 65 characters, Temp\$ is assigned to that length.

Then, the segment and address where Temp\$ is stored are passed to DOS in the DS and SI registers. Note that DOS is not very consistent in its use of registers. Where the service that finds the first matching file name uses DS:DX to point to the file specification, this service uses DS:SI to point to the string.

Like the FnFileExist function, you must change the statement that assigns Registers.DS if you plan to use this one with QuickBASIC. The BASIC PDS version of that statement is left active rather than the QuickBASIC version, so QuickBASIC will highlight that line as an error to remind you. Although FnFileExist uses VARSEG for the DS value when used with QuickBASIC, FnGetDir uses -1. Both methods work, and I used -1 here just to show that in context.

After DOSInt is called to load Temp\$ with the current directory name, the Carry Flag is tested to see if an error occurred. The only error that is possible here is "Invalid drive", in which case FnGetDir\$ is assigned a null value as a flag to indicate that. Otherwise, INSTR is used to locate the CHR\$(0) zero byte that DOS assigned to mark the end of the name.

This error testing can be left out to save code if you prefer. You could also validate the drive using the FnDriveValid function, either by adding the code within FnGetDir, or separately prior to invoking it.

Reading Files and Directory Names

One important service that many programs need and which BASIC has never provided is the ability to read directory names from disk. Any word processor worth its salt will let you view a list of files that match, say, a *.DOC extension, and then select the one you want to edit. With the introduction of BASIC PDS Microsoft added the DIR\$ function, which lets you read file names. However, there is no way to specify file attributes (hidden, read-only, and so forth), and also no way to read directory names. To add insult to injury, the PDS manuals do not show clearly how to read a list of file names, and store them into a string array.

The program that follows counts the number of files or directories that match a given specification, and then dimensions and loads a string array with their names.

```
DEFINT A-Z
DECLARE SUB LoadNames (FileSpec$, Array$(), Attribute%)

'$INCLUDE: 'REGTYPE.BI'

TYPE DTA
Reserved AS STRING * 21      'used by find first/next
Attribute AS STRING * 1     'reserved for use by DOS
FileTime AS STRING * 2      'the file's attribute
FileDate AS STRING * 2     'the file's time
FileSize AS LONG            'the file's date
FileName AS STRING * 13    'the file's size
                           'the file's name
END TYPE

DIM SHARED DTAData AS DTA    'shared so LoadNames can
DIM SHARED Registers AS RegType ' access them too
```



```

DEF FnFileCount% (Spec$, Attribute)
    STATIC Count                                'make this private

    Registers.DX = VARPTR(DTADData) 'set new DTA address
    Registers.DS = -1                    'the DTA is in DGROUP
    Registers.AX = &H1A00                 'specify service 1Ah
    CALL DOSInt(Registers)               'DOS set DTA service

    Count = 0                               'clear the counter
    Spec$ = Spec$ + CHR$(0)               'make an ASCIIIZ string
    IF Attribute AND 16 THEN              'find directory names?
        DirFlag = -1                       'yes
    ELSE
        DirFlag = 0                       'no
    END IF

    Registers.DX = SADD(Spec$)             'the file spec address
    Registers.DS = SSEG(Spec$)            'this is for BASIC PDS
    Registers.DS = -1                     'this is for QuickBASIC
    Registers.CX = Attribute              'assign the attribute
    Registers.AX = &H4E00                 'find first matching name
    DO
        CALL DOSInt(Registers)           'see if there's a match
        IF Registers.Flags AND 1 THEN EXIT DO 'no more
        IF DirFlag THEN
            IF ASC(DTADData.Attribute) AND 16 THEN
                IF LEFT$(DTADData.FileName, 1) <> "." THEN
                    Count = Count + 1     'increment the counter
                END IF
            END IF
        ELSE
            Count = Count + 1             'they want regular files
        END IF

        Registers.AX = &H4F00             'find next name
    LOOP

    FnFileCount% = Count                  'assign the function
END DEF

REDIM Names$(1 TO 1)                     'create a dynamic array
Attribute = 19                            'matches directories only Attribute = 39
'matches all files

INPUT "Enter a file specification: ", Spec$
CALL LoadNames(Spec$, Names$(), Attribute)

FOR X = LEN(Spec$) TO 1 STEP -1           'isolate the drive/path
    Temp = ASC(MID$(Spec$, X, 1))
    IF Temp = 58 OR Temp = 92 THEN      "":" or "\"
        Path$ = LEFT$(Spec$, X)        'keep what precedes that
        EXIT FOR                       'and we're all done
    END IF
NEXT

FOR X = 1 TO UBOUND(Names$)             'print the names
    PRINT Path$; Names$(X)
NEXT

```

```

PRINT
PRINT UBOUND(Names$); "matching file(s) "
END

SUB LoadNames (FileSpec$, Array$(), Attribute) STATIC
  Spec$ = FileSpec$ + CHR$(0)      'make an ASCIIZ string
  NumFiles = FnFileCount%(Spec$, Attribute) 'count names
  IF NumFiles = 0 THEN EXIT SUB      'exit if none
  REDIM Array$(1 TO NumFiles)      'dimension the array

  IF Attribute AND 16 THEN          'find directory names?
    DirFlag = -1                    'yes
  ELSE
    DirFlag = 0                     'no
  END IF

  '---- The following code isn't strictly necessary
  ' because we know that FnFileCount already set the
  ' DTA address.
  'Registers.DX = VARPTR(DTADData) 'set new DTA address
  'Registers.DS = -1              'the DTA in DGROUP
  'Registers.AX = &H1A00          'specify service 1Ah
  'CALL DOSInt(Registers)        'DOS set DTA service

  Registers.DX = SADD(Spec$)        'the file spec address
  Registers.DS = SSEG(Spec$)        'this is for BASIC PDS
  'Registers.DS = -1             'this is for QuickBASIC
  Registers.CX = Attribute          'assign the attribute
  Registers.AX = &H4E00            'find first matching name
  Count = 0                        'clear the counter

DO
  CALL DOSInt(Registers)           'see if there's a match
  IF Registers.Flags AND 1 THEN EXIT DO 'no more
  Valid = 0
  IF DirFlag THEN                  'directories?
    IF ASC(DTADData.Attribute) AND 16 THEN
      IF LEFT$(DTADData.FileName, 1) <> "." THEN
        Valid = -1                'this name is valid
      END IF
    END IF
  ELSE
    Valid = -1                    'they want regular files
  END IF

  IF Valid THEN                   'process the file if it
    Count = Count + 1             ' passed all the tests
    Zero = INSTR(DTADData.FileName, CHR$(0))
    Array$(Count) = LEFT$(DTADData.FileName, Zero - 1)
  END IF
  Registers.AX = &H4F00           'find next matching name
LOOP
END SUB

```

These routines call upon the DOS Find First and Find Next services, which performs the actual searching and loading of the names. Before the names can be loaded into an array, you need some way to know how many files there are. Therefore, the FnFileCount function makes repeated calls to DOS to find another file, until there are no more.

The general strategy is to request service &H4E to find the first matching file. If a file is found then the Carry Flag is returned clear; otherwise it is set and the function returns with a count of zero. If a file is found Registers.AX is assigned a value of &H4F, and this tells DOS to resume searching based on the same file specification as before. Where the FnFileExist function merely needed to check for the presence of a file using the Find First service, this one continues in a DO loop until no more matching files are found.

Understand that these DOS services accept either a partial file specification such as "*.BAS" or "D:\PATHNAME*.*", or a single file name such as "CONFIG.SYS" or "C:\AUTOEXEC.BAT".

File Attributes

The DOS Find services also accept and require a file attribute indicating the type of files that are being sought. The method of specifying and isolating files and their attributes is convoluted and confusing to be sure. Figure 11-2 lists each of the six file attributes, and shows which corresponds to each bit in the attribute byte.

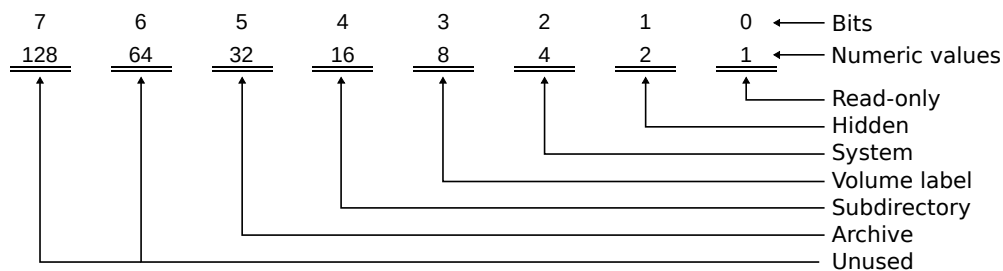


Figure 11-2: The makeup of the bits in the attribute byte, and the individual decimal value of each.

In most cases, the attribute bits are cumulative. For example, if you specify that you want to locate files marked as read-only, you will also get files that are not. But if you leave that bit clear, then read-only files will not be included. The same logic is used for reading directory names. If the directory bit is set then you will read directories, and also regular files whose directory bit is not set. This requires that you perform additional qualifications when the file name is read into the DTA. To make matters even worse, there is an exception to this rule whereby an attribute of zero will still read file names whose archive bit is set.

Before considering how to qualify the names as they are read, you must first understand what attributes are and how to specify them to begin with. Every file has an attribute, which is set by DOS to Archive at the time it is created. The archive bit is used solely to tell if the file has been backed up using the DOS BACKUP utility. When BACKUP copies the file to a floppy disk, it clears the Archive bit in the file's directory entry, then if the file is written to again later, DOS sets that bit. This way, BACKUP can tell which files need to be backed up, and which ones haven't changed since the last backup was

performed. Most modern commercial backup utilities also manipulate the archive bit, for the same reason that DOS' BACKUP does.

The hidden bit tells the DOS DIR command not to display that file's name. Although it won't display in a directory listing, a hidden file may be opened, read from, and written to. The system bit is similar in that it also tells DIR not to display the file. The IO.SYS and MSDOS.SYS files that come with MS-DOS are hidden system files, so to read their names you must set those bits in the search attribute. Note that IBM's version of DOS uses the names IBMBIO.COM and IBMDOS.COM respectively for the same files.

The label bit identifies a file as the disk's volume label, which isn't really a file at all. Every disk is allowed to have one volume label entry in its root directory, which lets an application identify the disk. This feature is not particularly important with hard disks, but when floppy-only systems were the norm this let programs ensure that the correct data diskette was installed in the drive. Even though a volume label is stored in the disk's directory like a regular file name, no sectors are allocated to it. Note that a bug in DOS 2.x versions causes a search for a volume label to fail. The only work-around is to use the more complex DOS 1.x Find First/Next services that are still supported in later versions for compatibility with older programs.

Finally, the subdirectory attribute bit identifies a file as a directory. From DOS' perspective a subdirectory is a file, with fixed-length records that hold the names, attributes, and other information for the files it contains. Notice that the "." and ".." directory entries that appear when you type DIR are in fact present in that directory.

Every directory except the root contains these entries, and they also have a directory attribute. The single dot refers to the current directory, and the double dots to the parent directory one level above. I mention this because these *dot* entries are reported by the Find First and Find Next services, and in many cases you will want to filter them out.

To specify a file attribute you must determine the correct value, based on the individual bits to be included in the search. As I stated earlier, setting the attribute to zero includes all normal files, and exclude any marked as read-only, hidden, system, or subdirectory. Therefore, to include all files but not subdirectories you will use an attribute value of 39. This value is derived by adding up the bit values for each desired attribute as shown in Figure 11-3.

When you add all of the values for each bit of interest, the answer is 32 (archive) + 4 (system) + 2 (hidden) + 1 (read-only) = 39. In a similar fashion, you will use 16 to read directory names, but hidden or read-only directories will not be included unless you also add 2 + 1 = 3, resulting in a final value of 19.

Although you can specify attribute bits in nearly any combination, DOS returns all of the names that match any of the bits. Therefore, you must further qualify the files by examining the attribute DOS returns in the DTA TYPE variable. A typical search for directory names will ask to include all three

attribute bits (directory, hidden, and read-only), but the qualification test merely tests if the directory bit is set. The following excerpt shows this in context.

```
Registers.CX = 19
CALL DOSInt(Registers)
IF ASC(DTADData.Attribute) AND 16 THEN 'it is a directory
```

Even if the directory was in fact hidden or read-only, the test for the directory bit will succeed regardless of any other bits that may be set. Unfortunately, the reverse is not true. If the directory is not hidden or read-only, then testing for those bits will fail. Both the FnFileCount function and the LoadNames subprogram include an explicit test for directory searches, and contain additional logic to check for this case.

You could also add similar logic to the FnFileExist function, or create a separate version perhaps called FnDirExist that adds a test for the directory bit and also filters out the "dot" entries.

REDIM Preserve

One glaring shortcoming you have probably already noticed is the enormous amount of code that is duplicated in both the FnFileCount and LoadNames routines. In fact, the two are almost identical, except that LoadNames also assigns elements in the array. Worse, having to count all of the names before they can be read greatly increases the amount of time needed to process a directory when there are many files. Until you know how many files are present, there's no way to know how large to dimension the string array.

One solution is to create an array with, say, 500 elements, and hope that the actual number of files does not exceed that. But if there are only a few files this wastes a lot of memory, and when there are more than 500, then you're still out of luck. In fact, this is one of the few features that C offers but QuickBASIC does not. C programs can allocate memory that will be treated as an array, and then repeatedly request more memory for that same array as it is needed.

Fortunately, BASIC PDS version 7.1 includes the PRESERVE option to the REDIM statement. This allows you to increase or decrease the size of an array, but without destroying its current contents. Thus, REDIM PRESERVE is ideal for applications like this that require an array's size to be altered. The next, much shorter program uses REDIM PRESERVE to advantage, and avoids the extra step that counts how many files match the search specification. Of course, this program requires BASIC PDS.

```
DEFINT A-Z
DECLARE SUB LoadNames (FileSpec$, Array$(), Attribute%)
'$INCLUDE: 'REGTYPE.BI'

TYPE DTA
  Reserved AS STRING * 21      'used by find first/next
  Attribute AS STRING * 1      'reserved for use by DOS
  FileTime AS STRING * 2      'the file's attribute
  FileDate AS STRING * 2      'the file's time
  FileSize AS LONG            'the file's date
  FileName AS STRING * 13     'the file's size
                              'the file's name
```

```

END TYPE

DIM SHARED DTADData AS DTA      'shared so LoadNames can
DIM SHARED Registers AS RegType '  access them too

REDIM Names$(1 TO 1)           'create a dynamic array
Attribute = 19                  'matches directories only Attribute = 39
'matches all files
Spec$ = "*.*"                   'so does this
CALL LoadNames(Spec$, Names$(), Attribute)

IF Names$(1) = "" THEN         'check for no files
  PRINT "No matching files"
ELSE
  FOR X = 1 TO UBOUND(Names$)  'print the names
    PRINT Path$; Names$(X)
  NEXT
END IF
END

SUB LoadNames (FileSpec$, Array$(), Attribute) STATIC
  Spec$ = FileSpec$ + CHR$(0)  'make an ASCIIIZ string
  Count = 0                     'clear the counter

  Registers.DX = VARPTR(DTADData) 'set new DTA address
  Registers.DS = -1              'the DTA is in DGROUP
  Registers.AX = &H1A00          'specify service 1Ah
  CALL DOSInt(Registers)        'DOS set DTA service

  IF Attribute AND 16 THEN      'find directory names?
    DirFlag = -1                'yes
  ELSE
    DirFlag = 0                 'no
  END IF

  Registers.DX = SADD(Spec$)     'the file spec address
  Registers.DS = SSEG(Spec$)     'this is for BASIC PDS
  Registers.CX = Attribute      'assign the attribute
  Registers.AX = &H4E00          'find first matching name
  DO
    CALL DOSInt(Registers)      'see if there's a match
    IF Registers.Flags AND 1 THEN EXIT DO 'no more

    Valid = 0                    'invalid until qualified
    IF DirFlag THEN              'find directories?
      IF ASC(DTADData.Attribute) AND 16 THEN 'yes, is it?
        IF LEFT$(DTADData.FileName, 1) <> "." THEN
          Valid = -1             'this name is valid
        END IF
      END IF
    ELSE
      Valid = -1                 'they want regular files
    END IF

    IF Valid THEN                'process the file if it
      Count = Count + 1          ' passed all the tests
      REDIM PRESERVE Array$(1 TO Count) 'expand the array
      Zero = INSTR(DTADData.FileName, CHR$(0)) 'find zero
      Array$(Count) = LEFT$(DTADData.FileName, Zero - 1)
    END IF
  LOOP

```

```
Registers.AX = &H4F00      'find next matching name
LOOP
END SUB
```

Managing Files

Chapter 6 explained in great detail how files are opened, closed, read, and written using BASIC. I mentioned there that BASIC imposes a number of arbitrary limitations on what you can and cannot do with files. Indeed, DOS allows almost any action except writing to a file that has been opened for input. As you can imagine, CALL Interrupt—or in this case the DOSInt replacement routine—can be used to circumvent BASIC and access your files directly.

Although BASIC expects you to state how the file will be accessed with the various OPEN options, to DOS all files are considered as being opened for binary access. There is no equivalent DOS service for BASIC's INPUT # or PRINT # commands. Therefore, it is up to you to write subroutines that look for a terminating carriage return and optional line feed when reading sequential text. Likewise, it is up to you to manually append a carriage return and line feed to the end of each line of text written to disk.

Frankly, sequential file access is often best left to BASIC, since a lot of time-consuming tests are needed when reading sequential data. You could, however, use the BufIn function shown in Chapter 6, or similar logic of your own devising. There are many types of file access that can be performed using direct DOS calls, and I will show those that are the most useful and appropriate here.

The program that will follow shortly is a combination demonstration, and suite of twelve subprograms and functions that perform most of the services necessary for manipulating files. Subprograms are provided to replace BASIC's OPEN, CLOSE, GET, and PUT statements, as well as LOCK and UNLOCK, SEEK, and KILL.

There are also replacement functions for LOC and LOF, as well as two additional subprograms that have no BASIC equivalent. All of the routines use the DOSInt interface routine, and avoid using BASIC's file handling statements. The demonstration is comprised of a series of code blocks that exercise each routine showing how it is used. Comments at the start of each block explain what is being demonstrated.

One reason to go behind BASIC's back this way is to avoid its many restrictions. For example, BASIC will not let you read from a file that has been opened for output, even though DOS considers this to be perfectly legal. Another is to avoid the need for ON ERROR. As you learned in Chapter 3, ON ERROR can make a program run more slowly, and also increase its size. By going directly to DOS you can avoid the burden of ON ERROR, which is otherwise needed to prevent your program from terminating if an error occurs. These replacement routines avoid errors such as those caused by attempting to open a file that does not exist, or trying to lock a network file that has already been locked by someone else.

As with some of the other programs in this book that combine a demonstration and subroutines, you should make a copy of the file, and then delete all of the code in the main portion of the program. The only lines that must not be deleted are the DEFINT, DECLARE, and INCLUDE statements, and also the two DIM SHARED statements. Then, you can load the resultant module into the BASIC editor along with your own main application.

```
'DOS.BAS, demonstrates the direct DOS access routines

DEFINT A-Z
DECLARE FUNCTION DOSError% ()
DECLARE FUNCTION ErrorMessage$ (ErrNumber)
DECLARE FUNCTION LocFile& (Handle)
DECLARE FUNCTION LofFile& (Handle)
DECLARE FUNCTION PeekWord% (BYVAL Segment, BYVAL Address)

DECLARE SUB ClipFile (Handle, NewLength&)
DECLARE SUB CloseFile (Handle)
DECLARE SUB FlushFile (Handle)
DECLARE SUB KillFile (FileName$)
DECLARE SUB LockFile (Handle, Location&, NumBytes&, Action)
DECLARE SUB OpenFile (FileName$, OpenMethod, Handle)
DECLARE SUB ReadFile (Handle, Segment, Address, NumBytes)
DECLARE SUB SeekFile (Handle, Location&, SeekMethod)
DECLARE SUB WriteFile (Handle, Segment, Address, NumBytes)

'$INCLUDE: 'REGTYPE.BI'

DIM SHARED Registers AS RegType 'so all can access it
DIM SHARED ErrCode              'ditto for the ErrCode
CRLF$ = CHR$(13) + CHR$(10)      'define this once now

COLOR 15, 1                      'this makes the DOS
CLS                               'messages high-intensity
COLOR 7, 1

'---- Open the test file we will use.
FileName$ = "C:\MYFILE.DAT"      'specify the file name
OpenMethod = 2                   'read/write non-shared
CALL OpenFile(FileName$, OpenMethod, Handle)
GOSUB HandleErr
PRINT FileName$; " successfully opened, handle:"; Handle

'---- Write a test message string to the file.
Msg$ = "This is a test message." + CRLF$
Segment = SSEG(Msg$)             'use this with BASIC PDS
'Segment = VARSEG(Msg$)          'use this with QuickBASIC
Address = SADD(Msg$)
NumBytes = LEN(Msg$)
CALL WriteFile(Handle, Segment, Address, NumBytes)
GOSUB HandleErr
PRINT "The test message was successfully written."

'---- Show how to write a numeric value.
IntData = 1234
Segment = VARSEG(IntData)
Address = VARPTR(IntData)
```



```

NumBytes = 2
CALL WriteFile(Handle, Segment, Address, NumBytes)
GOSUB HandleErr
PRINT "The integer variable was successfully written."

'----- See how large the file is now.
Length& = LofFile&(Handle)
GOSUB HandleErr
PRINT "The file is now"; Length&; "bytes long."

'----- Seek back to the beginning of the file.
Location& = 1 'specify file offset 1
SeekMethod = 0 'relative to beginning
CALL SeekFile(Handle, Location&, SeekMethod)
GOSUB HandleErr
PRINT "We successfully seeked back to the beginning."

'----- Ensure that the Seek worked by seeing where we are.
CurSeek& = LocFile&(Handle)
GOSUB HandleErr
PRINT "The DOS file pointer is now at location"; CurSeek&

'----- Read the test message back in again.
Buffer$ = SPACE$(23) 'the length of Msg$
Segment = SSEG(Buffer$) 'use this with BASIC PDS
'Segment = VARSEG(Buffer$) 'use this with QuickBASIC
Address = SADD(Buffer$)
NumBytes = LEN(Buffer$)
CALL ReadFile(Handle, Segment, Address, NumBytes)
GOSUB HandleErr
PRINT "Here is the test message: "; Buffer$

'----- Skip over the CRLF by reading it as an integer.
Address = VARPTR(Temp) 'read the CRLF into Temp
Segment = VARSEG(Temp)
NumBytes = 2
CALL ReadFile(Handle, Segment, Address, NumBytes)
GOSUB HandleErr

'----- Read the integer written earlier, also into Temp.
Address = VARPTR(Temp)
Segment = VARSEG(Temp)
NumBytes = 2
CALL ReadFile(Handle, Segment, Address, NumBytes)
GOSUB HandleErr
PRINT "The integer value just read is:"; Temp

'----- Append a new string at the end of the file.
Msg$ = "This is appended to the end of the file." + CRLF$
Segment = SSEG(Msg$) 'use this with BASIC PDS
'Segment = VARSEG(Msg$) 'use this with QuickBASIC
Address = SADD(Msg$)
NumBytes = LEN(Msg$)
CALL WriteFile(Handle, Segment, Address, NumBytes)

```

```

GOSUB HandleErr
PRINT "The appended message has been written, ";
PRINT "but it's still in the DOS file buffer."

'---- Flush the file's DOS buffer to disk.
CALL FlushFile(Handle)
GOSUB HandleErr
PRINT "Now the buffer has been flushed to disk. ";
PRINT "Here's the file contents:"
SHELL "TYPE " + FileName$

'---- Display the current length of the file again.
PRINT "Before calling ClipFile the file is now";
Length& = LofFile(Handle)
GOSUB HandleErr
PRINT Length&; "bytes long."

'---- Clip the file to be 2 bytes shorter.
NewLength& = LofFile(Handle) - 2
CALL ClipFile(Handle, NewLength&)
PRINT "The file has been clipped successfully. ";

'---- Prove that the clipping worked successfully.
Length& = LofFile(Handle)
GOSUB HandleErr
PRINT "It is now"; Length&; "bytes long."

'---- Close the file.
CALL CloseFile(Handle)
GOSUB HandleErr
PRINT "The file was successfully closed."

'---- Open the file again, this time for shared access.
OpenMethod = 66 'full sharing, read/write
CALL OpenFile(FileName$, OpenMethod, Handle)
GOSUB HandleErr
PRINT FileName$; " successfully opened in shared mode";
PRINT ", handle:"; Handle

'---- Lock bytes 50 through 59.
Start& = 50
Length& = 10
Action = 0 'specify locking
CALL LockFile(Handle, Start&, Length&, Action)
GOSUB HandleErr
PRINT "File bytes 50 through 59 are successfully locked."

'---- Prove that it is locked by asking DOS to copy it.
PRINT "DOS (another process) fails to access the file:"
SHELL "COPY " + FileName$ + " NUL"

'---- Unlock the same range of bytes (mandatory).
Start& = 50

```

```

Length& = 10
Action = 1                                'specify unlocking
CALL LockFile(Handle, Start&, Length&, Action)
GOSUB HandleErr
PRINT "File bytes 50 through 59 successfully unlocked."

'---- Prove the unlocking worked by having DOS copy it.
PRINT "Once unlocked DOS can access the file:";
SHELL "COPY " + FileName$ + " NUL"

CloseIt:
'---- Close the file
CALL CloseFile(Handle)
GOSUB HandleErr
PRINT "The file was successfully closed, ";

'---- Kill the file to be polite
CALL KillFile(FileName$)
GOSUB HandleErr
PRINT "and then successfully deleted."

END

'=====
' Error handler
'=====
HandleErr:

TempErr = DOSError%                       'call DOSError% just once
IF TempErr = 0 THEN RETURN                 'return if no errors
PRINT ErrorMessage$(TempErr)              'else print the message
IF TempErr = 1 THEN                        'we failed trying to lock
    COLOR 7 + 16
    PRINT "SHARE must be installed to continue."
    COLOR 7
    RETURN CloseIt
ELSE                                       'otherwise end
    END
END IF

SUB ClipFile (Handle, Length&) STATIC
'-- Use SeekFile to seek there, and then call WriteFile
' specifying zero bytes to truncate it at that point.
' Length& + 1 is needed because we need to seek just
' PAST the point where the file is to be truncated.
CALL SeekFile(Handle, Length& + 1, Zero)
IF ErrCode THEN EXIT SUB                  'exit if an error occurred
CALL WriteFile(Handle, Dummy, Dummy, Zero)
END SUB

SUB CloseFile (Handle) STATIC
ErrCode = 0                               'assume no errors
Registers.AX = &H3E00                      'close file service
Registers.BX = Handle                      'using this handle
CALL DOSInt(Registers)
IF Registers.Flags AND 1 THEN ErrCode = Registers.AX

```

END SUB

```
FUNCTION DOSError%
    DOSError% = ErrCode          'simply return the error
END FUNCTION
```

```
FUNCTION ErrorMessage$ (ErrNumber) STATIC
    SELECT CASE ErrNumber
        CASE 2
            ErrorMessage$ = "File not found"
        CASE 3
            ErrorMessage$ = "Path not found"
        CASE 4
            ErrorMessage$ = "Too many files"
        CASE 5
            ErrorMessage$ = "Access denied"
        CASE 6
            ErrorMessage$ = "Invalid handle"
        CASE 61
            ErrorMessage$ = "Disk full"
        CASE ELSE
            ErrorMessage$ = "Undefined error: " + STR$(ErrNumber)
    END SELECT
END FUNCTION
```

```
SUB FlushFile (Handle) STATIC
    ErrCode = 0                'assume no errors
    Registers.AX = &H4500      'create duplicate handle
    Registers.BX = Handle     'based on this handle

    CALL DOSInt(Registers)
    IF Registers.Flags AND 1 THEN 'an error, assign it
        ErrCode = Registers.AX
    ELSE
        TempHandle = Registers.AX 'no error, so closing the
        TempHandle 'dupe flushes the data
        CALL CloseFile(TempHandle)
    END IF
END SUB
```

```
SUB KillFile (FileName$) STATIC
    ErrCode = 0                'assume no errors
    LocalName$ = FileName$ + CHR$(0) 'make an ASCIIIZ string

    Registers.AX = &H4100      'delete file service
    Registers.DX = SADD(LocalName$) 'using this handle
    Registers.DS = SSEG(LocalName$) 'use this with PDS
    'Registers.DS = -1         'use this with QB

    CALL DOSInt(Registers)
    IF Registers.Flags AND 1 THEN ErrCode = Registers.AX
END SUB
```

```
FUNCTION LocFile& (Handle) STATIC
    ErrCode = 0                'assume no errors
    Registers.AX = &H4201      'seek to where we are now
    Registers.BX = Handle     'using this handle
```

```

Registers.CX = 0          'move zero bytes from here
Registers.DX = 0

CALL DOSInt(Registers)
IF Registers.Flags AND 1 THEN 'an error occurred
    ErrCode = Registers.AX
ELSE
    'adjust to one-based
    LocFile& = (Registers.AX + (65536 * Registers.DX)) + 1
END IF
END FUNCTION

SUB LockFile (Handle, Location&, NumBytes&, Action) STATIC
    ErrCode = 0          'assume no errors
    LocalLoc& = Location& - 1 'adjust to zero-based

    Registers.AX = Action + (256 * &H5C) 'lock/unlock
    Registers.BX = Handle
    Registers.CX = PeekWord%(VARSEG(LocalLoc&), VARPTR(LocalLoc&) + 2)
    Registers.DX = PeekWord%(VARSEG(LocalLoc&), VARPTR(LocalLoc&))
    Registers.SI = PeekWord%(VARSEG(NumBytes&), VARPTR(NumBytes&) + 2)
    Registers.DI = PeekWord%(VARSEG(NumBytes&), VARPTR(NumBytes&))
    CALL DOSInt(Registers)
    IF Registers.Flags AND 1 THEN ErrCode = Registers.AX
END SUB

FUNCTION LofFile& (Handle)
    '---- first get and save the current file location
    CurLoc& = LocFile&(Handle) 'LocFile also clears ErrCode
    IF ErrCode THEN EXIT FUNCTION

    Registers.AX = &H4202 'seek to the end of the file
    Registers.BX = Handle 'using this handle
    Registers.CX = 0 'move zero bytes from there
    Registers.DX = 0

    CALL DOSInt(Registers)
    IF Registers.Flags AND 1 THEN 'an error occurred
        ErrCode = Registers.AX
        EXIT FUNCTION
    ELSE
        'assign where we are
        LofFile& = Registers.AX + (65536 * Registers.DX)
    END IF

    Registers.AX = &H4200 'seek to where we were before
    Registers.BX = Handle 'using this handle
    Registers.CX = PeekWord%(VARSEG(CurLoc&), VARPTR(CurLoc&) + 2)
    Registers.DX = PeekWord%(VARSEG(CurLoc&), VARPTR(CurLoc&))
    CALL DOSInt(Registers)
    IF Registers.Flags AND 1 THEN ErrCode = Registers.AX
END FUNCTION

SUB OpenFile (FileName$, Method, Handle) STATIC
    ErrCode = 0          'assume no errors
    Registers.AX = Method + (256 * &H3D) 'open file service
    LocalName$ = FileName$ + CHR$(0) 'make an ASCIIIZ string

    DO
        Registers.DX = SADD(LocalName$) 'point to the name

```

```

Registers.DS = SSEG(LocalName$) 'use this with PDS
'Registers.DS = -1             'use this w/QuickBASIC
CALL DOSInt(Registers)         'call DOS
IF (Registers.Flags AND 1) = 0 THEN 'no errors
    Handle = Registers.AX       'assign the handle
    EXIT SUB                   'and we're all done
END IF

IF Registers.AX = 2 THEN        'File not found error
    Registers.AX = &H3C00      'so create it!
ELSE
    ErrCode = Registers.AX     'read the code from AX
    EXIT SUB
END IF
LOOP
END SUB

SUB ReadFile (Handle, Segment, Address, NumBytes) STATIC
    ErrCode = 0                 'assume no errors

    Registers.AX = &H3F00       'read from file service
    Registers.BX = Handle       'using this handle
    Registers.CX = NumBytes     'and this many bytes
    Registers.DX = Address      'read to this address
    Registers.DS = Segment     'and this segment

    CALL DOSInt(Registers)
    IF Registers.Flags AND 1 THEN ErrCode = Registers.AX
END SUB

SUB SeekFile (Handle, Location&, Method) STATIC
    ErrCode = 0                 'assume no errors
    LocalLoc& = Location& - 1   'adjust to zero-based

    Registers.AX = Method + (256 * &H42)
    Registers.BX = Handle
    Registers.CX = PeekWord%(VARSEG(LocalLoc&), VARPTR(LocalLoc&) + 2)
    Registers.DX = PeekWord%(VARSEG(LocalLoc&), VARPTR(LocalLoc&))
    CALL DOSInt(Registers)
    IF Registers.Flags AND 1 THEN ErrCode = Registers.AX
END SUB

SUB WriteFile (Handle, Segment, Address, NumBytes) STATIC
    ErrCode = 0                 'assume no errors

    Registers.AX = &H4000
    Registers.BX = Handle
    Registers.CX = NumBytes
    Registers.DX = Address
    Registers.DS = Segment

    CALL DOSInt(Registers)
    IF Registers.Flags AND 1 THEN
        ErrCode = Registers.AX
    ELSEIF Registers.AX <> Registers.CX THEN
        ErrCode = 61
    END IF
END SUB

```

This program begins by dimensioning two variables as `SHARED` throughout the entire module. By establishing the `Registers TYPE` variable as `SHARED`, all of the routines can use the same portion of `DGROUP` memory. If a separate `DIM` statement were used within each procedure, that many copies of this 20-byte variable would reside in memory at once. The `CRLF$` variable does not need to be shared, because it is used only by the demonstration portion of the program.

Before I describe each of these routines and how they are used, it is important to explain how DOS uses file handles. `BASIC` is unique among languages in that it allows you to make up an arbitrary file number that is used to access the files. With most languages and operating systems—and `DOS` is no exception—it is the operating system that assigns a number which your program must remember. Therefore, when you call the `OpenFile` routine to open a file, the `Handle` parameter is returned to you and you will use that number for subsequent file operations.

Another important point is how errors are handled by these routines. Since you do not use `ON ERROR` to trap those situations another method is needed. Each routine clears or sets a global `SHARED` variable named `ErrCode`, which indicates its success or failure. After each call to one of these routines you will then check this variable, to see if it was successful. For the most efficiency, this program invokes a central error checking `GOSUB` routine that performs the actual testing. If an error occurs this routine prints an appropriate message using the `ErrMsg$` function, and then ends. The `DOSError` function is provided to allow access to `ErrCode` from other modules.

In practice, it is not strictly necessary to add an explicit test after each subroutine call. For example, if you know the file has been opened successfully and you are sure the disk drive has sufficient space, then it is probably safe to assume that subsequent file writes will be okay. However, if you do call a routine that causes an error and don't check for that error, the next successful call to another routine will clear `ErrCode` and you will have no way to know about the earlier error.

Opening a File

The demonstration begins by first assigning a file name and open method, and then calling `OpenFile` to open the file. The open method lets you indicate the file access mode (reading, writing, or both), and also if the file will be accessed on a network. This parameter is bit-coded, and each bit has a parallel equivalent in `BASIC`'s `ACCESS READ`, `WRITE`, `SHARED`, `LOCK READ`, and `LOCK WRITE` options. Figure 11-3 shows how these bits are organized.

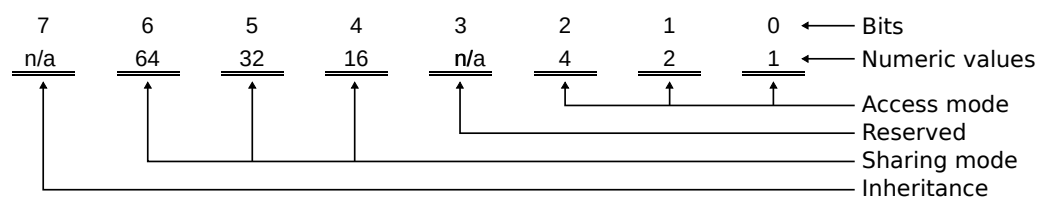


Figure 11-3: The organization of the bits that establish how a file is to be opened.

As with the file attribute bits shown earlier in Figure 11-2, you also need to set bits individually here to fully control the various file permission privileges. The access mode bits are valid with DOS versions 2.0 or later, and are equivalent to BASIC's ACCESS arguments. The sharing mode bits require DOS 3.0 or later, and also require SHARE.EXE to be installed. Note that some network software does not explicitly require SHARE, and provides the same functionality as part of its normal operation.

The three lower bits control the file access, using the following binary code: 000 establishes read-only access, 001 allows writing only, and 010 allows both reading and writing. The term access as used here means what actions your program can perform, and has nothing to do with network or file sharing privileges.

File sharing privileges are controlled by the three bits in the upper nybble (half-byte), and these determine what actions may be performed by other programs while your file is open. Regardless of what sharing (or locking) options you choose, your program always has full permission to access the file. The share bits are organized as follows: 000 means sharing is disabled, and this is what you must specify if you are not running on a network or when DOS 2.x is installed. A code of 001 denies other programs access to either read from or write to the file, 010 allows other programs to read but not write, and 011 allows writing but not reading. A code of 100 indicates full sharing, which lets other programs read and write, as long as that part of the file is not locked explicitly.

Again, these codes are presented as binary values, and it is up to you to determine the correct value based on the settings of the individual bits. This is not as hard as it may sound at first, because you simply add up the bit values shown in the table. For example, to open a file for non-network read/write access under any version of DOS you use $000 + 010 = 2$, which is the value used in the first OPEN example. To open a file for reading and writing and also allow other applications to access it fully you instead use $100 + 010 = 64 + 2 = 66$. This is shown in the second OPEN statement. Table 11-2 lists a few of the possible bit combinations, with the equivalent BASIC OPEN options.

BASIC OPEN Statement	Bits	Value
OPEN FOR BINARY	00000010	2
OPEN FOR BINARY ACCESS READ	00000000	0
OPEN FOR BINARY ACCESS WRITE	00000001	1
OPEN FOR BINARY ACCESS READ WRITE	00000010	2
OPEN FOR BINARY ACCESS READ SHARED	01000000	64
OPEN FOR BINARY LOCK READ	00110010	50
OPEN FOR BINARY LOCK WRITE	00100010	34

Table 11-2: Bit equivalents for some of BASIC's OPEN options.

Reading and Writing

Once the file has been opened successfully, the next step is to show how to write a string variable in the same way BASIC does when you use PRINT #. The WriteFile and ReadFile routines each expect four

arguments: the DOS file handle, the segment and address to save from or read into, and the number of bytes. These are the same parameters that DOS expects, and you can see by examining the subprograms that they merely pass this information on to DOS.

Just before the first call to WriteFile, Msg\$ is assigned a short test string, and a carriage return and line feed are appended to it manually. Remember, when you use BASIC's PRINT # command it is BASIC that adds these bytes for you. When dealing with DOS directly it is up to you to append these characters. Of course, you would omit these to mimic appending a semicolon at the end of a BASIC print line:

```
PRINT #1, Msg$;
```

SSEG then determines where the string data segment is, and SADD reports its address within that segment. The QuickBASIC version is shown as a comment, and it uses VARSEG instead. The number of bytes is obtained using LEN, and DOS accepts any value up to 65535. It is imperative that you never pass a value of zero for the number of bytes, or DOS will truncate the file at the current seek location.

The next example that writes an integer variable to the file is similar, except it uses a fixed length of 2. BASIC will not let you pass different types of data to one subprogram or function, which is why these read and write routines are designed to accept a segment and address.

ReadFile is not called until later in the demonstration; however, it is nearly identical to WriteFile. Because you must tell ReadFile how many bytes are to be read, you should establish some type of system. One good one is the method used by Lotus and described in Chapter 6. For programs that do not need such a heavy-handed approach or that write only strings, you could use a simpler technique. For example, each string could be preceded by an integer length word, and that word would be read prior to reading each string. The short code fragment that follows shows how this might work.

```
Segment = VARSEG(Length)      'Length is what gets read first
Address = VARPTR(Length)
CALL ReadFile(Handle, Segment, Address, 2)

Work$ = SPACE$(Length)       'make a string that long
Segment = SSEG(Work$)        'then read Length bytes into the string
Address = SADD(Work$)
CALL ReadFile(Handle, Segment, Address, Length)
```

Setting and Reading the DOS Seek Location

The LocFile and LofFile functions are similar to their BASIC LOC and LOF counterparts, except that LocFile is really equivalent to the SEEK function. Chapter 6 described the difference between the LOC and SEEK functions, and came to the inescapable conclusion that LOC is not nearly as useful as SEEK in most situations.

The SeekFile subprogram on the other hand is equivalent to the statement form of BASIC's SEEK, and offers an interesting twist as an enhancement. Where BASIC's SEEK statement expects an offset from the beginning of the file, DOS provides additional seek methods. One lets you seek relative to where

you are now in the file, and the other is relative to the end of the file. Therefore, I have included a `SeekMethod` parameter with my version of `SeekFile`, letting you enjoy the same flexibility.

If `SeekMethod` is set to zero, DOS behaves the same as BASIC does and bases the new seek location from the beginning of the file. If `SeekMethod` is instead assigned to 1, the new offset into the file will be based on the current location. Note that you may use both positive and negative seek values, to move forward and backwards respectively. Finally, using a `SeekMethod` value of 2 tells DOS to consider the new location as being relative to the end of the file.

For this method you may also use either a positive or negative value, to go beyond the end of the file or some offset before the end. While there is nothing inherently wrong with seeking past the end of a file, if any data is written at that point DOS will make that the new file length. And as explained in Chapter 6, the portion of the file that lies between the previous end of the file and the current end will hold whatever junk happened to be in the sectors that were just assigned to extend the length.

One slight complication arises if you are dealing with fixed-length record data: you must calculate the appropriate file offset manually. The short one-line DEF FN function below shows how to do this.

```
DEF FNSeekLoc&(RecNumber, RecLen) = ((RecNumber - 1) * CLNG(RecLen)) + 1
```

Locking a File

The `LockFile` subprogram serves the same purpose as BASIC's `LOCK` and `UNLOCK` statements. Because the code to lock and unlock a file are identical except for a single instruction, it seemed reasonable to combine the two services into one routine. `LockFile` expects four arguments: a handle, a starting offset, the number of bytes, and an action code. The starting offset and number of bytes use long integer values, to accommodate large files.

Because DOS's `Lock` and `Unlock` services require you to specify the range of bytes to be locked, additional effort may be needed on your part. For example, if you are manipulating fixed-length records it is up to you to translate record numbers and record ranges to an equivalent binary offset and number of bytes. Fortunately, these values are very easy to determine using the following formulas:

```
Location& = (RecNumber - 1) * CLNG(RecLength)  
NumBytes& = RecLength * CLNG(NumRecords)
```

Note how `CLNG` is necessary to prevent BASIC from creating an overflow error if the result of the multiplications exceeds 32767.

`LockFile` can also be used with normal BASIC file handling statements, if you merely want to avoid an error from attempting to lock a file that is already locked by another process. This requires you to use BASIC's `FILEATTR` function to obtain the equivalent DOS handle, thus:

```
Handle = FILEATTR(FileNumber, 2)
```

Here, FileNumber is the BASIC file number that was specified when the file was first opened. For example, if you used this:

```
OPEN FileName$ FOR RANDOM SHARED AS #4 LEN = RecLength
```

then the correct value for FileNumber will be 4.

Beyond BASIC's File Handling

Aside from SeekFile's ability to use the end of a file or the current seek location as a base point, the routines presented so far merely mimic the same capabilities BASIC already provides. Two notable exceptions, however, are ClipFile and FlushFile.

The ClipFile subprogram lets you set a new length for a file, and that length may be either longer or shorter than the current length. ClipFile takes advantage of a little-known DOS feature that sets a new length for a file when you tell it to write zero bytes. This technique was used in the DBPACK.BAS program from Chapter 7, and it let that program remove deleted records from the end of a dBASE file.

ClipFile begins by calling SeekFile to move the DOS file pointer just past the new length specified. If no error occurred it then calls WriteFile to write zero bytes at that point, thus establishing the new length. Notice the way the undefined variable Zero is used rather than a literal constant 0. As you already learned in Chapter 2, when a constant is passed to a subprogram or function, BASIC creates code to store a copy of the constant in DGROUP, and then passes the address of that copy. Although the variable Zero also requires two bytes of DGROUP memory for storage, the code to explicitly place the value there is avoided. Since an unassigned variable is always zero this method can be used with confidence.

FlushFile also provides an important service that BASIC does not. When data is written to disk using either BASIC or DOS via direct interrupt calls, the last portion that was written is not necessarily on the physical disk. DOS buffers all file writes to minimize the number of disk accesses needed, thereby improving the speed of those writes. BASIC performs additional buffering as well, which further improves your program's performance. However, this creates a potential problem because a power outage or other disaster will cause any data in the file buffer to be lost.

FlushFile calls upon another little-known DOS service called *Duplicate Handle*. When this service is called with the handle of a file that is already open, DOS creates a duplicate handle for the same file. This service is not that useful in and of itself, except for one important exception: When the duplicate handle is subsequently closed, DOS also writes the original file's contents to disk and updates the directory entry to reflect the current length. This is exactly what FlushFile does to flush the file buffer to disk.

Error Messages

The ErrorMessage\$ function is designed to display an appropriate message if an error occurs while using these routines. DOS has fewer error codes than BASIC, and it also uses a completely different

numbering system. The ErrMessage\$ function returns an error message that is equivalent to BASIC's where possible, but based on the DOS error return codes.

Potential Problems

Although this collection of file handling routines offers many improvements over using equivalent BASIC statements, there is one important issue I have not addressed here: handling critical errors. A critical error is caused by attempting to access a floppy disk drive with the drive door open, or no disk in place. At the DOS command line critical errors result in the infamous "Abort, Retry, Fail" message.

Handling critical errors requires pure assembly language and is a fairly complex undertaking. Therefore, I have purposely omitted that functionality from these routines. However, add-on library products such as QuickPak Professional and P.D.Q. from Crescent Software are written in assembly language, and include critical error handling.

There is another potential problem you must be aware of when using these routines. When you open a file using BASIC's OPEN statement, and then restart the program before the file has been closed, BASIC closes the file before running your program again. This is done automatically and without your knowing about it.

If you call OpenFile to open a file and then restart the program, the original file remains open. This causes no harm by itself. Your program will simply receive the next available handle when it calls OpenFile. But at some point you will surely exhaust the available handles. The problem is that you will not be able to save your program, because the BASIC editor needs a handle when writing your source code to disk.

The solution is to press F6 to go to the Immediate window, and then type the following line:

```
FOR X% = 5 TO 20: CALL CloseFile(X%): NEXT
```

This closes all of the files your program opened, thus freeing them for use by the BASIC editor. It is essential that you never close DOS handles zero through four, because they are in use by the PC. Since DOS uses these handles itself to print to the screen and read keyboard input, closing those handles will effectively lock up your PC.

It is okay to close handles 5 through 20, even if your program hasn't opened that many. That is, asking DOS to close a file handle that was never opened does no harm.

Accessing The Mouse

All of the DOS and BIOS system services we have looked at so far rely on either the Interrupt routine that comes with BASIC, or the simplified DOSInt replacement. In a similar fashion, accessing the

mouse driver also requires you to call interrupts. All of the mouse services are invoked using Interrupt &H33, and like DOS and the BIOS they require you to load the processor's registers to pass information, and then read them again afterward to obtain the results.

In this section I will present several useful subroutines that show how to access the mouse interrupt. The first portion discusses the various utility routines, and shows how they are used. Following that, I will explain how the routines actually work and interface with the mouse driver.

Mouse Services

The important mouse services provided here are those that turn the mouse cursor on and off, position it on the screen and control its color, and let you determine which buttons are being pressed and where the cursor is presently located. Other routines show how to restrict the range of the mouse cursor's travel, and show how to define new, custom cursor shapes.

To reduce the size of your programs I have written a short assembly language subroutine called MouseInt. This is similar to the DOSInt routine introduced in Chapter 6, except it is intended for use with the mouse interrupt &H33.

```
;MOUSEINT.ASM

.MODEL Medium, Basic

MouseRegs Struct
    RegAX  DW ?
    RegBX  DW ?
    RegCX  DW ?
    RegDX  DW ?
    Segmnt DW ?
MouseRegs Ends

.CODE

MouseInt Proc Uses SI DS ES, MRegs:Word
    Mov  SI,MRegs           ;get the address of MouseRegs
    Mov  AX,[SI+RegAX]      ;load each register in turn
    Mov  BX,[SI+RegBX]
    Mov  CX,[SI+RegCX]
    Mov  DX,[SI+RegDX]

    Mov  SI,[SI+Segmnt]     ;see what the segment is
    Or   SI,SI             ;is it zero?
    Jz   @F                ;yes, skip ahead and use default

    Cmp  SI,-1             ;is it -1?
    Je   @F                ;yes, skip ahead
    Mov  DS,SI             ;no, use the segment specified

@@:
    Push DS                ;either way, assign ES=DS
    Pop  ES
    Int  33h               ;call the mouse driver
```

```

Push SS                ;regain access to MouseRegs
Pop  DS

Mov  SI,MRegs          ;access MouseRegs again
Mov  [SI+RegAX],AX    ;save each register in turn
Mov  [SI+RegBX],BX
Mov  [SI+RegCX],CX
Mov  [SI+RegDX],DX

Ret                    ;return to BASIC
MouseInt Endp
End

```

Like DOSInt, this routine also uses a TYPE variable to define the various CPU registers that are needed by the mouse driver. However, fewer registers are needed simplifying the TYPE structure. You should define this TYPE variable as follows:

```

TYPE MouseType
  AX      AS INTEGER
  BX      AS INTEGER
  CX      AS INTEGER
  DX      AS INTEGER
  Segment AS INTEGER
END TYPE
DIM MouseRegs AS MouseTYPE

```

Since the mouse driver uses only these few registers, you can save a few bytes of DGROUP memory by using this subset TYPE instead of the full Registers TYPE that DOSInt requires. Notice the last component called Segment. Unlike the Mouse routine that Microsoft sells as an add-on library, MouseInt lets you specify a segment for passing far data to the mouse interrupt handler. For most mouse services you can leave the segment set to zero or -1. Either value tells MouseInt to use BASIC's default data segment. But some services that accept the address of incoming data also need to know the data's segment.

In the Microsoft version you have no choice but to use static data and near memory arrays. Obviously, this precludes being able to use BASIC PDS far strings with that interface routine. You would instead have to create a single fixed-length string or TYPE variable, just to force the data to reside in near memory. When calling MouseInt with a value other than zero or -1 for the segment, MouseInt loads both DS and ES with that value.

As with the collection of DOS file access routines, the following subprograms and functions can be added as a module to your program. Again, you should first make a copy of the source file that is included on the accompanying floppy disk, and then delete the demonstration portion of the program. This way, you can also run the original demonstration, and trace through it to test each of the mouse services. Of course, be sure to leave the commands that dimension the MouseRegs and MousePresent variables as being shared, and also the relevant DECLARE and DEFINT statements.

```

'MOUSE.BAS, demonstrates the various mouse services
DEFINT A-Z

```

```

'----- assembly language functions and subroutines
DECLARE FUNCTION PeekWord% (BYVAL Segment, BYVAL Address)
DECLARE SUB MouseInt (MouseRegs AS ANY)

'----- BASIC functions and subprograms
DECLARE FUNCTION Bin2Hex% (Binary$)
DECLARE FUNCTION MouseThere% ()
DECLARE FUNCTION WaitButton% ()
DECLARE SUB CursorShape (HotX, HotY, Shape())
DECLARE SUB HideCursor ()
DECLARE SUB MouseTrap (ULRow, ULCol, LRRow, LRCol)
DECLARE SUB MoveCursor (X, Y)
DECLARE SUB ReadCursor (X, Y, Buttons)
DECLARE SUB ShowCursor ()
DECLARE SUB TextCursor (FG, BG)

DECLARE SUB Prompt (Message$) 'used for this demo only

TYPE MouseType 'similar to DOS RegType
    AX AS INTEGER
    BX AS INTEGER
    CX AS INTEGER
    DX AS INTEGER
    Segment AS INTEGER
END TYPE

DIM SHARED MouseRegs AS MouseType
DIM SHARED MousePresent
REDIM Cursor(1 TO 32)

IF NOT MouseThere% THEN 'ensure a mouse is present
    PRINT "No mouse is installed" ' and initialize it if so
    END
END IF
CLS

DEF SEG = 0 'see what type of monitor
IF PEEK(&H463) <> &HB4 THEN 'if it's color
    ColorMon = -1 'remember that for later
    SCREEN 12 'this requires a VGA
    LINE (0, 0)-(639, 460), 1, BF 'paint a blue background
END IF

DIM Choice$(1 TO 5) 'display some choices
LOCATE 1, 1 'for something to point at FOR X = 1 TO 5
    READ Choice$(X)
    PRINT Choice$(X);
    LOCATE , X * 12
NEXT
DATA "Choice 1", "Choice 2", "Choice 3"
DATA "Choice 4", "Choice 5"

IF NOT ColorMon THEN 'if it's not color
    CALL TextCursor(-2, -2) 'select a text cursor
END IF

```

```

CALL ShowCursor
CALL Prompt("Point the cursor at a choice, and press _
a button.")

DO                                'wait for a button press
  CALL ReadCursor(X, Y, Button)
LOOP UNTIL Button
IF Button AND 4 THEN Button = 3 'for three-button mice

CALL Prompt("You pressed button" + STR$(Button) + _
" and the cursor was at location" + STR$(X) + ", " + _
STR$(Y) + " - press a button.")

IF ColorMon THEN                 'if it is a color monitor
  RESTORE Arrow                  ' load a custom arrow
  GOSUB DefineCursor
END IF
Dummy = WaitButton%

IF ColorMon THEN                 'the hardware can do it
  RESTORE CrossHairs            'set a cross-hairs cursor
  GOSUB DefineCursor
  CALL Prompt("Now the cursor is a cross-hairs, press _
a button.")
  Dummy% = WaitButton%
END IF

IF ColorMon THEN                 'now set an hour glass
  RESTORE HourGlass
  GOSUB DefineCursor
END IF

CALL Prompt("Now notice how the cursor range is _
restricted. Press a button to end.")
CALL MouseTrap(50, 50, 100, 100)
Dummy = WaitButton%

IF ColorMon THEN                 'restore to 640 x 350
  CALL MouseTrap(0, 0, 349, 639)
ELSE                              'use CGA bounds for mono!
  CALL MouseTrap(0, 0, 199, 639)
END IF

Dummy = MouseThere%             'reset the mouse driver
CALL HideCursor                 'and turn off the cursor
SCREEN 0                         'revert to text mode
END

DefineCursor:
FOR X = 1 TO 32                  'read 32 words of data
  READ Dat$                     'read the data
  Cursor(X) = Bin2Hex%(Dat$)    'convert to integer
NEXT

```



```
CALL CursorShape(Zero, Zero, Cursor())
RETURN
```

Arrow:

NOTES:

```
'The first group of binary data is the screen mask.
'The second group of binary data is the cursor mask.
'The cursor color is black where both masks are 0.
'The cursor color is XORed where both masks are 1.
'The color is clear where the screen mask is 1 and the
' cursor mask is 0.
'The color is white where the screen mask is 0 and the
' cursor mask is 1.
,
'Mouse cursor designs by Phil Cramer.
```

```
'--- this is the screen mask
```

```
DATA "1110011111111111"
DATA "1110001111111111"
DATA "1110000111111111"
DATA "1110000011111111"
DATA "1110000001111111"
DATA "1110000000111111"
DATA "1110000000011111"
DATA "1110000000001111"
DATA "1110000000000111"
DATA "1110000000000011"
DATA "1110000000000001"
DATA "1110000000011111"
DATA "1110001000011111"
DATA "1111111100001111"
DATA "1111111100001111"
DATA "1111111110001111"
```

```
'---- this is the cursor mask
```

```
DATA "0001100000000000"
DATA "0001010000000000"
DATA "0001001000000000"
DATA "0001000100000000"
DATA "0001000010000000"
DATA "0001000001000000"
DATA "0001000000100000"
DATA "0001000000010000"
DATA "0001000000001000"
DATA "0001000000000100"
DATA "00010000000111110"
DATA "0001001100100000"
DATA "0001110100100000"
DATA "0000000010010000"
DATA "0000000010010000"
DATA "000000001110000"
```

CrossHairs:

```
DATA "1111111101111111"
DATA "1111111101111111"
DATA "1111111101111111"
DATA "1111000000000111"
```

DATA "1111011101110111"
DATA "1111011101110111"
DATA "1111011111110111"
DATA "1000000111000000"
DATA "1111011111110111"
DATA "1111011101110111"
DATA "1111011101110111"
DATA "1111000000000111"
DATA "1111111101111111"
DATA "1111111101111111"
DATA "1111111101111111"
DATA "1111111111111111"

DATA "0000000010000000"
DATA "0000000010000000"
DATA "0000000010000000"
DATA "0000111111111000"
DATA "0000100010001000"
DATA "0000100010001000"
DATA "0000100000001000"
DATA "0111111000111111"
DATA "0000100000001000"
DATA "0000100010001000"
DATA "0000100010001000"
DATA "0000111111111000"
DATA "0000000010000000"
DATA "0000000010000000"
DATA "0000000010000000"
DATA "0000000000000000"

HourGlass:

DATA "1100000000000111"
DATA "1100000000000111"
DATA "1100000000000111"
DATA "1110000000001111"
DATA "1110000000001111"
DATA "1111000000011111"
DATA "1111100000111111"
DATA "1111110001111111"
DATA "1111110001111111"
DATA "1111100000111111"
DATA "1111000000011111"
DATA "1110000000001111"
DATA "1110000000001111"
DATA "1100000000001111"
DATA "1100000000001111"
DATA "1100000000001111"

DATA "0000000000000000"
DATA "0001111111110000"
DATA "0000000000000000"
DATA "0000111111110000"
DATA "0000100110100000"
DATA "0000010001000000"
DATA "0000001010000000"
DATA "0000000100000000"
DATA "0000000100000000"
DATA "0000000101000000"
DATA "0000011111000000"
DATA "0000110001100000"

```

DATA "000010000001000000"
DATA "000000000000000000"
DATA "000111111111100000"
DATA "000000000000000000"

```

```

FUNCTION Bin2Hex% (Binary$) STATIC 'binary to integer
    Temp& = 0
    Count = 0

    FOR X = LEN(Binary$) TO 1 STEP -1
        IF MID$(Binary$, X, 1) = "1" THEN
            Temp& = Temp& + 2 ^ Count
        END IF
        Count = Count + 1
    NEXT

    IF Temp& > 32767 THEN Temp& = Temp& - 65536
    Bin2Hex% = Temp&
END FUNCTION

```

```

SUB CursorShape (HotX, HotY, Shape()) STATIC
    IF NOT MousePresent THEN EXIT SUB

    MouseRegs.AX = 9
    MouseRegs.BX = HotX
    MouseRegs.CX = HotY
    MouseRegs.DX = VARPTR(Shape(1))
    MouseRegs.Segment = VARSEG(Shape(1))

    CALL MouseInt(MouseRegs)
END SUB

```

```

SUB HideCursor STATIC 'turns off the mouse cursor
    IF NOT MousePresent THEN EXIT SUB

    MouseRegs.AX = 2
    CALL MouseInt(MouseRegs)
END SUB

```

```

FUNCTION MouseThere% STATIC 'reports if a mouse is present
    MouseThere% = 0 'assume there is no mouse
    IF PeekWord%(Zero, (4 * &H33) + 2) = 0 THEN 'segment = 0
        EXIT FUNCTION ' means there's no mouse
    END IF

    MouseRegs.AX = 0
    CALL MouseInt(MouseRegs)
    MouseThere% = MouseRegs.AX
    IF MouseRegs.AX THEN MousePresent = -1
END FUNCTION

```

```

SUB MouseTrap (ULRow, ULColumn, LRRow, LRColumn) STATIC
    IF NOT MousePresent THEN EXIT SUB

    MouseRegs.AX = 7 'restrict horizontal movement
    MouseRegs.CX = ULColumn

```

```

MouseRegs.DX = LRColumn
CALL MouseInt(MouseRegs)

MouseRegs.AX = 8           'restrict vertical movement
MouseRegs.CX = ULRow
MouseRegs.DX = LRRow
CALL MouseInt(MouseRegs)
END SUB

SUB MoveCursor (X, Y) STATIC 'positions the mouse cursor
  IF NOT MousePresent THEN EXIT SUB

  MouseRegs.AX = 4
  MouseRegs.CX = X
  MouseRegs.DX = Y
  CALL MouseInt(MouseRegs)
END SUB

SUB Prompt (Message$) STATIC 'prints prompt message
  V = CSRLIN                'save current cursor position
  H = POS(0)
  LOCATE 30, 1              'use 25 for EGA SCREEN 9
  CALL HideCursor          'this is very important!
  PRINT LEFT$(Message$, 79); TAB(80);
  CALL ShowCursor          'and so is this
  LOCATE V, H              'restore the cursor
END SUB

SUB ReadCursor (X, Y, Buttons) 'returns cursor and button
  ' information
  IF NOT MousePresent THEN EXIT SUB

  MouseRegs.AX = 3
  CALL MouseInt(MouseRegs)

  Buttons = MouseRegs.BX AND 7
  X = MouseRegs.CX
  Y = MouseRegs.DX
END SUB

SUB ShowCursor STATIC          'turns on the mouse cursor
  IF NOT MousePresent THEN EXIT SUB

  MouseRegs.AX = 1
  CALL MouseInt(MouseRegs)
END SUB

SUB TextCursor (FG, BG) STATIC
  IF NOT MousePresent THEN EXIT SUB

  MouseRegs.AX = 10
  MouseRegs.BX = 0
  MouseRegs.CX = &HFF
  MouseRegs.DX = 0

  IF FG = -1 THEN             'maintain FG as the cursor moves?
    MouseRegs.CX = MouseRegs.CX OR &HF00

```

```

ELSEIF FG = -2 THEN      'invert FG as the cursor moves?
  MouseRegs.CX = MouseRegs.CX OR &H700
  MouseRegs.DX = &H700
ELSE
  MouseRegs.DX = 256 * (FG AND &HFF)
END IF

IF BG = -1 THEN          'maintain BG as the cursor moves?
  MouseRegs.CX = MouseRegs.CX OR &HF000
ELSEIF BG = -2 THEN      'invert BG as the cursor moves?
  MouseRegs.CX = MouseRegs.CX OR &H7000
  MouseRegs.DX = MouseRegs.DX OR &H7000
ELSE
  MouseRegs.DX = MouseRegs.DX OR Temp
  Temp = (BG AND 7) * 16 * 256
END IF

CALL MouseInt(MouseRegs)
END SUB

FUNCTION WaitButton% STATIC      'waits for a button press
  IF NOT MousePresent THEN EXIT FUNCTION

  X! = TIMER                    'pause to allow releasing
  WHILE X! + .2 > TIMER          ' the button
  WEND

  DO                            'wait for a button press
    CALL ReadCursor(X, Y, Button)
  LOOP UNTIL Button

  IF Button AND 4 THEN Button = 3 'for three-button mice
  WaitButton% = Button           'assign the function
END FUNCTION

```

This program begins by declaring all of the support functions, and then defines and dimensions the MouseRegs TYPE variable. The integer array is used to hold the custom graphics cursor shape information, which the CursorShape routine requires. The remainder of the program illustrates how to use the various mouse routines in your own programs.

Determining if a Mouse is Present

The first function is MouseThere, which serves two important purposes: The first is to determine if a mouse is present. The second purpose of MouseThere is to initialize the mouse driver to its default parameters. This lets you be sure that the mouse color, shape, and other parameters are in a known state. Resetting the mouse is strongly recommended because some programs do not bother to reset the mouse when they are finished.

Although there is a mouse service to determine if the driver is installed, you must also perform an additional test to prevent problems with early computers running DOS version 2. The problem arises because these computers leave the mouse interrupt (&H33) undefined if no mouse is present, and calling this interrupt is likely to make the PC crash.

As you already know, the interrupt vector table in low memory holds the segment and address for every interrupt service routine that is present in the PC. But who puts those addresses into the interrupt vector table? All of the BIOS interrupt addresses are assigned by the BIOS as part of the power-up code in your PC's ROM. Likewise, DOS installs the addresses it needs while it is being loaded from disk.

The BIOS in modern computers assigns every interrupt vector to a valid address, even those that it (the BIOS) does not use. The code pointed to by the unused interrupts is an assembly language Iret (Interrupt Return) instruction. So if no other routine is servicing that interrupt, calling it merely returns with no change to the register contents. But early computers and early versions of DOS ignored Interrupt &H33, and left the values in that vector address set to zero. Calling the code at address zero is guaranteed to fail, since address zero holds other addresses and not executable code. Therefore, to safely detect the presence of a mouse requires first looking in low memory, to ensure that the interrupt address there is valid.

It is important to understand that you must use MouseThere once at the start of your program, before any of the other mouse routines will work. All of the mouse routines check the global variable MousePresent before calling MouseInt, and do nothing if it is zero. This safety mechanism lets you freely call the various mouse services without regard to whether or not a mouse is installed, to avoid the DOS 2 problem described earlier. Thus, the same program statements can accommodate a mouse if one is present or not, without requiring many separate IF tests.

For example, you will probably want to write programs that use a mouse if one is present, but don't require it. If you had to have a separate block of code for each case, your program would be much larger and slower than necessary. Therefore, you can simply call these mouse routines whether or not a mouse is present. The code fragment that follows shows a simple example of this in context.

```
PRINT "Press a key or mouse button to continue: ";
DO
  Temp$ = INKEY$
  CALL ReadCursor(X, Y, Buttons)
LOOP UNTIL LEN(INKEY$) OR Buttons
PRINT "Thank you."
```

If MouseThere determined that no mouse was present when it was called earlier, then ReadCursor will do nothing and return no values. Of course, you will have to check for mouse events and act on them, but these can be handled within the same blocks of code that also handle keyboard input.

Once the program knows that a mouse is in fact present, it checks to see if the display adapter is color or monochrome. A color monitor supports more mouse options such as changing the shape of the mouse cursor. In this case the program assumes that you have a VGA adapter. If you have only an EGA, simply change the SCREEN 12 statement to SCREEN 9. You will also have to change the LOCATE command in the Prompt subprogram to use line 25 instead of line 30. Although the cursor shape can be altered with CGA and Hercules adapters, those are not accommodate here.

Once the screen display mode is set, a filled box is drawn covering the entire screen, to create an attractive blue background. You should be aware that the drivers included come with many older, inexpensive clone mouse devices do not support the EGA and VGA display modes. This is not a limitation with the mouse hardware; rather, the problem lies in the driver software. Fortunately, the MOUSE.COM and MOUSE.SYS drivers that Microsoft includes with BASIC work with most brands of mouse. Furthermore, you are allowed to distribute those drivers with your own programs, as long as you include an appropriate copyright notice. See the license agreement that came with your version of BASIC for more information on displaying the Microsoft copyright.

Controlling the Text Cursor

After reading and displaying a list of sample choices that serve as a menu, the program again checks to see which type of adapter is present. If it is monochrome, then a custom text cursor is defined using the TextCursor routine. This routine is appropriate for both monochrome and color adapters, and offers several useful options that let you control fully how the foreground and background colors will appear. Also, an initial call to TextCursor is needed with some non-Microsoft mouse drivers to ensure that the cursor is displayed after calling ShowCursor.

TextCursor expects two parameters to control the cursor's foreground and background colors. If a positive value is given for either parameter, then that is the color the mouse cursor assumes as it travels around the screen. For example, if you use a color combination of 0, 4 the character under the mouse cursor will be shown in black on a red background. It is important to understand that the normal mouse cursor color is actually the character's background color. The foreground indicates what color the text is to become as the cursor passes over it.

Using a value of -1 for either parameter tells the mouse driver to leave that portion of the color alone when the cursor is positioned over a character. If you use a color combination of 7, -1 the text under the mouse cursor will be shown in white and the background will be unchanged. Of course, if both the foreground and background are set to -1, the cursor will never be visible.

A value of -2 causes that color portion to be inverted using an XOR process as the cursor moves around the screen. That is, white becomes black, green turns to magenta, and blue is translated to brown. Although a value of -2 for the background guarantees that the cursor is always visible, it can also be distracting to see the mouse cursor color change constantly when the screen itself uses many colors. If you want to experiment with the various TextColor options, add remarking apostrophes to deactivate the three statements after the line `IF PEEK(&H463) <> &HB4 THEN` near the beginning of the program.

The ShowCursor subprogram simply tells the mouse drive to make the mouse cursor visible, in much the same way LOCATE , , 1 option does with the normal screen cursor. The companion routine HideCursor turns the mouse cursor off again. These are very simple routines that do not require much explanation; however, please understand that until you turn the cursor on explicitly it remains hidden.

As a rule, you also want to ensure that the cursor is turned off before you end your program and return to DOS.

There is one irritating quirk about how the mouse driver keeps track of whether the mouse cursor is currently visible or not. When you use the statement `LOCATE , , 0` to turn off the regular text cursor, the BIOS remembers that it is off. And if you subsequently use the same statement again the request is ignored. The mouse driver, on the other hand, remembers how many times you called `HideCursor` and requires a corresponding number of calls to `ShowCursor` before it becomes visible. However, the reverse is not true. If you turn on the cursor, say, five times in a row, only one call to `HideCursor` is needed to turn it off.

Reading the Mouse Buttons and Cursor Position

The next mouse routine is called `ReadCursor`, and it calls the service that returns both the current mouse cursor position and also which buttons are currently pressed. Notice that the X and Y values returned assume graphics pixel coordinates even when the display screen is in text mode. Therefore, when a monochrome display adapter is being used, the values returned range from 0 to 639 horizontally (X), and 0 through 199 vertically (Y). These are the same values you would receive when in CGA black and white screen mode 2. When in graphics mode, the X and Y values are based on the current `SCREEN` setting. For example, in EGA screen mode 9, the returned value for X ranges from 0 through 639, and Y is between 0 and 349.

When your program is in text mode (`SCREEN 0`), the current X and Y cursor location is based on the upper-left corner of the mouse cursor box. Therefore, the actual horizontal range (X) is usually returned between 0 and 632 to account for a box width of 8 pixels. The vertical location (Y) ranges from 0 to 192 for the same reason: If the bottom of the cursor is at the bottom of the screen, then the top is eight pixels higher. In graphics mode you are allowed to establish any portion of the mouse cursor as being the hot spot, and this is discussed below in the section `Changing the Mouse Cursor Shape`.

The buttons are returned bit coded—the lowest bit is set if button 1 is pressed, and the next bit is set when the second button is pressed. If a mouse has three buttons, the third bit may also be set to indicate that. Isolating which bit or combination of bits is set is done using the AND logic operator. If `Button AND 1` is non-zero then the first button is pressed. Similarly, `Button AND 2` means the second button is being pressed. However, testing for button 3 requires a value of 4, since that is the value of the third bit. The program fragment that follows shows this in context, and you can press one or more buttons at a time.

```
DO
  PRINT "Press Ctrl-Break to end."
  CALL ReadCursor(X, Y, Button)

  LOCATE 10, 1
  IF Button AND 1 THEN
    PRINT "BUTTON 1"
  ELSE
    PRINT "      "
```



```

END IF

LOCATE 10, 11
IF Button AND 2 THEN
    PRINT "BUTTON 2"
ELSE
    PRINT "      "
END IF

LOCATE 10, 21
IF Button AND 4 THEN
    PRINT "BUTTON 3"
ELSE
    PRINT "      "
END IF
LOOP

```

Besides the ReadCursor routine which returns the cursor position and button status, I have also included a related function called WaitButton. If your program will be waiting for a button and needs to know which button was pressed, WaitButton does this using fewer bytes of compiler-generated code. Since there are no passed parameters only five bytes are needed to call WaitButton, compared to 17 needed to call ReadCursor. WaitButton simply waits in an empty loop until a button is pressed, and then reports which button it was.

Changing the Mouse Cursor Shape

The CursorShape routine lets you change the size and shape of the mouse cursor when the display is in graphics mode. The mouse driver routine that is called requires the address of a block of memory 32 words long that holds the new shape and color information. The data in this memory block is organized into two sections. The first 16 words hold what is called the *screen mask*, and the second 16 words hold the *cursor mask*.

The bits in these masks interact to change the way the foreground and background colors on the screen change as the cursor passes over them. The method used by the mouse driver to control the cursor shape and colors is very complex, and the examples and discussions in Microsoft's documentation do little to assist the programmer. Therefore, I have provided a simple mechanism that lets you draw the cursor shape using a series of BASIC DATA statements.

Using this method it is easy to control each individual pixel in the mouse cursor, and determine if it is white, black, or transparent. When the bits in both the screen and cursor masks are both zero, the cursor will be black. And when the bits in both masks are set to 1, the color is XORed (reversed) at that pixel position. If a screen mask bit is 1 and its corresponding bit in the cursor mask is 0, the cursor is transparent. Reversing this to make the screen mask 0 and the cursor mask 1 makes the cursor white at that position. Thus, you can create nearly any shape for the mouse cursor, and a wide variety of interesting color effects.

If your needs are modest or to minimize the number of DATA statements, you can define only the cursor mask and use -1 for the first 16 elements in the array by changing that portion of the program like this:

```
DefineCursor:
FOR X = 1 TO 32                'read 32 words of data
  IF X < 17 THEN              'set first 16 elements = -1
    Cursor(X) = -1
  ELSE                          'and for the second 16
    READ Dat$                  ' read the data and then
    Cursor(X) = Bin2Hex%(Dat$) ' convert to an integer
  END IF
NEXT

DATA "110000000000000000"      'use only 16 DATA items
DATA "111000000000000000"      ' in this section
:
```

The other two parameters required by CursorShape are the X and Y cursor hot spots. When you call ReadCursor to return the current mouse cursor location and button information, the X and Y position returned identifies a single pixel on the screen. Which pixel within the mouse cursor that is reported is the cursor hot spot. When you use an arrow cursor shape, the hot spot is typically the tip of the arrow. This is located in the upper left corner of the cursor box and is identified as location 0, 0. However, you can also make any other portion of the cursor the hot spot. For simplicity, the GOSUB routine at the DefineCursor label always uses 0, 0. However, the cross hairs cursor really should use the values 8, 8 to set the hot spot at the center of the block.

Controlling the Mouse Cursor Position and Range

The MoveCursor routine lets you set a new position for the mouse cursor, and it too expects pixel values even when the screen is in text mode. Although MoveCursor is not demonstrated in this program, it is included in the interest of completeness.

The final mouse subprogram included lets you restrict the range of mouse cursor travel, and it is called—appropriately enough—MouseTrap. You pass the upper-left and lower-right boundaries to MouseTrap, and it in turns passes those values on to the mouse driver. Internally, the mouse driver lets you restrict the range for horizontal and vertical motion independently. But for simplicity this routines requires both sets of values at one time.

Like the services that ReadCursor and MoveCursor call, these services also expect the cursor bounds to be given as pixels even when in text mode. Also, notice that the mouse driver always forces the cursor into the restricted region for you. That is, if the cursor is in the upper-left corner and you call MouseTrap forcing it to stay inside the bottom half of the screen, it will be moved to the top of that region.

Be aware that MouseTrap is also required if you plan to use the 43 or 50-line EGA and VGA text modes. By default, the mouse driver assumes that a text screen has only 25 lines, and will not normally let the mouse cursor be placed below that line. If you have used `WIDTH , 50` to put the screen into the 50-line mode, the mouse cursor will not be allowed below line 25. Therefore, you must use MouseTrap to increase the allowable cursor region beyond the default range. Also be aware that using values larger than the current screen dimensions let the mouse disappear off the bottom of the screen, or wrap around past the right edge and reappear on the left side.

Accessing the Mouse Driver

All of the mouse routines considered so far are comprised of a simplified interface to the mouse driver through the MouseInt routine. MouseInt lets you access any service supported by the mouse driver, including those that I have not described here. Similar to the various DOS and BIOS services, the mouse driver expects a service number in the AX register. The other registers contain the various expected parameters and returned information, and they vary from service to service.

There are no errors returned by the mouse driver, so no mechanism is needed to handle errors. For example, if you tell the mouse driver to position the cursor off the top edge of the screen, it simply ignores you.

Unfortunately, discussing every possible mouse service goes beyond what I could ever hope to include in a book about BASIC. If you want to learn more about the services that are available to you, I recommend purchasing a good technical reference such as the *Microsoft Mouse Programmer's Reference*. Other mouse manufacturers also publish their own technical manuals, and make them available to the public for a small charge. Thankfully, all of the mouse services are consistent across brands, although some brands include more features than defined by Microsoft. Unless you write programs only for your own use, you should avoid relying on services that are specific to a single manufacturer.

Accessing Expanded Memory

The last set of routines I will present show how you can use interrupts to access an expanded memory (EMS) driver. Expanded memory has been available for many years, and it provides a way to exceed the normal 640K RAM barrier imposed by the 8088 microprocessors. Newer computers that use an 80286 or later processors can use what is called Extended Memory (XMS), and this type of memory will eventually become the standard way for all computers in the future to access more than 1MB of memory. Unfortunately, accessing the extended memory beyond 1MB on an 80286-based PC is complicated by a design deficiency in that CPU chip. Many people are confused about the difference between Expanded and Extended memory, so perhaps a brief explanation is in order.

Extended memory is a single contiguous block that starts at address zero and extends through the highest address available, based on the amount of memory that is present in a PC. *Expanded memory*,

on the other hand, is more complex, and uses a technique called *bank switching*. With bank switching, a large amount of memory (up to 16 megabytes) is made available to the CPU in 16K blocks. Each of these blocks is called a page, and only four of them can be accessed at one time. Thus, the term bank switching is appropriate because various banks of far memory are switched in and out of a near memory address space.

The EMS standard requires a 64K contiguous area of near memory within the 1MB addressable range to be reserved for use by the EMS driver as a *page frame*. On my own PC the 64k address range from &HE000:0000 through &HE000:FFFF is not used for any other purpose, and is therefore available for use by an EMS driver. At any given time, the four 16K blocks of memory within this segment can be connected to memory that lies outside of the 1MB normal address range.

Hardware plug-in EMS boards such as the Intel Above Board contain their expanded memory on the board itself. EMS emulator software instead converts the Extended memory on computers so equipped to be accessible through the 64K segment within the EMS page frame. This is achieved through hardware switches that allow any area of memory to be remapped to any other range of addresses. In either case, however, Expanded memory is made available to an application one page at a time as near memory.

Each of the four 16K near memory pages in the EMS page frame are called *physical pages*, because they reside in physical memory that can be accessed directly by the CPU. However, many pages of far EMS memory are available—up to four at a time—and these are called *logical pages*. This is shown graphically in Figure 11-4.

Here, physical page 0 is connected to logical page 38 in expanded memory, physical page 1 to logical page 45, and so forth. Whenever a program wants to access a particular logical page in expanded memory, it calls the EMS driver telling it to map that page to one of the four physical pages in the page frame segment. Then, the EMS logical page can be accessed at the near memory address within the page frame.

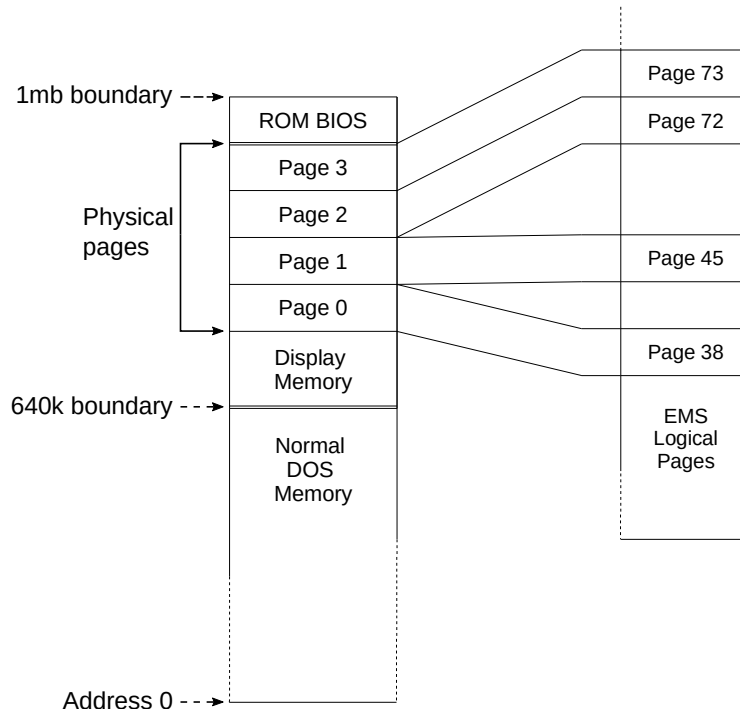


Figure 11-4: How EMS logical pages in far memory are mapped onto physical pages in conventional memory.

For simplicity, all of the routines provided here to handle Expanded memory use physical page 0 only. Since these routines merely copy array data back and forth between conventional and Expanded memory, the data can be copied in blocks of 16K and there is no need to have to map multiple pages simultaneously. Therefore, these routines always map physical page 0 to whichever logical page needs to be accessed, and then copy the data in that page only.

EMS Services

As with the DOS services accessed through Interrupt &H21, the EMS driver also uses handles to identify which data you are working with. When memory is allocated using EMS Interrupt &H67, you tell the driver how many 16K pages you are requesting, and if there is sufficient memory available it returns a handle. It should come as no surprise to learn that these parameters are passed using the CPU registers. Also like DOS and the BIOS, the EMS driver expects a service number in the AH Register. For example, the service that requests memory is specified with AH set to &H43.

To minimize the amount of code that is added to your programs, I have created a short assembly language subroutine called EMSInt that replaces the Interrupt routine included with BASIC. As with DOSInt and MouseInt, this routine lets you pass only the parameters that are actually needed, to reduce the amount of compiler-generated code. EMSInt needs access only to the AX, BX, CX, and DX registers, so these are the only components in the EMSType TYPE structure shown below.

```

TYPE EMSType
  AX AS INTEGER
  BX AS INTEGER
  CX AS INTEGER
  DX AS INTEGER
END TYPE

```

Unlike BASIC's Interrupt routine that has to deal with three parameters and code to generate any interrupt number, EMSInt itself is relatively simple:

```

;EMSINT.ASM

.Model Medium, Basic

EMSRegs Struc
  RegAX DW ?
  RegBX DW ?
  RegCX DW ?
  RegDX DW ?
EMSRegs Ends

.Code

EMSInt Proc Uses SI, ERegs:Word
  Mov  SI,ERegs          ;get the address of EMSRegs
  Mov  AX,[SI+RegAX]    ;load each register in turn
  Mov  BX,[SI+RegBX]
  Mov  CX,[SI+RegCX]
  Mov  DX,[SI+RegDX]

  Int  67h              ;call the EMS driver

  Mov  SI,ERegs        ;access EMSRegs again
  Mov  [SI+RegAX],AX   ;save each register in turn
  Mov  [SI+RegBX],BX
  Mov  [SI+RegCX],CX
  Mov  [SI+RegDX],DX

  Ret                  ;return to BASIC
EMSInt Endp
End

```

If you plan to use the mouse and EMS routines in the same program, you could use the MouseRegs variable for both and ignore the Segment portion when call EMSInt.

The program that follows combines a demonstration portion and a collection of subprograms and functions. Notice that like the various mouse services, you must query EMSThere to ensure that an EMS driver is loaded before any of the other routines can be used.

```

'EMS.BAS, demonstrates the EMS memory services

DEFINT A-Z

DECLARE FUNCTION Compare% (BYVAL Seg1, BYVAL Adr1, BYVAL Seg2, _
  BYVAL Adr2, NumBytes)

```

```

DECLARE FUNCTION EMSErrorMessage$ (ErrNumber)
DECLARE FUNCTION EMSError% ()
DECLARE FUNCTION EMSFree& ()
DECLARE FUNCTION EMSThere% ()
DECLARE FUNCTION PeekWord% (BYVAL Segment, BYVAL Address)

DECLARE SUB EMSInt (EMSRegs AS ANY)
DECLARE SUB EMSStore (Segment, Address, ElSize, NumEls, Handle)
DECLARE SUB EMSRetrieve (Segment, Address, ElSize, NumEls, Handle)
DECLARE SUB MemCopy (BYVAL FromSeg, BYVAL FromAdr, BYVAL ToSeg, _
    BYVAL ToAdr, NumBytes)

TYPE EMSType                                'similar to DOS Registers
    AX    AS INTEGER
    BX    AS INTEGER
    CX    AS INTEGER
    DX    AS INTEGER
END TYPE

DIM SHARED EMSRegs AS EMSType
DIM SHARED ErrCode
DIM SHARED PageFrame

CLS
IF NOT EMSThere% THEN                        'ensure EMS is present
    PRINT "No EMS is installed"
END
END IF

PRINT "This computer has"; EMSFree&;
PRINT "kilobytes of EMS available"

REDIM Array#(1 TO 20000)
FOR X = 1 TO 20000
    Array#(X) = X
NEXT

CALL EMSStore(VARSEG(Array#(1)), VARPTR(Array#(1)), 8, 20000, Handle)
IF EMSError% THEN
    PRINT EMSErrorMessage$(EMSError%)
END
END IF

REDIM Array#(1 TO 20000)
CALL EMSRetrieve(VARSEG(Array#(1)), VARPTR(Array#(1)), 8, 20000, Handle)
IF EMSError% THEN
    PRINT EMSErrorMessage$(EMSError%)
END
END IF

FOR X = 1 TO 20000                            'prove it worked
    IF Array#(X) <> X THEN PRINT ".";
NEXT
END

FUNCTION EMSErrorMessage$ (ErrNumber) STATIC
    SELECT CASE ErrNumber
        CASE 128
            EMSErrorMessage$ = "Internal error"
    
```

```

CASE 129
    EMSErrorMessage$ = "Hardware malfunction"
CASE 131
    EMSErrorMessage$ = "Invalid handle"
CASE 133
    EMSErrorMessage$ = "No handles available"
CASE 135, 136
    EMSErrorMessage$ = "No pages available"
CASE ELSE
    IF PageFrame THEN
        EMSErrorMessage$ = "Undefined error: " + STR$(ErrNumber)
    ELSE
        EMSErrorMessage$ = "EMS not loaded"
    END IF
END SELECT
END FUNCTION

FUNCTION EMSError% STATIC
    Temp& = ErrCode
    IF Temp& < 0 THEN Temp& = Temp& + 65536
    EMSError% = Temp& \ 256
END FUNCTION

FUNCTION EMSFree& STATIC
    EMSFree& = 0 'assume failure
    IF PageFrame = 0 THEN EXIT FUNCTION

    EMSRegs.AX = &H4200
    CALL EMSInt(EMSRegs)
    ErrCode = EMSRegs.AX 'save possible error from AH

    IF ErrCode = 0 THEN EMSFree& = EMSRegs.BX * 16
END FUNCTION

SUB EMSRetrieve (Segment, Address, ElSize, NumEls, Handle) STATIC
    IF PageFrame = 0 THEN EXIT SUB

    LocalSeg& = Segment 'use copies we can change
    LocalAdr& = Address

    BytesNeeded& = NumEls * CLNG(ElSize)
    PagesNeeded = BytesNeeded& \ 16384
    Remainder = BytesNeeded& MOD 16384
    IF Remainder THEN PagesNeeded = PagesNeeded + 1

    NumBytes = 16384 'assume we're copying a
                    ' complete page
    ThisPage = 0 'start copying to page 0

    FOR X = 1 TO PagesNeeded 'copy the data
        IF X = PagesNeeded THEN 'watch out for last page
            IF Remainder THEN NumBytes = Remainder
        END IF

        IF LocalAdr& > 32767 THEN 'handle segment boundaries
            LocalAdr& = LocalAdr& - &H8000&
            LocalSeg& = LocalSeg& + &H800
            IF LocalSeg& > 32767 THEN

```



```

        LocalSeg& = LocalSeg& - 65536
    END IF
END IF

EMSRegs.AX = &H4400      'map physical page 0 to the
EMSRegs.BX = ThisPage   ' current logical page
EMSRegs.DX = Handle     ' for the given handle
CALL EMSInt(EMSRegs)    'then copy the data there
ErrCode = EMSRegs.AX    'save possible error from AH
IF ErrCode THEN EXIT SUB
CALL MemCopy(PageFrame, Zero, CINT(LocalSeg&), CINT(LocalAdr&), _
NumBytes)

    ThisPage = ThisPage + 1
    LocalAdr& = LocalAdr& + NumBytes
NEXT

EMSRegs.AX = &H4500      'release memory service
EMSRegs.DX = Handle
CALL EMSInt(EMSRegs)
ErrCode = EMSRegs.AX    'save possible error
END SUB

SUB EMSStore (Segment, Address, ElSize, NumEls, Handle) STATIC
    IF PageFrame = 0 THEN EXIT SUB

    LocalSeg& = Segment      'use copies we can change
    LocalAdr& = Address

    BytesNeeded& = NumEls * CLNG(ElSize)
    PagesNeeded = BytesNeeded& \ 16384
    Remainder = BytesNeeded& MOD 16384
    IF Remainder THEN PagesNeeded = PagesNeeded + 1

    EMSRegs.AX = &H4300      'allocate memory service
    EMSRegs.BX = PagesNeeded
    CALL EMSInt(EMSRegs)

    ErrCode = EMSRegs.AX    'save possible error from AH
    IF ErrCode THEN EXIT SUB
    Handle = EMSRegs.DX     'save the handle returned

    NumBytes = 16384        'assume we're copying a
                            ' complete page
    ThisPage = 0           'start copying to page 0

    FOR X = 1 TO PagesNeeded 'copy the data
        IF X = PagesNeeded THEN 'watch out for last page
            IF Remainder THEN NumBytes = Remainder
        END IF

        IF LocalAdr& > 32767 THEN 'handle segment boundaries
            LocalAdr& = LocalAdr& - &H8000&
            LocalSeg& = LocalSeg& + &H800
            IF LocalSeg& > 32767 THEN
                LocalSeg& = LocalSeg& - 65536
            END IF
        END IF
    NEXT X

    EMSRegs.AX = &H4400      'map physical page 0 to the

```

```

    EMSRegs.BX = ThisPage      ' current logical page
    EMSRegs.DX = Handle       ' for the given handle
    CALL EMSInt(EMSRegs)      'then copy the data there
    ErrCode = EMSRegs.AX      'save possible error from AH
    IF ErrCode THEN EXIT SUB
    CALL MemCopy(CINT(LocalSeg&), CINT(LocalAdr&), PageFrame, Zero, NumBytes)

    ThisPage = ThisPage + 1
    LocalAdr& = LocalAdr& + NumBytes
NEXT
END SUB

FUNCTION EMSThere% STATIC
    EMSThere% = 0             'assume the worst
    DIM DevName AS STRING * 8
    DevName = "EMMXXXX0"     'search for this below

    '---- Try to find the string "EMMXXXX0" at offset 10 in the EMS handler.
    '      If it's not there then EMS cannot possibly be installed.
    Int67Seg = PeekWord%(0, (&H67 * 4) + 2)
    IF NOT Compare%(Int67Seg, 10, VARSEG(DevName$), VARPTR(DevName$), 8) THEN
EXIT FUNCTION
    END IF

    EMSRegs.AX = &H4100       'get Page Frame Segment service
    CALL EMSInt(EMSRegs)
    ErrCode = EMSRegs.AX     'save possible error from AH

    IF ErrCode = 0 THEN
        EMSThere% = -1
        PageFrame = EMSRegs.BX
    END IF
END FUNCTION

```

EMS.BAS begins by declaring all of the subprograms and functions that it uses, as well as the EMSType structure. The three shared variables are used by the various procedures, and should not be removed when you delete the demo portion to create a reusable module.

Determining if EMS is Present

The first function used is EMSThere, which reports if an EMS driver is loaded and operative. EMSThere begins by assuming that an EMS driver is not loaded, and assigns a function output value of 0. Then it attempts to find the device name "EMMXXXX0" in the header portion of the EMS device driver. Like the MouseThere function that checked the interrupt vector table for a non-zero segment value, this preliminary check is also needed to prevent a system lockup on older computers running DOS version 2.

To search for this string EMSThere uses PeekWord to retrieve the segment for Interrupt &H67, and then looks at the eight bytes at offset 10 within that segment. If the Compare function finds the unique identifying string, it knows that the driver is loaded and it is safe to invoke Interrupt &H67. Service &H41 returns either -1 in AX if the driver is active, or 0 if it is not. This service also returns the page

frame segment the driver is using in near memory, and EMSThere saves this value in the shared variable PageFrame for access by the other routines.

Determining Available EMS Memory

The second function, EMSFree, returns the number of 16K EMS pages that are available to your program. The remainder of the demonstration simply dimensions a 20,000 element double precision array, and then saves it to expanded memory. Because this array exceeds 64K, you must start BASIC with the /ah command line switch. Otherwise you will receive a "Subscript out of range" error message.

EMSFree uses function &H42 to ask the EMS driver for the number of free pages, and the driver returns the page count in BX. Although it is not shown here, service &H42 also returns the total number of pages in the DX register. Therefore, you could easily create a TotalPages function from a copy of EMSFree by changing the line that assigns the function output to instead be `IF ErrCode = 0 THEN TotalPages& = EMSRegs.DX * 16.`

Storing and Retrieving Data

The actual storing and retrieving of data to and from Expanded memory is fairly complicated, because of the need to map different logical pages to physical page zero. Although Figure 11-6 shows a single group of logical pages, the EMS driver really maintains a separate series of logical pages for each active handle.

EMSStore and EMSRetrieve store and retrieve data in Expanded memory respectively, and both of these subprograms are designed to accommodate huge arrays larger than 64k. Therefore, additional work is needed to calculate new segment values as each 16K portion has been processed.

As with all of the EMS procedures shown here, EMSStore begins by verifying that EMSThere has already been invoked, and that a valid page frame segment has been obtained. The next step is to make long integer copies of the incoming segment and address parameters. Because of the segment arithmetic that is performed later in the routine, long integers are needed to allow values greater than 32,767 to be compared. Equally important, a routine should never alter incoming parameters unless they also return information or such changes are expected.

Next, EMSStore determines the total number of bytes of EMS storage that are needed, and from that calculates the total number of 16K pages. Because the EMS driver allocates entire pages only, an odd number of bytes requires an entire additional page. BASIC's MOD function is used for this, and if the result is non-zero, the TotalPages variable is incremented.

Once the number of pages is known, service &H43 is called to allocate the Expanded memory. The remainder of the procedure walks through the array data in 16K increments, mapping physical page zero to the next logical page in sequence. Note the code that tests the current address to see if it is

within 32K of spanning a segment boundary. In that case, the address is dropped by 32K, and the segment is increased by an equivalent amount. Because each new segment starts 16 bytes higher than the previous one, $32K \setminus 16$ is added to LocalSeg& rather than a full 32K.

After the array is stored in EMS, it is redimensioned in the demonstration and then retrieved using the EMSRetrieve subprogram. EMSRetrieve is nearly identical to EMSStore, except it copies from EMS to the array, and releases memory when it is finished rather than claim it at the beginning. The final step in the demonstration is to examine the value in each element, to prove that the array was restored correctly.

Detecting EMS Errors

The EMSError function retrieves the current value of ErrCode, and manipulates it into a form usable by your programs. EMS errors are returned in the AH register, which requires dividing by 256 to derive a single byte value. But since EMS error numbers start at 128, the value returned in AX appears negative to BASIC programs which treat all integers as being signed. This is why a long integer is used initially and then converted to a positive value, before dividing to produce the final result.

The EMSErrorMessage function can be used to display an appropriate message if an error is detected. The incoming error code is filtered through a series of CASE statements, based on the error values defined by the EMS specification.

Suggested Enhancements

The routines presented herein provide a limited set of services for accessing Expanded memory. However, there are several improvements you can make, and a few other uses that I have not shown. If you are using BASIC PDS [or VB/DOS], one useful enhancement you can add is to change the subprograms and functions to receive their parameters by value using the BYVAL option. In fact, this can also be done with the DOS and mouse routines, to minimize the amount of code the BASIC compiler adds to your final executable program.

Although this demonstration shows storing array data only, you can also use these routines to store and retrieve text and graphics screens. This is much quicker than saving them to disk, as was shown in Chapter 6. For example, to save a 25 line by 80 column color text screen in Expanded memory you would use the appropriate segment and address like this:

```
CALL EMSStore(&HB800, 0, 1, 4000, Handle)
CALL EMSRetrieve(&HB800, 0, 1, 4000, Handle)
```

Just as you can cause problems by failing to close DOS handles during the development of a program, the same problem can happen with an EMS driver. Unfortunately, it is not as easy to know which handle numbers are still open if you have not kept track of them yourself manually. DOS issues its handles using a sensible series of sequential numbers. This is not necessarily the case with EMS

handles. The EMM386.EXE driver provided by Microsoft does issue sequential handles, starting with handle 1. But many drivers use other starting values, some work from high numbers backwards, and yet others use a handle number sequence that is not in order.

Finally, to learn about all of the possible EMS services you need a good reference. Although the primary services are shown here, there are several others you may find useful. For example, service &H46 lets you retrieve the EMS version number, and service &H4C lets you see how many pages are currently allocated for a given handle. The EMS driver version can be valuable, because newer drivers offer more features which you may want to take advantage of. Ray Duncan's book *Advanced MS-DOS* mentioned earlier is one good source, and it lists each EMS service and the possible errors that can be returned.

Summary

In this chapter you learned how BASIC—and indeed, all languages—use interrupts to communicate with the operating system. You learned what interrupts are and how to access them, and how the CPU registers are used to communicate information between your program and the interrupt handler being invoked. You also learned how some of the two-byte registers can be treated as two one-byte registers, which requires multiplying and dividing to access those portions individually.

A number of complete programs were presented showing how to access the BIOS, DOS, the mouse driver, and Expanded memory. In the section on BIOS interrupts, examples were given that showed how to simulate pressing the PrtSc key, and also how to call the video service that clears or scrolls only a portion of the display screen.

The DOS examples included a complete set of subroutines to replace BASIC's file handling statements. One advantage gained by bypassing BASIC is to read and write large amounts of data at one time. Another is to avoid the need for ON ERROR in certain programming situations. Although calling the DOS services directly can be beneficial in many cases, it also requires more work on your part. However, some services cannot be accessed using BASIC alone, such as reading file and directory names, or determining a file's attribute. Where BASIC employs string descriptors to know how long a string is, DOS instead uses a CHR\$(0) zero byte to mark the end.

The mouse and Expanded memory discussions described how those interrupt services are accessed, and provided practical advice and warnings where appropriate. Although a large number of interrupt routines were described, there is a practical limit to how much information can be provided here. In particular, you will need a separate reference manual that describes the details of each interrupt service routine in depth.

In the next and final chapter you will learn how to program in assembly language, and how to add assembly language routines to programs you write using BASIC. Assembly language is unlike any high-level language, and it provides the ultimate means to exploit fully all of the resources in a PC.

12

Assembly Language Programming

This book has consistently presented programming techniques that reduce the size of your programs, and make them run faster. Most of the discussions focused on ways to write efficient BASIC code, and several showed how to access system interrupt services. Where speed was critical or BASIC was inflexible, I presented subroutines written in assembly language.

Assembly language is the most powerful way to communicate with a PC, and it offers speed and flexibility unmatched by any other language. Indeed, assembly language is in many ways the ultimate programming language because it lets you control fully every aspect of your PC's operation. Anything that a PC is capable of doing can be accomplished using assembly language. This final chapter explains assembly language in terms that most BASIC programmers can understand.

Why, you might ask, would a BASIC programmer be interested in assembly language? After all, the whole point of a high-level language such as BASIC is to shield the programmer from the underlying hardware. Without having to worry about CPU registers and memory addresses, a BASIC programmer can be immediately productive, and probably write programs with fewer initial bugs. However, there are three important reasons for using assembly language:

- To speed up selected portions of a program
- To reduce the size of a program
- To perform services that BASIC simply cannot

It is important to understand that any high-level language will benefit from the appropriate use of assembler. And while it is possible to write a major application using only assembly language, the increased complexity and added time to develop and debug it are often not worth the trouble. Using a high-level language—especially BASIC—for the majority of a program and then coding the size and speed-critical portions in assembly language often is the most practical solution.

Many BASIC programmers mistakenly believe that to achieve the fastest and smallest programs they should learn C. In my opinion, nothing could be further from the truth. Assembly language is barely more difficult to use than C, and in fact the code is often more readable. Further, no high-level language can come even close to what raw 8086 code can achieve. If you truly desire to become an advanced programmer, you owe it to yourself to at least see what assembly language is all about. I believe there is no deeper satisfaction than that gained by understanding fully what your computer is doing at the lowest level.

This chapter assumes that you already understand basic programming concepts such as variables, arrays, and subroutines. As we proceed, most of the examples will provide parallels to BASIC where possible. But please remember one important point: There is nothing inherently difficult about

assembly language. Attitude is everything, and if you can think of assembler as a stripped-down version of BASIC, you will be successful that much sooner.

For ease of reading, I will refer to the 8088 microprocessor used in the IBM PC throughout this chapter. However, everything said about the 8088 also applies to the 8086, the 80286, the 80386/486, and the NEC V series found in some older PC compatible computers. I will also use the terms *assembly language* and *assembler* interchangeably, although assembler can also be used to mean the program that assembles your source files.

All of the examples in this chapter are meant to be assembled with the Microsoft Macro Assembler (MASM) version 5.1 or later. MASM requires that you save your source files as standard ASCII text, and most word processor programs can do this.

Some of the examples in this chapter are derived from those that used CALL Interrupt in Chapter 11. In most cases I have not bothered to restate the same information from that chapter, and you may want to refer back for additional information.

Finally, many entire books have been written about assembly language, and there is no way I can possibly teach you everything you need to know here. Rather, my intent is to provide a gentle introduction to the concepts using practical and useful examples.

As Easy as BASIC

Assembly language uses the same general form as a BASIC program. That is, commands are performed in sequence until a GOTO or GOSUB is encountered. In assembly language these are called Jump and Call, respectively. Many BASIC instructions have a direct assembler equivalent, although the syntax is slightly different. One important difference, however, is that the 8088 microprocessor can operate on integer numbers only. Another is that for the most efficiency, you are limited to only a few working variables. I will begin by showing some rudimentary assembly language instructions, so you can see how they are analogous to similar commands in BASIC. Consider the following BASIC program fragment:

```
AX = 5
```

Here, the value 5 is assigned to the variable AX. The 8088 has several built-in variables called *registers*, and one of them is called AX. To move the value 5 into the AX register you use the Mov instruction:

```
MOV AX, 5
```

As with BASIC, the destination variable in an assembly language program is always shown on the left, and the source is on the right. Now consider addition and subtraction. To add the value 12 to AX in BASIC you do this:

```
AX = AX + 12
```

The equivalent 8088 command is:

```
Add AX,12
```

Again, the variable or register on the left is always the one that receives the results of any adding, moving, and so on. Subtraction is very similar to addition, replacing Add with Sub:

BASIC:

```
AX = AX - 100
```

Assembler:

```
Sub AX,100
```

Comparing and branching in assembly language is also quite similar to BASIC. But instead of this:

```
AX = AX + 2  
IF AX > 60 GOTO Finished
```

You'd do it in assembler this way:

```
Add AX,2  
Cmp AX,60  
Ja Finished
```

This tells the 8088 to add 2 to AX, then compare AX to 60, and finally to *jump if above* to the code at label Finished. There are several kinds of conditional jump instructions in assembly language, and they often follow a comparison as shown here. In fact, all you can really do after a compare is jump somewhere based on the results. And while there is no direct equivalent for this BASIC statement:

```
IF AX = 10 THEN BX = BX - 1
```

You can change the strategy to this:

```
IF AX <> 10 GOTO Not10  
BX = BX - 1  
Not10:  
.  
.
```

Now a direct translation is simple:

```
Cmp AX,10  
Jne Not10  
Dec BX  
Not10:
```


Jne stands for *Jump if Not Equal*. Also, notice the command Dec, which means decrement by 1. This is one case in which an assembler instruction is actually more to the point than its BASIC counterpart, and is equivalent to the BASIC command `BX = BX - 1`. While `Sub BX, 1` would work just as well, using Dec is faster and generates less code, and we all know that speed is the name of the game.

The complement to Dec is Inc, short for *increment by one*. You can use Inc and Dec with most of the 8088's registers, as well as on the contents of any memory location, which brings up an important issue. At some point, many programs will require more variables than can be held within the CPU's registers. All of the available free memory in a PC can be used as variable storage, with only a few limitations:

- You must first tell the assembler how much space to set aside, much like you would when dimensioning an array. Moreover, MASM is pretty friendly and lets you use names for the memory locations. In fact, in most cases you do not need to know the memory addresses variables will be stored in. The assembler handles that for you as well.
- Adding, subtracting, incrementing, and decrementing are all much faster when done within registers. When an operation is performed on a memory variable, it must first be fetched by the CPU, manipulated, and then stored again. Because the registers are within the CPU chip, those extra steps are not needed. The steps to retrieve and then store memory variables is handled transparently by the 8088; I mention this merely to explain why register operations are faster.
- Some operations can be done only using registers. If you want to multiply the memory variable Counter by 12, you first have to move the variable into AX, do the multiplication, and then move it back into memory again. And if AX is currently holding a needed value, it must be saved before multiplying and restored again afterward. Although assembly language is not as complicated as many people think, it surely can be tedious at times.

Besides the CPU registers and conventional memory addresses, a special portion of memory called the *stack* is also available for storage. The stack is much like the temporary memory on a four-function calculator, and it is often used to store intermediate results. The stack is also commonly used to pass variables between programs, because all programs can access it without having to know exactly where in memory it is located. Again, assembly language doesn't usually require you to deal with absolute memory addresses at all—especially for subroutines that will be added to a BASIC program. The only exceptions might be when writing directly to the display screen, or when looking at low memory, perhaps to see whether the Caps Lock key is engaged.

Spaghetti Code?

To write a routine that converts lower case letters to capital letters in BASIC, you might use something like this:

```
IF AL$ => "a" AND AL$ <= "z" THEN
  AL$ = CHR$(ASC(AL$) - 32)
END IF
```

In assembly language each compare must be done separately, followed by a jump based on the results. Let's rephrase the BASIC example slightly:

```
IF AL$ < "a" GOTO Done
IF AL$ > "z" GOTO Done
AL$ = CHR$(ASC(AL$) - 32)
Done:
.
```

Now a conversion to assembler is easy:

```
Cmp AL,"a"      ;compare AL to "a"
Jb Done        ;Jump if Below to Done
Cmp AL,"z"      ;compare AL to "z"
Ja Done        ;Jump if Above to Done
Sub AL,32       ;subtract 32 from AL
Done:
.
```

Notice how the assembler allows the use of quoted constants. When it sees a character or string in double or single quotes, it knows you mean to use the character's ASCII value. Unlike BASIC with its strong variable typing that prevents you from performing numeric operations on a string, assembly language has very few such restrictions. Also notice how much jumping around is necessary to accomplish even the simplest of actions.

As I mentioned earlier, assembly language can certainly be more tedious than BASIC, although the logic is not really that different. Such frequent jumping around is called spaghetti code by some programmers, and it is often used in a derogatory fashion when discussing BASIC's GOTO statement. But this is the way that computers work, and I am amused by programmers who argue so strongly against all use of the GOTO command. While nobody could seriously object to a well organized and structured programming style, all programs are eventually converted to equivalent assembly language jumps and branches.

The Registers

There are six general purpose registers available for you to use: AX, BX, CX, DX, SI, and DI. Each register may be used for the most common operations like adding and subtracting, although some are specialized for certain other operations. However, most of the registers also have a specialty. For example, AX is the only register that can be multiplied or divided. The A in AX stands for Accumulator, and it often used for math operations such as accumulating a running total. Also, several assembler instructions result in one byte less code when used with AX, when compared to the same instructions using other registers.

The B in BX means Base, and this register is frequently used to hold the base address of a collection of variables or other data. If you have a text string in memory to be examined, you could put the address of the first character in BX. The rest of the string can then be found by referencing BX.

BX can also be used to specify computed addresses using addition or subtraction. For example, the instruction `Mov AX, [BX+4]` means to load AX with the word four bytes beyond the address held in BX. Likewise, the instruction `Add DL, [BX+SI-10]` adds the value of the byte at that computed address to the current contents of DL. You may use BX this way with either a constant number, the SI or DI register, or one of those registers and a constant number. However, only addition and subtraction may be used, as opposed to multiplication or division. I will return to computed and indirect addressing later in this chapter.

The C in CX stands for Count, since CX is most often used as the counter in an assembly language FOR/NEXT loop. In fact, the assembly language command `Loop` uses CX to perform an operation a specified number of times. The comparison below illustrates this.

```
BASIC:
  FOR CX = 1 TO 5
    GOSUB BeepTone
  NEXT

Assembler:
  Mov  CX, 5
  Do:  Call Beep_Tone
  Loop Do
```

Here, the `Loop` instruction automatically branches to the label `Do:` CX times. That is much faster and more efficient than this:

```
Mov  CX, 5
Do:  Call Beep_Tone
Dec  CX
Cmp  CX, 0
Jne  Do
```

The DX register is a general purpose Data register, and is named accordingly. DX is also used in conjunction with AX when multiplying and dividing.

The last two general purpose registers are SI and DI. SI stands for Source Index, while DI means Destination Index. It is not hard to guess that these registers are well suited for copying data from one memory location to another. The 8088 has a rich set of instructions for moving and comparing strings, using SI and DI to show where they are.

Like BX, SI and DI may be used with a constant offset such as `[SI+100]` to compute a memory address, or with a constant value and/or BX. But again, SI and DI are still general purpose registers, and they can be used for common chores as well. In many situations it really doesn't matter whether you use BX or DI or SI or AX.

There are two specialized registers called BP and SP. BP (Base Pointer) is another Base register like BX, only it is intended for use with the stack. When you need to access data on the stack, BP is the most appropriate register to use. Like BX, BP can reference computed addresses with a constant offset, with SI or DI, or with a constant and SI or DI.

The SP (Stack Pointer) register holds the current address of the stack, and it should never be altered unless you have a very good reason to do so.

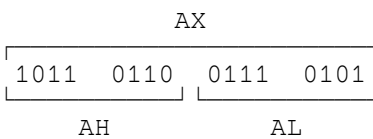
The last four registers are the segment registers, but I will mention them only briefly right now. As you undoubtedly know, the 8088 used a segmented architecture; although it can utilize a megabyte of memory, it can do so only in 64K portions at a time. The CS register holds the current Code Segment (your program code), DS holds the Data Segment (your memory variables), SS holds the Stack Segment, and ES is an Extra Segment that is often used to access arrays located in far memory.

Each of the 8088 registers can hold one word (two bytes), allowing you to store any integer number between 0 and 65535. This range of values can also be considered as -32768 to 32767. But AX, BX, CX, and DX may also be used as two separate one-byte registers with a range of either 0 to 255 or -128 to 127. One byte is often sufficient—for example, when manipulating ASCII characters—and this ability to access each half individually effectively adds four more registers. Remember, the more variables you can keep within registers, the faster and more efficient a program will be.

When using the registers separately, the two halves are identified by the letters H and L, for High and Low. That is, the high portion of AX is referred to as AH, while the low portion of DX is called DL. This would be represented with BASIC variables as follows:

$$AX = AL + 256 * AH$$

Each half can also be represented as bit patterns:



Notice that SI, DI, BP, and SP cannot be split this way, nor can the segment registers CS, DS, SS, and ES.

There is also another register called the Flags register, though it is not intended for you to use directly. After performing calculations and comparisons, certain bits in the Flags register are set or cleared by the CPU automatically, depending on the results. For example, if you add a register that holds the value 40000 to another register whose value is 30000, the Carry flag will be set to show that the result exceeded 64K. The 8088 flags are also set or cleared to reflect the result of a Cmp (Compare) instruction. Although you will not usually access these flags directly, they are used internally to process Jne, Ja, and the other conditional jump commands.

Variables in Assembly Language

All of the example routines shown so far have used the 8088 registers as working variables. Indeed, using registers whenever possible is always desirable because they can be accessed very quickly. But in many real-world applications, more variables are needed than can fit into the few available registers. As with BASIC, MASM lets you define variables using names you choose, and you must also specify the size of each variable.

The first step is to define the amount of space that will be set aside with the assembler instructions DB and DW. These stand for Define Byte and Define Word respectively, and they allocate either one byte of storage or two. You can also use DD to define a double word long integer variable. Notice that these are not commands that the 8088 processor will execute; rather, they inform the assembler to leave room for the data. Some examples are shown below:

```
MyByte DB 12h                ;one byte, preset to 12h
Buffer DB 15 Dup(0)          ;fifteen bytes, all 0
Dummy  DW ?                  ;one word (two bytes), 0
Msg    DB "Test message",13,10 ;message, CR, LF
```

In the first example one byte of memory is allocated using the name MyByte, and the value 12 Hex is placed there at assembly time. The second example illustrates using the Dup (duplicate) command, and tells MASM to set aside fifteen bytes filling each with the specified value. In this case that value is zero. Initialized data is an important feature of assembly language, and one that is sorely missing from BASIC. By being able to allocate data values at assembly time, additional code to assign those values at runtime is not needed.

Filling an area with zeroes can also be accomplished with a question mark, and this is frequently used when the value that will eventually end up there is not known in advance. Both do the same thing in most cases, however using "?" implies an unknown, as opposed to an explicit zero. You may use whichever method seems more appropriate at the time. The last example shows how text may be specified, as well as combining values in a single statement.

Since the assembler lets you use names for your data, fetching or storing values can be done with the normal Mov instruction like this:

```
Error_Code DB ?
Mov Error_Code,AL
```

This puts the contents of register AL into memory location Error_Code. Getting it back again later is just as easy:

```
Mov DH,Error_Code
```

Sometimes the assembler needs a little help when you assign variables. When you move AL or DH in and out of a memory location, the assembler knows that you are dealing with a single byte. And if you specify BX or SI as the source or destination operand, the assembler understands this to mean two

bytes, or one word. But when literal numbers are used, the size of the value is not always obvious. Consider the following:

```
Mov [BX],3Ch
```

Does this mean that you want to put the value 3Ch into the byte at the address held in BX, or the value 003Ch into the *word* at that address? There is no way for MASM to know what your intentions are, so you must specify the size explicitly. This is done with the Byte Ptr and Word Ptr directives. Here, Ptr stands for Pointer, and two examples are shown:

```
Mov Byte Ptr [BX],15  
Mov Word Ptr ES:[DI],100
```

The first example specifies that the memory at address BX is to be treated as a single byte. Had Word been used instead, a 15 would be placed into the byte at address held in BX, and a zero would be put into the byte immediately following. Words are always stored with the low-byte before the high-byte in memory.

Memory variables are accessed using the normal complement of instructions. For example, to add 15 to the variable Counter you will use `Add Counter,15`. And to multiply AX by the word variable Number you will use `Mul Word Ptr Number`. In MASM versions 5.0 and later, the Word Ptr argument is not strictly necessary. That is, if Number had been defined using DW, then MASM knows that you mean to multiply by a word rather than a byte. But earlier versions of the assembler were not so smart, and an explicit Word Ptr or Byte Ptr was required.

Note, however, that you must still use Byte Ptr or Word Ptr to override a variable's type. For example, if Value was defined as a word but you want to access just its lower byte, you must use `Mov AL,Byte Ptr Value`. Here, stating Byte Ptr explicitly tells MASM that you are intentionally treating Value as a different data type. Otherwise, it will issue a non-fatal warning error message.

Sometimes you may want to refer to the address of a variable, as opposed to its contents. For example, `Mov AX,Variable` tells MASM to move the value held in Variable into the AX register. But many DOS services require that you specify a variable's address in a register. This is done using the Offset operator: `Mov DX,Offset Buffer`. Where `Mov DX,Buffer` places the first two bytes of the buffer into DX, using Offset tells MASM that you instead want the starting address of the buffer.

You can also use the Lea (Load Effective Address) command to obtain an address, but that is less frequently used. Although `Lea DX,Buffer` can be used to load DX with the starting address of Buffer, it is a slightly slower instruction. Lea is needed only when an address must be computed. For example, the instruction `Lea SI,[BX+DI]` loads SI with the sum of the BX and DI registers. You may notice that Lea can provide a shortcut for adding or subtracting certain register combinations. Although this use of Lea is uncommon, Lea can replace the following two instructions:

```
Mov SI,BX  
Add SI,DI
```

To subtract two registers or a register and a constant value you could use `Lea AX, [BX-DI]` or `Lea SI, [BP-10]`.

Calculations in Assembly Language

When adding or subtracting you may use two registers, or a register and a memory variable. It is not legal to specify two memory variables as in `Add Var1, Var2`.

Multiplying and dividing are not so flexible; only AL and AX may be multiplied. When dividing, the numerator must be either in AX, or the long integer comprised of DX:AX. In this case, DX holds the upper word and AX holds the lower one. However, you may multiply or divide these registers using either a register or a memory location. Because of this restriction, it is not necessary to specify the target operand size. That is, `Mul CL` means to multiply AL by CL leaving the result in AX, and `Div WordVariable` divides DX:AX by the contents of WordVariable leaving the result in AX and the remainder in DX. Although you could use the commands `Mul AL, CL` and `Div AX, WordVariable`, this is not necessary or common.

All of the allowable combinations for multiplying and dividing are shown in Table 12-1.

Instruction	Operand	Result	Remainder
Mul ByteRegister	AL	AX	n/a
Mul ByteVariable	AL	AX	n/a
Mul WordRegister	AX	DX:AX	n/a
Mul WordVariable	AX	DX:AX	n/a
Div ByteRegister	AX	AL	AH
Div ByteVariable	AX	AL	AH
Div WordRegister	DX:AX	AX	DX
Div WordVariable	DX:AX	AX	DX

Table 12-1: The allowable register/memory combinations for multiplying and dividing.

In Table 12-1 ByteRegister means any byte-sized register such as AL or CH; WordRegister indicates any word-sized register like CX or BP. Likewise, ByteVariable and WordVariable specify byte and word-sized integer memory variables respectively.

It's important to understand that you must never divide by zero, because that will generate a critical error. Because the result from dividing by zero is infinity, the 8088 has no way to handle that—it can't simply ignore the error. Therefore, dividing by zero causes the CPU to generate an Interrupt 0. In a BASIC program that error is routed to BASIC's internal error handling mechanism which either invokes the ON ERROR handler if one is in effect, or ends your program with an error message. In a

purely assembly language program, DOS intervenes printing an error message on the screen, and then it ends the program.

Related to division by zero is dividing when the result cannot fit into the destination register. For example, if AX holds the value 20000 and you divide it by 2, the resulting 10000 cannot fit into AL. Since this is another unrecoverable error that cannot be ignored, the 8088 generates an Interrupt 0 there as well.

Besides the Div and Mul instructions, there are also signed versions called Idiv and Imul. Where Div and Mul treat the contents of AX or DX:AX as an unsigned value, Idiv and Imul treat them as being signed. You'll use whichever command is appropriate, so the 8088 knows if values having their highest bit set are to be treated as negative. BASIC always uses Idiv and Imul in the code it generates, since all integer and long integer values are treated by BASIC as signed.

Because only AX and DX:AX may be used for multiplying and dividing, this affects your choice of registers. The short example that follows shows how you might select registers when translating a simple BASIC-like expression that uses only integer (not long integer) variables.

BASIC:

```
Result = (Var1 + Var2 * (Var3 - Var4)) \ 100
```

Assembler:

```
Mov  AX,Var3           ;work from the innermost level out
Sub  AX,Var4           ;so first perform Var3 - Var4
Imul Word Ptr Var2    ;then multiply that by Var2
Add  AX,Var1           ;add Var1 to what we have so far
Mov  DX,0              ;next prepare to divide DX:AX
Mov  CX,100            ;use CX for the divisor
Idiv CX                ;do the division
Mov  Result,AX        ;then assign Result ignoring the remainder left in DX
```

Because dividing by an integer value uses both DX and AX, it is necessary to clear DX explicitly as shown unless you are certain it is already zero. The use of CX to hold the value 100 is arbitrary. If CX were currently in use, any available word-sized register or memory location could be used. If you compile this program statement and view the resultant code using CodeView, you will see that BASIC does an even better job of translating this particular expression to assembly language.

String Processing Instructions

Besides being able to add, subtract, multiply, and divide, the 8088 provides four very efficient instructions for manipulating strings and other data in memory. Movs copies, or moves a string from place to another; Cmps compares two ranges of memory; Stos fills, or stores one or more addresses with the same value; and Scas scans a range of memory looking for a particular value. These instructions require either a byte or word specifier. For example, you would use Movsb to copy a byte, and Cmpsw to compare two words.

There are two important factors that contribute to the power and usefulness of these string instructions: each is only one byte long, and they automatically increment or decrement the SI and DI registers that point to the data being manipulated. Thus, they are both convenient to use, and also very fast. Because it is common to access blocks of memory sequentially a byte or word at a time, automatically advancing SI and DI saves you from having to do that manually with additional instructions. For example, after one pair of words has been compared, SI and DI are already set to point at the next pair.

You can also specify that SI and DI are to be decremented by first using the `Std` (Set Direction) command. The Direction Flag stores the current string operations direction, which is either up or down. If a previous `Std` was in effect, then you'd use `Cld` (Clear Direction) to force copying and moving to be forward. In fact, BASIC requires you to clear the direction flag to forward before returning from an assembler routine that set it to backwards.

Movs and Cmps

`Movs` and `Cmps` use the DS:SI register pair to point to the first range of memory being copied or compared, and ES:DI to point to the second range. Each time a byte is being copied or compared, SI and DI are incremented or decremented by one to point to the next address. And when a word is being accessed, SI and DI are incremented or decremented by two.

Notice that there is no protection against SI or DI being incremented or decremented through address zero, nor is there any indication that this has happened. Also notice that the name `Movs` is somewhat of a misnomer. To me, moving something implies that it is no longer at its original location. `Movs` does not alter the source data at all—it merely places a new copy at the specified destination address.

Scas and Stos

`Scas` compares the value in AL or AX with the range of memory pointed to by ES:DI. That is, `Scasb` compares AL and `Scasw` uses AX. `Stos` also uses ES:DI to show where the data being written to is located; `Stosb` stores the contents of AL in the address at ES:[DI] and then increments or decrements DI by one. Likewise, `Stosw` stores the value in AX there and increments or decrements DI by two.

Repeating String Operations

If these four instructions merely acted on the data and incremented SI and DI automatically, that would be very useful indeed. But they also have another talent: they recognize a `Rep` (Repeat) prefix to perform their magic a specified number of times. The number of iterations is specified by the count held in CX. Furthermore, the number of repetitions can be made conditional when comparing and scanning, based on the data encountered.

If you have, say, 20 bytes of data that need to be copied from one place to another, you would first set CX to 20 and then use `Rep Movsb`. And to compare 100 words you would load CX with the value 100 and use `Rep Cmpsw`. `Stos` also accepts a `Rep` prefix; `Rep Stosb` places the value in AL into CX bytes of contiguous memory starting at the address specified in ES:DI. For each iteration the 8088 decrements CX, and when it reaches zero the copying or comparing is complete.

It is usually not valuable to scan a range of memory unconditionally and repeatedly. Therefore `Scas` is generally used in conjunction with either `Repe` (Repeat while Equal) or `Repne` (Repeat while Not Equal). `Cmps` is also generally used with these conditional prefixes, to avoid wasting time comparing bytes after a match or a difference was found. In either case, however, you load CX with the total number of bytes or words being compared or scanned.

Because each iteration decrements CX, you can easily calculate how many bytes or words were actually processed. Also, you can test the results of scanning and comparing using the normal methods such as `Je` and `Jne`. The following few examples show some ways these commands can be used.

```
See if two 40-byte ranges of memory are the same:
Mov  CX,20                ;comparing 20 words is faster than 40 bytes
Repe Cmpsb               ;compare them
Je   Match                ;they matched
```

```
Copy a 2000-element integer array to color screen memory:
Mov  AX,ArraySeg         ;set DS to the source segment
Mov  DS,AX               ;through AX
Mov  SI,ArrayAdr         ;point SI to the array start
Mov  AX,&HB800           ;the color text screen segment
Mov  ES,AX               ;assign that to ES
Mov  DI,0                ;clear DI to point to address 0
Mov  CX,2000            ;prepare to copy 2000 words
Rep  Movsw               ;copy the data
```

```
Search a DOS string looking for a terminating zero byte:
Mov  AX,StringSeg       ;set ES to the string's segment
Mov  ES,AX               ;(ES cannot be assigned directly)
Mov  DI,Offset ZString  ;point DI to the string data
Mov  CX,80               ;search up to 80 bytes
Mov  AL,0                ;looking for a zero value
Repne Scasb             ;while ES:[DI] <> AL
;-- Now DI points just past the terminating zero byte.
;-- The length of the string is (80 - CX + 1).
```

In the first example, it is assumed that DS:SI and ES:DI already point to the correct segment and address. By asking to compare only while the bytes are equal, the result of the most recent byte comparison can be tested using `Je`. A common mistake many programmers make is comparing the bytes, and then checking if CX is zero. The reasoning is that if CX is zero then they must have all matched; otherwise the 8088 would have aborted the comparisons early. But CX will also be zero if all but the last byte matched! Therefore, you must check the zero flag using `Je` (or `Jne` if that is more appropriate).

Notice in the first example how 20 words are compared, rather than 40 bytes. Although the net result is the same, word operations are faster on 80286 and later processors when the blocks of memory begin at an even numbered address.

```
Though you can't always know if a variable or block of
memory will begin at an even address, using the word
version will be more efficient at least some of the
time.
```

The second and third examples include the code needed to set up the appropriate segment and address values in DS:SI and ES:DI. Although this may seem like a lot of work, you can often do this setup only once and then use the same registers repeatedly within a routine. Unfortunately, you are not allowed to assign a segment register from a constant number. You must first assign the number to a conventional register, and then use Mov to copy it to the segment register.

The Stack

The primary purpose of the stack is to retain the return address of a program when a subroutine is called. This is true not only for assembly language, but for BASIC as well. For example, when you use the BASIC statement `GOSUB 1200`, BASIC must remember the location in memory of the next command to execute when the routine returns. It does this by placing the address of the next instruction onto the stack before it jumps to the subroutine. Then when a RETURN instruction is encountered, the address to return to is available. The 8088 understands Calls and Returns directly, and it places and restores the addresses on the stack automatically.

The stack is not unlike a stack of books on a table, and one of its great advantages is that you don't need to know where in memory it is actually located. Items can be placed onto the stack either manually with the Push instruction, or automatically by the 8088 processor as part of its handling of Call and Return statements. Values are retrieved from the stack with the Pop command, among other methods.

One important feature of the stack is when items are added and removed, the stack pointer register is updated automatically to reflect the next available stack location. Thus, a program can access items on the stack based on the stack pointer, rather than have to know the exact address at any given time. This simplifies exchanging information between programs, since neither has to know how the other operates. This mechanism also makes it possible for programs written in one language to communicate with subroutines written in another. Figure 12-1 shows how the stack operates.

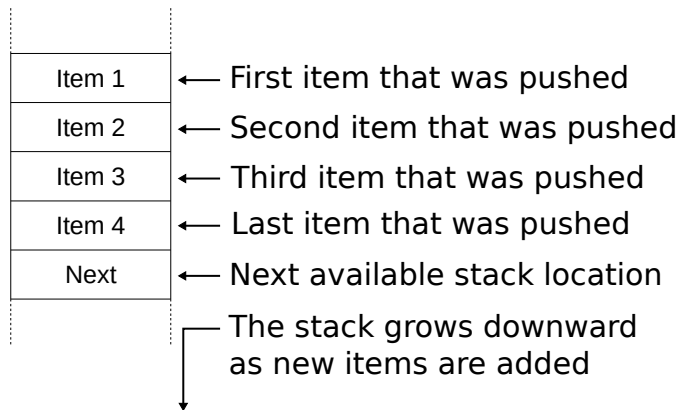


Figure 12-1: The organization of the CPU stack.

As each item is pushed onto the stack, it is placed two bytes below the address held in the stack pointer. Then the stack pointer is decremented by two, to show the next available stack location. Therefore, the stack grows downward as new items are added. Note that only full words may be pushed onto the stack, so all of the items shown here are two bytes in size. Also note that the stack pointer holds the address of the last item that was pushed.

Passing Parameters

Imagine you have a BASIC subroutine that does something to the variable X. The code to assign X, process, and print X might look like this:

```
X = 12
GOSUB 2000      'the routine at line 2000 manipulates X
PRINT X
```

In assembly language you could push the value 12 onto the stack, and then call the subroutine. The subroutine, expecting the value there would retrieve it, do its work, and then place the result back again before returning. This is similar, but not identical, to how variables are passed between programs. Most high-level languages including BASIC pass variables to subroutines by placing their addresses on the stack. A called routine can then access the variable via its address, either to read it or to assign a new value.

If BASIC let you access the registers directly, it could pass variables through them, as you saw when telling DOS which of its services to do. But BASIC doesn't allow that and moreover, with a limited number of registers, only a few variables or addresses could be accommodated. The stack can hold any number of arguments, by pushing the address of each in turn.

When you use the BASIC CALL command and pass a variable name to a SUB or FUNCTION procedure, BASIC first pushes the address of that variable onto the stack, before jumping to the code being called. And if more than one variable is specified, all of the addresses are pushed. The example below shows how you might call a routine that returns the current default drive.

```
CALL GetDrive(Drive%)
```

When GetDrive begins, it knows that the stack is holding the address of Drive%. The segment and address of the calling BASIC program is also on the stack; however, GetDrive is not concerned with that. The important point is that it can find the address on the stack using the SP (Stack Pointer) register. When GetDrive begins the stack is set up as shown in Figure 12-2.

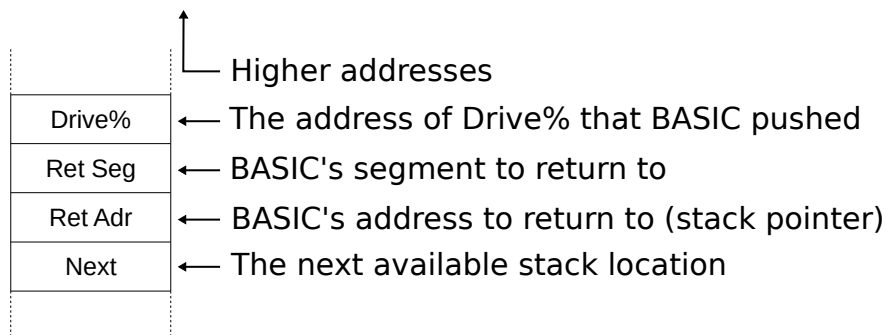


Figure 12-2: The state of the stack within a procedure when one variable address was passed.

Notice that while GetDrive can get at the address of Drive% through SP, an extra step is still required to get at the data held in Drive%. Let's digress for a moment to reconsider the difference between memory addresses and values. The assembler command `Mov AX,12` puts the value 12 into register AX. But suppose you want to put the contents of *memory location* 12 into AX. You indicate this to the assembler by using brackets, as shown in the two equivalent examples following.

```
Mov AX, [12]      ;load AX from address 12
Mov BX, 12        ;assign BX to the value 12
Mov AX, [BX]     ;load AX from the address held in BX
```

The first statement loads AX from the contents of memory at address 12. The second first loads BX with the number 12, and then uses BX to identify that address, moving the contents of that address into AX. This is an important distinction, and is illustrated in Table 12-2 using parallels to BASIC's PEEK and POKE commands.

BASIC	Assembler
BP = SP	Mov BP,SP
AL = PEEK(BP + 8)	Mov AL,[BP+8]
SI = 12	Mov SI,12
POKE SI, 12	Mov Byte Ptr [SI],12

Table 12-2: Similarities between BASIC's PEEK and POKE, and the assembly language Mov instruction.

Although you can easily find the address of Drive% by looking at SP, an extra step is required to get at the actual value. The example that follows shows how to do this, except there is one added complication. You are not allowed to use SP for addressing, except with 386 and later microprocessors. Since you undoubtedly want your programs to work with as many computers as possible, a different strategy must be used.

As I mentioned earlier, the BP register is a base register that is meant for accessing data on the stack. Therefore, you must first copy SP into BP, and then use BP to access the stack. Then you can find where Drive% is located, and put the current drive number into that address as shown following:

```
Mov BP,SP      ;put the current stack pointer into BP
Mov SI,[BP+4]  ;put the address of Drive% into SI
Mov AH,19h     ;tell DOS we want the default drive
Int 21h        ;call DOS to do it
Mov [SI],AL    ;put the answer into Drive%
```

Notice how brackets are used to indicate the addresses. You must first determine the address of Drive%'s address (whew!), before you can put the value held in AL there. This is called indirect addressing, because a register is used to hold the address of the data. Again, notice how the 8088 accepts addition on the fly when you tell it BP+4.

The complete working GetDrive routine has two small added complications. Beside being unable to use SP for addressing memory, BASIC also requires you to not change BP either. The obvious solution, therefore, is to first save BP on the stack before changing it, and then restore BP later before returning to BASIC. The other complication is caused by the very fact that BASIC put extra information (Drive%'s address) onto the stack. But neither is insurmountable, as shown here:

```
Push BP        ;save BP before changing it
Mov BP,SP      ;put the stack pointer into BP
Mov SI,[BP+6]  ;put the address of Drive% into SI
Mov AH,19h     ;tell DOS we want default drive
Int 21h        ;call DOS to do it
Mov [SI],AL    ;put the answer into Drive%
Pop BP         ;restore BP to its original value
Ret 2          ;return to BASIC
```

Notice that here, the address of Drive% is at [BP+6] rather than [BP+4] as it was in the previous listing. Since BP was pushed at the start of the procedure, the stack pointer is two bytes lower when it is

subsequently assigned to BP. When SI is loaded, [BP] points to the saved version of itself, [BP+2] and [BP+4] point to the address and segment to return to, and [BP+6] holds the address of Drive%'s address. This is illustrated in Figure 12-3.

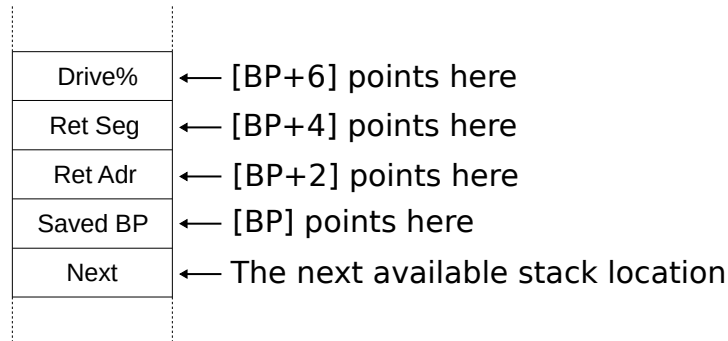


Figure 12-3: The state of the stack within a procedure after BP has been pushed.

Normally when a Ret command is encountered, the 8088 pops the last four bytes from the stack automatically, and returns to the segment and address contained in those bytes. But that would leave the 2-byte address of Drive% still cluttering up the stack. To avoid this problem the 8088 lets you specify a *parameter count* as part of the Ret instruction.

For each variable address that is passed with a CALL from BASIC, you must add 2 to the Return instruction in your assembler routine. This is the number of bytes to remove from the stack, with two being used for each incoming two-byte address. Had two variables been passed, the program would have used Ret 4 instead. Although it is possible to have the calling program clean up the stack itself, that would be wasteful.

For every occurrence of every call that passes parameters, BASIC would have to include additional code following the call to increment SP accordingly. Pushing a parameter's address onto the stack leaves that much less stack space available. Therefore, someone has to reverse the process and either pop the addresses or use Add SP,Num to adjust the stack pointer. By having the called routine handle it, that code is needed only once. In fact, this is an important deficiency of C, because by design C requires the caller to clean up the stack.

If you've managed to persevere this far you'll be pleased to know that in practice, the assembler can be told to handle most or all aspects of stack addressing for you. This is discussed in the sections that follow.

It is also possible to tell BASIC to pass some types of parameters by value using the BYVAL option in the DECLARE or CALL statements. When BYVAL is used, BASIC places the actual value of the variable onto the stack, rather than its address. This has several important benefits. First, the assembly

language routine can use one less instruction. Second, when a constant number is passed, BASIC does not need to make a copy of it in DGROUP. This copying was described in Chapter 2.

However, BYVAL is appropriate only when a parameter does not have to be returned, and only when the values are integers. If you pass a double precision parameter using BYVAL, all eight bytes are placed on the stack using four separate instructions rather than only two needed to pass the address. You can also instruct BASIC to pass the full, segmented address of a parameter, and that is discussed in the section **Dynamic Arrays**.

Procedures in Assembly Language

All of the discussions so far have focused on how to write the instructions for an assembly language subroutine. However, none have described how these routines are added to a BASIC program, or how a complete procedure is defined. Furthermore, the previous examples have not shown a key step that is needed with all such external routines: establishing the code and data segments.

Before an external routine can be linked to a BASIC program you must establish a public procedure name that LINK can identify. I will first show the formal method for defining a procedure and its segments, and then show the newer, simplified methods that were introduced with MASM version 5.1. The simplified syntax is used for all of the remaining examples in this chapter, so don't worry if the setup details for this first example appear overwhelming.

The simplest complete subprogram you are likely to encounter is probably the PrtSc routine that follows—all it does is call Interrupt 5 to send the contents of the current display screen to LPT1.

```
Code      Segment Word Public 'Code'
Assume    CS:Code
Public    PrtSc
PrtSc     Proc Far           ;this is equivalent to SUB PrtSc STATIC in BASIC

Int 5
Ret                               ;call BIOS interrupt 5
                               ;return to BASIC

PrtSc     Endp               ;this is equivalent to BASIC's END SUB
Code      Ends
End
```

The first three lines tell the assembler that the code is to be placed in the segment named Code, and that the name PrtSc is to be made public. The fourth line defines the start of a procedure. The actual code occupies the next two lines. Of course, you must tell the assembler where the procedure ends, which in this case is also the end of the code segment. Had several procedures been included within the same block of code, each procedure would show a start and end point, but there would only be a single code segment. The final End statement is needed to tell the assembler that this is the end of listing, although you might think that MASM would be smart enough to figure that out by itself!

Notice that there are two kinds of procedures: Far and Near. External routines that are called from BASIC are always Far, because BASIC uses what is called a *medium model*. This means the procedure does not necessarily have to be within the same code segment as the main BASIC program. The medium model allows the combined programs to exceed the usual 64k limit when linked to a final .EXE file.

When BASIC executes a CALL command, it uses a two-word address as the location to jump to. One of the words contains a segment, and the other an address within that segment. Then when your program finally returns, the 8088 must know to remove two words from the stack—a segment and an address—to find where to return to in the calling BASIC program.

A near procedure, on the other hand, calls an address that is only one word long. And when the procedure returns, only a single word is popped from the stack. Again, the assembler does the bulk of the dirty work for you. You just have to remember to use the word Far.

Simplified Directives

Fortunately, Microsoft realized what a pain dealing with segments and procedures and offsets from BP can be, and they enhanced MASM beginning with version 5.0 to handle these details automatically for you. Rather than require the programmer to define the various code and data segments, all that is needed are a few simple key words.

The first is `.Model Medium`, which tells MASM that the procedures that follow will be Far. Used in conjunction with `.Code` and `.Data`, `.Model Medium` tells MASM that any data you define should be placed into a group named DGROUP. Adding `„Basic` after the `.Model` directive also declares your procedures as `Public` automatically, so BASIC can access them when your program is linked.

By using the name DGROUP, the linker automatically gathers all of your DB and DW data variables, and places them into the same segment that BASIC uses. While this has the disadvantage of impinging on BASIC's near data space, it also means that on entry to the routine the DS register (which BASIC sets to hold the DGROUP segment) hold the correct segment value for your variables as well.

To show the advantages of simplified directives, contrast the earlier `PrtSc` with this version that does exactly the same thing:

```
.Model Medium, Basic
.Code

PrtSc Proc
    Int 5
    Ret
Endp
End
```

MASM 5.1 introduced additional simplified directives that let you access incoming parameters by name, rather than as offsets from BP. All of the remaining examples in this chapter take advantage of simplified directives, as the following revised listing for GetDrive illustrates.

```
;Syntax: CALL GetDrive(Drive%)

.Model Medium, Basic
.Data
    ;-- if variables were needed they would be placed here

.Code
GetDrive Proc, Drive:Word

    Mov  AH,19h      ;tell DOS we want the default drive
    Int  21h        ;call DOS to do it
    Mov  BX,Drive    ;put the address of Drive% into BX
    Cbw                    ;clear AH to make a full word
    Mov  [BX],AX     ;then store the answer into Drive%
    Ret                                ;return to BASIC

GetDrive Endp      ;indicate the end of the procedure
End                ;and the end of the source file
```

As you can see, this looks remarkably like a BASIC SUB or FUNCTION procedure, with the incoming parameter listed by name and type as part of the procedure declaration. This greatly simplifies maintaining the code, especially if you add or remove parameters during development. If incoming parameters are defined as shown here using Drive%, code to push BP and then move SP into BP is added for you automatically. When you refer to one of the parameters, the assembler substitutes [BP+##] in the code it generates. Note, however, that the Word identifier for Drive refers to the 2-byte size of its address, and not the fact that Drive% is a 2-byte integer.

Also notice the new *Cbw* command, which is used here to clear the AH register. *Cbw* (Convert Byte to Word) expands the byte value held in AL to a full word in AX. A full word is needed to ensure that both the high and low-byte portions of Drive% are assigned, in case it held a previous value. If the value in AL is positive (between 0 and 127), AH is simply cleared to zero. And if AL is negative (between -128 and -1 or between 128 and 255), *Cbw* instead sets all of the bits in AH to be on. Thus, the sign of the original number in AL is preserved.

A complementary statement, *Cwd* (Convert Word to Double Word), converts the word in AX to a double-word in DX:AX. Again, if AX is positive when considered as a signed number, DX is cleared to zero. And if AX is currently negative, DX is set to FFFFh (-1) to preserve the sign. *Cbw* and *Cwd* are both one-byte instructions, so even with unsigned values they are always smaller and faster for clearing AH or DX than `MOV AH, 0` and `MOV DX, 0` which require two bytes and three bytes respectively.

Finally, the `Ret` command that exits the procedure is translated by MASM to include the correct stack adjustment value, based on the number of incoming parameters. If you have multiple exit points from the procedure (equivalent to EXIT SUB), the exit code will be generated multiple times. That is, each occurrence of `Ret` is replaced with a code sequence to pop the saved registers, and preform the 3-byte

Ret # instruction. Therefore, you should always use a single exit point in a routine, and jump to that when you need to exit from more than one place.

Calling Interrupts

Chapter 11 explained how interrupts work, and mentioned that only assembly language can call an interrupt directly. An assembler program uses the Int instruction, and this tells the 8088 to look in the interrupt vector table in low memory to obtain the interrupt procedure's segment and address. Then the procedure is called as if it were a conventional subroutine.

All of the DOS and BIOS services are accessed using interrupts, though there are so many different services that you also have to pass a service number to many of them. Most of the DOS services are accessed through interrupt 21h. Where BASIC uses the &H prefix to indicate a hexadecimal value, assembly language uses a trailing letter H. If you specify a number without an H it is assumed by MASM to be regular decimal. Note that MASM doesn't care if you use upper or lowercase letters, and knows that either means hexadecimal.

When specifying hexadecimal values to MASM, the first character must always be a digit. That is, 1234h is acceptable, but &HB800 must be entered as 0B800h. Using B800h will generate a syntax error.

DOS and BIOS Services

You have already seen how to call the BIOS routine that prints the screen and the DOS routine that returns the current drive. Let's continue and see how to call some of the other useful routines in the BIOS and DOS.

The next example program, DosVer, shows how to call the DOS service that returns the DOS version number. Like many of the assembler routines that you can use with BASIC, DosVer relies on an existing DOS service to do the real work. In this program you will also learn how to push and pop values on the stack.

The syntax for DosVer is `CALL DosVer(Version%)`, where Version% returns with the DOS version number times 100. That is, if your PC is running DOS version 3.30, then Version% will be assigned the value 330. Manipulating floating point numbers is much more difficult than integers, and the added complexity is not justified for this routine.

The DOS service that retrieves the version number returns with two separate values—the major version number (3 in this case) and the minor number (30). These values are returned in AL and AH respectively. The strategy here is to first multiply AL by 100, and then add AH. The last step is to assign the result to the incoming parameter Version%.

Unfortunately, when you use AL for multiplication, the value 100 must be in a register or memory location. You can't just use `MUL AL, 100` though it would sure be nice if you could. Further, whenever AL is multiplied the result is placed into the entire AX register. Therefore, `DosVer` also uses BX to temporarily store the original contents of AX before the two are added together.

As you already have learned, the only register that can be multiplied is AX, or its low-byte portion, AL. MASM knows if you plan to multiply AX or AL based on the size of the argument. For example, `Mul BX` means to multiply AX by BX and leave the result in DX:AX. `Mul CL` instead multiplies AL by CL and leaves the answer in AX.

The complete `DosVer` routine is shown following, and comments explain each step.

```
;DOSVER.ASM, retrieves the DOS version number

.Model Medium, Basic
.Code

DOSVer Proc, Version:Word

    Mov  AH,30h      ;service 30h gets the version
    Int  21h        ;call DOS to do it

    Push AX         ;save a copy of the version for later
    Mov  CL,100     ;prepare to multiply AL by 100
    Mul  CL         ;AX is now 300 if running DOS 3.xx

    Pop  BX         ;retrieve the version, but in BX
    Mov  BL,BH      ;put the minor part into BL for adding
    Mov  BH,0       ;clear BH, we don't want it anymore
    Add  AX,BX      ;add the major and minor portions

    Mov  BX,Version ;get the address for Version%
    Mov  [BX],AX    ;assign Version% from AX
    Ret             ;return to BASIC

DOSVer Endp
End
```

Notice the extra switch that is done with BH and BL. AX is saved onto the stack because multiplying the byte in AL leaves the result as a full word in AX, thus destroying AH. When the version is popped into BX, the minor part is in BH. But you are not allowed to add registers that are different sizes (AX and BH). Further, any number in the high half of a register is by definition 256 times the value of the same number in a low half. Therefore, BH is first copied to BL to reflect its true value. BH is then cleared so it won't affect the result, and finally AX and BX are added.

A better way to save AX and then restore it to BX would be to simply use `MOV BX, AX` immediately after the call to Interrupt 21h. I used `Push` and `Pop` just to show how this is done. As you can see, it is not necessary to pop the same register that was pushed. However, every `Push` instruction must always have a corresponding `Pop`, to keep the stack balanced. If a register or other value is on the stack when the final `Ret` is encountered, that value will be used as the return address which is of course incorrect.

Division also acts on AX, or the combination of DX:AX. When you use the command `Div BL`, the 8088 knows you want to divide AX because BL is a byte-sized argument. It then leaves the result in AL and the remainder, if any, is placed into AH. Similarly, `Div DX` means that you are dividing the long integer in DX:AX, because DX is a word. The result of this division is assigned to AX, with the remainder in DX.

Accessing BASIC Strings in Assembly Language

As Chapter 2 explained, strings are stored very differently than regular numeric variables. BASIC lets you find the address of any variable with the `VARPTR` function. For integer or floating point numbers, the value `VARPTR` returns is the address of the actual data. But for strings, `VARPTR` instead returns the address of a string descriptor.

DOS employs a different method entirely for its strings, using a `CHR$(0)` to mark the end. This is described separately later in the section DOS Strings.

BASIC Near Strings

A BASIC string descriptor is a table containing information about the string—that is, its length and address. In Microsoft compiled BASIC a string descriptor is comprised of two words of information. For QuickBASIC and near strings when using BASIC PDS, the first word contains the length of the string and the second holds the address of the first character. Consider the following BASIC instructions:

```
X$ = "Assembler"  
V = VARPTR(X$)
```

V now holds the starting address of the four-byte descriptor for X\$. For the sake of argument, let's say that V is now 1234. Addresses 1234 and 1235 will together contain the length of X\$ which is 9, and addresses 1236 and 1237 will contain yet another address—that of the first character in X\$. You can therefore find the length of X\$ using this formula:

```
Length = PEEK(V) + 256 * PEEK(V + 1)
```

And the first character "A" can be located with this:

```
Addr = PEEK(V + 2) + 256 * PEEK(V + 3)
```

You could then print the string on the screen like this:

```
FOR C = Addr TO Addr + Length - 1  
  PRINT CHR$(PEEK(C));  
NEXT
```

Therefore, this is a BASIC model for how strings are located by an assembly language program. When you call an assembler routine with a string argument, BASIC first pushes the address of the descriptor onto the stack, before calling the routine. The next example is called Upper, because it capitalizes all of the characters in a string. Even though BASIC offers the UCASE\$ and LCASE\$ functions, these are relatively slow because they return a copy of the data that has been manipulated. Upper instead capitalizes the data in place very quickly.

The strategy is to first get the descriptor address from the stack. Then Upper puts the length into BX and the address of the string data into SI. Upper steps through the string starting at the end, decrementing BX by one for each character. When BX crosses zero, it is done. A BASIC version is shown first, followed by the assembly language equivalent.

Upper in BASIC:

```
SUB Upper(Work$) STATIC
  '-- load SI with the address of Work$ descriptor
  SI = VARPTR(Work$)

  '-- assign LEN(Work$) to BX
  BX = PEEK(SI) + 256 * PEEK(SI + 1)

  '-- the address of the first character goes in SI
  SI = PEEK(SI + 2) + 256 * PEEK(SI + 3)

More:
  BX = BX - 1                'point to the end of Work$
  IF BX < 0 GOTO Exit        'no more characters to do
  AL = PEEK(SI + BX)         'get the current character
  IF AL < ASC("a") GOTO More 'skip conversion if too low
  IF AL > ASC("z") GOTO More 'or if too high
  AL = AL - 32               'convert to upper case
  POKE SI + BX, AL          'put character back in Work$
  GOTO More                  'go do it all again

Exit:                        'return to caller
END SUB
```

Upper in assembly language:

```
Upper Proc, Work:Word

  Mov  SI,Work      ;load SI with Work$'s descriptor address
  Mov  BX,[SI]      ;put LEN(Work$) into BX
  Mov  SI,[SI+2]    ;SI holds address of the first character
More:
  Dec  BX           ;point to the next prior character
  Js   Exit         ;if sign is negative BX is less than 0
  Mov  AL,[BX+SI]   ;put the current character into AL
  Cmp  AL,"a"       ;compare it to ASC("a")
  Jb   More         ;jump if below to More
  Cmp  AL,"z"       ;compare AL to ASC("z")
  Ja   More         ;jump if above to More
  Sub  AL,32        ;convert AL to upper case
  Mov  [BX+SI],AL   ;put AL back into Work$
  Jmp  More         ;jump to More
```

```
Exit:
  Ret          ;return to BASIC

Upper Endp
End
```

What's Your Sign?

Notice that for expediency, these routines work backwards from the end of the string. There are a number of shortcuts that you can use in assembly language, and one important one is being able to quickly test the result of the most recent numeric operation. If the program worked forward through the string, it would take three lines of code to advance to the next character, and also require saving the string length separately:

```
Inc  BX          ;point to the next character
Cmp  BX,Length  ;are we done yet?
Jne  More       ;no, continue
```

Notice the use of a new form of conditional jump—*J*s which stands for *Jump if Signed*. Here the code tests the sign of the number in BX, and jumps if it is negative. Though I haven't mentioned this yet, a conditional jump doesn't always have to follow a compare. Although a comparison will set the flags in the 8088 that indicate whether a particular condition is true, so will several other instructions. Some of these are Add, Sub, Dec, and Inc, but not Mov. So instead of having to include an explicit comparison:

```
Dec  BX          ;decrement BX
Cmp  BX,0       ;compare it to zero
Jl   More       ;jump if less to More
```

All that is really needed is this:

```
Dec  BX
Js   More
```

The Dec instruction sets the Sign Flag automatically, just as if a separate compare had been performed.

Conditional Jump Instructions

Besides Je, Jne, and Js, there are a few other forms of conditional jump instructions you should understand. Table 12-3 lists all of the ones you are likely to find useful.

Command	Meaning
Je	Jump if equal
Jne	Jump if not equal
Ja	Jump if above (unsigned basis)
Jna	Jump if not above (unsigned basis)
Jb	Jump if below (unsigned basis)

Command	Meaning
Jnb	Jump if not below (unsigned basis)
Jg	Jump if greater (signed basis)
Jng	Jump if not greater (signed basis)
Jl	Jump if less (signed basis)
Jnl	Jump if not less (signed basis)
Jc	Jump if Carry Flag is set
Jnc	Jump if Carry Flag is clear
Js	Jump if sign flag is set
Jns	Jump if sign flag is not set
Jcxz	Jump if CX is zero

Table 12-3: The 8088 conditional jump instructions.

You should know that *Je* and *Jne* also have an alias command name: *Jz* and *Jnz*. These stand for *Jump if Zero* and *Jump if Not Zero* respectively, and they are identical to *Je* and *Jne*. In fact, though I didn't mention this earlier, the *Repe* and *Repne* string repeat prefixes are sometimes called *Repz* and *Repnz*.

Because *Je* and *Jz* cause MASM to generate the identical machine code bytes, they may be used interchangeably. In some cases you may want to use one instead of the other, depending on the logic in your program. For example, after comparing two values you would probably use *Je* or *Jne* to branch if they are equal or not equal. But after testing for a zero or non-zero value using `Or AX, AX` you would probably use *Jz* or *Jnz*. This is really just a matter of semantics, and either version can be used with the same results.

Also, please understand that *Jnb* is not the same as *Ja*. Rather, the case of being Not Below is the same as being Above Or Equal. In fact, MASM recognizes *Jae* (Jump if Above or Equal) to mean the same thing as *Jnb*. Likewise, *Jbe* (Jump if Below or Equal) is the same as *Jna*, *Jge* (Jump if Greater or Equal) is the same as *Jnl*, and *Jle* (Jump if Less or Equal) is identical to *Jng*. Again, which form of these instructions you use will depend on how you are viewing the data and comparisons.

Note the special form of conditional jump, *Jcxz*. *Jcxz* stands for Jump if CX is Zero, and it combines the effects of `Cmp CX, 0` and `Je label` into a single fast instruction. *Jcxz* is also commonly used prior to a *Loop* instruction. When you use *Loop* to perform an operation repeatedly, CX must be assigned initially to the number of times the loop is to be executed. But if CX is zero the loop will execute 65536 times! Thus, adding `Jcxz Exit` avoids this undesirable behavior if zero was passed accidentally.

Finally, you must be aware that a conditional jump cannot be used to branch to a label that is more than 128 bytes earlier, or 127 bytes farther ahead in the code. A conditional jump instruction is only two bytes, with the first indicating the instruction and the other holding the branch distance. If you need to jump to a label farther away than that you must reverse the sense of the condition, and jump to a near label that skips over another, unconditional jump:

```
Cmp AX, BX           ;we want to jump to Label: if AX is greater
```



```

Jna NearLabel          ;so jump to NearLabel if it's NOT greater
Jmp Label              ;this goes to Label: which is farther away
NearLabel:
:
:

```

As used here, the unconditional `Jmp` instruction can branch to any location within the current code segment. There is also a short form of `Jmp`, which requires only two bytes of code instead of three. If you are jumping backwards in the program and the address is within 128 bytes, MASM uses the shorter form automatically. But if the jump is forward, you should specify `Short` explicitly: `Jmp Short Label`. Some non-Microsoft assemblers do not require you to specify `Short`; the newest MASM version 6.x also adjusts its generated code to avoid the extra wasted byte.

DOS Strings

When string information is passed to a DOS routine, for example when giving a file or directory name, the string must end with a `CHR$(0)`. In DOS terminology this is called an ASCIIZ string. (Do not confuse this with a `CHR$(26)` Ctrl-Z which marks the end of a file.) Unlike BASIC, DOS does not use string descriptors, so this is the only way DOS can tell when it has reached the end. By the same token, when DOS returns a string to a calling program, it marks the end with a trailing zero byte.

When passing a string to a DOS service from BASIC you must either concatenate a `CHR$(0)` manually, or add extra code within the assembler routine to copy the name into local storage and add a zero byte to the copy. From BASIC you would therefore use something like this:

```
CALL Routine(FileName$ + CHR$(0))
```

BASIC Fixed-Length Strings

Fixed-length strings and the string portion of a `TYPE` variable do not use a string descriptor, which you might think would require a different strategy to access them. But whenever a fixed-length string is used as an argument to an assembler routine or BASIC subprogram, BASIC first copies it into a temporary conventional string, and it is the temporary string that is passed to the routine. When the routine returns, BASIC copies the characters back into the original fixed-length string. Thus, any routine written in assembly language that expects a descriptor will work correctly, regardless of the type of string being sent.

Of course, this copying requires BASIC to generate many extra bytes of assembler code for each call. If you do not want BASIC to create a temporary string copy from one of a fixed-length, you must first define the string as a `TYPE` like this:

```

TYPE FLen
  S AS STRING * 20
END TYPE
DIM FString AS FLen

```

Though this appears to be the same as defining FString as a string with a fixed length of 20, there is an important difference: declaring it as a TYPE tells BASIC not to make a copy. That is, BASIC does not treat FString as a string, as long as the ".S" portion that identifies it as a string is not used. Here's an example based on the FLen TYPE that was defined above:

```
DIM FString AS FLen          'FString is a TYPE variable
FString.S = "This is a test" 'assign the string portion
CALL Routine(FString)       'call the routine without .S
```

Here, the address of the first character in the string is passed to the routine, as opposed to the address of a temporary string descriptor. We have told BASIC to call Routine, and pass it the entire FString TYPE but without interpreting the .S string component. This next example does cause BASIC to create a temporary copy:

```
CALL Routine(FString.S)
```

The short assembly language routine that follows expects the address of a fixed-length string with a length of 20, as opposed to the address of a string descriptor. The routine then copies the characters to the upper-left corner of a color monitor.

```
Push BP          ;access the stack as usual
Mov  BP,SP
Mov  SI,[BP+6]   ;SI points to the first character
Mov  DI,0        ;the first address in screen memory
Mov  AX,0B800h   ;color monitor segment when in text mode
Mov  ES,AX       ;move into ES through AX
Mov  CX,20       ;prepare to copy 20 characters
Cld              ;clear the direction flag to copy forward
More:
Movsb            ;copy a byte to screen memory
Inc  DI          ;skip over the attribute byte
Loop More        ;loop until done
Pop  BP          ;restore BP
Ret  2           ;return to BASIC
```

Recall that the color monitor segment value of 0B800h must be assigned to ES through AX, because it is not legal to assign a segment register from a constant. Also, notice the way that DI is cleared to zero. Although `Mov DI, 0` indeed moves a zero into DI, this is not the most efficient way to clear a register. Any time a numeric value is used in a program (0 in this case), that much extra space is needed to store the actual value as part of the instruction. A preferred method for clearing a register is with the `Xor` instruction. That is, `Xor DI, DI` gives the same result as `Mov DI, 0` except it is one byte shorter and slightly faster.

When `Xor` is performed on any two values, only those bits that are different are set to 1. But since the same register is used here for both operands, all of the result bits will be cleared to 0. The code for using `Xor` is decidedly less obvious, but you'll see `Xor` used this way very often in assembly listings in magazines and books. Another, equally efficient way to clear a register is to subtract it from itself using `Sub AX, AX`.

Far Strings in BASIC PDS

Accessing near strings in QuickBASIC and BASIC PDS is a relatively simple task, because both the descriptor and the string data are known to be in near DGROUP memory. But BASIC PDS also supports far strings, where the data may be in a different segment. The composition of a far string descriptor was shown in Chapter 2; however, you do not need to manipulate these descriptors yourself directly.

BASIC PDS includes two routines—StringLength and StringAddress—that do the work of locating far strings for you. Further, because Microsoft could change the way far strings are organized in the future, it makes the most sense to use the routines Microsoft supplies. If the layout of far string descriptors changes, your program will still work as expected.

StringLength and StringAddress expect the address of the string descriptor, and they return the string's length and segmented address respectively. Note that while far string data may be in nearly any segment, the descriptors themselves are always in DGROUP. Also note that these routines are not very well-behaved. In particular, registers you may be using are changed by the routines. To solve this problem and also to let you get all of the information in a single call, I have written the StringInfo routine. StringInfo is contained in the FAR\$.ASM file on the accompanying disk.

```
;from an idea originally by Jay Munro
.MODEL Medium, Basic
  Extn StringAddress:Proc ;these are part of PDS
  Extn StringLength:Proc

.CODE
StringInfo Proc Uses SI DI BX ES

  Pushf                ;save the flags manually

  Push ES              ;save ES for later
  Push SI              ;pass incoming descriptor
  Call StringAddress   ;call the PDS routine

  Pop  ES              ;restore ES for StringLength
  Push AX              ;save offset and segment
  Push DX              ; returned by StringAddress

  Push SI              ;pass incoming descriptor
  Call StringLength    ;get the length
  Mov  CX,AX           ;copy the length to CX

  Pop  DX              ;retrieve the saved Segment
  Pop  AX              ;and the address

  Popf                 ;restore the flags manually
  Ret                 ;restore registers and return

StringInfo Endp
End
```

StringInfo is called with DS:SI pointing to the string descriptor, and it returns the length in CX and the address of the string data in DX:AX. Although StringInfo could be designed to return the segment in DS or ES, it is safer to assign the segment registers yourself manually.

Notice the Uses clause—this tells MASM that the named registers must be preserved, and generates additional code to push those registers upon entry to the procedure, and pop them again upon exit.

Also notice the new Extrn directive at the beginning of the source file. These tell the assembler that the stated routines are not in the current source file. MASM then places the external name in the object file header, with instructions to LINK to fill in the address portion of the Call. Data must also be declared as external if it is not in the same source file as the routine being assembled. When a data item is to be made available to other modules, you must also have a corresponding Public statement in that file for the same reason:

```
.Model Medium, Basic
.Data
Public MyData
MyData DW 12345
:
```

Accessing Arrays

As you have seen, a conventional variable is passed to an assembly language subroutine by placing its address onto the stack. If the variable is a string, then the address passed is that of its descriptor, and the string data address is read from there. Accessing array elements is only slightly more involved, because array elements are always stored in adjacent memory locations. Let's look first at integer arrays.

When BASIC encounters the statement `DIM X%(100)` in your program, it allocates a contiguous block of memory 202 bytes long. (Unless you first used the statement `OPTION BASE 1`, dimensioning an array to 100 means 101 elements.) The first two bytes in this block hold the data for `X%(0)`, the next two bytes hold `X%(1)`, and so forth. When you ask `VARPTR` to find `X%(0)`, the address it returns is the start of this block of memory.

The address of subsequent array elements may then be easily computed from this base address. But with a dynamic array, the segment that holds the array may not be the same as the segment where regular variables are stored. Also, huge arrays that span more than 64K require extra care when crossing a 64K segment boundary.

String arrays are structured in a similar fashion, in that each element follows the previous one in memory. For each string array element that is dimensioned, four bytes are set aside. These bytes comprise a table of descriptors which contain the length and address words for each element in the array. But the important point is that once you know where one element or string descriptor is located, it is easy to find all of those that are adjacent. Following is a QuickBASIC example that shows how to locate `Array$(15)`, based on the `VARPTR` address of `Array$(0)`.

```

DIM Array$(100)
Array$(15) = "Find me"

Descriptor = VARPTR(Array$(0))
Descriptor = Descriptor + (4 * 15)

Length = PEEK(Descriptor) + 256 * PEEK(Descriptor + 1)
PRINT "Length ="; Length

Addr = PEEK(Descriptor + 2) + 256 * PEEK(Descriptor + 3)
PRINT "String = ";
FOR X = Addr TO Addr + Length - 1
  PRINT CHR$(PEEK(X));
NEXT

```

Dynamic Arrays

Most of the routines shown so far manipulated variables that are located in near memory. BASIC can store numeric, TYPE, and fixed-length string arrays in far memory, and additional steps are needed to read from and write to those arrays.

When an assembly language routine receives control after a call from BASIC, it can access your regular variables because they are in the default data segment. Most memory accesses assume the data is in the segment held in the DS register. For example, the statement `MOV [BX], AX` assigns the value in AX to the memory location identified by BX within the segment held in DS. Likewise, `Sub [DI+10], CX` subtracts the value held in CX from the memory address expressed as DI+10, where that address is again in the default data segment.

It is also possible to specify a segment other than the current default. One way is with a *segment override* command, like this:

```
Mov ES:[BX], AX
```

Here, the segment held in ES is used instead of DS. A segment override adds only one byte of code, so it is quite efficient. If you plan to access data in a different segment many times, you can optionally set DS to that segment. However, it is mandatory that you reset DS to its original value before returning to BASIC. You must also understand that changing DS means you no longer have direct access to DGROUP anymore. In that case you could use the stack segment as an override, since the stack segment is always the same as the data segment in a BASIC program. The next short example shows this in context.

```

Push DS                ;save DS
Mov DS,FarSegment     ;now DS points to your far data
.                     ;access that far data here
.
Mov AX,SS:[Variable]  ;access Variable in DGROUP
.                     ;access more far data here
Pop DS                ;restore DS before returning

```

When Microsoft introduced QuickBASIC version 2.0, one of the most exciting new features it offered was support for dynamic numeric arrays. Unlike QuickBASIC near strings, string arrays, and non-array variables, these arrays are always located outside of BASIC's near 64K data segment. This means that an assembler routine needs some way to know both the address and the segment for an array element that is passed to it.

In general, routines you design that work on an entire array will be written to expect a particular starting element. The routine can then assume that all of the subsequent elements lie before or after it in memory. Unfortunately, this does not always work unless you add extra steps. If you call an assembly language routine passing one element of a far-memory dynamic array like this:

```
CALL Routine(Array(1))
```

BASIC makes a copy of the array element into a temporary variable in near memory, and then passes the address of that copy to the routine. Thus, while the routine can still receive an array element's value, it has no way to determine its true address. And without the address, there is no way to get at the rest of the array.

Since being able to pass an entire array is obviously important, BASIC supports two options to the CALL command—SEG and BYVAL. The SEG keyword indicates that both the address and the segment are to be passed on the stack, and it also tells BASIC not to make a copy of the array element. SEG is used with an array element (or any variable, for that matter) like this:

```
CALL Routine(SEG Array%(1))
```

You could also send the segment and address manually, like this:

```
CALL Routine(BYVAL VARSEG(Array%(1)), BYVAL VARPTR(Array%(1)))
```

In both cases, BASIC first pushes the segment where the element resides onto the stack, followed by the element's address within that segment. By pushing them in this order the routine can conveniently use either Lds (Load DS) or Les (Load ES) to get both the segment and address in one operation:

```
Les DI,[BP+6]           ;if using manual stack addressing
```

or

```
Les BX,[StackArg]      ;if using MASM's simplified directives
```

Les loads four bytes in one operation, placing the lower word at [BP+6] into the named register (DI in the first example case), and the higher word at [BP+8] into ES. Lds works the same, except the higher word is instead moved into DS. Once the segment and address are loaded, you can access all of the array elements:

```
Push DS                ;save DS
Lds SI,[BP+6]          ;now DS:SI points at first element
Mov [SI],AX            ;assign Array%(1) from AX
```

```

Add  SI,2           ;now SI points at the next element
Mov  [SI],BX       ;assign Array%(2) from BX
Pop  DS            ;restore DS
.                  ;continue
.

```

If Les were used instead of Lds, then an ES: override would be needed to assign the elements. Although you must always preserve the contents of DS regardless of the version of BASIC, some registers need to be saved only when using BASIC PDS far strings. Other registers do not need to be saved at all. Table 12-4 shows which registers must be preserved based on the version of BASIC.

QuickBASIC and PDS near strings	BASIC PDS far strings
DS	DS
SS	SS
BP	BP
SP	SP
	ES
	SI
	DI

Table 12-4: The registers that must be preserved in an assembly language subroutine.

Besides having to save and restore the registers shown in Table 12-4, you must also be sure that the Direction Flag is cleared to forward before returning to BASIC. The Direction Flag affects the 8088 string operations, and is by default set to forward. You can usually ignore the direction flag unless you set it to backwards explicitly with the Std instruction. In that case, you must use a corresponding Cld command.

Huge Arrays

A huge array is one that spans more than one 64K segment, and as you can imagine, it requires extra steps to access all of the elements. That is, the assembler routine must know which elements are in what segment, and manually load those segments as needed. The following code fragment shows how to walk through all of the elements in a huge integer array, and just for the sake of the example adds each element to determine the sum of all of them.

A simple setup example and call syntax for this routine is as follows:

```

REDIM Array$(1 TO 30000)
FOR X% = 1 TO 30000
  Array$(X%) = X%
NEXT

CALL SumArray(SEG Array$(1), 30000, Sum&)
PRINT "Sum& ="; Sum&

```

And here's the code for the SumArray routine:

```

.Model Medium, Basic

```

.Code

SumArray Proc Uses SI, Array:DWord, NumEls:Word, Sum:Word

```
Push DS          ;save DS so we can restore it later
Push SI          ;PDS far strings require saving SI too

Xor  AX,AX       ;clear AX and DX which will accumulate
Mov  DX,AX       ; the total

Mov  BX,NumEls   ;get the address for NumElements%
Mov  CX,[BX]     ;read NumElements% before changing DS
Lds  SI,Array    ;load the address of the first element
Jcxz Exit       ;exit if NumElements = 0

Do:
Add  AX,[SI]     ;add the value of the low word
Acd  DX,[SI+2]   ;and then add the high word
Add  SI,4        ;point to the next array element

Or   SI,SI       ;are we beyond a 32k boundary?
Jns  More        ;no, continue

Sub  SI,8000h    ;yes, subtract 32k from the address
Mov  BX,DS       ;copy DS into BX
Add  BX,800h     ;adjust the segment to compensate
Mov  DS,BX       ;copy BX back into DS

More:
Loop Do          ;loop until done

Exit:
Pop  SI          ;restore SI for BASIC
Pop  DS          ;restore DS and gain access to Sum&
Mov  BX,Sum      ;get the DGROUPE address for Sum&
Mov  [BX],AX     ;assign the low word
Mov  [BX+2],DX   ;and then the high word

Ret             ;return to BASIC

SumArray Endp
End
```

The segment bounds checking is handled by the six lines that start with `Or SI,SI`. The idea is to see if the address is beyond 32767, subtract 32768 if it is, and then adjust the segment to compensate. The most direct way would have been with `Cmp SI,32767` and then `Ja More`, but `Cmp` used this way generates three bytes of code, whereas `Or` creates only two bytes. Since `Or` sets the Sign flag if the number is negative (above 32767), you can use it to know when the address adjustment is needed.

Because it is not legal to add or subtract a segment register, DS is first copied to BX, 800h is added to that, and the result is then copied back to DS. 800h is used instead of 8000h (32768) because a new segment begins every 16 bytes; that is, adding 800h to a segment value is the same as adding 8000h to the address.

`SumArray` also introduces a new instruction: `Acd` means Add with Carry, and it is used to add long integer values that by definition span two words. When you add two registers—say, AX and BX—if the

result exceeds 65535 only the remainder is saved. However, the Carry Flag is set to indicate the overflow condition. `Adc` takes this into account, and adds one extra to its result if the Carry Flag is set. Therefore, whenever two long integers are added you'll use `Add` to combine the lower words, and `Adc` for the high words. Similarly, subtracting long integers requires that you use `Sub` to subtract the lower words and then `Sbb` (Subtract with Borrow) on the upper words.

Although the details are hidden from you, when more than one parameter is passed to an assembly language routine it is the last in the list that is at `[BP+6]` on the stack. The previous argument is at `[BP+8]`, and the one before that is at `[BP+10]`. Because the stack grows downward as new items are pushed onto it, each subsequent item is at a lower address.

Finally, in a real program this routine would probably be designed as a function. Using a function avoids having to pass the `Sum&` parameter to receive the returned value, and helps reduce the size of the program.

Assembler Functions

Designing a procedure as a function lets you return information to a program, but without the need for an extra passed parameter. Functions are also useful because BASIC performs any necessary data type conversion automatically. For example, if you have written a function that returns an integer value, you can freely assign the result to a single precision variable.

You can also test the result of a function directly using `IF`, display it directly with `PRINT`, or pass it as a parameter to another procedure. Some typical examples are shown here:

```
SingleVar! = MyFunction%  
  
IF YourFunction&(Argument%) > 1004 THEN ...  
  
PRINT HisFunction$(Any$)
```

Beginning with QuickBASIC version 4.0, functions written in assembly language may be added to a BASIC program. To have a function return an integer value, simply place the value into the `AX` register before returning to BASIC. If the function is to return a long integer, both `DX` and `AX` are used. In that case, `DX` holds the higher word and `AX` holds the lower one.

String Functions

String functions are only slightly more complicated to design. A string function also uses `AX` as a return value, but in this case `AX` holds the address of a string descriptor you have created. The complete short string function that follows accepts an integer argument, and returns the string "False" if the argument is zero or "True" if it is not.

```

;Syntax:
;DECLARE FUNCTION TrueFalse$(Argument%)
;Answer$ = TrueFalse$(Argument%)

.Model Medium, Basic
.Data
  DescLen DW 0
  DescAdr DW 0
  True    DB "True"
  False   DB "False"

.Code
TrueFalse Proc, Argument:Word

  Mov  DescLen,4           ;assume true
  Mov  DescAdr,Offset True

  Mov  BX,Argument        ;get the address for Argument%
  Cmp  Word Ptr [BX],0    ;is it zero?
  Jne  Exit               ;no, so we were right
  Inc  DescLen            ;yes, return five characters
  Mov  DescAdr,Offset False ;and the address of "False"

Exit:
  Mov  AX,Offset DescLen  ;show where the descriptor is
  Ret                                ;return to BASIC

TrueFalse Endp
End

```

Although the function is declared using a dollar sign in the name, the actual procedure omits that. The dollar sign merely tells BASIC what type of information will be returned. It is not part of the actual procedure name. TrueFalse begins by defining a string descriptor in the .Data segment. It is also possible to store strings and other data in the code segment and access it with a CS: segment override. However, data that is returned as a function must be in DGROUP, and so must the descriptor.

The first two statements assign the descriptor to an output string length of four characters, and the address of the message "True". Then, the address of Argument is obtained from the stack, and its value is compared to zero. If it is not zero, then the descriptor is already correct and the function can proceed. Otherwise, the descriptor length is incremented to reflect the correct length, and the address portion is reassigned to show where the string "False" begins in memory. In either case, the final steps are to load AX with the address of the descriptor, and then return to BASIC.

MASM also lets you access data using simple arithmetic. For example, the descriptor could have been defined as a single pair of words with one name, and the second word could be accessed based on the address of the first one like this:

```

.Data
  Descriptor DW 0, 0
  True      DB "True"
  False     DB "False"

.Code
  .
  .

```

```

Inc  Descriptor
Mov  Descriptor+2,Offset False
:

```

Far String Functions

Far string functions require more work to write than near string functions, because of the added overhead needed to support far strings. Fortunately, BASIC includes routines that simplify the task for you. Actually, the routines to create and assign strings have always been included; it's just that Microsoft never documented how to do it before BASIC 7.0. Later in this chapter I'll show code to create strings that works with all versions of BASIC 4.0 or later.

The StringAssign routine expects six arguments on the stack, for the segment, address, and length of both the source and destination strings. StringAssign can assign from or to any combination of fixed and variable-length strings. If the length argument for either string is zero, then StringAssign knows that the address is that of a descriptor. Otherwise, the address is of the data in a fixed-length string.

Because of the added overhead of obtaining values and pushing them on the stack, I have created a short wrapper program that does this for you. MakeString accepts the same arguments as StringAssign, but they are passed using registers rather than on the stack. Of course, calling one routine that in turn calls another takes additional time. But the savings in code size when MakeString is called repeatedly will overshadow the very slight additional delay.

MakeString is called with DX:AX holding the segmented address of the source string, and CX holding its fixed length. If the source is a conventional string, CX is set to zero to indicate that. The destination address is identified with DS:DI, using BX to hold the length. Again, BX holds zero if the destination is not a fixed-length string.

```

;from an idea originally by Jay Munro
.MODEL Medium, Basic
EXTRN STRINGASSIGN:PROC

.CODE
MakeString PROC USES DS

    PUSH DX          ;push the segment of the source string
    PUSH AX          ;push the address of the source string
    PUSH CX          ;push the string length
    PUSH DS          ;push the segment of the destination
    PUSH DI          ;push the address of the destination
    PUSH BX          ;push the destination length

    CALL STRINGASSIGN ;call BASIC to assign the string
    RET

MakeString ENDP
END

```

Now, with the assistance of MakeString, TrueFalse\$ can be easily modified to work with BASIC 7 far strings:

```
.Model Medium, Basic
Extrn MakeString:Proc          ;this is in FAR$.ASM

.Data
Descriptor DW 0, 0            ;the output string descriptor
True       DB "True"
False     DB "False"

.Code
TrueFalse Proc Uses ES DS SI DI, Argument:Word

Mov  CX,4                      ;assume true
Mov  AX,Offset True

Mov  BX,Argument               ;get the address for Argument%
Cmp  Word Ptr [BX],0           ;is it zero?
Jne  @F                        ;no, so we were right

Inc  CX                        ;yes, assign five characters
Mov  AX,Offset False           ;and use the address of "False"

@@:
Mov  DX,DS                     ;assign the segment and address
Mov  DI,Offset Descriptor      ; of the destination descriptor
Xor  BX,BX                     ;assign to a descriptor
Call MakeString                ;let MakeString do the work

Mov  AX,DI                     ;AX = address of output descriptor
Ret                                ;return to BASIC

TrueFalse Endp
End
```

Notice the introduction of the new at-symbol (@) assembler directive. The at-symbol and double at-symbol label are quite useful, because they let you avoid having to create unique label names each time you specify the target of a jump. As with BASIC, creating many different label names is a nuisance, and also impinges on the assembler's working memory. When a label is defined using @@: as a name, you can jump forward to it using @F or backwards using @B. Multiple @@: labels may be used in the same program, and @F and @B always branch to the nearest one in the stated direction.

Floating Point Functions

Single and double precision functions are handled in yet another manner. Although a single precision value could be returned in the DX:AX register combination, a double precision result would need four registers, which is impractical. Further, a floating point number is most useful to BASIC if it is stored in a memory location, rather than in registers.

When BASIC invokes a floating point function it adds an extra, dummy parameter to the end of the list of arguments you pass. If no parameters are being used, it creates one. This parameter is the address

into which your routine is to place the outgoing result. Because of this added parameter, it is essential that you account for it when returning to BASIC. Thus, a function without arguments must use `Ret 2`, a function with one argument needs `Ret 4`, and so forth. Since we're using MASM's simplified directives, all that is needed is to create an extra parameter name.

The short double precision function that follows squares a double precision number much faster than using `Value# ^ 2`, and also shows how to perform simple floating point math using assembly language. You will declare and invoke `Square` like this:

```

DECLARE FUNCTION Square#(Variable#)
Result = Square#(Variable#)

;SQUARE.ASM, squares a double precision number
;
;WARNING: This file must be assembled using /e (emulator).
.Model Medium, Basic
.Code
.8087                ;allow 8087 instructions

Square Proc, InValue:Word, OutValue:Word

Mov  BX,InValue      ;get the address for InValue
FLd  QWord Ptr [BX] ;load InValue onto the 8087 stack
FMul QWord Ptr [BX] ;multiply InValue by itself

Mov  BX,OutValue     ;get the address for OutValue
FStp QWord Ptr [BX] ;store the result there
FWait                ;wait for the 8087 to finish

Mov  AX,BX           ;return DX:AX holding the full
Mov  DX,DS           ; address of the output value
Ret                    ;return to BASIC

Square Endp
End

```

This `Square` function illustrates several important points. The first is the use of MASM's `/e` switch, which lets an assembly language routine share BASIC's floating point emulator. When a BASIC program begins, it looks to see if an 8087 coprocessor is installed in the host PC. If so, it uses one set of library routines; otherwise it uses another.

The library routines that use an 8087 simply modify the caller's code to change the floating point interrupts that BASIC generates into actual 8087 instructions. It then returns to the instruction it just created and executes it. Although this adds to the time needed to perform a floating point operation, the code is patched only once. Thus, statements within a `FOR` or `DO` loop operate very quickly after the first iteration. This is very much like the method used by the BRUN library described in Chapter 1.

When no coprocessor is detected, the floating point interrupts that BASIC generates are used to invoke routines in BASIC's floating point software emulator. As its name implies, an emulator imitates the behavior of a coprocessor using assembly language commands. A coprocessor can perform a variety of floating point operations, including addition, multiplication, and rounding, as well as some transcendental functions such as logarithms and arctangents.

When you use the /e switch, MASM adds extra information to the object file header that tells LINK where to patch your 8087 instructions. LINK can then change your code to the equivalent floating point instructions, similar to the way BASIC patches its own code to change the instructions to 8087 instructions. Therefore, when you write floating point code that will be called from BASIC, your routine can tie into BASIC's emulator, and use it automatically if no coprocessor is installed.

Also, notice the .8087 directive which tells MASM not to issue an error message when it sees those instructions. Other, similar directives are .80287 and .80387, and also .80286 and .80386. These directives inform MASM that you are intentionally using advanced commands that require these processors, and have not made a typing error.

The actual body of the Square function is fairly simple. First, the address of the incoming value is retrieved from the system stack, and then the data at that address is loaded onto the coprocessor's stack using the FLd (Floating point Load) instruction. Since this is a double precision value, QWord Ptr (Quad Word Pointer) is needed to indicate the size of the data. Had the incoming value been single precision, DWord Ptr (Double Word Pointer) would be used instead. One important feature of an 8087 or software emulator is that a number may be converted from one numeric format to another simply by loading it as one data type, and then saving it as another.

The next instruction, FMul (Floating point Multiply), multiplies the value currently on the 8087 stack by the same address. Since the original value is still present, there's no need to make a new copy. Next, the destination address is placed into BX, and the result now on the 8087 stack is stored there. The trailing letter p in the FStp instruction specifies that the value loaded earlier is to be popped from the coprocessor stack.

A complete discussion of 8087 instructions and how the coprocessor stack operates goes beyond what I can hope to cover here. When in doubt about what instruction is needed, I suggest that you code a similar sample in BASIC, and then examine the code BASIC generates using CodeView. There are also several books that focus on writing floating point instructions in assembly language.

The last 8087 instruction is FWait, and it tells the 8088 to wait until the coprocessor has finished, before continuing. Because an 8087 is a true coprocessor, it operates independently of the main 8088 CPU. Once a value is loaded and the 8087 is instructed to perform an operation, the 8087 returns immediately to the program that issued the instruction and continues to process the numbers in the background. If Square exited immediately and BASIC read the returned value, there's a good chance that the 8087 did not finish and the value has not yet been stored. In that case, whatever happened to be in memory at that time would be the value that BASIC uses, which is obviously incorrect.

Experienced 8087 programmers know how long the various coprocessor instructions take to complete, and with careful planning the number of FWait commands can be kept to a minimum. However, the code that BASIC generates always finishes with an FWait. Of course, there is no need to wait when the

emulator is in use. In fact, an FWait is patched by BASIC to do nothing (`MOV AX, AX`), rather than waste time invoking an empty interrupt handler repeatedly.

As shown, Square can be added to a Quick Library for use with either QuickBASIC or BASIC PDS. Unfortunately, the information link needs to patch 8087 instructions is available only with BASIC PDS. Therefore, the following file is included in the libraries on the accompanying disk, to supply the external data that LINK requires.

```
;FIXUPS.ASM, deciphered by Paul Passarelli

FIARQQ Equ 0FE32h
FJARQQ Equ 04000h
FICRQQ Equ 00E32h
FJCRQQ Equ 0C000h
FIDRQQ Equ 05C32h
FIERQQ Equ 01632h
FISRQQ Equ 00632h
FJSRQQ Equ 08000h
FIWRQQ Equ 0A23Dh

Public FIARQQ
Public FJARQQ
Public FICRQQ
Public FJCRQQ
Public FIDRQQ
Public FIERQQ
Public FISRQQ
Public FJSRQQ
Public FIWRQQ
End
```

These values are added to the floating point instruction bytes during the linking process, and the addition converts those statements into equivalent BASIC floating point interrupt commands. For example, the 8087 statement `Fld DWord Ptr [1234h]` is represented in memory as the following series of Hexadecimal bytes:

```
9B D9 06 34 12
```

After LINK adds the value FIDRQQ (5C32h) to the first two bytes of this command the result is:

```
CD 35 06 34 12
```

And when disassembled back to assembler mnemonics, the CD35h displays as `Int 35h`. The three bytes that follow are always left unchanged, and they specify the type of operation—DWord Ptr on a memory location—and the address of that location.

Floating Point Comparisons

At the core of any sorting or searching routine is an appropriate comparison function. Previous chapters showed how to compare string data, and as you can imagine comparing floating point values is much

more complex. But now that you know how to tap into BASIC's floating point routines it is almost trivial to effect a floating point comparison. The routines that follow let you compare either single or double precision values, by passing them as arguments.

```
;COMPAREFP.ASM, compares floating point values

;WARNING: This file must be assembled using /e (emulator)

.Model Medium, Basic
  Extrn B$FCMP:Proc      ;BASIC's FP compare routine

.8087                    ;allow coprocessor instructions
.Code

CompareSP Proc, Var1:Word, Var2:Word

  Mov  BX,Var2           ;get the address of Var1
  Fld  DWord Ptr [BX]   ;load it onto the 8087 stack
  Mov  BX,Var1           ;same for Var2
  Fld  DWord Ptr [BX]
  FWait
  Call B$FCMP           ;compare the values, (and pop both)

  Mov  AX,0              ;assume they're the same
  Je   Exit             ;we were right
  Mov  AL,1              ;assume Var1 is greater
  Ja   Exit             ;we were right
  Dec  AX                ;Var1 must be less than Var2
  Dec  AX                ;decrement AX to -1

Exit:
  Ret                   ;return to BASIC

CompareSP Endp

CompareDP Proc, Var1:Word, Var2:Word

  Mov  BX,Var2           ;as above
  Fld  QWord Ptr [BX]
  Mov  BX,Var1
  Fld  QWord Ptr [BX]
  FWait
  Call B$FCMP

  Mov  AX,0
  Je   Exit
  Mov  AL,1
  Ja   Exit
  Dec  AX
  Dec  AX

Exit:
  Ret

CompareDP Endp
End
```


Like the Compare3 function shown in Chapter 8, CompareSP and CompareDP are integer functions that return -1, 0, or 1 to indicate if the first value is less than, equal to, or greater than the second. Therefore, to use these from BASIC you would invoke them like this:

```
IF CompareSP%(Value1!, Value2!) = -1 THEN
  'the first value is smaller than the second
END IF
```

And to test if the first is equal to or greater than the second you would instead do this:

```
IF CompareSP%(Value1!, Value2!) >= 0 THEN
  'the first value is equal or greater
END IF
```

You can also use these functions from assembly language. But if you do this, I suggest a simple modification. A comparison routine meant to be called from another assembler routine would not generally return the result in the registers. Rather, it would leave the flags set appropriately for a subsequent Ja or Jne branch.

Fortunately, BASIC's B\$FCMP routine already does this. Therefore, you will make a copy of the COMPAREF.ASM source file, and delete the six lines between the call to B\$FCMP and the Ret instruction. You can also remove the Exit: label if you like, although its presence causes no harm. Of course, the code itself is so simple that the best solution may be to simply duplicate the same instructions inline in your routine.

Exploiting MASM's Features

Each example I have shown so far introduced another useful MASM feature. For example, you learned how MASM lets you establish data memory with an initial value, so you don't have to assign it explicitly. But there are several other features you should know about as well. One is conditional assembly.

Conditional Assembly

With conditional assembly you can specify that only certain portions of a file are to be assembled. This makes it easier to maintain two different versions of a routine, for example one for near strings and one for far strings. If you had to create two separate copies of the source file, any improvements or bug fixes that you add would have to be done twice.

There are two ways that a section of code can be optionally included or excluded. One is to define a constant at the beginning of the source file, and then test that constant using a form of IF and ELSE test. Like BASIC, MASM lets you define constant values using meaningful names. The problem with this method—albeit a minor one—is that you must alter the code prior to assembling each version. The example that follows shows how this kind of conditional assembly is employed.

```

MyConst = 1
.
.
IF MyConst
    ;do whatever you want here
ELSE    ;the ELSE is optional
    ;do whatever else you want here
ENDIF
.
.

```

The idea is that if you want the code that follows the IF test to be assembled, you would use a non-zero value for MyConst. If you wanted to create an alternate version using the code within the optional ELSE block, you would change the value to be zero.

You can also use IFE (If Equal to zero) to test if a constant is zero. And this brings up another interesting MASM feature. There are actually two types of constants you can define. The constant MyConst shown above is called a *redefinable constant*, because you can actually change its value during the course of a program. The other type of constant is defined using the Equ (Equate) directive, and may not be changed:

```
YourConst Equ 100
```

Redefinable constants are often used in repeating macros, and macros are discussed later in this section.

The other way to tell MASM that it is to assemble just a portion of the file is with IFDEF. IFDEF (If Defined) tests if a constant has been defined at all, as apposed to comparing for a specific value. The value of this approach is that you can define a constant on the MASM command line when you run it. The first example below tells MASM to assemble the code within the IFDEF block, and the second tells it to not to.

```

C:\ASM\> masm program /def myconst ;
C:\ASM\> masm program ;

```

Here's the portion of the routine that is being assembled conditionally:

```

IFDEF MyConst
    ;do something optional here
ENDIF

```

Likewise, IFNDEF (If Not Defined) tests if a constant has not been defined when reversing the logic is more sensible to you. MASM includes a great number of such conditional tests, and only by reading that section of the MASM manual will you become familiar with those that are the most useful.

Comment Blocks

Another useful MASM feature that I personally would love to see added to BASIC is multi-line comment blocks. The Comment command accepts any single character you choose as a delimiter, and considers everything thereafter to be comments until the same character is encountered. Many programmers use a vertical bar, because it is not a common character:

```
Comment |
This program is intended to blah blah blah, and it works by loading AX with blah
blah blah.
|
```

Besides avoiding the need to place an explicit semicolon on each comment line, this also makes it easy to remark out large sections of code while you are debugging a routine.

Quoted Strings

Yet another useful feature is MASM's willingness to use either single or double quotes to indicate ASCII text and individual characters. In BASIC, if you want to specify a double quote you must use CHR\$(34)—it simply is not legal to use "", where the quote in the middle is the character being defined.

```
With the introduction of VB/DOS triple quotes may now
be used for this purpose.
```

If you need to define a double quote simply surround it with apostrophes like this:

```
SomeData DB '''
Mov AH, '''
```

Or you can place a single quote within double quotes like this:

```
Add DL, '''
```

MASM can use either convention as needed, which is a feature I personally like a lot.

Length and Address Self-Calculation

Whenever MASM sees the dollar sign (\$) operator it interprets that to mean here, or the current address. This can be used both for data and code, though it is more common with data as the example below illustrates.

```
.Data
Descriptor DW MsgLen, Address
Message DB "This is a message."
Address = Offset Message
MsgLen = $ - Address
```

The expression `$ - Address` tells the assembler to take the current data address, and subtract from that the address where `Message` begins. This is a very powerful concept because it frees the programmer from many tedious calculations. In particular, if the string contents are changed at a later time, the new length is recalculated by MASM automatically.

Defining Data Structures

To assist you in manipulating data structures, MASM offers the `Struc` directive. This is identical to BASIC's `TYPE` statement, whereby you define the organization of a collection of related data items. The example below shows how to define a custom data structure using BASIC, followed by an equivalent MASM `Struc` definition.

BASIC:

```
TYPE MyType
  LastName AS STRING * 15
  FirstName AS STRING * 12
  ZipCode AS STRING * 5
  RecordPtr AS LONG
END TYPE
DIM MyVar AS MyType
```

MASM:

```
Struc MyStruc
  LastName DB 15 Dup (?)
  FirstName DB 12 Dup (?)
  ZipCode DB 5 Dup (?)
  RecordPtr DD ?
MyStruc Ends
MyVar DB Size MyStruc Dup (?)
```

Like BASIC, defining a structure merely establishes the number and type of data items that will be stored; memory is not actually set aside until you do that manually. In BASIC, you must use `DIM` to establish the memory that will hold the `TYPE` variable. In assembly language you instead use `DB` in conjunction with the `Size` directive, to set aside the appropriate number of bytes.

Each component of the Structure is defined using an identifying name and a corresponding data type. Then, whenever a structure member is referenced in your assembler routine, MASM replaces it with a number that shows how far into the structure that member is located. MASM uses the same syntax as BASIC, with a period between the data name and the structure identifier. Here are a few examples:

```
Mov AL, [BX+MyVar.LastName] ;same as Mov AL, [BX+15]
Les DI, [MyVar.RecordPtr] ;loads ES:DI from RecordPtr
```

Minimizing DGROUP Usage

In many cases you will store the variables your routines need in DGROUP using the .Data directive. As with static subprograms and functions in BASIC, this data will not change between subroutine calls. But this also means that these variables are combined into the same 64k segment that is shared with BASIC. When there are many variables or many different routines each with their own variables, this can significantly reduce the amount of near memory available to BASIC. There are two effective solutions to this problem.

Local Variables

One way to reduce the DGROUP impact of many variables is to place some of them onto the system stack. MASM lets you do this automatically with its Local directive, or you can do it manually by subtracting the requisite number of bytes from SP. Of course, there is only so much room on the stack, so this approach is most useful when there are many routines and each has less than 1K or so of data. Stack variables are also useful when programming for OS/2 or Windows. These operating systems require that all of your procedures be reentrant so static variables cannot be used.

The example below creates room for fifty words of local storage on the stack, and then clears the variables to zero.

```
Routine Proc Uses ES DI, Param1:Word, Param2:Word
  Sub  SP,100          ;50 words = 100 bytes
  Push SS              ;assign ES from SS
  Pop  ES
  Mov  DI,SP           ;point DI to the start of storage
  Xor  AX,AX           ;fill with zeros
  Mov  CX,50           ;clear fifty words
  Rep  Stosw           ;store AX CX times at ES:[DI]
  .                  ;the routine continues
  .
  Add  SP,100          ;restore SP to what it had been
  Ret                  ;return to BASIC
Routine Endp
```

MASM can also allocate the stack memory automatically for you using Local like this:

```
Routine Proc Uses ES DI, Param1:Word, Param2:Word
  Local Buffer [100]:Byte
  Lea  DI,Buffer       ;clear the stack variables here
  .                  ;the routine continues
  .
  Ret                  ;return to BASIC
Routine Endp
```

As you can see, Local lets you refer to the start of the local stack data area by name. Notice how Lea is required here, because the address of Buffer is expressed as an offset from BP. That is, MASM translates the Lea instruction to Lea DI, [BP-100]. You cannot use Mov DI, Offset Buffer because Buffer's address—which is based on the current setting of the stack pointer—is not known when the routine is assembled or linked.

In this case only one local block is defined, so you could also use `MOV DI, SP` to set `DI` to point to the start of the data. It is not strictly necessary to clear the stack space before using it, but it is important to understand that whatever junk happened to be in memory at that time will still be there after using `Local`.

It is also important to be aware of a number of bugs with the `Local` directive. I have found that limiting the use of `Local` to a single set of data as shown here is safe with all MASM versions through 5.1. Using multiple `Local` directives defined with data structures can result in the wrong part of the stack being written to when a structure member is accessed by name.

Storing Data in the Code Segment

Another time-honored technique for conserving `DGROUP` memory is to place selected variables into the code segment. In most cases storing data for a routine in the code segment will make your programs slightly larger and slower, because of the need for an added `CS: segment override`. But when large amounts of data must be accommodated, this can be very valuable indeed. One advantage to using the code segment is that you can establish initial values for the data, which is not possible when using the stack.

As an example of this technique, I have written a string function called `Message$` that stores a series of messages in the code segment. In this case only a single `CS: segment override` is needed, so the impact of using the code segment for data is insignificant. `Message$` is designed to be declared and invoked as follows:

```
DECLARE FUNCTION Message$(BYVAL MsgNumber%)
Result$ = Message$(AnyInt%)
```

`Message$` is table driven, which makes it simple to modify the routine to change or add messages without having to make any changes to the function's structure. As shown here, `Message$` is designed to return the name of a weekday, given a value between one and seven. You can easily modify it to return other strings of nearly any length.

```
.Model Medium, Basic
  Extrn B$ASSN:Proc          ;BASIC's assignment routine

.Data
  Descriptor DD 0           ;the output string descriptor
  Null$      DD 0           ;use this to return a null
                                   ; (needed for BASIC PDS only,
                                   ; but okay with QuickBASIC)

.Code

Message Proc Uses SI, MsgNumber:Word

  Mov  SI,Offset Messages    ;point to start of messages
  Xor  AX,AX                 ;assume an invalid value

  Mov  CX,MsgNumber          ;load the message number
  Cmp  CX,NumMsg             ;does this message exist?
```

```

Ja    Null                ;no, return a null string
Jcxz  Null                ;ditto if they pass a zero

Do:                                     ;walk through the messages
Lods  Word Ptr CS:0        ;load and skip over this message's length
Dec   CX                  ;show that we read another
Jz    Done                ;this is the one we want

Add   SI,AX                ;skip over the message text
Jmp   Short Do            ;continue until we're there

Done:
Or    AX,AX                ;are we returning a null?
Jz    Null                ;yes, handle that differently
Push  CS                  ;no, pass the source segment

Done2:
Push  SI                  ;and the source address
Push  AX                  ;and the source length

Push  DS                  ;pass the destination segment
Mov   AX,Offset Descriptor ;and the destination address
Push  AX
Xor   AX,AX                ;0 means assign a descriptor
Push  AX                  ;pass that as well

Call  B$ASSN              ;let B$ASSN do the dirty work
Mov   AX,Offset Descriptor ;show where the output is
Ret

Null:
Push  DS                  ;pass the address of Null$
Mov   SI,Offset Null$
Jmp   Short Done2

Message Endp

;----- DefMsg macro that defines messages
DefMsg Macro Message
LOCAL MsgStart, MsgEnd    ;;local address labels
NumMsg = NumMsg + 1      ;;show we made another one
IFB <Message>            ;;if no text is defined
    DW 0                  ;;just create an empty zero
ELSE                      ;;else create the message
    DW MsgEnd - MsgStart  ;;first write the length
    MsgStart:              ;;identify the starting address
    DB Message            ;;define the message text
    MsgEnd Label Byte     ;;this marks the end
ENDIF
Endm

Messages Label Byte       ;the messages begin here
NumMsg = 0                ;tracks number of messages
                           ;DO NOT MOVE this constant

DefMsg "Sunday"
DefMsg "Monday"
DefMsg "Tuesday"
DefMsg "Wednesday"
DefMsg "Thursday"
DefMsg "Friday"

```

```
DefMsg "Saturday"  
End
```

After declaring BASIC's B\$ASSN routine as being external, Message\$ defines two string descriptors in the Data segment. The first is used for the function output when returning a normal message, and the second is used only when returning a null string. In truth, the need for a separate output descriptor and the slight added steps to detect the special case of a null output string is needed only with BASIC PDS far strings. And this brings up an important point.

It is impossible to write one assembly language subroutine that can work with both QuickBASIC and BASIC PDS far strings using the normal, documented methods. To create a string function for use with QuickBASIC and PDS near strings, you define and fill in a string descriptor in DGROUP, and assign its address in AX before returning to BASIC. And to return a far string as a function for PDS requires calling the internal STRINGASSIGN routine that Microsoft provides with PDS. STRINGASSIGN works with both near and far strings in PDS, but is not available in QuickBASIC.

The trick is to use the undocumented name B\$ASSN, which is really the same thing as STRINGASSIGN. The big difference, though, is that B\$ASSN is available in all versions of BASIC 4.0 and later. When near strings are used the B\$ASSN routine is extracted from the near strings library. When linking with far strings a different version is used, extracted by LINK from the far strings library. This is a powerful concept to be sure, and one we will use again for other examples later on in this chapter.

Message\$ begins by loading SI with the starting address of a table of messages. These messages are located at the end of the source file in the code segment, and each is preceded with the length of the text. Although it may not be obvious from looking at the source listing, the message data is actually structured like this:

```
DW 6  
DB "Sunday"  
DW 6  
DB "Monday"  
.  
.
```

Next, AX is cleared to zero just in case the incoming string number is illegal. Later in the program AX holds the length of the output string; clearing it here simply makes the program's logic more direct.

CX is then loaded with the message number the caller asked for. If CX is either higher than the available number of messages or zero, the program jumps to the code that returns a null string. Otherwise, a small loop is entered that walks through each message, decrementing CX as it goes. When CX reaches zero, SI is pointing at the correct message and AX is holding its length. Otherwise, the current length is added to SI, thus skipping over that data.

Notice the unusual form of the Lodsw statement, to allow it to work with a CS: override. MASM has a number of quirks that are less than intuitive, and this is but one of them. Normally you would use either Lodsb or Lodsw, to indicate loading either a byte into AL or a word into AX. But when you use a

segment override MASM requires omitting the "b" or "w" Lods suffix, and you must state Byte Ptr or Word Ptr explicitly. Then, a dummy argument must be placed after the override colon.

MASM Macros

The last new feature this listing introduces is the use of macros. The most basic use of MASM macros is to define a block of code once, and then repeat it multiple times with a single statement. This is not unlike keyboard macro programs such as Borland's SuperKey, that let you assign a string of text to a single key. For example, you could press Alt-S and SuperKey will type "Very truly yours", five Enter keys, and then your name. MASM macros also offer many other interesting and useful capabilities, including the ability to accept arguments.

```
I should mention that the main point of the DefMsg
macro is to make this function easy to modify, so you
can create other, similar string functions from this
same routine.
```

Before attempting to explain the DefMsg (Define Message) macro I designed for use with Message\$, let's consider some macro basics.

Say, for example, you find that a particular routine needs to push the same five registers many times during the course of a procedure. To simplify this task you could define a macro—perhaps named PushRegs—that performs the code sequence for you. Such a macro definition would look like this:

```
PushRegs Macro
  Push AX
  Push BX
  Push SI
  Push DS
  Push ES
PushRegs Endm
```

Now, each time you want to execute this series of instructions you would simply use the command PushRegs. Please understand that a macro is not the same as a called subroutine. The assembler still places each Push command in sequence into your source code each time the macro is invoked. But a simple macro like this can reduce the amount of typing you must do, and minimize errors such as pushing registers in the wrong order. And in some cases Macros also make your code easier to read.

As I mentioned, a MASM macro can accept arguments, and it can even be designed to accept a varying number of them. If you need to push three registers but which ones may change, you would define PushRegs like this:

```
PushRegs Macro Reg1, Reg2, Reg3
  Push Reg1
  Push Reg2
  Push Reg3
Endm
```

Then to push AX, SI, and DI you would invoke PushRegs as follows:

```
PushRegs AX, SI, DI
```

Of course, a corresponding PopRegs macro would be defined similarly. Once a macro has been defined you can pass any legal argument to it. For example, you could also use this:

```
PushRegs AX, Word Ptr [BP-20], IntVar
```

Here, you are pushing AX, the word 20 bytes below where BP points to on the stack, and the integer variable named IntVar.

A useful enhancement to this macro would let you pass it a varying number of parameters. The PushM macro that follows accepts any number of arguments (up to eight), and pushes each in sequence.

```
PushM Macro A,B,C,D,E,F,G,H      ;;add more place-holders to suit
IRP CurArg, <A,B,C,D,E,F,G,H>   ;;repeat for each argument
IFNB <CurArg>                  ;;if this arg is not blank
Push CurArg                      ;;push it
    ENDF
Endm                             ;;end of repeat block
Endm                             ;;end of this macro
```

From this you can create a complementary PopM macro by changing the name, and also changing the Push instruction to Pop.

The IRP command works much like a FOR/NEXT loop in BASIC, and tells MASM to repeat the following statements for each argument that was given. IFNB (If Not Blank) then tests each argument to see if it was in fact present in the incoming list of parameters. In this case, CurArg assumes the name of the argument, and the Push instruction is expanded to specify that name.

There is no disputing that the syntax of a MASM macro is confusing at best. Having to enclose some arguments in angle brackets but not others requires frequent visits to the MASM manual. Further, a MASM macro is virtually impossible to debug. If you write a macro incorrectly or create a syntax error, MASM reports an error at the line where the macro was invoked, rather than at the line containing the error in the macro. It is not uncommon to receive a number of errors all pointing to the same source line, with no indication whatsoever where the error really is.

Now consider how the DefMsg macro operates. DefMsg begins by defining a single incoming parameter named Message. Two local labels—MsgStart and MsgEnd—are defined, and these are needed so MASM can calculate the length of the messages. Although labels within a macro do not have to be declared as local, you would get an error if the macro were used more than once. Like BASIC, the assembler requires that each label have a unique name. By using local labels MASM generates a new, unique internal name for each macro invocation, instead of the actual label name given.

The next statement increments a MASM variable named NumMsg. To avoid an error caused by calling Message\$ with an invalid message number, it compares the number you pass to the number of messages that are defined. This test occurs in the fourth line of the procedure, at the `Cmp CX, NumMsg` statement. NumMsg is a constant, except it may be redefined within the routine. When a constant is assigned using the word `Equate`, its value may not be changed by either your source code or by a macro. But when a variable is defined using an equals sign (`=`), MASM allows it to be altered as it assembles your program. Understand that the resulting number is added to your program as a constant. However, its value can be changed during the course of assembly. Therefore, each time `DefMsg` is invoked, it increments NumMsg. MASM places the final value into the `Cmp` instruction, as if you had defined it using a fixed known value.

The IFB (If Blank) test checks to see if `DefMsg` was given a parameter when it was invoked. In most cases you will probably want to define a series of consecutive messages. As it is used here, seven different day names are returned in sequence. But there may be times when you want to leave a particular message number blank. For example, you could create a series of messages that correspond to BASIC's error numbers. BASIC file error numbers range from 50 through 76, but there are no messages numbers 60, 65, or 66. You could therefore leave those blank, and invoke a modified copy of `Message$` like this:

```
CALL DOSMessage$(51 - ERR)
```

When `DefMsg` is used with no argument, it merely creates a zero word at that point in the code segment. Otherwise, the length of the message is stored, followed by the message text. The statement `DW MsgEnd - MsgStart` is replaced with the difference between the addresses, which MASM calculates for you. This is similar to the earlier example that showed how a dollar sign (\$) can simplify defining strings that may change.

The last macro I will describe here is `Rept`, which means "Repeat the following statements a given number of times". In the simplest sense, `Rept` could be used to generate a series of the same instructions:

```
Rept 100
  Xor  AX,AX
  Push AX
  Call SomeProc
Endm
```

A `Rept` macro is not invoked by name; rather, it is added inline to a program (or included within a macro that is called by name). In most cases you would use a coding loop to repeat a block of code, since a `Rept` macro actually generates the same code repeatedly in the program. But there are situations where timing is very critical, and a loop is always somewhat slower than a sequence of inline instructions.

Another good use for `Rept` is in conjunction with redefinable equates, such as this example which defines the letters of the alphabet:

```

Alphabet:
Char = 0
Rept 26          ;;do this 26 times
  DB "A" + Char  ;;define ASC("A") + Char
  Char = Char + 1 ;;increment Char
Endm

```

Although the MASM manual states that you must use double semicolons for remarks within a macro as shown here, I have used a single semicolon without problems.

There are other macro commands and features I will not describe here, because I have not found them to be particularly useful. However, macros can be recursive, multiple macros may be nested, and even redefined on the fly. I urge you to refer to the documentation that Microsoft provides for more information on those advanced features.

Segment Naming

Aside from the short `PrtSc` example shown earlier in this chapter, we have relied upon MASM's simplified segmentation directives to spare us from the nuisance of defining and naming segments. Indeed, when writing routines that will be added to BASIC it is rarely necessary to do this manually, so why bother?

One place where naming segments explicitly is useful is when you have many internal procedures that are never called from BASIC directly. If, for clarity and organization reasons, you decide to store those routines in different files, you still may want to access the routines using near calls. Since a near call is two bytes shorter than a far call and also operates slightly faster, this can make a difference when there are many `Call` commands within the routines.

As `LINK` pulls all of the various pieces of your program together from separate object and library files, it reads the segment names and combines those with the same name. Thus, a routine in one source file can call a routine in a different file, and `LINK` will place both routines into the same segment if they use the same segment name. This is of course needed to ensure that the called routine is reachable by the caller (within 64K).

All of the standard segment names that Microsoft recommends are listed in the MASM manual, along with instructions for creating your own names.

Accessing BASIC Internals

In preceding sections you learned that it is possible—even desirable—to call BASIC's internally routines directly. Besides those that have already been described, there are several other useful routines that can be accessed from assembly language. One of these is `B_ONEXIT`, which lets you tap into BASIC's termination procedure.

When a BASIC program ends by running out of statements, or by using END, STOP, or SYSTEM, BASIC makes a call to a central routine that in turn tells DOS to end the program. If a fatal error occurs and there is no ON ERROR handler, BASIC also calls a routine that prints an error message. B_ONEXIT lets you tell BASIC the segment and address of a routine you want called as part of the termination process. B_ONEXIT is supported only in QuickBASIC version 4.5 and BASIC PDS.

One reason you might want to use B_ONEXIT is to ensure that interrupts taken over by your assembler routine are restored properly. Taking over interrupts will be described later in the section Handling Interrupts. Here's a program fragment showing how B_ONEXIT is set up and called:

```
Extrn B_ONEXIT:Proc      ;declare B_ONEXIT as external
Push CS                  ;pass your code segment
Mov AX,Offset TermProc  ;and the address of the routine
Push AX                  ; that is to be called
Call B_ONEXIT            ;register it with B_ONEXIT
.
.

TermProc Proc            ;this is the routine to be called
.                          ;do whatever you need to here
.
Ret                      ;don't forget to return!
TermProc Endp
```

BASIC's Internal Data

There are two internal variables BASIC maintains that you will find useful. One is the current DEF SEG setting, and it is stored in the integer variable named B\$SEG. The other is the current color value that is used by PRINT and CLS. The foreground and background colors are stored combined in a single word named B\$FBColors. The reason these are useful is because you may want to change and then restore them from inside a BASIC subprogram. Much of the benefit of reusable programming is lost if you cannot put things back to the way they were originally.

For example, if you have written a BASIC routine that prints an error message in bright red at the bottom of the screen, you will need to use a subsequent COLOR command to put the color back to what it had been. But what color do you use? The same holds true for a routine that changes the current DEF SEG setting, perhaps before loading or saving a file using BLOAD or BSAVE. If you cannot return that to its original value, extra work is needed in the main program each time the routine is used.

Access to B\$SEG requires a single assembler instruction, as shown in the complete GetSeg function shown following. Declare and use GetSeg like this:

```
DECLARE FUNCTION GetSeg%()
SavedSeg = GetSeg%
.
.
```

```

DEF SEG = SavedSeg

;GETSEG.ASM
.MODEL Medium, Basic
.DATA
    Extrn B$Seg:Word

.CODE
GetSeg Proc

    Mov  AX,B$Seg    ;load the value from B$Seg
    Ret              ;return with the function output in AX

GetSeg Endp
End

```

Because BASIC combines its colors into a single word, a few extra steps are needed to separate them. Call GetColor like this:

```
CALL GetColor(FG%, BG%)
```

FG% and BG% are returned to you holding the current foreground and background color values. Here's how GetColor works:

```

;GETCOLOR.ASM
.MODEL Medium, Basic
.DATA
    Extrn B$FBColors:Word

.CODE
GetColor Proc, FG:Word, BG:Word

    Mov  DX,B$FBColors    ;load the combined colors
    Mov  AL,DL             ;copy the foreground portion
    Cbw                      ;convert it to a full word
    Mov  BX,FG             ;get the address for FG%
    Mov  [BX],AX           ;assign FG%
    Mov  AL,DH             ;load the background portion
    Mov  BX,BG             ;get the address for BG%
    Mov  [BX],AX           ;assign BG%
    Ret                    ;return to BASIC

GetColor Endp
End

```

One unfortunate problem is that GetColor cannot be used in the editing environment. When BASIC compiles a PEEK or POKE statement, it generates inline code that loads ES with the segment from B\$SEG, and then reads or writes the data at the specified address. Therefore, the current segment must be available to BASIC routines that use PEEK or POKE in a Quick Library. But the color values are accessed only by routines in BASIC's runtime library, so the information is not made available to procedures in a Quick Library. Because of this issue, the GetColors routine is provided on the accompanying disk only in the BASIC.LIB and BASIC7.LIB linking libraries.

There are several other internal data items you may want to know about, and one that I have found useful is called `__osversion`. This byte holds the major DOS version number; for example, if DOS 3.x is running then `__osversion` will hold the value 3. Even though it is trivial to query DOS for the number, why bother since you can get it this way with a single `Mov`.

BASIC's Internal Routines

Besides the procedures and internal data I have described previously, there are many others you will no doubt find useful. You can, for example, call `SETMEM` prior to claiming memory from DOS. And although the `B$ASSN` routine can assign any type of data from any other type including strings, a simplified version is also present to assign to and from conventional strings only.

As you have seen, the beauty of using BASIC's own routines is that identical code can be used for both near and far strings. In either case, the string descriptors are known to reside in `DGROUP`, and the internal routines are designed to operate on those descriptors. You don't even have to know which of the string libraries (near or far) is being used.

There are also several math routines that can be accessed directly, including those that multiply, divide, and compare long integers. Even if you know how to do that, it's always easier to call BASIC's routines. This results in less code as well. And if you need to read the current cursor position, you can access `CSRLIN` and `POS(0)` directly. In some cases, you can't read that information from the BIOS, so calling BASIC is the only reliable way to get it.

The following section documents the BASIC internal routines that I have found useful when called from assembly language. I have purposely omitted routines that handle BASIC commands such as `PRINT`, `INKEY`, `GET`, and `PUT`. Even though several of these were described throughout the course of this book, they have little relevance within a called assembler routine.

BASIC's internal services that follow are listed in alphabetical order, based on their call names. Be sure to declare them as external procedures in your routine's source code.

B\$CPI4: Compare Two Long Integers

`B$CPI4` expects two long integer arguments to be placed onto the stack by value, and it returns the result of its comparison in the `Flags` register. For example, to see if `Var1` is greater than `Var2` you'd use code like this:

```
Push Word Ptr [Var1+2] ;first push Var1's high word
Push Word Ptr [Var1]   ;and then its low word
Push Word Ptr [Var2+2] ;next do the same for Var2
Push Word Ptr [Var2]
Call B$CPI4           ;compare them
Jg Label             ;Var1 is indeed greater
```

Remember that long integers are compared by BASIC on a signed basis, so you should use Jg or Jl rather than Ja or Jb. The letters CPI4 stand for Compare Integer 4 bytes.

B\$CSRL: CSRLIN Function

B\$CSRL is called with no arguments, and it returns BASIC's current row in AX as follows:

```
Call B$CSRL
.           ;do what you want with AX
```

B\$DVI4: Divide Two Long Integers

Like B\$CPI4, B\$DVI4 (Divide Integer 4 bytes) expects the incoming integer arguments to be passed by value on the stack. The result is then returned in DX:AX as a long integer:

```
Push Word Ptr [Var2+2] ;always push the high word first
Push Word Ptr [Var2]   ;then the low word
Push Word Ptr [Var1+2] ;ditto for Var2
Push Word Ptr [Var1]
Call B$DVI4            ;divide them
.                     ;now DX:AX holds Var1 \ Var2
```

Notice that with B\$DVI4, the divisor is pushed first onto the stack, followed by the dividend.

B\$FPOS: POS(0) Function

Even though the argument passed to BASIC's POS(0) is ignored, it is still expected mainly for historical reasons. Therefore, you must push something—anything—onto the stack before calling B\$FPOS:

```
Push AX
Call B$FPOS
.           ;now AX holds the column
```

As with all of BASIC's functions that return an integer, B\$FPOS returns the current column in AX. The leading F in FPOS stands for Function.

B\$FRI2: FRE() Function

B\$FRI2 (Free Integer 2 bytes) requires an incoming integer argument by value on the stack, and for safety you should use this for the -1 and -2 variations only.

Using -1 reports the total amount of memory that is available to BASIC, so you might use this before calling SETMEM to release memory for your own uses. Although B\$FRI2 uses an integer for an argument, it returns a long integer in DX:AX. You can also use an argument of -2 to see how much stack space is available:


```

Mov  AX,-2
Push AX
Call B$FRI2
.      ;now DX:AX holds the available stack space

```

B\$RDIM: REDIM Statement

In most cases you will probably not find the ability to call REDIM directly very valuable. One notable exception is explained later in the section entitled Reading the Array Descriptor, where I show how to size and then load a string array with all of the files that match a given search specification.

B\$RDIM is fairly complicated to set up and call, because it accepts a varying number of parameters. This is needed because BASIC accepts a variable number of dimensions, and the same routine is used for all cases. The following example shows how to prepare and call this routine when resizing a one-dimensional array.

```

Mov  AX,LBound      ;first pass the lower bound value
Push AX
Mov  AX,UBound      ;then pass the upper bound
Push AX
Mov  AX,ElementLength ;next the length of each element
Push AX
Mov  AX,Features     ;see the accompanying text for
Push AX             ; information on these two items
Mov  AX,Offset ArrayDescriptor
Push AX
Call B$RDIM         ;call REDIM to do it

```

Chapter 2 described the array descriptor in detail, including the Features word. However, you must not use REDIM to create a new array where none existed before. Instead, you will read the current features from the existing array descriptor, and pass the same values on again to B\$RDIM. This will be shown in context momentarily.

B\$STD L: String Delete

You can call B\$STD L to delete a string or string array element, and it requires less code than assigning the string from another, null string. The single argument is the address of a string descriptor:

```

Mov  AX,Offset Descriptor
Push AX
CALL B$STD L

```

B\$SETM: SETMEM Function

B\$SETM expects a long integer argument by value on the stack; if the value is negative then that much memory is released back to DOS, and thus taken from your BASIC program. However, you should call B\$SETM again later with a positive value when you are finished, so the BASIC program can reclaim

that memory. Since SETMEM is a function, B\$SETM also returns the amount of memory currently available in the DX:AX register pair.

B\$\$ASS: String Assign

Where B\$ASSN is capable of assigning any mix of conventional and fixed-length strings, B\$\$ASS works with conventional strings only. However, it requires only two parameters instead of six:

```
Mov  AX,Offset Source$
Push AX
Mov  AX,Offset Destination$
Push AX
CALL B$$ASS
```

Note that if the destination string is not null, its current contents are released after assigning it from Source\$. This is the normal way that strings are assigned, and B\$ASSN also works like this.

Finding Other Routines

The routines just described are those that I personally have found to be useful. Discovering other routine names and how they are called is in fact quite simple. If you wanted to access, say, COMMAND\$, you would write a one-line BASIC program, and then examine the code that is generated using Microsoft CodeView. CodeView lets you see which and how many parameters are being passed as well as the routine name being called, making exploration both easy and fun.

BASIC string functions such as COMMAND\$ and ENVIRON\$ return the DGROUP address of the result string descriptor in AX, just like an assembly language function you would write. If you do call a built-in BASIC function, be sure to also pass its output descriptor to B\$STD (String Delete) when you are done with it. Otherwise, the string space it uses, and the temporary output descriptor, will never be released.

Reading the Array Descriptor

Chapter 2 described the BASIC array descriptor in detail, and discussed each of the components it contains. Understanding how an array descriptor works opens many opportunities to assembly language programmers, because it lets you write routines that accept an array passed with empty parentheses. This was shown in the Sort routine introduced in Chapter 8, although the techniques used there were not detailed.

As an example of the possibilities direct access to an array descriptor offers, I will show a subroutine that accepts a file specification, and returns a string array filled with the names of all matching files. GetNames calls upon three internal BASIC routines: B\$FLEN, B\$RDIM, and B\$ASSN. B\$FLEN returns the length of a string, and is used here to know how long the file specification is. B\$RDIM

redimensions the passed string array to the correct number of elements, based on the number of matching file names that are found. B\$ASSN then assigns each element to those names.

This next short BASIC program shows how GetNames is set up and used.

```

DECLARE FUNCTION GetNames%(Array$())
REDIM Array$(1 TO 1)           'use REDIM, not DIM
Array$(1) = "*.*"             'any valid spec is okay
NumFiles% = GetNames%(Array$()) 'load all names at once

IF NumFiles% = 0 THEN          'were any files found?
  PRINT "No matching files."   'no, say so and end
  END
END IF

FOR X% = 1 TO NumFiles%       'yes, print each name
  PRINT Array$(X%)
NEXT
PRINT NumFiles; "matching files were found"

```

As you can see, you must establish the array initially using REDIM. To avoid the need for an extra parameter, the file specification is passed in the first element of the array. Furthermore, GetNames returns the number of files that matched as an integer result. If no files were encountered, GetNames leaves the array as it was.

When GetNames is called, the array may already contain other data, and it can have any legal upper and lower bounds. As long as the lowest element number contains a valid search specification, the spec can be found and the array will be redimensioned starting at element number one. The GETNAMES.BAS demonstration program on the accompanying disk adds to this short example by sorting the names after they are read.

A complete description of how GetNames works follows this source listing.

```

;GETNAMES.ASM, loads a group of file names into an array

.Model Medium, Basic
Extrn B$RDIM:Proc           ;this redimensions an array
Extrn B$ASSN:Proc           ;this assigns a string
Extrn B$FLEN:Proc           ;this returns a string's length

DTAType Struc               ;define the DOS DTA structure
  Intern  DB 21 Dup (?)     ;this is used by DOS internally
  FATtr   DB ?              ;this holds the file attribute
  FTime   DW ?              ;this holds the file time
  FDate   DW ?              ;this holds the file date
  FSize   DD ?              ;this holds the file size
  FName   DB 13 Dup (?)    ;this holds each file name
DTAType Ends

.Data
DTA DB Size DTAType Dup (?) ;DOS places file info here
NumFiles DW 0                ;how many names were read
SpecLength DW 0              ;remembers file spec length

.Code

```

GetNames Proc Uses SI DI, Array:Word

```
Local Buffer[80]:Byte ;copy the spec here, add a zero

;-- Create a local Disk Transfer Area for our own use.
Lea DX,DTA ;show DOS where the new DTA goes
Mov AH,1Ah ;set DTA service
Int 21h ;call DOS to do it

;-- Read the array descriptor, get the search spec from the first element,
; then copy it to the stack appending a CHR$(0) byte (ASCIIIZ string).
Mov SI,Array ;get address of array descriptor
Mov BX,[SI+0Ah] ;now BX holds adjusted offset
Mov AX,4 ;each element is four bytes long
Mul Word Ptr [SI+10h] ;multiply by first element number
Add BX,AX ;BX holds first element's address
Push DS ;push source segment and address
Push BX ; for call to B$ASSN later on
Xor AX,AX ;tell B$ASSN source is descriptor
Push AX ;using a value of zero

Push BX ;pass descriptor addr to B$FLEN
Call B$FLEN ;this returns the length in AX
Mov SpecLength,AX ;save length locally for a moment
Lea AX,Buffer ;get the destination address
Push SS ;pass the segment to assign into
Push AX ;and then the address
Push SpecLength ;we're assigning a fixed length
Call B$ASSN ;copy the file spec to the stack

Lea BX,Buffer ;retrieve start address of spec
Mov DX,BX ;copy to DX where DOS expects it
Add BX,SpecLength ;point just past end of string
Mov Byte Ptr [BX],0 ;and append trailing zero byte

;-- Count the number of names that match the search specification.
Mov AH,4Eh ;specify Find First matching name
Mov CX,00100111b ;this matches any type of file
Xor BX,BX ;BX counts the number of names

CountNames:
Int 21h ;see if there's a matching name
Jc DoneCount ;carry set means no more names
Inc BX ;otherwise, we found another one
Mov AH,4Fh ;find the next matching name
Jmp CountNames ;continue until there are no more
DoneCount:
Mov NumFiles,BX ;remember how many files we found
Or BX,BX ;did we fail on the first name?
Jz Exit ;yes, return a count of zero

;-- Now that we know how many file names there are, REDIM the string array.
Mov AX,1 ;specify an LBOUND of 1
Push AX ;pass that on to B$RDIM
Push BX ;and pass on the new UBOUND value
Mov AL,4 ;each descriptor takes four bytes
Push AX ;pass that on too

Mov BX,Array ;get array descriptor again
Mov AX,[BX+08] ;load the existing Features word
Push AX ;use that again for this call
```

```

Push BX                ;show where array descriptor is
Call B$RDIM           ;finally, redimension the array

;-- This is the main processing loop that reads and assigns each name
;   that is found.
Mov  AH,4Eh           ;specify Find First matching name
Lea  DX,Buffer       ;load address of file spec again
Mov  BX,Array        ;get array descriptor address too
Mov  BX,[BX+0Ah]     ;reload the adjusted offset value
Add  BX,4            ;BX is first descriptor address

Do:
Mov  CX,00100111b    ;specify any type of file again
Int  21h            ;see if there's a matching name
Jc   Exit           ;carry set means no more names
Push BX            ;otherwise, save the address

;-- Search for the zero that marks the end of this name.
Mov  DI,Offset DTA.FName
Push DS            ;in anticipation of call below
Push DI           ;DI too while the address handy

Push DS           ;ensure that ES=DS
Pop  ES
Mov  CL,13        ;search up to 13 characters
Repne Scasb      ;do the search
Mov  AL,CL        ;save the remainder in AL

Mov  CL,13        ;calc number of chars to copy
Sub  CL,AL        ;the answer is now in CX
Dec  CX           ;don't include the zero byte
Push CX          ;pass that on to B$ASSN

Push DS          ;show where destination string is
Push BX
Xor  AX,AX        ;zero means B$ASSN is assigning
Push AX          ; to a conventional string
Call B$ASSN      ;assign this element to the name

Pop  BX          ;retrieve the descriptor address
Add  BX,4        ;point to the next element
Mov  AH,4Fh      ;specify Find Next matching name
Jmp  Do          ;and keep on keepin' on

Exit:
Mov  AX,NumFiles  ;assign the function output
Ret                    ;return to BASIC

GetNames Endp
End

```

GetNames begins by declaring the three BASIC routines it will call as being external. Next the DTA structure is defined, to simplify access to the file name address when it assigns each element in the string array. The only data items are the DTA itself, two working variables, and the local stack buffer. Since the incoming file specification needs to be converted to an ASCIIZ string for DOS, GetNames copies that specification into Buffer and then appends a CHR\$(0) zero byte to the end.

Once the DTA has been established, the next step is to read the file specification passed in the first element, and copy it into local storage. B\$FLEN is used to obtain the length of the string, so GetNames will know how far into the buffer the zero byte will be placed. The last preparatory steps call B\$ASSN telling it to copy from a conventional string (the array element) to a fixed-length string (Buffer), and then store the zero byte.

The actual body of the program is broken into two portions. The first simply calls DOS repeatedly to count the file names, to know how many elements are needed. The count is then saved in NumFiles; if none were found GetNames exits without doing anything else. Otherwise, the incoming string array is redimensioned from 1 to the number of files.

The second portion again reads each file name through DOS, but this time the names are actually assigned to the array elements using B\$ASSN. This time, however, B\$ASSN assigns a conventional string from the fixed-length string portion of the DTA. Since the source is now of a fixed-length, GetNames needs to know how long each name is. The longest possible name is 13 bytes long (eight for the name, a period, three for an extension, and one more for the terminating zero byte). Therefore, ES:DI is set to point to the start of the DTA, AX is set to zero to search for the zero byte, and CX is loaded with the number of characters to scan.

Once the zero is found—and it always will be—the count that remains in CX is subtracted from 13 to obtain the actual length of the current name. Because that calculation includes the unwanted CHR\$(0), CX is decremented by one.

There is one small related trick that bears explaining. Just before the call to B\$RDIM, AX is loaded with the number 1, to specify that as the first element number. This three-byte instruction sets AL to 1, and clears AH to 0. Three lines below that only AL is assigned, which is sufficient because we know that AH is already zero. Because the number being assigned is one byte long, assigning AL requires only two bytes.

Admittedly, the savings is small, but the affect on code readability is minimal once you know about such tricks. And a byte saved is always welcome in assembly language programming. The same trick is used when setting CL to 13, where CH is known to be zero after assigning the file attribute of 00100111b to all of CX.

Handling Interrupts

The last programming technique I want to describe is writing an interrupt handler you can attach to a BASIC program. There are several applications for this, such as tapping into the timer interrupt to display an on-screen clock. Instead of having to constantly print TIME\$ during your INKEY\$ input loops, such a routine would act as a sort of TSR, getting control at each timer tick and displaying the time automatically.

The example I will show here takes over the keyboard interrupt, and disables the Ctrl-Alt-Del key sequence. This lets you prevent rebooting with its corresponding loss of data, should someone press those keys inadvertently (or on purpose). NoReboot is called as follows:

```
CALL NoReboot (BYVAL InstallFlag%)
```

If InstallFlag is non-zero, you are telling NoReboot to install itself and take over the keyboard interrupt to prevent rebooting. An argument of zero instead unhooks the interrupt, and re-enables those keys. Although you could certainly modify NoReboot to use BASIC's B_ONEXIT service to uninstall itself automatically, I have left that feature out on purpose in the interest of clarity. This also lets you activate NoReboot selectively in your program, since there is no way to revoke a request to B_ONEXIT.

```
;NOREBOOT.ASM, traps Ctrl-Alt-Del within a BASIC program

.Model Medium, Basic
.Code

NoReboot Proc Uses DS, InstallFlag:Word

    Cmp  InstallFlag,0      ;are they asking to install?
    Je   Deinstall        ;no, so deinstall it

    Cmp  CS:Old9Seg,0      ;yes, are we already installed?
    Jne  Exit             ;yes, and don't do that again!

    Mov  AX,3509h          ;ask DOS for current Int 9 vector
    Int  21h              ;DOS returns it in ES:BX
    Mov  CS:Old9Adr,BX     ;save it locally
    Mov  CS:Old9Seg,ES

    Mov  AX,2509h          ;point Int 9 to our own handler
    Mov  DX,Offset NewInt9
    Push CS                ;copy CS into DS
    Pop  DS
    Int  21h

Exit:
    Ret                   ;return to BASIC

;-- Control comes here when a key is pressed or released.
NewInt9:
    Sti                ;enable further interrupts
    Push AX            ;save the registers we're using
    Push DS

    In   AL,60h        ;read the keyboard scan code
    Cmp  AL,83         ;is it the Delete key?
    Jnz  Continue     ;no, continue on to the BIOS

    Xor  AX,AX         ;see if Alt and Ctrl are pressed
    Mov  DS,AX         ;by looking at address 0:417h

    Mov  AL,DS:[417h]  ;get shift status at 0000:0417h
    Test AL,8          ;is Alt key depressed?
    Jz   Continue     ;no, continue on to the BIOS
    Test AL,4          ;is Ctrl key depressed?
```

```

Jz    Continue                ;no, continue on to the BIOS

In    AL,61h                  ;send an acknowledge to keyboard
Mov   AH,AL                   ;otherwise the Ctrl-Alt-Del
Or    AL,80h                  ; keystroke will still be
Out   61h,AL                   ; hanging around the next time
Mov   AL,AH                   ; a program asks for a key
Out   61h,AL
Mov   AL,20h                  ;indicate end of interrupt to the
Out   20h,AL                   ; 8259 interrupt controller chip
Pop   DS                       ;ignore, simply return to caller
Pop   AX
Iret                                ;use this special Ret when
                                ; returning from an interrupt

Continue:
Pop   DS                       ;restore the saved registers
Pop   AX
Jmp   DWord Ptr CS:Old9Adr      ;continue on to the BIOS by
                                ; jumping to the address
                                ; that was saved during
                                ; initialization

DeInstall:
Mov   AX,2509h                 ;restore original Int 9 handler
Mov   DX,CS:Old9Adr            ;from segment and address saved
Mov   DS,CS:Old9Seg            ; earlier
Int   21h                       ;DOS does this for us
Mov   CS:Old9Seg,0             ;clear this as an installed flag
Jmp   Short Exit                ;and then exit back to BASIC

NoReboot Endp

Old9Adr DW 0                    ;remembers original Int 9 address
Old9Seg DW 0                    ;these must be stored in the code
                                ; segment because DS is undefined
                                ; when NewInt9 receives control

End

```

The first thing NoReboot does is look to see if the caller is installing or uninstalling. If installation is requested, the saved Interrupt 9 segment is checked, to be sure that it holds the initial value of zero. It is important to prevent multiple installations, because installing saves the current interrupt handler's address. If NoReboot installed itself twice, it would save its own address on top of the original BIOS handler's saved address. And once that address is lost, it is impossible to restore it again later.

Assuming it is safe to be installed, the next step is to ask DOS for the current interrupt handler's address using service 35h. This service expects the service number in AH, and the interrupt number in AL. To save a byte, both values are loaded at once. Service 35h returns the segment and address in ES:BX, and these are saved in the code segment. Because the original address will be called from within the interrupt handler, CS is the only register whose contents are known. Accessing data in DGROUPE is more difficult, because an interrupt can occur at any time, and DS will likely not be holding the correct segment.

Execution could be at any point in the program when Ctrl-Alt-Del is pressed, including within a routine that has changed DS. So when NoReboot receives control


```
it can't be certain that DS holds the segment for .Data
variables it has defined.
```

Once the original interrupt handler address has been saved, NoReboot calls DOS again, but this time to assign the segment and address of its replacement handler in the interrupt vector table. It is easy to access the interrupt vector table directly using Mov instructions, but it is even easier to have DOS do that.

Finally, NoReboot returns to the calling BASIC program, and all subsequent key presses are now routed to the NewInt9 procedure.

NewInt9 must perform a few tricks, partly because it is handling a hardware interrupt. All interrupt handlers begin with the instruction Sti, which tells the 8088 to allow further interrupts to occur and be processed. Next, the two registers being used are saved on the stack, so they can be restored again later. Because a keyboard interrupt can occur at any time interrupting the process that is currently running, it is imperative that you not alter any aspect of the 8088's current state. This includes the settings of the Flags register as well. However, the Flags register is saved automatically by the 8088 as part of its handling of interrupts, so the flags don't have to be saved or restored manually using Pushf and Popf.

The next sequence of instructions reads the key that was pressed from the keyboard's I/O port (60h), and compares that to the scan code for the Del key. If any other key was pressed, NoReboot jumps to the original keyboard handler in the ROM BIOS. Otherwise, it examines low memory to see if both the Ctrl and Alt keys are also currently pressed. Unless all three conditions are met, control passes on to the BIOS. But if Ctrl-Alt-Del is pressed, NoReboot handles the keystroke entirely on its own and ignores it. In that case DS and AX are restored, and NoReboot exits back to the underlying program.

Notice the special form of return command, Iret (Interrupt Return). Like a conventional far return, Iret pops the address and segment to return to from the stack, but it also pops the Flags register that was stored there by the 8088 automatically.

The final section of code restores the original interrupt vector, and clears the Old9Seg variable to zero. This lets NoReboot know that it is not installed, in case you call it again later.

This same technique can be applied to handle other interrupt services, and I encourage you to experiment on your own. You could, for example, write a routine that takes over the communications interrupt, and displays a flashing box in a corner of the screen whenever characters are received. Likewise, you could modify this routine to create an on-screen display of the Caps Lock and Num Lock state. Each time one of those keys is pressed you would either print or clear a status message.

Debugging With CodeView

As useful as CodeView can be for a purely BASIC program, it is even more necessary when writing in assembly language. CodeView lets you step through the code that BASIC generates to set up and call your subroutine, and then step through the routine a line at a time. Being able to watch your program as

it executes helps you to quickly zero in on any problems. Further, CodeView shows you the current CPU register contents, as well as the value of memory locations about to be read from or written to.

To debug an assembly language subroutine with CodeView, you must first assemble it using the /Zi option switch:

```
masm routine /zi;
```

Then you link the routine to your BASIC program using the /Co option. Of course, the BASIC program must also have been compiled using /Zi:

```
bc program /o /zi;  
link program routine /co;
```

Finally, you start CodeView specifying the name of the BASIC program:

```
cv program
```

Once the BASIC source code is showing on the screen you can step and trace through it as described in Chapter 4. As with BASIC subprograms and functions, to step into an assembler routine you press F8 at the CALL statement. If the routine is designed as a function you instead press F8 at the line in which the function is referenced.

Once CodeView has traced into the routine, you can press F3 to view the source code only, the assembly code only, or both intermixed. I usually prefer to view only my original source, but that hides the data memory addresses that MASM and LINK assigned. Usually you will not need to know those addresses, but there are times when this can be helpful. For example, when a program is not working correctly, the bug could be caused by a different portion of the program overwriting the named variables.

Besides the F3 key, you can also use F4 and F7, and these have the same meaning as the same keys when used in the BASIC editor. Indeed, debugging an assembly language subroutine is quite similar to debugging a BASIC program as far as which keys are used.

MASM 6.0 Enhancements

All of the discussions in this chapter have focused on using MASM version 5.1. However, Microsoft's more recent version 6.0 introduces a number of significant changes and new features. Perhaps the most useful new feature in this release is the greatly improved documentation. The manuals that came with past versions of MASM were very dry, containing reams of facts but no practical advice or guidance. The new documentation include both facts and programming tips, and this addition is welcome indeed.

If you already have existing assembly language source code, you may have to change it to accommodate the new MASM 6.0 conventions. In particular, MASM's handling of data structures has

changed substantially, and in many cases code that used to work correctly no longer does. However, you can optionally use the /Zm command line switch, to tell MASM 6.0 to behave like the earlier 5.1 version.

A new MASM.EXE program launcher is also included to offer a similar capability. Where older versions of MASM were named MASM.EXE, the new program is called ML.EXE. The MASM.EXE that now comes with MASM 6.0 simply passes the /Zm option on to ML, along with some other option switches that are needed to tell ML to mimic the older assembler's behavior.

Improved Assembly Optimizations

Before MASM 6.0, a conditional jump was limited to a distance no greater than 128 bytes earlier or 127 bytes farther ahead in the code. When there was no way to restructure your code to accommodate this inherent 8088 limitation, you had to use a conditional jump around another unconditional jump like this:

```
;if AX < 12 go to FarLabel
Cmp  AX,12                ;compare AX to 12
Jnl  NearLabel           ;jump if not less over far jump
Jmp  FarLabel            ;perform the far jump
NearLabel:
.                          ;program continues
.
.                          ;this label is more than
FarLabel:                 ; 127 bytes past Jnl
```

MASM 6.0 avoids this limitation and lets you use JI to the far label directly, although it really just replaces your use of JI with code equivalent to that shown above.

Another, similar optimization affects unconditional jumps. As I mentioned earlier, each time MASM 5.1 encounters a label in your source code, it remembers its address in the resultant object code. Then if you jump backwards to that label later, MASM knows if it can use the shorter two-byte form of the Jmp instruction. But a forward jump to a near label requires you to explicitly state Jmp Short to obtain this code savings, since MASM 5.1 does not yet know the target label's address. Without Short, MASM 5.1 uses a long jump on a trial basis. If the jump turns out to be within the near range MASM goes back and patches the code to a short jump followed by a byte-wasting Nop (No Operation) instruction.

MASM 6.0 avoids this problem by processing your source file in multiple passes. That is, MASM reads your code and assembles what it can, using far jumps when the target address has not yet been encountered. Then it processes that intermediate code again modifying its earlier output as appropriate. If a three-byte jump can be replaced with the two-byte version, MASM 6.0 rewrites the code sliding subsequent instructions back a byte. MASM 6.0 is called an *n-pass assembler*, because as many passes as needed are performed until the code is as small as possible.

New Simplified Directives

Besides the improved optimizing, MASM 6.0 offers several features borrowed from high-level languages. These include `.IF`, `.ELSE`, and `.ELSEIF`; `.WHILE` and `.ENDW`; and `.REPEAT` and `.UNTIL`. Unfortunately, these new constructs are modelled after the C language, and provide little if any clarification to BASIC programmers. For example, you can now write code such as this:

```
.IF (AL < "0") || (AL > "9")
```

which is equivalent to this BASIC statement:

```
IF AL < ASC("0") OR AL > ASC("9")
```

Even worse, the MASM manual does not document each directive showing precisely what it does to your code.

Like C, BASIC's AND is replaced with a double ampersand (&&), testing for equality uses a double equals sign (==), and NOT is replaced with an exclamation point (!). Therefore, you could write assembly language source statements like these next two examples:

```
.IF (AX != 14) && (BX < 10) ;IF AX <> 14 AND BX < 10 THEN
Mov  AX,SomeVar           ;divide SomeVar by CX
Cwd
Div  CX
Mov  SomeVar,AX
.ENDIF

.REPEAT
Mov  AH,1                 ;ask for a keyboard character
Int  21h                  ;through DOS
.UNTIL (AL == 13)         ;loop until they press Enter
```

`PROTO` and `INVOKE` are two other new simplified directives, and it's hard for me to recommend using them for similar reasons. `PROTO` mimics C's function prototype capability, and lets you define a called procedure and its arguments. `INVOKE` then calls that routine passing the arguments you give it. To define a procedure called, say, `MyProc`, you would use `PROTO` like this:

```
MyProc PROTO Var1:Word, Var2:Word, Var3:DWord
```

Then to call `MyProc` you use `INVOKE` as follows:

```
INVOKE MyProc, BX, 100, LongVar
```

Thus, `PROTO` and `INVOKE` are very similar to `DECLARE SUB` and `CALL` in BASIC. The problem is that you have no way to know what code MASM generates for this command unless you create a sample program, assemble it, and examine the result using CodeView. In particular, how does the value 100 used here get onto the stack? As it turns out, assembling the preceding `INVOKE` command results in the following code:

```
Push BX
Mov AX,100
Push AX
Push Word Ptr [LongVar+2]
Push Word Ptr [LongVar]
```

As you can see, even if AX is holding an important value, its contents are destroyed when MASM assigns the value 100 prior to placing it on the stack. While I applaud Microsoft's attempts to make assembly language easier to use, such behavior can and will introduce subtle bugs. These bugs can be even harder to track down than usual, because you did not make the coding error, the assembler did. Since the whole point of programming in assembly language is to control fully what the CPU is doing, such hidden behavior can have disastrous effects.

One new feature that I do find useful, however, is the ability to continue a line with a trailing comma. Often, a single source statement will extend into the comments column, spoiling the appearance of your listing. You can now avoid this by placing a comma in the middle of a logical line, and then continuing the remainder of the statement on the next line.

Another very useful feature is MASM 6's ability to accept wild cards on the command line. For example, you can assemble all of the files in the current directory using the command `masm *.asm;`

Tricks of the Trade

The final topic I want to present is a variety of assembly language programming short cuts and other techniques I have developed over the years. In preceding sections you saw how Xor or Sub can be used to clear a register, using less code than Mov. And if you know that the high-byte portion of a register or memory variable is already zero, you can save a byte by assigning only the lower byte. And to clear both AX and DX you can use Xor with AX, and then Cwd to extend the zero into DX using only one additional byte. As you might imagine, there are many other ways to be clever in assembly language.

Minimize Code to Access Parameters

When parameters are accessed within an assembly language subroutine, the usual way to get at them is through BP. Even when you use MASM's simplified directives, code to push BP, assign it from SP, and then reference the address on the stack is added to your program. In that case, the steps are simply hidden from you. Because BASIC (and indeed, every high-level language) requires you to preserve BP, one byte each is needed for the Push and Pop instructions.

You can eliminate that overhead by taking advantage of the fact that the stack is always kept in DGROUP, and that SS and DS are equal. The trick is to use BX as a stack reference, because it doesn't need to be preserved. Unfortunately, this precludes using the simplified methods for parameter access. But when speed or code size are paramount or you have many routines, stack addressing via BX

affords a real savings. Here's how you will design the routine, using an example that accesses an incoming string:

```
GetString Proc      ;one parameter, not shown

  Mov  BX,SP        ;address the stack manually using BX
  Mov  BX,[BX+04]   ;get the address for the string
  Mov  CX,[BX]      ;get the length of the string
  Jczz Exit         ;quit if the string is null
  Mov  BX,[BX+02]   ;get address of first character

Exit:
  Retf 2           ;specify far return with 2 bytes

GetString Endp
End
```

Because BP has not been pushed onto the stack, the incoming string descriptor address is at [BX+4] rather than [BX+6]. Other than that, the remainder of the routine proceeds as usual.

Byte Savers

Another useful trick lets you save a byte when adding two to a variable. As you know, Inc and Dec when used with a register are always better than Add and Sub, because they are one-byte instructions. Therefore, two Inc or Dec commands in a row are still better than Add AX, 2 which requires three bytes. However, you must never do this with SP. The stack pointer must always hold an even number, and it is possible that an interrupt could come along after the first Inc or Dec, but before the second has executed. Which brings up a related byte saver.

If you need only a single word of local stack storage, don't use Sub SP, 2 to allocate the space and Add SP, 2 later to clear it. Instead, simply use Push AX, or Push with any other register. Likewise, just before returning to BASIC, pop any register that doesn't return information, such as CX or BX.

Rep Always Clears CX

Another trick you can take advantage of is that CX is often zero after a repeating string command that uses Rep. Zero is a common value in assembly language programming, and you can usually save a byte by using a register instead of a constant zero. In particular, if you are copying a file name to a buffer and adding a CHR\$(0) to the end, you can use code like this:

```
...                ;set up DS:SI and ES:DI here
Mov  CX,NumBytes
Rep  Movsb
Mov  [DI],CL       ;tack a zero byte onto the end
...
```

This trick is made even more valuable by the fact that DI is left pointing at the byte just past the data that was just copied. Of course, CX is not necessarily zero after Repe or Repne, because those forms of Rep can terminate before CX is exhausted.

Use AX Where Possible

Another little-known fact is that memory operations that use AX are one byte smaller than equivalent operations on any other register. That is, `Mov BX, KeyCode` results in four bytes of code, whereas `Mov AX, KeyCode` creates only three. I often use the DOS DEBUG program for quick tests, just to see which sequence of instructions results in less code. Since DEBUG does not let you specify a variable name, use `[100]` or any other address instead:

```
-a 100
-####:0100 Mov AX, [100]
-####:0103 Mov BX, [100]
-####:0107 <press Enter to stop assembling>
-u 100,106
####:0100 A10001      MOV    AX, [0100]
####:0103 8B1E0001    MOV    BX, [0100]
-q
```

This sample session tells DEBUG to begin assembling at address 100 (the default for .COM files), and then assemble the two instructions shown. When you are done press Enter at the dash prompt, and then unassemble the results and quit. As you can see, using AX creates one less byte of code.

Multiplying and Dividing By a Power of 2

Because of the way binary numbers are organized, shifting the bits left or right can provide a very fast way to multiply or divide by a power of two. And because the bit shifting commands can be used with all but the segment registers, this can also save you from having to copy the data to AX or DX:AX first. To divide a register by two simply shift the bits right one position:

```
Shr CX, 1
```

And to multiply by two shift them left:

```
Shl SI, 1
```

If you need to multiply or divide by four, eight, sixteen, and so forth, the shift count must first be placed into the CL register:

```
Mov CL, 5      ;prepare to divide BP by 32
Shr BP, CL
```

On 80186 and later processors you can specify a shift count directly. Unfortunately, this doesn't work with an 8088, so CL must be used. Still, multiplying and dividing are extremely slow instructions on an 8088, so the added setup will be more than offset if speed is the primary factor.

Low Memory is at Segment Zero

Another useful byte saver is to treat the BIOS data area in low memory as being at segment zero, instead of the more commonly used segment 40h. By convention, the BIOS data area is said to reside at segment 40h, even though a number of segment/address pairs can be used to access that data. I mentioned this briefly in Chapter 11, in the discussions about using BASIC's CALL Interrupt. Since Xor or Sub can be used to clear a register to zero with one byte less code than assigning it a value of 40, I use this technique frequently:

This example generates 9 bytes:

```
Xor  AX,AX
Mov  DS,AX
Test Byte Ptr [417h],8 ;see if the Alt key is depressed
```

And this example creates 10 bytes:

```
Mov  AX,40h
Mov  DS,AX
Test Byte Ptr [17h],8
```

Scanning An ASCIIZ String

Because ASCIIZ strings are used in programs that access DOS services, searching those strings to find the end is a common operation. For example, the GetNames function does this to determine the length of each file name before assigning it to elements in the incoming string array. In that routine CX is assigned to 13, which is the maximum length a file name can be. Since CX is decremented for each character that is examined, the length is calculated by subtracting CX from 13, which requires an extra register.

As long as you are certain that a zero byte is present, you can use a clever trick to determine directly the number of bytes that were searched. Instead of loading CX with the maximum number of bytes to scan, assign it to -1. As each character is searched CX is decremented, which results in a negative version of the number of bytes. Then the NOT instruction can be used to revert that to a positive number:

```
Mov  ES,Segment      ;point ES:DI to the start of the data
Mov  DI,Address
Cld                    ;ensure that scanning is forward
Mov  CX,-1           ;set CX to -1
Mov  AL,0            ;search for a zero byte

Repne Scasb          ;scan the string
Not  CX              ;convert to a positive number
Dec  CX              ;don't include the zero byte itself
Mov  AX,CX           ;now AX holds the length of the string
```


As you learned in Chapter 2, BASIC's NOT instruction flips all of the bits, converting ones to zeros and vice versa. The assembly language version works the same way, and can be used with registers or memory locations.

Cycle Savers

Besides savings bytes when possible, most assembly language programmers also like to save clock cycles. Every assembler instruction requires a certain amount of CPU timing cycles to execute, although there are other factors that also affect the actual throughput of a given piece of code. But instructions with the fewest number of clock cycles as published by Intel are always faster than those that require more cycles.

Move and Store Words Instead of Bytes

One very effective speed enhancement is to copy and store words when possible, instead of bytes. On 80286 and later processors, words are moved and stored as quickly as bytes. Therefore, moving 50 words is much faster than moving 100 bytes. If you know ahead of time how many bytes are going to be processed and that the number is even, you can simply load CX with half the value, and use Rep Movsw or Rep Stosw instead of Rep Movsb or Rep Stosb. This trick can be used even if the program runs on an 8088, but the speedup only occurs with 80286 and later CPUs. With only a little added code you can also use this technique to determine at runtime if an odd byte needs to be processed. Here's one way to do that:

```
Shr  CX,1      ;divide CX by 2
Rep  Movsw     ;copy the words
Jnc  Done     ;the Carry Flag is clear
Movsb                    ;copy the odd byte
Done:
.           ;program continues
.
```

First, CX is divided by 2, and the odd bit, if there was one, is stored by the CPU in the Carry Flag. Then the data words are copied to their destination. Finally, the Carry flag is tested and the program either copies a single additional byte or skips over that command.

A Jump Not Taken is Faster Than One That is

And this brings us to yet another cycle saver. In some cases the Jnc will be executed, and in others it will not. And in most programs, the chances of either happening are about fifty-fifty. But if you know ahead of time that a particular action will happen less often than another, you can take advantage of another 8088 fact: A jump not taken is always faster than one that is taken.

Each time the 8088 jumps to a new location or calls a procedure, it discards its *pre-fetch queue*. The pre-fetch queue is a small area of memory on the CPU itself that holds the next few instructions to be executed. In many cases, the 8088 can do several things at once. So while it is adding or subtracting

numbers, it simultaneously fetches instruction bytes from your code, in anticipation of what it will do next. This lets the CPU act on the subsequent instructions very quickly, because they are already in its own local on-chip memory. Just as data in registers can be accessed faster than data that must be read from memory, so too can instructions that are already in the CPU.

But when execution branches to a new location, any bytes present in the pre-fetch queue are obsolete. Therefore, the 8088 must read the new bytes at the new location, which takes additional time. If you have a routine that makes a test repeatedly within a loop you should change the logic as necessary, to branch on the less likely situation. That is, instead of `Jne` you might use `Je`, or vice versa.

Miscellaneous Techniques

One very powerful technique you will surely find useful is self-modifying code. As its name implies, self-modifying code actually writes new instructions into its own code segment, and this is useful in a variety of situations. For example, if you are writing a routine that accepts a variable number of parameters this lets you patch the `Ret` instruction to be `Ret 2`, `Ret 4`, and so forth.

One warning, however, is related to the pre-fetch queue. If a byte or word has already been read into the CPU, changing it in the code segment has no effect. Worse, there is no way to know for certain which bytes will have already been read, because the size of the pre-fetch queue has grown with each new CPU from Intel. For example, only four bytes are allocated for a pre-fetch queue on an 8088, but the 80386 uses 16 bytes.

In general, if the code you are patching is located at least a few dozen bytes farther in the program, you should be safe. Such self-modifying code was used in the `SORT.ASM` routine shown in Chapter 8, to let the same code sort either forward or backward. There, the bytes that represent `Jae` and `Jbe` were assigned to `AL` and `AH`, and the code was patched based on the incoming sort direction. Since the patching takes place a hundred or so bytes earlier in the program, it is unlikely that this routine will fail with future processors.

Static-Free CGA Text Display

The final technique you will find useful is writing to CGA text mode video memory without creating a disturbance. When IBM designed the original CGA adapter they skimmed on the design, using circuitry that shares a single address line for both the 8088 CPU and the video hardware that updates the screen. Even when a program is not reading from or writing to display memory, that memory is still read periodically by the display adapter and sent to the monitor. Therefore, accessing that memory directly from an assembly language routine creates a disturbing burst of static that is visible on the monitor. This is caused by the conflict of the CPU and the video adapter accessing the same video memory addresses at the same time.

Newer CGA adapters employ a dual-port design that arbitrates simultaneous read and write requests, thereby eliminating this problem. And, of course, EGA and VGA adapters are much more sophisticated

than the CGA, and fortunately also more common these days. However, you can avoid the screen disturbance on older CGA adapters by synchronizing your reading and writing with the horizontal retrace timing.

As you undoubtedly know, the image on a CRT is drawn by scanning a single dot horizontally across each successive row. This happens so quickly that the eye perceives the moving dot as an entire image. After each row is drawn, the dot is turned off, quickly placed at the start of the next row below, and then turned on again. By writing to the screen only while the dot is turned off you can hide the memory conflicts that cause static.

The short code fragment below shows how to synchronize video writing with the CGA's horizontal retrace. In a windowing routine that also needs to read video memory, you would use the same technique just before each byte or word is read.

```

.
.
Mov  SI,Descriptor  ;get the incoming descriptor address
Mov  CX,[SI]        ;the string's length goes in CX
Mov  SI,[SI+2]      ;and the address of the data in SI

Mov  AX,&HB800      ;load ES with the CGA video segment
Mov  ES,AX          ;through AX
Xor  DI,DI          ;point DI to the upper left corner

Mov  AH,Color       ;load color parameter (passed BYVAL)
Jczz Done           ;don't try to print a null string!

No_Retrace:
  In  AL,DX          ;get the video status byte
  Test AL,1          ;test the horizontal retrace bit
  Jnz No_Retrace    ;if doing retrace, wait until done
  Cli               ;disable interrupts until we're done
Retrace:
  In  AL,DX          ;get the status byte again
  Test AL,1          ;are we currently doing a retrace?
  Jz  Retrace        ;no, wait until we are
  Lodb               ;load the current character
  Stosw              ;store the character and attribute

  Sti               ;re-enable interrupts
  Loop No_Retrace   ;loop until the string is printed

Done:
.                  ;program continues or exits here
.

```

The current horizontal retrace status can be read using the In instruction, and then masking off all but the lowest bit. To protect against the case where the print loop is entered just as the retrace is about to end, this routine waits until a new period has just begun. This is not unlike the empty loop used in the benchmark examples in Chapter 9, that waited for a new system clock cycle to begin.

Summary

In this final chapter you have learned what assembly language programming is all about, and how it can help you as a BASIC programmer. There is no doubt that using assembly language is more tedious than BASIC, but the overall methods and code structures are similar.

You learned about the 8088's registers, and why operations that use them are faster than similar operations on memory variables. The string instructions are particularly useful, because they are very small and do several things at once. Coupled with the Rep prefix these commands can replace many separate Mov and Inc and Cmp statements. You also learned how to perform simple calculations in assembly language, and an example showed how to translate simple BASIC integer and floating point expressions.

This chapter explained how the stack operates, and how procedures are designed to accept passed parameters. The new simplified directives introduced with MASM 5.1 eliminate the need to define segments and figure parameter stack displacements in your routines. This chapter also explained how to call DOS and BIOS interrupts from assembly language.

You learned how to access every kind of data a BASIC program can pass to a routine, including near and far strings, integers, and even floating point values. The section that described arrays showed how to access both near and far data, and even huge arrays that span multiple segments.

Besides conventional called procedures, you also learned how to create functions that can return any type of data. Several innovative techniques were presented, including a method for creating a single procedure that can work with both near and far strings, and even with different versions of the BASIC compiler. Equally innovative are the methods that show how to write floating point instructions and tie them into BASIC's software emulator. And if you are not certain how to code a particular floating point instruction, you can create a short BASIC program and then examine its code using CodeView.

This chapter explained many of MASM's features, such as initialized data, conditional assembly, and defining structures and macros. In particular, macros can greatly simplify coding redundant instructions and data definitions. Furthermore, MASM can calculate data addresses and lengths automatically, reducing your work when the data must be changed later on.

Because so many different data items all compete for the same 64K near memory segment, it is often desirable to store working variables on the system stack. Likewise, when large amounts of data are involved, variables and tables can be stored in the code segment. Both of these techniques were described in depth, and accompanying examples showed how to do this in context.

Several of BASIC's most useful internal variables and procedures were described, showing their public names and parameter requirements. The GetNames function brought all of this information together, showing how to read an array descriptor, redimension a string array, and assign individual elements—all using code that works identically with both near and far strings.

You also learned how to write an interrupt handler that can be installed and uninstalled from within a BASIC program. The example showed how to take over the keyboard interrupt; however, the same technique can be applied to nearly any other hardware or software interrupt as well.

Finally, this chapter described many useful tricks and techniques that help to reduce the size of your assembly language routines, and also make them faster. Many operations that use the AX register result in less code than the same operations using other registers. And when moving or storing contiguous data, accessing the data as words instead of bytes can sometimes yield a nearly two-fold speed improvement. When in doubt about which of several sequences of code is smaller, you can use the DOS DEBUG utility to quickly determine that.

Appendix

Overview of the Accompanying Files

BTU.ZIP contains all of the example code, BASIC programs, and assembly language source files used in this book. The example BASIC files have names like CHAPn-n.BAS, where the first part of the name indicates which chapter the example was used in, and the second part is the listing number within that chapter. The remaining BASIC files use more descriptive names, and those are the ones you're most likely to actually use and add to your own programs. Likewise, the shorter assembly language examples from Chapter 12 are in files named CHAP12-n.ASM, but source code for the complete routines are in files having names based on the actual routine names.

The library files named BASIC.* are meant for use with QuickBASIC version 4.0 or later, and the files named BASIC7.* are for use with BASIC PDS and VB/DOS. Files with a .QLB extension are Quick Libraries that you load along with QB.EXE or QBX.EXE or VBDOS.EXE, depending on your version of BASIC. The .LIB files are intended for use with LINK, when you create executable programs. There are also a few .BI (BASIC Include) and .MAK (Make) files used to support some of the programs. I did not bother to include separate .OBJ files for the assembly language routines, since you can easily extract them from BASIC.LIB or BASIC7.LIB if you need them.

Starting BASIC

To start QuickBASIC and load the BASIC.QLB library, enter this from the DOS command line:

```
qb [program] /l basic.qlb /ah
```

If you specify the optional BASIC source program name, that is loaded into the QuickBASIC editor along with the BASIC.QLB library. The /ah switch tells QuickBASIC to allow huge (greater than 64K) arrays, which is needed for some of the demonstration programs.

If you are using BASIC PDS, start QBX as follows:

```
qbx [program] /l basic7.qlb /ah /es
```

The /es switch is needed for the EMS.BAS demonstration, and it tells QBX to cooperate with your use of Expanded memory. When /es is omitted, QBX assumes no other programs are using EMS, which lets it access that memory slightly faster. Since EMS.BAS stores its sample data in Expanded memory, this option is needed to avoid corrupting EMS memory. Even if you do not plan to run EMS.BAS, using /es is harmless.

If you have VB/DOS you should start it like this:

```
vbdos [program] /l basicvbd.qlb /ah /es
```

Once the appropriate Quick Library has been loaded, you may use the File Open menu sequence to load the BASIC programs. Note that with VB/DOS, the Open menu defaults to a .MAK extension, so you'll have to enter *.BAS or type the complete name of a BASIC program source file.

Some of the example programs use BASIC's CALL Interrupt command, and to run those you will have to quit the BASIC editor, and restart it loading the default Quick Library that comes with your version of BASIC. You do not need to specify a Quick Library name when loading the default library; using only the /l switch is sufficient.

Linking

When you compile and link programs manually from the DOS prompt and you want to use the .LIB libraries supplied with this book, you must specify the library name manually on the LINK command line:

```
link program [/options] , , nul, basic[7] ;
```

The BASIC7 library works with both BASIC PDS and also VB/DOS, so it was not necessary to provide a separate BASICVBD.LIB file. You can also compile and link from within the QuickBASIC or QBX editors using the menu options. When a Quick Library is loaded, the BASIC editor uses the same first name for the LINK library when it shells to run BC and LINK. For example, if you started QBX like this:

```
qbx /l basic7.qlb
```

QBX tells LINK to use a parallel .LIB library named BASIC7.LIB. But since there is no BASICVBD.LIB file you must compile and link manually when using VB/DOS. If you don't use BASIC PDS you can optionally rename BASIC7.LIB to be BASICVBD.LIB. Then when you start VB/DOS as shown above it can specify the correct library name when it shells to LINK.

The BASIC editor limits you to using either the Quick Library from this book or BASIC's version that contains CALL Interrupt—you cannot load two Quick Libraries at one time. However, you can link with more than one library when creating an executable program manually. This example is for QuickBASIC, and you would substitute QBX.LIB and BASIC7.LIB with PDS, or VBDOS.LIB and BASIC7.LIB when using VB/DOS:

```
bc program [/o] ;  
link [/options] program [other modules], , nul, qb.lib basic.lib ;
```



What really happens inside a compiled program?

BASIC Techniques and Utilities takes the QuickBASIC and BASIC 7 programmer into the heart of the BASIC language, exploring low-level details not found in any other book. It explains precisely how the BASIC compiler works, how memory is organized, and how to increase your productivity with a variety of original programming techniques.

BASIC Techniques and Utilities reaches an unprecedented level of coverage on BASIC's internal operation, and contains many undocumented BASIC routines that will dramatically increase productivity. Winer also presents instructions for accessing DOS and BIOS services using CALL INTERRUPT and assembly language.