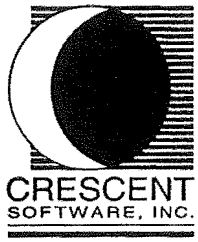
Professional Programming Tools
for BASIC Compilers

P.D.Q.

Owner's Manual



P.D.Q.
A New Concept in High-Level Programming Languages

Version 3.13

P.D.Q. was conceived and written by Ethan Winer, with substantial contributions (that is, the really hard parts) by Robert L. Hummel.

The example programs were written by Ethan Winer, Don Malin, and Nash Bly, with additional contributions by Crescent and Full Moon customers. The floating point math package was written by Paul Passarelli. This manual was written by Ethan Winer. The section that describes how to use P.D.Q. with assembly language was written by Hardin Brothers.

Full Moon Software
34 Cedar Vale Drive
New Milford, CT 06776
Sales: 860-350-6120
Support: 860-350-8188 (voice); 860-350-6130 (fax)

Sixth printing.

LICENSE AGREEMENT

Crescent Software grants a license to use the enclosed software and printed documentation to the original purchaser. Copies may be made for back-up purposes only. Copies made for any other purpose are expressly prohibited, and adherence to this requirement is the sole responsibility of the purchaser. However, the purchaser does retain the right to sell or distribute programs that contain P.D.Q. routines, so long as the primary purpose of the included routines is to augment the software being sold or distributed. Source code and libraries for any component of the P.D.Q. program may not be distributed under any circumstances. This license may be transferred to a third party only if all existing copies of the software and documentation are also transferred.

WARRANTY INFORMATION

Crescent Software warrants that this product will perform as advertised. In the event that it does not meet the terms of this warranty, and only in that event, Crescent Software will replace the product or refund the amount paid, if notified within 30 days of purchase. Proof of purchase must be returned with the product, as well as a brief description of how it fails to meet the advertised claims.

CRESCENT SOFTWARE'S LIABILITY IS LIMITED TO THE PURCHASE PRICE. Under no circumstances shall Crescent Software or the authors of this product be liable for any incidental or consequential damages, nor for any damages in excess of the original purchase price.

To My Honey, Elli Mastrangelo Winer

Table of Contents

Chapter 1: Introduction

Chapter 2: Compiling and Linking

Chapter 5: P.D.Q. Extensions

Chapter 6: Using P.D.Q. With Assembly Language Programs

Chapter 7: Programmer's Reference

Appendices

Appendix A:

Appendix B:

Appendix C:

Appendix D:

Appendix E:

Appendix F:

Appendix G:

Appendix H:

Appendix I:

Chapter 1: Introduction

■ ■ ■

Section I, Chapter 1: Introduction

Thank you for purchasing P.D.Q. We have made every effort in designing P.D.Q. to create a powerful, yet easy to use replacement library for linking with Microsoft compiled BASIC. We sincerely hope that you find P.D.Q. both useful and informative. If you have a comment, a complaint, or perhaps a suggestion for another BASIC-related product, please let us know. We want to be your favorite software company.

Before we begin discussing the contents of the P.D.Q. disk and manual, please take a few moments to fill out the enclosed registration card. Doing this entitles you to free technical support by phone, as well as ensuring that you are notified of possible enhancements and new products. Many upgrades are offered at little or no cost, but we can't tell you about them unless we know who you are! Note, however, that if you purchased P.D.Q. directly from us, the mail-in portion of the registration card may have been removed. In this case, you are already registered.

Please mark the P.D.Q. product serial number on your disk label or manual cover. License agreements and registration forms have an irritating way of becoming lost, and doing this will insure that the number is handy if you need to contact us. You may also want to note the product version number in a convenient location; this is stored on the distribution disks in the volume label. If you ever have occasion to call us for assistance, we will need to know your serial number, and probably the version you are using as well. To determine the version number for any Crescent Software product, simply use the DOS VOL command, which will display the disk volume label:

```
VOL A:
Volume in drive A is PDQ V3.XX
```

We are constantly improving all of our products, and you may want to call us periodically and ask for the current version number. Major upgrades are always announced, however minor additions or fixes are generally not. If you are having any problems at all—even if you are sure it is not caused by one of our products—please call us. We support all versions of compiled BASIC, and can often provide better assistance than Microsoft.

About This Manual

This manual is divided into several sections—the important ones are an overview which describes what P.D.Q. is all about, a section that discusses various aspects of P.D.Q. programming, and an appendix that also includes several in-depth tutorials and programming hints. Newly added with

version 3.00 is a complete tutorial with examples showing how to use P.D.Q. as an assembly language toolbox.

The overview provides an important first look at BASIC programming with P.D.Q., and explains which BASIC statements have been omitted and why. The details section begins with a discussion of the many language extensions that are included with P.D.Q. Other important topics include file operations, TSR and interrupt handling considerations, and a complete description of each external P.D.Q. subroutine. A "How We Did It" section explains how P.D.Q. works internally and how it was developed, and several tutorials are included which cover CALL INTERRUPT and the DOS environment. Finally, a table of common error messages and their cause is given, along with a section that describes various ways to optimize your programs.

Besides the information in this manual, there are a number of files that contain additional information you may find useful. Perhaps most important, we include all of the assembly language source code for P.D.Q. This will be invaluable if you are interested in learning more about BASIC's internal workings. Programmers who need to create embedded ROM applications using P.D.Q. should also be sure to see the EMBEDDED.DOC file.

Installing P.D.Q.

All of the files that constitute P.D.Q. are compressed and stored in the .ZIP files on the accompanying disks. We use .ZIP files because the entire contents of P.D.Q. encompasses more than 2MB of library, example, and source files. To help you get these files copied correctly onto your hard disk we have included an automated installation utility. The list below shows which files are provided in the order they occur on the disks, along with a brief description of each.

INSTALL.EXE	The Crescent Software installation program
PKUNZIP.EXE	PKWARE's .ZIP file extract utility
PDQ.ZIP	Libraries and example programs
SOURCE.ZIP	Assembly language source code for routines in PDQ.LIB
FPSOURCE.ZIP	Assembly language source code for the floating point routines
ASM.ZIP	Files for using P.D.Q. as an assembly language toolbox
SMALLDOS.ZIP	Assembly language source code for routines in SMALLDOS.LIB
BASIC7.ZIP	Assembly language source code for routines in BASIC7.LIB

CURRENCY.ZIP Assembly language source code for the Currency data routines

Installing P.D.Q. is very easy. Simply log on to Drive A, place Disk 1 into that drive, and enter INSTALL at the DOS prompt. As the files from each disk are unpacked and copied to your hard disk, INSTALL prompts you to insert the next disk. This continues until all of the files have been copied.

On-screen instructions explain how to use INSTALL. Notice, however, that F2 lets you see the file names inside each .ZIP file, and selectively mark or unmark them for installation. This feature lets you install only certain files. You can also mark and unmark entire .ZIP files.

It is not necessary to install the assembly language source code to use P.D.Q., and we include it solely for those people who are interested. If you do not plan to install the assembly language source code, you can simply exit the program by pressing F4 after PDQ.ZIP on disk 1 has been installed.

By default, installation is to C:\PDQ, though you can change that to reflect any valid drive and directory. If the directory you specify does not exist, INSTALL will create it. You can also switch directories during installation by pressing Tab and editing the directory name. We recommend that you install the assembler source code into its own directory.

Notice that the assembler source files for BASIC 7 PDS far strings and the SMALLDOS.LIB library have the same names as the equivalent files for use with near strings. Therefore, if you plan to install all of the source code you should create four directories: \PDQ, \PDQ\SOURCE, \PDQ\SOURCE7, and \PDQ\SMALLDOS.

Please note that there are a few empty files with names like "-" and "--" that serve as separators between logically grouped sets of files. Because these files have a zero length they do not take up any disk space other than the 32 bytes used by all directory entries. However, most hard disk tune-up programs will not move zero-length files, so you may want to delete them after you have installed P.D.Q.

If you are familiar with the PKUNZIP program, you can optionally run it manually. Entering PKUNZIP with no arguments displays a help screen that shows all of the option switches it recognizes. PKUNZIP is provided under license from PKWARE, Inc.

P.D.Q. Overview

P.D.Q. is a replacement linking library for use with Microsoft QuickBASIC version 4.0 or later or BASIC 7 PDS. When a compiled BASIC program is linked with PDQ.LIB instead of the usual BCOM library supplied with BASIC, the .EXE file size will be reduced dramatically. Code size reductions of six to one are typical for very small source files, however the actual improvement will of course depend on the individual program. Program execution speed when using P.D.Q. will also be improved in many cases. Most applications that have been linked with P.D.Q. will be noticeably smaller than an equivalent written in C, and will in fact be closer to pure assembly language.

P.D.Q. also features a number of important language extensions, including TSR program support and interrupt handling. Writing TSR programs and interrupt handlers usually requires an extensive knowledge of assembly language; however, P.D.Q. includes a full complement of routines that allow you to do this using only BASIC commands and simple extensions. Many other important features are provided with P.D.Q. and we will get to those shortly.

The primary purpose of P.D.Q. is to create .EXE programs that are as small and fast as possible. There is very little error checking beyond simple syntax errors which are caught at compile time. However, because P.D.Q. is based on Microsoft BASIC, programs may also be developed and tested in the more secure environment BASIC offers, and then linked for maximum efficiency using the P.D.Q. library once they are working correctly.

P.D.Q. has been designed as a subset of the recognized industry standard BASIC that has been established by Microsoft. Programs you create using P.D.Q. are inherently well-behaved, and may thus be run under operating systems such as Microsoft Windows and Quarterdeck's DESQview without any additional effort. Because P.D.Q. programs are also BASIC programs, they may be further enhanced with add-on libraries such as our QuickPak Professional and Graphics Workshop products.

By default P.D.Q. is highly compatible with Microsoft's BASIC compilers and QuickBASIC. In many cases, existing programs may be linked with P.D.Q. with little or no change to the program's source code. However, we have provided a number of options that let you reduce the size of your programs even further. For example, P.D.Q. includes a reduced-capability version of LOCATE which adds only 29 bytes of code to your program. The only restriction is that you must use two—and only two—arguments to specify the new row and column.

Contrast that with the default LOCATE in P.D.Q. which adds 187 bytes. Because the default LOCATE must also be able to turn the cursor on and off and change its size, that much additional code is added to your programs, even if they only need to position the cursor! A number of other alternate BASIC statement support routines are included with P.D.Q., and each is described in detail in the section *Linking With Stub Files*.

Finally, two Quick Libraries are provided with P.D.Q., containing most of the P.D.Q. extensions for use in the QB and QBX editing environments. The remaining extensions are in a BASIC source file named PDQSUBS.BAS, for those that we could not implement in a form suitable for use in a Quick Library. Please understand that when you link with P.D.Q. *all* of the routines and extensions are written in assembly language. The BASIC versions are solely for use in the QB and QBX environments.

Why BASIC?

Some programmers, particularly those who don't ordinarily program in BASIC, may wonder why we would select BASIC as the core language for P.D.Q. Simply put, BASIC is the easiest of all the high-level languages to use, and nearly every programmer is already familiar with it. Microsoft BASIC provides all of the features necessary for modern, structured programming. Further, the BC.EXE compiler supplied with BASIC is as powerful and capable as any available language compiler. P.D.Q. programs may be written and debugged in the convenient environment Microsoft BASIC offers, and then linked for maximum performance when creating the final program.

But Isn't That What C Is For?

One of the promises of the C language was to provide smaller and faster programs, in exchange for additional effort. If you were willing to step down to a lower level language nearer to assembler, the compiler would reciprocate by generating a more efficient program. Unfortunately, this simply isn't the case—the current generation of C compilers offers little if any improvement over QuickBASIC 4.5 or BASIC 7 PDS. Further, C programs are notoriously tedious to write and difficult to debug. A wise programmer we know once called C a "write-only" language. While that may be a bit of an overstatement, most programmers would agree that C is not for the timid. There are very few things that C can accomplish which BASIC cannot, compared with the many capabilities in BASIC that C could never match.

Of course, any compiler should provide at least some amount of protection to prevent a programmer from simple mistakes. For example, a single mistyped variable name should not crash the system. Clearly, a well designed language will offer an effective compromise between features, ease of use, and efficiency of the generated code. That language is P.D.Q.

The Spirit Of Performance

Our goal in designing P.D.Q. was to place code size and execution speed above all other considerations. Many of BASIC's most advanced features are not included, and some commands have been implemented in a slightly different manner. Therefore, we'll begin by looking at what has been omitted. Please understand that in all cases where a BASIC feature is not supported, the improvement in code size or speed was the deciding factor.

Also understand that in exchange for only a slight increase in programming effort, P.D.Q. will reciprocate with incredible improvements in program performance and size. The primary purpose of P.D.Q. is to create extremely small programs that execute very quickly. If you intend to write a major accounting program or engineering application, you will probably be better off using regular Microsoft BASIC.

P.D.Q. Supported Key Words

Table I-1 lists all of the BASIC commands and functions that are supported by P.D.Q. Some keywords either have a slightly different syntax, some restrictions, or enhancements. Other keywords are not supported at all, but have an equivalent P.D.Q. extension routine. These are identified with bullets in that table.

TABLE I-1
P.D.Q. Supported Key Words

ABS	ABSOLUTE	ALIAS	AND
ANY	AS	ASC	BASE
BEEP	BINARY	BLOAD	BSAVE
BYVAL	CALL	CALLS	CCUR
CDBL	CDECL	CHDIR	CHDRIVE
CHR$	CINT	CLNG	CLOSE
CLS	COLOR	COMMAND$	COMMON
CONST	CSNG	CSRLIN	CURDIR$
CVD	CVI	CVL	CVS
DATA	DATE$	DECLARE	DEF FN
DEF SEG	DEF*type*	DIM	DIR$
DO	$DYNAMIC	ELSE	ELSEIF
END	ENDIF	ENVIRON	ENVIRON$
EOF	EQV	ERASE	ERR
ERROR	EXIT	FILEATTR	FILES
FIX	FOR	FRE	FREEFILE
FUNCTION	GET	GOSUB	GOTO
HEX$	IF	IMP	$INCLUDE
INKEY$	INP	INPUT	INPUT#
INPUT$	INPUT$ #	INSTR	INT
INTEGER	IOCTL	IOCTL$	IS
KILL	LBOUND	LCASE$	LEFT$
LEN	LET	LINE INPUT	LOC
LOCATE	LOCK	LOF	LONG
LOOP	LPRINT	LSET	LTRIM$
MID$	MKC$	MKD$	MKDIR
MKI$	MKL$	MKS$	MOD
NAME	NEXT	NOT	OCT$
ON ERROR	ON GOSUB	ON GOTO	OPEN
OPTION BASE	OR	OUT	OUTPUT
PEEK	PLAY	POKE	POS
PRINT	PRINT #	PUT	RANDOMIZE
READ	REDIM	REM	RESUME line
RETURN	RIGHT$	RMDIR	RND
RSET	TRIM$	■ RUN	SADD
SCREEN	SEEK	SELECT	SGN
SHARED	SHELL	SLEEP	SOUND
SPACE$	SPC	SSEG	SSEGADD
STATIC	STEP	STOP	STR$
STRING	STRING$	SUB	SWAP
SYSTEM	THEN	TIME$	TIMER
TO	TYPE	UBOUND	UCASE$
UNLOCK	UNTIL	■ USING	VAL
VARPTR	VARSEG	WAIT	WEND
WHILE	XOR	+	-
*	/	\	

Table I-2 lists all of the BASIC keywords that cannot be used with P.D.Q. These are primarily related to advanced math and BASIC 7 ISAM statements. All of the differences between standard BASIC commands and their P.D.Q. implementation are described in the next section.

TABLE I-2

Keywords *Not* Supported By P.D.Q.

ALL	ATN	BEINGTRANS	BOF
CHAIN	CHECKPOINT	CIRCLE	CLEAR
COM	COMMITTRANS	COS	CREATEINDEX
DELETE	DELETEINDEX	DELETETABLE	DRAW
ERDEV	ERDEV$	EVENT	EXP
GETINDEX$	INSERT	ISAM	KEY
LINE	LOCAL	LOG	LPOS
MOVEFIRST	MOVELAST	MOVENEXT	MOVEPREVIOUS
OFF	PAINT	PALETTE	PCOPY
PEN	PMAP	POINT	PRESET
PSET	RETRIEVE	ROLLBACK	RUN
SAVEPOINT	SEEKEQ	SEEKGE	SEEKGT
SETINDEX	SIGNAL	SIN	SQR
STACK	STICK	STRIG	TAN
UEVENT	UPDATE	USING	VARPTR$
VIEW	WINDOW	WRITE #	^

Differences Between P.D.Q. And Microsoft BASIC

This section lists all of the differences between P.D.Q. and regular Microsoft BASIC. The section that follows lists each BASIC keyword that is supported by P.D.Q. but is different than regular BASIC. Note that some of these differences are limitations, while others are useful enhancements to the language.

Floating Point Math

P.D.Q. provides only limited support for floating point numbers. Only the four basic functions (add, subtract, multiply, and divide) may be used; however, P.D.Q. does support the ABS, INT, FIX, SGN, RND, and TIMER floating point functions. P.D.Q. also supports CINT, CLNG, CSNG, and CDBL, as well as MKS$, CVS, MKD$, and CVD. Of course, floating point comparisons such as IF X! > Y# may also be used.

In many cases floating point math is not truly needed, and a program's size can be made much smaller by using only integers and long integers. When 2-place fixed point numbers are sufficient you can treat them as long integers, and then format them when needed. A special routine named Dollar$ is provided for this purpose. This is a well-known technique, and it lets you accommodate dollar amounts up to plus or minus 21 million dollars.

Instead of supporting the PRINT USING statement, P.D.Q. instead comes with a function that returns a string formatted in a similar fashion. Adding a USING option to the PRINT statement the way regular BASIC does would mean adding extra code to the core PRINT routine. This would have made all programs larger, or required yet another link step to remove that support.

By returning a string FUsing is more flexible than PRINT USING, which only lets you display the formatted result. FUsing is from our QuickPak Professional library, and it handles all of the formatting options of PRINT USING except for scientific notation.

Communications and Graphics

There is no built-in support for either communications or graphics. Though you can easily switch the PC's screen to any of the graphics modes supported by the hardware using SCREEN, we do not support the BASIC syntax for drawing lines, boxes, or circles. A collection of BASIC subroutines are provided for drawing lines and circles in the EGA and VGA modes only. Although communications and graphics are not supported directly, Crescent Software offers the PDQComm and Graphics Workshop libraries separately for these purposes.

DOS and Other Errors

DOS critical errors such as an open drive door will result in the familiar Abort/Retry/Fail message. This is also true for the P.D.Q. extensions such as PDQExist, which checks for the presence of a file. Unlike a BASIC program that prints "Drive not ready at address ####:####" and then dies, a P.D.Q. program at least gives the user a chance to retry. However, we have also provided routines that let you easily trap and act upon critical errors if you prefer.

We assume that you have already gotten your program to work using regular BASIC *before* attempting to link it with P.D.Q. Most runtime errors such as "Out of string space" or "Subscript out of range" are ignored. Therefore, if you dimension an array that is larger than 64K but

forget to compile with the /ah switch, your program will not work and no error will be reported. However, you can detect most such errors by compiling with the /d (Debug) switch. Please remember, P.D.Q. is intended primarily as an alternative to writing in C or assembly language. P.D.Q. provides you with an incredible amount of power, and what you do with it is limited only by your imagination.

The most important difference is the way errors—especially DOS file errors—are detected and handled. With P.D.Q., errors such as "Out of string space" or "File not found" do not end your program suddenly and without warning. Instead, the offending statement simply sets BASIC's ERR function to indicate the type of error that occurred. This is in many ways more sensible than crashing your program, or requiring ON ERROR as the only practical alternative.

The following short example shows how you would open a file for input, and detect a "File not found error":

```
FileName$ = "\AUTOEXEC.BAT"
OPEN FileName$ FOR INPUT AS #1
IF ERR = 53 THEN
  PRINT "File not found"
  END
END IF
 ...
 ...
```

Although P.D.Q. partially supports ON ERROR, we recommend it primarily for debugging your programs. In particular, ON ERROR in a P.D.Q. program does not always recognize critical errors such as an open drive door or a printer that is off line.

Please see the section *File Handling in P.D.Q.* for more information on how DOS errors are handled by P.D.Q.

Ctrl-C and Ctrl-Break

As with regular BASIC, a P.D.Q. program ignores the Ctrl-C and Ctrl-Break keys unless it is compiled with /d (Debug). However, one very important exception is when using PRINT, INPUT, or LINE INPUT. Because these BASIC statements call upon the built-in DOS services, DOS will end the program if Ctrl-C or Ctrl-Break are pressed either during or prior to console input or output.

There are several solutions at your disposal. One is to use the _CPRINT.OBJ stub file that is discussed in the section *Linking With Stub Files* elsewhere in this manual. _CPRINT replaces the default PRINT with an alternate version that uses the BIOS instead of DOS. Likewise,

the BIOSInput routine can be used to replace INPUT and LINE INPUT, and this too avoids calling DOS services. You can also use the BreakOff and BreakOn routines to disable Ctrl-C and Ctrl-Break entirely, and these are described in the reference section of this manual.

Recursive Procedures

P.D.Q. supports creating recursive subprograms and functions with only one exception: string functions. Failing to add the STATIC identifier as part of a string FUNCTION definition causes LINK to report B$SCPF as an unresolved external procedure.

Dynamic Arrays

The P.D.Q. version of DIM when used with dynamic numeric, TYPE, and fixed-length string arrays operates differently than the same command in regular BASIC. In a P.D.Q. program once a dynamic array has been dimensioned, it will never move around in memory. In contrast, QuickBASIC's far heap manager often moves arrays around, as it maintains memory to keep it contiguous.

Each method has its advantages, however the P.D.Q. method requires far less code to implement which was our main concern. When an array is guaranteed not to move as in P.D.Q. (and also C and Pascal), you can create linked lists between data items, confident that the pointers will always be valid. The downside is that memory fragmentation can occur when other arrays are dimensioned and erased. In that case it is possible to have plenty of memory available, but as many small blocks and not enough for a single large array.

Huge Arrays

Another important difference that isn't directly related to a BASIC key word is improved support for huge (/ah) arrays. In regular BASIC, you can create a huge array that exceeds 128K in size only if the length of each element is a power of 2. This is not a problem with numeric arrays, since they all have lengths that are 2, 4, or 8 bytes. But with fixed-length string and TYPE arrays, an element can be nearly any length. If the length of each element is not power of 2 and the entire array is larger than 128K, one or more elements will straddle a segment boundary which Microsoft BASIC cannot handle properly.

P.D.Q. does not have this limitation, and a huge array can have elements of any size, up to the maximum of 32,767 elements per dimension or the limits of DOS memory. The only limitation P.D.Q. does impose is that no single array element may be larger than 32,767 bytes. Regular BASIC

allows a single TYPE element to be as large as 64K, which in turn causes the power of 2 limitation.

Be aware, however, that many third-party library routines cannot properly handle huge arrays whose element length is not a power of 2. In particular, the TYPE sort routines in our QuickPak Professional product will not work with arrays whose elements span a segment boundary.

BASIC 7 Advanced Features

Although P.D.Q. programs may be compiled using the BC.EXE program that comes with BASIC 7 PDS, it does not support the most advanced features of that version. Specifically, P.D.Q. does not support far strings, procedure overlays, custom runtime modules, ISAM files, or the alternate math library. If you really need those features then you really must expect larger programs than P.D.Q. can produce.

Using Overlays

P.D.Q. does support the use of program overlays, but not using the BASIC 7 PDS overlay manager. P.D.Q. has been successfully tested with the commercial programs PLINK and RT-LINK, and with the shareware LOVR program by Michael Devore (available on CompuServe). Overlays may be used both in conventional and simplified TSR programs, but not within a TSR that takes over interrupts manually without also using the PopRequest routine.

If you plan to use overlays in a TSR program it is important that you not use the interrupt method, since that could conflict with a foreground program's use of the same interrupts if it too uses overlays. LOVR does not offer an option to have its overlay manager be called directly, so you should use that product with care.

Differences By Key Word

ASC

Where BASIC's ASC() function creates an "Illegal function call" error when it is used on a null string, the P.D.Q. implementation instead returns -1. This can save unnecessary BASIC code by eliminating an extra test just for a null string:

```
IF LEN(Work$) THEN        'this is needed in QuickBASIC
  IF ASC(Work$) = 65 THEN
    ...
    ...
```

```
  END IF
END IF

IF ASC(Work$) = 65 THEN   'this is all P.D.Q. needs
   ...
   ...
END IF
```

BLOAD

In regular BASIC, an address parameter for the BLOAD command is optional. If omitted, BASIC uses the segment and address that were stored in the file's header when it was created. P.D.Q. does not support that feature, therefore you must specify an explicit address to load to, using the current DEF SEG segment:

```
DEF SEG = Segment
BLOAD FileName$, Address
```

CALL INTERRUPT

Where the Microsoft BASIC version of CALL INTERRUPT expects two TYPE variables to define the CPU registers, the P.D.Q. version uses only one. That is, the same set of registers are used both going into and coming out of the call to INTERRUPT. We made this change because it reduces the size of the INTERRUPT routine, and also avoids the memory needed for a second copy of the Registers variable. Two copies of a Registers variable is rarely needed; however, if you really do need to maintain separate variables, simply make a copy before calling INTERRUPT:

```
DIM InRegs AS Registers, OutRegs AS Registers
  ...
  ...                                    'set up InRegs here
LSET OutRegs = InRegs                    'make the copy
CALL INTERRUPT(IntNum, OutRegs)          'InRegs is still intact
```

CHDRIVE

CHDRIVE is a BASIC PDS enhancement that lets you change the current default drive. Because it is in the P.D.Q. library, QuickBASIC programmers can also use CHDRIVE by declaring the internal routine and calling it directly. Here's the DECLARE and usage:

```
DECLARE SUB CHDRIVE ALIAS "B$CHDR" (Drive$)
Drive$ = "A"
CHDRIVE Drive$
```

Note that CHDRIVE may not be used in the QuickBASIC editing environment.

CHR$

Rather than report an "Illegal function call" error if an illegal value is used, the P.D.Q. version of CHR$ handles an invalid argument by ignoring the excess beyond 255. Specifically, negative numbers are treated as **256 - Number**, and numbers greater than 255 are considered as **Number MOD 256**.

COLOR

Because of the way printing is handled in P.D.Q., the COLOR command affects only CLS and the PDQCPrint "quick print" routine. However, you can link with the _CPRINT.OBJ stub file to have printing also honor the current COLOR settings. See the section entitled *Linking With Stub Files* for more information about using replacement BASIC statements.

COMMAND$

Like regular BASIC, the P.D.Q. version of COMMAND$ strips leading blank spaces. Unlike BASIC it removes leading Tab characters and also honors capitalization.

CURDIR$

Like CHDRIVE, CURDIR$ is a BASIC PDS enhancement that lets you determine the current directory for any drive. If you are using QuickBASIC you may declare the internal routine and call it directly like this.

```
DECLARE FUNCTION CURDIR$ ALIAS "B$FCD1" (Drive$)
Drive$ = "A"
Directory$ = CURDIR$(Drive$)
```

If Drive$ is null, CURDIR$ uses the current default drive.

Note that CURDIR$ may not be used within the QuickBASIC editing environment.

DIR$

DIR$ is another BASIC PDS function that can be used from QuickBASIC. DIR$ returns either the first or next file name that matches a given search specification. Declare the internal P.D.Q. routine as follows:

```
DECLARE FUNCTION DIR$ ALIAS "B$FDR1" (FileSpec$)
```

To find the first file that matches a given file specification use **D$ = DIR$(Spec$)**, where Spec$ is *.* or C:\BATCH*.BAT or the like. To find successive files that match the same spec use a null string (or no argument) for Spec$.

Note that DIR$ may not be used within the QuickBASIC editing environment.

ENVIRON and ENVIRON$

P.D.Q. provides a number of important enhancements to BASIC's ENVIRON and ENVIRON$ routines, such as accessing the parent's environment, and ignoring or honoring capitalization. These are discussed in depth in the section entitled *The Environment* elsewhere in this manual.

ERR

P.D.Q. supports only a subset of BASIC's error values. In most cases, the missing error codes are irrelevant. For example, "Syntax error" is a compiler error rather than a run time error. Further, errors are handled very differently by P.D.Q. Please see the section entitled *DOS Error Handling* for a complete discussion of this topic. Additional errors and their codes are described in the section entitled *The Environment.*

All of the errors that are supported by P.D.Q. are listed in Table I-3. Many of the error codes listed are used by BASIC; however, errors 83 through 127 are unique to P.D.Q. Also see the PDQMessage$ function, which returns a string containing the appropriate text for each error.

The P.D.Q. extended error codes may require additional explanation. Error 83 is relevant only if you are linking with the alternate SMALLDOS.LIB, which is described in the *SMALLDOS* section. Errors 101 through 112 are returned by the various TSR support routines, and they are described in the section that covers TSR programming. Errors 125 through 127 are used to report BASIC errors that either have no number, or where it was difficult for us to honor the normal Microsoft number. Note that these errors (125 through 127) are relevant only if you are compiling using the BC.EXE /d debug switch. Error 11 can only occur when using the HookIntØ routine.

ERROR

The ERROR statement may be used for assigning a value that will be returned by ERR. This is useful mostly in modules that need to convey

error information back to a caller, but without requiring an extra passed parameter. Of course, using ERROR does not halt your program as BASIC's implementation would (unless you are using ON ERROR). Rather, it simply sets the value that will be returned the next time ERR is queried. Since any value between 0 and 255 may be used, ERROR can also provide a simple way to pass information between modules without needing COMMON.

FILEATTR

BASIC's FILEATTR function may be used either to obtain the equivalent DOS handle for a file, or the mode with which a file had been opened. Because P.D.Q. allows nearly any operation on any file regardless of how it was opened, the second argument is ignored. Therefore, FILEATTR under P.D.Q. returns the DOS handle only, as shown below:

```
DOSHandle = FILEATTR(FileNum, Ignored)
```

FRE(-2)

In P.D.Q. the FRE function with an argument of -2 always returns the number of bytes of stack space that are currently available. In QuickBASIC and BASIC PDS, FRE(-2) instead returns a "low water mark" showing how much stack space was available at the deepest level encountered thus far in the program. This is a very minor difference to be sure, that is unlikely to affect most programs.

TABLE I-3
P.D.Q. Error Codes And Equivalent Messages

NUMBER	ERROR MESSAGE (also returned by PDQMessage$)
4	Out of DATA
5	Illegal function call
7	Out of memory
9	Subscript out of range
11	Division by zero
14	Out of string space
16	String formula too complex
52	Bad file number
53	File not found
54	Bad file mode
55	File already open
61	Disk is full
62	Input past end
67	Directory is full
68	Device unavailable
71	Disk not ready
75	Access denied
76	Path not found
83	Buffer too small
101	COMSPEC not found
102	Environment not found
103	ENVIRON string invalid
104	Out of string pool memory
105	Out of environment space
111	Pop-up already installed
112	PopUpHere already called
125	Overflow
126	Out of stack space
127	RETURN without GOSUB

FREEFILE

Unlike regular BASIC, the FREEFILE function as implemented in P.D.Q. will return -1 if there are no more file numbers available.

GET (binary file version)

When the P.D.Q. version of GET reads past the end of a file, it simply stops transferring data from the disk to the destination variable. For example, if you are twenty bytes from the end of a file and you ask GET to fill a TYPE variable that is thirty bytes long, GET leaves whatever

remains in the last ten bytes of the TYPE. Contrast that behavior with regular BASIC, which includes extra code to clear the remainder of the variable to null bytes in that special case. (In a binary access context, the end of the file is the physical length as reflected in the directory entry for that file.)

INKEY$

Although P.D.Q. fully supports BASIC's INKEY$ function, for the smallest code you should use the alternate PDQInkey routine as a substitute because it returns an integer value rather than a character string. Integer operations and comparisons are always faster than string operations, and this is discussed further in the tutorial section of this manual. Please note that INKEY$ may not be used within a P.D.Q. "simplified" TSR program. You must instead use the BIOSInkey function, which bypasses DOS and goes directly to the BIOS.

INPUT$

The INPUT$ function in P.D.Q. does not behave exactly the same as regular BASIC. If you ask for INPUT$(1) in a QuickBASIC program and the user presses an extended key (such as a function key or the up arrow key), the leading zero byte is returned and the second character is discarded. The INPUT$ statement in P.D.Q. returns the zero byte like QB, and a subsequent INPUT$ or (INKEY$) returns the extended key code. We would have gladly changed P.D.Q. to work like QuickBASIC, except that method is much less useful.

LPRINT

Like BASIC's LPRINT, the P.D.Q. implementation intercepts some control characters and acts on them in a special manner. To circumvent that behavior QuickBASIC and BASIC PDS let you use the ":BIN" option as shown below, however this is not supported by P.D.Q.

```
OPEN "LPT1:BIN" FOR OUTPUT AS #1
```

If you really need to send binary information to your printer, we recommend that you call the BIOS directly. This is illustrated in both the SETUP.BAS and DEFFN.BAS example programs.

LTRIM$

The P.D.Q. version of LTRIM$ removes CHR$(0) null bytes as well as blank spaces. This is a useful enhancement because the BASIC compiler

initializes fixed-length strings (and the string portion of TYPE variables) to null bytes rather than CHR$(32) blanks.

NAME

Unlike regular Microsoft BASIC, the P.D.Q. version of NAME does not support the DOS wild cards, "*" or "?".

OPEN

In a P.D.Q. program, file numbers used with OPEN must range from 1 through 15 inclusive. For each file that BASIC or P.D.Q. is to maintain, two integer words are required—one to remember the equivalent DOS file handle, and the other to remember the record length if the file was opened for random access. By limiting the range of possible file numbers to 15 rather than 255 like BASIC, nearly 500 bytes are saved from all programs.

OPEN in P.D.Q. further improves on BASIC because once a file has been opened, you may do nearly anything with it. That is, if a file has been opened for output, you may freely GET or INPUT from it as well as PRINT or PUT to it. Regular BASIC does not allow this flexibility, and will report a "Bad file mode" error if you attempt to do that. The only exception is that when a file has been opened for INPUT, DOS itself will refuse to write to it. OPEN for BINARY will fail on read-only files because it assumes both read and write access. Use ACCESS READ to avoid that if you need read-only permission.

PLAY

The PLAY statement as provided with P.D.Q. does not support background operation (using the "MB" command string). Therefore, using "MB" in a PLAY command will be ignored. The only other limitation is that dotted notes are not supported (a period following a note indicates that the length is to be extended an extra fifty percent). You may get around that problem simply by specifying "ML" to make the notes flow together, and then adding a second note with the appropriate duration. This is shown below.

```
PLAY "mn o3 l8 abcde."                  'the trailing dot is not
                                        '  supported
PLAY "mn o3 l8 abcd ml e el6 mn"        'this works in P.D.Q.
```

PRINT

Unlike QuickBASIC version 4.00 and later, the PRINT statement in P.D.Q. uses the DOS Write services for all printing. This allows program

output to be redirected, but without including additional code to also write directly to screen memory. Further, PRINT always uses the current screen color, which is generally white on black.

We recommend calling the PDQPrint routine when colors are required, or when screen output must be as fast as possible. Also see the PDQCPrint routine which honors the current COLOR settings, thereby eliminating one of the parameters required by PDQPrint. The _CPRINT.OBJ stub file can also be used to honor the current COLOR value, and it is ideal you are converting a large program and do not want to rewrite every PRINT statement. See the section entitled *Linking With Stub Files* for more information about using replacement BASIC statements.

PRINT

P.D.Q. adds an enhancement to the PRINT # statement whereby the reserved file number 255 sends its output to the DOS STDERR (standard error) device. This guarantees that the message will appear on the display screen, even when program output has been redirected by the user. Use PRINT #255 like this:

```
PRINT #255, "This message always goes to the screen"
```

PRINT USING

PRINT USING is not supported by P.D.Q. Instead, the FUsing function is provided. FUsing is more flexible than PRINT USING because it returns the formatted value in a string. You can then do whatever you want with that string. To mimic PRINT USING you will use code such as this:

```
PRINT FUsing$(STR$(Number), "####,.##")
```

RANDOMIZE

Under regular BASIC, using RANDOMIZE without an argument displays a prompt to input a seed value. This is not supported in P.D.Q. since you can easily add that yourself manually using INPUT.

RND

The P.D.Q. RND function always returns the next random number in sequence. BASIC's RND lets you use a 0 argument to return the same number as the last time RND was used, or a negative number to return a fixed value based on the number you specify.

RTRIM$

The P.D.Q. version of RTRIM$ removes CHR$(0) null bytes as well as blank spaces. This is a useful enhancement because the BASIC compiler initializes fixed-length strings (and the string portion of TYPE variables) to null bytes rather than CHR$(32) blanks.

RUN

Regular BASIC provides two variations on the RUN command, neither of which is supported by P.D.Q. To run the current program again you must clear any variables if that is important, and then use GOTO to jump to the start of the program. To run another program, use the supplied StuffBuf routine instead.

Notice that unlike BASIC's RUN, StuffBuf also allows you to run .COM and .BAT files, as well as .EXE programs.

SCREEN (statement form)

The SCREEN statement in P.D.Q. supports only a single argument to specify a video mode. Using the BASIC options for active and viewed display pages is not supported. See the MULTPAGE.BAS demonstration program for an example of accessing multiple text display pages.

SLEEP

In the P.D.Q. version of SLEEP, the optional seconds parameter is limited to 1820 (about 30 minutes), and is controlled by the BIOS timer count in low memory. The calculations are based on 18 ticks per second rather than 18.206481, therefore the elapsed time will be very slightly less than what is specified. Also, the P.D.Q. implementation of SLEEP flushes the keyboard buffer when it is done, thus removing any keystrokes that may have been pending.

SOUND

Although SOUND is supported by P.D.Q., the tones that are played are always handled as a foreground task. Where BASIC's SOUND returns immediately to your program but continues to run in the background, the P.D.Q. version does not return until the sound has completed.

Note that SOUND requires floating point support, which can add appreciably to the size of a P.D.Q. program. We recommend instead using

the PDQSound replacement routine. Internally the routines are identical, however the BC compiler uses floating point interrupts when it invokes SOUND.

SSEG

In Microsoft BASIC 7 the SSEG function returns a value of zero for null strings, but in P.D.Q. SSEG always returns the normal DGROUP data segment. This is because all strings in P.D.Q. are near data.

STOP

Using STOP in a P.D.Q. program is exactly the same as END, and BASIC's "Stop" message is not displayed.

STRING$

Both versions of STRING$() are supported by P.D.Q. However, using a numeric ASCII value results in slightly less code than a string argument. Therefore, the first example below is preferred.

```
STRING$(NumChars, CharNum)        'this adds less code
STRING$(NumChars, Char$)          'this adds slightly more code
```

SWAP

When swapping fixed-length or TYPE variables, both must be same length. With regular BASIC, if the variables are not the same length SWAP uses the shorter of the two. And if the destination is longer, the excess is cleared to blanks as if LSET had been used. We have assumed that you will not need this added capability, thereby reducing the amount of code that SWAP adds to your programs. Further, SWAP in P.D.Q. will not exchange a mix of regular and fixed-length strings. You must instead use a temporary string as an intermediary.

TIMER

Like VAL and SOUND, the P.D.Q. TIMER function requires floating point support, which is best avoided whenever possible. Therefore, we have provided a similar routine called PDQTimer which returns a long integer value that represents the BIOS timer count in low memory. See the TIMER.BAS demonstration program which shows how to easily simulate the resolution and convenience of BASIC's TIMER function.

VAL

Although VAL is fully supported by P.D.Q., using it adds floating point support routines to your programs. We have provided the PDQValI and PDQValL replacements for integer and long integer results respectively. Also, VAL does not recognize either the "&H" or "&O" prefixes to specify Hexadecimal or Octal notation. PDQValI and PDQValL do support using "&H", but not "&O". Since Hexadecimal notation implies integer or long integer values, it seemed pointless to add extra code to VAL to support that.

WIDTH

In P.D.Q. the WIDTH command is intended to specify the display screen only, and will not work with files or devices. Further, like regular BASIC, WIDTH may be used with color monitors only. However, WIDTH in P.D.Q. does support the optional second argument to set the number of rows for EGA and VGA displays:

```
WIDTH 40                    'color display only
WIDTH 80, 43                'EGA or VGA only
WIDTH ,50                   'VGA only
```

Please understand that unlike regular BASIC, P.D.Q. does not automatically issue a carriage return and line feed every eighty characters when printing to a device. Therefore, WIDTH #n, 255 is not needed to disable that "feature". You will not receive any errors if you use WIDTH for that purpose, but a few bytes of unnecessary code will be added to your program.

WRITE

Supporting the WRITE # statement in P.D.Q. would have required adding extra code to the PRINT # procedure, which we decided to avoid. Since WRITE # is so easy to emulate, you can create a short subprogram to do this as follows:

```
SUB WriteIt(FileNum%, Text$) STATIC
  PRINT #FileNum%, CHR$(34); Text$; CHR$(34)
END SUB
```

Then to write a string you would use **CALL WriteIt(1, Text$)**, and to write a number you could use **CALL WriteIt(1, STR$(Number))**.

Changes From Earlier Versions Of P.D.Q.

P.D.Q. version 3.0 provides a considerable number of enhancements over the earlier 2.*xx* versions. If you have just purchased P.D.Q. for the first time you can skip this section.

Most notable of the new features is support for floating point math and the corresponding increase in compatibility with regular Microsoft BASIC. During the life of P.D.Q. version 2, new statements and features were continually added. Therefore, depending on which 2.*xx* version you had previously, some of the new features listed here may already be familiar to you. The sections that follow describe all of the improvements and other changes we have made to P.D.Q. since the original 2.00 version.

Floating Point Math

P.D.Q. now supports floating point math and also the BASIC 7 PDS Currency data type. At this time only the basic four functions may be used (add, subtract, multiply, and divide), as well as SGN and ABS, INT and FIX, and the CINT, CLNG, CSNG, CDBL, and CCUR conversion functions. Of course, comparisons such as **IF X# > Y!** may be used, and MKC$, MKD$, MKS$, CVC, CVD, and CVS are also supported. Finally, the floating point commands TIMER, SOUND, RND, RND(argument), and RANDOMIZE(argument), are now supported as well.

There are a few very minor differences between BASIC's implementation of RND, RANDOMIZE, and SOUND, and what P.D.Q. does, and these are described in the section *Differences Between P.D.Q. and Microsoft BASIC*. Also, see the *Compiling and Linking* section for more information about including and avoiding floating point support in a P.D.Q. program.

Network Access and File Locking

P.D.Q. now supports network operation with the SHARED, LOCK READ, LOCK WRITE, and LOCK READ WRITE options of OPEN. You may also specify the ACCESS parameters with OPEN, such as ACCESS READ and ACCESS WRITE. Further, P.D.Q. now supports the LOCK and UNLOCK statements, to protect a file from access by other users. Unlike BASIC's LOCK and UNLOCK statements, P.D.Q. lets you lock any part of any file, regardless of the mode in which it was opened. (Regular BASIC doesn't allow you to lock a portion of a file that was opened for INPUT or OUTPUT.

The code to support network operations adds about 110 bytes to a program. Therefore, we now include the _NONET.OBJ stub file to *exclude* that

support from programs that do not need it. In interim 2.*xx* versions of P.D.Q. network support was added explicitly using the _OPENNET.OBJ stub file. To be compatible with regular BASIC, network capability is now the default.

All network errors are reported as "Bad file mode" (error 54). For example, if the host PC is running DOS 2 or if SHARE or network software are not installed you will receive this error. The only exception is "Bad file number", which you will receive if you attempt to lock or unlock a file that isn't open.

Arrays Containing More Than 65,535 Elements

In version 2.14 we introduced a new stub file named _DIM.OBJ that allowed arrays to exceed 65,535 total elements. Although BASIC does not allow more than 32,767 elements in a single dimension, arrays with more elements may be created by using more than one dimension. At a cost of about 100 added bytes of code, _DIM.OBJ allowed arrays to exceed 65,535 total elements. In this version of P.D.Q. that feature is now the default, and _DIM.OBJ is used to *remove* support for more than 65,535 elements.

EGA and VGA Graphics

We have added a set of BASIC subroutines for performing simple EGA and VGA graphics in P.D.Q. Although the routines are written in BASIC, they are very fast and also very small. See the section *Graphics And P.D.Q.* elsewhere in this manual for information on using these routines.

Changes to PRINT and STR$

In previous versions of P.D.Q. the STR$() function did not include a leading blank for positive numbers. PRINT calls upon STR$ to format numbers to ASCII digits, so it too was affected. Because having to strip the leading blank is so often a nuisance, we thought this would be a useful enhancement to the language.

Unfortunately, this caused problems for people with existing programs that counted on the blank, so we added the _STR$.OBJ stub file. Now that P.D.Q. supports floating point math, yet another stub file is needed for using STR$() with single and double precision values. As with OPEN and DIM, we are taking this opportunity to turn things around and do it properly.

By default, STR$() and PRINT now add a leading space with positive numbers. If you want the old way without the space, use _STR$.OBJ if you are using only integers (or long integers). If you are using single or double precision values, use _STR$FP.OBJ instead. If you are using both then you must use both stub files. Most of the example programs that in the past relied on STR$ not returning a blank have been changed; others now have a header comment showing that the program should be linked using _STR$.OBJ.

Note that if you use _STR$ or _STR$FP and also use the FUsing function, you must concatenate a leading blank like this:

```
Work$ = FUsing$(" " + STR$(Number), Mask$)
```

FUsing$ and the PUsing routine from our QuickPak Professional library *require* a leading bank with positive numbers.

Linking With /Stack:

Earlier versions of P.D.Q. did not allow reducing the size of the stack with the /Stack: LINK option. This has now been changed, and any reasonable stack size may be specified. See the section entitled *The Stack* in Section I, Appendix H, *Miscellaneous Considerations* for more information.

PopRequest Changes

The strategy for using PopRequest has been changed very slightly. In truth, PopRequest has not been changed, but we discovered a situation where reordering a program's statements can increase the likelihood of PopRequest being able to pop up successfully. Please see the section entitled *Using PopRequest* later in this manual for details about positioning the manual and simplified handlers in your programs.

MKD$, MKI$, MKL$, And MKS$

Previous versions of P.D.Q. did not allow concatenating these functions. You may now freely do so in any order or combination.

RESUME NEXT

We have removed the code that handled RESUME NEXT because it did not work properly, and was frankly too much trouble to make work.

Swapping TSR Programs

P.D.Q. simplified popup TSR programs may now be swapped out of memory to either a disk file or to expanded memory when they are not

active. See the description for the POPSWAP.OBJ stub file, and also the Swap2Disk and Swap2EMS routines.

New Routines and Programs

All of the new routines about to be described are detailed elsewhere in this owner's manual. The discussion simply lists each addition for owners of previous versions, and you should refer to the reference portion for complete details.

Because the added support for QuickBASIC compatibility in this version increases the size of your programs when compared to earlier versions, we have retained the original, limited versions of those statements that are affected. Some of these are provided as *stub files*, while others are contained in the SMALLDOS.LIB library file. This arrangement gives you complete control over which version of each statement is used, to achieve the smallest .EXE programs possible. See the section entitled *Linking With Stub Files* for more information.

The routines and programs that follow are listed chronologically in the order they were added. Therefore, if you previously had an interim 2.*xx* version, you can skip ahead looking for new items. New demonstration programs are described in detail; however, new routines and stub files are mentioned only briefly. The demonstration programs are described in the section *Files On The P.D.Q. Disk*. Please refer to the reference portion of this manual for more information on the new P.D.Q. extensions. All of the P.D.Q. stub files are described fully in the section *Linking With Stub Files*.

NOBEEP.BAS is a TSR utility that traps calls made by other programs to beep the speaker. It simply intercepts the BIOS interrupt, and ignores requests to print the CHR$(7) Ctrl-G beep character. You could also modify NOBEEP to replace the PC's beep with a less obnoxious tone, perhaps using PDQSound.

PDQZIP.BAS is an example program that shows how to read the header of a PKZIP file. PDQZIP was contributed by Crescent friend Dan Moore—simply run it specifying the name of an existing .ZIP file.

PDQShl and **PDQShr** are new functions that return an integer value with the bits shifted left or right a specified number of places. Shifting bits is one area where BASIC is particularly weak, and these functions can replace a substantial amount of code.

PDQMAKE.BAS lets you automate the building of multi-module programs using a batch file.

GetSeg is a function that lets your programs obtain the current DEF SEG setting.

ENVELOPE.BAS is a TSR envelope printing utility.

PopRequest is a major and important new feature that lets you perform nearly any DOS or BIOS service from within a manual interrupt handler.

APPOINT.BAS is another useful TSR example program, and it shows how to use the new PopRequest function (see above).

HercMode is a new subroutine that lets you enable and disable graphics on a Hercules display adapter.

PDQCompare is a new function that lets you compare any two blocks of memory (up to 64K) to see if they are identical. This type of routine is available in C, and we realized that it would also be useful with P.D.Q.

PDQCAP.BAS is a TSR screen capture utility that works in both text and graphics modes.

B_ONEXIT is now supported for use by add-on assembly language subroutines. B_ONEXIT is not something most BASIC programmers need to know about, but we have added it for our own use, and in response to requests from our customers. For your interest, B_ONEXIT lets an assembler routine tell BASIC, "When the program ends, execute the block of code at the following segment and address". This is a very powerful concept, because among other things, it lets a program that is trapping interrupts unhook those interrupts automatically, without putting the burden for remembering to do that on the programmer.

_INKEY$.OBJ is a new stub file that replaces INKEY$, using the BIOS rather than DOS to read the keyboard.

LPT2FILE.BAS is a new TSR demonstration program that captures all printed output and logs it to a disk file.

BufIn is a new function that performs buffered sequential file reading and serves as a LINE INPUT replacement. BufIn is much faster than the default P.D.Q. LINE INPUT routine, and also about four times faster than QuickBASIC's LINE INPUT. Added in this version is the ability to close a file in mid-read. Also, BufIn has been completely rewritten to store the

buffered file contents in far memory. BufIn is demonstrated in the DEMOBUF.BAS example program.

PDQSetWidth is a new routine that lets PDQPrint, PDQCPrint, and BIOSInput use a screen width other than 80 columns.

NoSnow lets you disable CGA *snow suppression* to achieve the fastest display speed when using that type of display adapter.

_DIM.OBJ is a stub file that limits dynamic numeric and TYPE arrays to a maximum of 64K elements. Although previous versions of P.D.Q. let you create huge arrays that use more than 64K of memory, the number of elements was not allowed to exceed 65,535. P.D.Q. now supports more than 64K elements by default, but at the expense of increasing the size of the DIM statement by about 100 bytes. This stub file removes that support for programs that don't need it, with a corresponding decrease in program size.

_STR$.OBJ and **_STR$FP.OBJ** are replacement stub files for BASIC's STR$ function that do not add a leading blank space when used with positive values. In previous versions of P.D.Q. _STR$.OBJ was used to *add* the blank space. By default, P.D.Q. now behaves the same as regular Microsoft BASIC.

_CPRINT.OBJ is a new stub file that lets PRINT honor the current color setting, without having to modify your program.

TEMPLATE.BAS is an empty "simplified" TSR program skeleton you can use as a starting point when writing your own TSR programs.

REBOOT.BAS shows how to easily reboot a PC.

WAITTIL.BAS is a simple utility that pauses a batch file until a specified time of day.

MULTPAGE.BAS is a new demonstration program that shows how to read and write to different text-mode display pages.

PDQCOPY.BAS is an intelligent COPY replacement utility that copies files only if they are newer or do not exist in the target directory or drive.

_SKIPEOF.OBJ is a new stub file that eliminates support for skipping past a CHR$(26) EOF (end of file) character when a file has been opened for APPEND.

MAKEPDQ.BAS is an automated program builder that lets you specify compile and link options and stub files from a menu.

HookInt0 is a new routine that lets you trap integer and long integer "Division by zero" errors without crashing.

KEY2FILE.BAS is a TSR keystroke logging utility that captures all keys as they are pressed, and saves them to a file.

ONMOUSE.BAS is an example program that shows how to take over hardware interrupts based on knowing the IRQ level.

MAKESTR.BAS creates custom STR*xxxxx*.OBJ files to control the amount of string memory that is allocated to a P.D.Q. program.

RedimAbsolute lets you assign any arbitrary segment to an existing dynamic array. For example, you could establish the color screen text display segment &HB800 as being a 2000-element integer array.

DEMOINT8.BAS shows how to perform periodic DOS services in a TSR program based on the system timer interrupt.

The **SOUND** statement is now supported using the normal BASIC syntax. Since SOUND merely calls PDQSound to do the real work, you can also use a negative value for the duration argument to tell SOUND to leave the speaker turned on.

MACRO.BAS is a TSR keyboard macro program that lets you assign nearly any number of characters to a single key.

IOCTL and **IOCTL$** are now fully supported, for applications that must communicate with device drivers.

FUsing$ is a numeric formatting function taken from our QuickPak Professional product, and it is similar to BASIC's PRINT USING feature.

MidCharS is a complement to the MidChar function, and it inserts a single character very quickly into a string.

_GET1BYT.OBJ is a new stub file that replaces the Get1Byte function with a version that returns unsigned values between 0 and 255.

DOSWATCH.BAS has been enhanced to detect prior installation, and to allow deinstallation by running a second copy with a /U command line switch.

FREEINTS.BAS is a new example program that shows how to determine which interrupt vectors are available.

BIOSInput2 is an enhanced version of BIOSInput that recognizes the Home, End, Ins, and Del keys, and returns the last key pressed. BIOSInput2 also accepts a row and column to specify the left edge of the field.

Files On The P.D.Q. Disk

Once you have installed P.D.Q., a number of different files will be present on your disk. Besides the actual P.D.Q. libraries, many example and utility programs are also provided. These will be described briefly below. Notice that additional programs may be included, which have been added after this manual was printed. Also notice that files with a .MAK extension are "make" files for the various P.D.Q. demonstration programs, to allow them to be run within the QuickBASIC editor.

README, if present, contains important information that is not covered in this printed manual.

HISTORY.DOC is a complete history of revisions and corrections since P.D.Q. version 2.05.

PDQ.LIB is the main linking library that contains all of the BASIC language routines supported by P.D.Q.

PDQ.QLB is a Quick Library containing most of the P.D.Q. extensions, and it is meant to assist you when developing programs in the QuickBASIC editor.

PDQ.RSP is the response file we use to create the PDQ.LIB library. This file and the other supplied response files are not likely to be useful to you unless you plan to extract files or recreate the PDQ.LIB library file from scratch. These files are provided primarily for informational purposes.

PDQFP.RSP is the response file we use to create an intermediate library named PDQFP.LIB. This library is incorporated into PDQ.LIB, and again, this file is provided only for completeness.

CURRENCY.RSP, like PDQFP.RSP above, is another intermediate response file, and it is used to create a library containing all of the Currency data routines for use with BASIC 7 PDS.

DONOTUSE.LIB contains BASIC 7-compatible versions of the P.D.Q. extensions, and is used by QUICK7.BAT. As its name implies, you should *not* attempt to link with this library. In previous versions of P.D.Q. this file was named PDQ7.LIB.

PDQ7.RSP is the response file we use to create the DONOTUSE.LIB library.

PDQ386.LIB contains 386-specific versions of the P.D.Q. routines that perform long integer multiplication, division, and comparisons.

PDQ386.RSP is the response file we use to create the PDQ386.LIB library.

SMALLDOS.LIB contains reduced-capability versions of several file-related BASIC language statements.

SMALLDOS.RSP is the response file used to create the SMALLDOS.LIB library file.

BASIC7.LIB contains BASIC 7-compatible versions of those P.D.Q. internal routines that had to be changed to support BASIC 7.

BASIC7.RSP is the response file we use to create the BASIC7.LIB library.

QUICKLIB.BAT is the batch file we use to create the PDQ.QLB Quick Library that contains the various P.D.Q. assembly language extensions.

QUICKLIB.RSP is the LINK.EXE response file containing the names of all the object files to be included in the Quick Library; it is used by QUICKLIB.BAT.

EXTRACT.RSP is the LIB.EXE response file containing the names of all the object files to be extracted from PDQ.LIB; it is used by QUICKLIB.BAT.

QUICK7.BAT is the batch file that creates the PDQ7.QLB Quick Library for use within the BASIC 7 PDS QBX.EXE environment.

QUICK7.RSP is the LINK.EXE response file used by QUICK7.BAT.

EXTRACT7.RSP is the LIB.EXE response file used by QUICK7.BAT.

PDQPRO.BAT and **PDQPRO7.BAT** are batch files that automatically build Quick Libraries combining the P.D.Q. extensions with routines from QuickPak Professional. You must also own QuickPak Professional to use

these batch files. LINK will probably issue a warning error that one or more routines with the same name were specified. You may ignore this error.

PDQPRO.RSP and **PDQPRO7.RSP** are response files used by PDQPRO.BAT and PDQPRO7.BAT respectively.

COMPILE.BAT is a sample batch file for compiling P.D.Q. BASIC programs. You may rename it to something shorter such as C.BAT if you prefer.

SMALLDOS.BAT is another sample batch file, and it contains the correct commands for linking with the SMALLDOS.LIB library.

EMBEDDED.DOC is a reprint of an article that appeared in Embedded Systems Programming magazine showing how to create ROMable applications using P.D.Q.

XREF.KEY is a copy of the file used by our XREF utility that identifies which keywords are valid or invalid to use with P.D.Q. If you purchased XREF prior to P.D.Q. this is a newer version of that file.

PDQDECL.BAS is an Include file that contains DECLARE statements for all of the P.D.Q. assembly language extensions. PDQDECL also contains the TYPE definition for the Registers variable used by CALL INTERRUPT and the P.D.Q. TSR extensions.

PDQSUBS.BAS contains BASIC versions of several P.D.Q. extensions for use within the QB and QBX environments. If you plan to develop programs using the QuickBASIC editor, you will load PDQ.QLB (or PDQ7.QLB with BASIC 7 PDS) and also load this file as a module. Do not, however, compile and link with this file when creating a final program. All of the routines contained in PDQSUBS.BAS are also in the main PDQ.LIB library.

APPOINT.BAS is a useful example program that demonstrates using the PopRequest function. APPOINT is a TSR appointment scheduler that lets you specify the time to pop up, and a message to display. For example, if you want to be reminded a few minutes before your appointment with Mr. Jones at 10:00 am, you would press Ctrl-A to pop up the display, enter 9:55:00, and then type a message that says "meet Mr. Jones in his office".

ASK.BAS is a simple program that allows keyboard input from within a batch file. When compiled and run, ASK accepts a command line message parameter, and pauses until a key is pressed. The ASCII value of that key

is then returned as a DOS ERRORLEVEL. ASK is demonstrated in the MENU.BAT batch file.

MENU.BAT shows how to create a simple DOS-only menu using the ASK.BAS program described above.

BIGPUT.BAS shows how to call the internal PUT and GET routines directly, to save and load entire arrays in a single operation. This technique can provide a dramatic improvement in file access time, and it works with conventional BASIC as well.

CDIR.BAS is a DOS directory changing utility. Simply highlight the directory you wish to change to (or navigate through as many directories as needed), and then press Escape to quit the program. CDIR illustrates several important concepts, including using DEF FN-style functions as procedures to reduce code size. Comments in the source code show how to implement CGA snow suppression, though at the expense of display speed. **CDIR.MAK** is a make file for running CDIR.BAS in the QuickBASIC environment, and it specifies PDQSUBS.BAS as a support module.

CLOCK.BAS is a TSR on-screen clock program, and it provides an example of intercepting hardware interrupts directly using P.D.Q. CLOCK also shows a clever technique that avoids adding the string handling routines to a program.

CMOS.BAS is a handy utility that saves or restores the current CMOS setup information in AT-class computers. Most PCs include a setup program to define the type of hard disk, memory, and so forth, which must be entered whenever the battery has been changed. Unfortunately, most people have no idea of the correct information to enter. Therefore, you should compile and run CMOS.BAS before your battery fails, saving the setup information to a *bootable* disk. Then, when the battery does need replacing, it is a simple matter to run CMOS again to restore the CMOS memory to its original state.

COLORS.BAS displays a chart of every possible color combination, and it is meant for identifying the color value to use when calling the PDQPrint "quick printing" routine.

DEFFN.BAS contains some useful DEF FN-style functions that may be added to your programs.

DEMOBUF.BAS is a demonstration program showing the P.D.Q. BufIn$ function in context. Comments in the program header show a novel way

of declaring string functions, to avoid the unnecessary string copying code that BASIC usually adds each time a string function is invoked. **DEMOBUF.MAK** is a .MAK file for use within the QuickBASIC editing environment.

DEMOINT8.BAS shows how to tap into the timer interrupt, and perform DOS services periodically. Many people have called us asking how to do this, mostly for developing network mail systems. This short program simply checks if a file is present every ten seconds, and if so sounds an alarm. You can easily modify the program skeleton to do whatever you need. For example, you can open files, read them, and so forth.

DEMOTSR.BAS shows all of the possible methods for detecting if a P.D.Q. TSR program is already installed, and deinstalling it if so. Although several other demos also show how to do this, DEMOTSR provides the most thorough example of these techniques.

DIALTSR.BAS is a TSR phone dialing utility. Simply compile and run it using the syntax shown in the file header comments, then press Ctrl-D whenever you want to dial a number.

DISKUSED.BAS is similar to the Norton Utility's FS.COM (File Size) program. It expects a file specification such as *.* or C:\QB*.BAS, and reports the total size of the files, the space actually occupied on the disk, and the percent wasted due to the way DOS organizes disk clusters. If the file specification is omitted, DISKUSED assumes *.* using the current drive and directory. DISKUSED also accepts a drive letter as an optional second argument, and uses that drive's cluster makeup to determine if the specified files will fit on the disk currently in that drive.

DOSWATCH.BAS is a TSR program that provides a "window" into DOS as it works. Whenever a program performs a DOS service, information about that service is displayed at the top of the screen. In many cases, additional information is also displayed, such as file and directory names, handles being read from and written to, and so forth. DOSWATCH is described in detail in the section entitled *The DOSWATCH Example Program*.

ENVEDIT.BAS is a DOS environment editor. Modifying the master DOS environment from a program is usually not possible, however P.D.Q. includes extensions that let you do this.

ENVELOPE.BAS is a TSR example program that is also very useful in its own right. Envelope lets you capture a name and address from the underlying screen (perhaps while a record in a database program is

displayed), and then send that to either a printer or a disk file. The header comments show how to compile and link it to take as little RAM as possible. Once Envelope has been run, press Ctrl-E to pop it up. Next, position the cursor at the upper left corner of the name and address on your screen and press Enter. Envelope then makes its best guess as to where the lower right corner is, and you can use the cursor arrow keys to mark the exact boundary. When you press Enter again the envelope is printed.

Envelope accepts command line arguments to specify the output device or file, form length, and top and left margins. These may be given in any order, and all but the device/file name are optional and preceded with a code. The complete syntax is as follows:

```
ENVELOPE outfile [/fl n] [/tm n] [/lm n]
```

outfile is the name of a file or device (LPT1, LPT2 or PRN) to send the captured address to (OPEN FOR APPEND is used to combine multiple envelopes into a single file). Notice that a trailing colon is *not* used for printer device names with P.D.Q.

/fl n (Form Length) specifies that the envelope (form) is "n" lines long. That is, extra blanks are printed to fill that many lines and to eject the envelope from the printer.

/tm n (Top Margin) specifies that "n" lines will be printed before the address.

/lm n (Left Margin) specifies that the address will be indented "n" spaces.

The following specifies a standard size (#10) envelope on printer port # 1, and also shows the default values Envelope uses for each parameter when they are omitted:

```
ENVELOPE lpt1 /fl 25 /tm 10 /lm 40
```

If you reinvoke the program from DOS with no parameters, it will deinstall itself. If you reinvoke the program with any parameters, the new parameters replace the resident program's current settings. Envelope therefore shows how to modify the resident copy of a TSR from a subsequent invocation of the same program.

EXE2COM.BAS is a clever program that reduces the size of your P.D.Q. programs even further. In truth, EXE2COM doesn't really create a .COM file. Rather, it creates a copy of the specified .EXE file, but with a reduced header size and .COM extension. Every .EXE file includes a header which is at least 512 bytes. This header contains relocation information

that DOS uses when it loads the program into memory. EXE2COM reduces the size of this header to eliminate the unused zero bytes, and names the output to have a .COM extension (great for fooling C programmers). Use it like this, without a file extension:

```
EXE2COM program
```

IMPORTANT:

When you type a program's name and then press Enter, DOS looks first for a file with a .COM extension, then .EXE, and finally .BAT. Therefore, if you process a program using EXE2COM but then make a new .EXE file and try to run it, DOS will execute the older .COM version. We suggest that you run EXE2COM as the last step when a project is completed, or be sure to delete interim .COM versions during development.

FILTER.BAS is a simple DOS filter that shows how P.D.Q. supports redirection. It accepts input from STDIN (DOS standard input), capitalizes it and also strips any high bits, and then sends it through STDOUT (DOS standard output). As provided, FILTER is not intended as a useful utility. However, it shows how such DOS filters may be written using P.D.Q. If you enter FILTER at the DOS command line, it will merely echo what you type, only capitalized. You must press Ctrl-Z to end the program. Of course, DOS filters are really meant to be used with redirected input and/or output. For example, to make a capitalized copy of your AUTOEXEC.BAT file you would enter:

```
FILTER < \AUTOEXEC.BAT > AUTOEXEC.CAP
```

FINDTEXT.BAS is a copy of the Norton Utility's TS.COM (Text Search) program. It accepts a file specification to indicate which files to search, and then prompts you for the text to search for. FINDTEXT examines every file that matches the file specification, and then displays the text in context (showing 20 characters before and after the search string). Besides being half the size of the Norton version, we have also added a handy feature that lets you skip to the next file. Note that FINDTEXT is ideal for searching through the .BAS files that come with P.D.Q., to find examples of a routine or keyword.

FREEINTS.BAS is a utility program that reports all of the interrupt vectors that are not active, and are thus available for use in a TSR. This program is described in the section *Accessing A Resident Program.*

HIGUY.BAS is an example of the simplest keyboard interrupt handler possible using P.D.Q. Be aware that HIGUY uses manual interrupt

handling, rather than the "simplified" pop-up method we recommend for most TSR applications.

KEY2FILE.BAS is our answer to requests from many customers asking how to write a TSR that captures keystrokes to a log file. Most folks attempt to capture the keyboard hardware Interrupt 9, which deals with scan codes rather than ASCII key values. KEY2FILE instead takes over Interrupt &H16, which is more direct. Simply compile and link it as shown in the header comments, and all keystrokes will be saved to a file named KEY2FILE.DAT.

Note that extended keys such as F1 and Alt-C are stored in the file in the same way that BASIC expects them. For example, F1 is written as CHR$(0) + CHR$(59). However, pressing Enter does not write a corresponding CHR$(10) line feed. Therefore, if you use the DOS TYPE command the file will not appear to be correct, even though it is. To verify a keystroke file's contents you will need a file viewer that can accommodate lines that end with a CHR$(13) only. You could optionally load the file into DEBUG and use a sequence of D (dump) commands to view it:

```
DEBUG KEY2FILE.DAT<Enter>
D 100<Enter>
D<Enter>
  ...
  ...
```

KEY2FILE may be deinstalled by pressing Ctrl-Alt-U while at the DOS command line. In fact, it must be deinstalled before the output file may be viewed or used, since the file is kept open while KEY2FILE is active.

LPT2FILE.BAS is a TSR utility that captures printed output and routes it to a file. By default, LPT1 is intercepted and the output file is called LPT2FILE.DAT, however these may easily be changed. See the program's source code and the KEY2FILE comments above for more information.

MACRO.BAS is a program that shows how to create a keyboard macro program. Although we include PDQKEY.BAS for this, that program is more complicated and does more. When all that is needed is keyboard macros, the additional interrupt handling in PDQKEY makes the program larger than is truly needed.

MAKEPDQ.BAS is a clever utility program written by Pierre Connolly that lets you easily specify all aspects of compiling and linking with P.D.Q. Instead of manually entering commands and stub files and alternate libraries, or writing batch files to do that for each program, you simply select them from a menu. Pierre has written his own .DOC file, which explains how to use MAKEPDQ in detail. There are some true program-

ming gems in MAKEPDQ, including a routine that loads and executes another program without using SHELL, and also retrieves the DOS error level of that program. **MAKEPDQ.CNF** is the configuration file used by MAKEPDQ, and you should edit this file to reflect the appropriate path names for your PC. **MAKEPDQ.DOC** describes how to use MAKEPDQ.BAS and also how to edit the MAKEPDQ.CNF configuration file.

MAKESTR.BAS is a utility program that directly generates a custom string pool object file. Simply compile it as shown in the program header comments, and then run it. At the prompt tell it the number of bytes of string space you need, and it will create a new STR#####.OBJ file. You may also specify the string pool size directly as a command line argument. String pool object files may be created having memory sizes ranging from 10 to 63,000 bytes. For information about why this program is useful see the manual section that describes stub files.

MAP.BAS is a memory map utility, and it shows all of the programs that are currently in memory, as well as any interrupt vectors they have intercepted. Unlike the popular public domain SMAP program which was written in C, MAP displays all of the programs that are loaded. Even if you have shelled from one program, run another, and shelled from that to run MAP! Our MAP is also much smaller than the C version. MAP.MAK is needed if you plan to run MAP.BAS in the QuickBASIC editor.

MULTPAGE.BAS is an example that shows how to simulate BASIC's **SCREEN , , apage, vpage** capability to select active and visual pages, because this features is not supported directly by P.D.Q. Comments in the source show what it is doing and how to adapt these techniques to your own programs.

NOBEEP.BAS is a TSR program that merely disables the Ctrl-G beep from sounding. Some programs are obnoxious and beep at every little error; we wrote NOBEEP to get around a flaky BIOS that sustained every beep for two full seconds.

NOBOOT.BAS shows how to trap the Ctrl-Alt-Del keys to prevent someone from rebooting the PC.

NUMOFF.BAS simply turns off the NumLock status. Most people want such a utility, because current versions of DOS turn NumLock on when they boot. This type of utility would normally be written in assembly language, and NUMOFF shows how adept P.D.Q. is at creating very small programs.

ONKEY.BAS shows how to simulate BASIC's ON KEY statement using direct interrupt handling.

ONMOUSE.BAS shows how to trap hardware interrupts and deal manually with the PIC (Programmable Interrupt Controller). ONMOUSE is intended for use with a serial mouse, and it is definitely not for the faint of heart.

ONTIMER.BAS, like ONKEY.BAS above, provides an example of imitating BASIC's ON TIMER statement.

PDQBLANK.BAS is a TSR screen blanking utility. It accepts a command line argument to tell it how many seconds of keyboard inactivity to wait before turning off the display. You may run PDQBLANK with a /U option to uninstall it. You may also run it multiple times to specify a different wait time. Having one copy of a program modify variables within another is a fairly tricky concept, and this too is shown in PDQBLANK.

PDQBLNK2.BAS is a stripped-down version of PDQBLANK that does not take command line arguments, but uses less memory when it is loaded.

PDQCALC.BAS is a basic TSR 4-function memory calculator. Run it once from the DOS command line, and then press Alt-C to pop it up; press Escape when you are finished to return to the underlying application. PDQCALC saves and restores the original screen automatically, and also remembers its current display and memory values between pop-ups. Keys supported by PDQCALC are as follows:

- Digits 1-9 for entering numbers, and the decimal point.
- The four math operator keys and Equals; +, -, *, /, and =.
- The up and down arrows, either of which exchanges the current display contents with memory.
- The "A" key (Clear All) which clears both the display and memory values.
- The "H" key which changes the display to Hexadecimal notation until another key is pressed.
- The "M" key (Memory Plus) which adds the current display value to memory.
- The "N" (Negate) key which changes the sign of the display value. This key is not shown on the calculator control panel.

- The "R" (Recall Memory) key which copies the value in memory into the active display.

PDQCALC.BAS also provides an example of using long integers as dollar amounts, and then multiplying and dividing them.

PDQCAP.BAS is a TSR screen capture program that works in every BASIC-supported text and graphics video mode, including CGA, EGA, VGA, and Hercules. The companion program SCRNSHOW.BAS is a slide show which displays the saved screens in sequence. See the header comments in these files for more details about how they are used. PDQCAP and SCRNSHOW were written by Nash Bly, and you can see by examining the source listing what an ambitious project this was!

PDQCOPY.BAS is a utility program that works just like the DOS COPY command, except it copies only files that are newer than the target or if the target does not exist. This will save you a lot of time when there are many files to update or back up, and only some of them have changed. The syntax is almost identical to the DOS COPY command, except both source and destination arguments are required:

```
PDQCOPY filespec destination
```

Filespec can be a complete file name, or a file specification such as *.* or *.BAS or E:\SOMEDIR*.*. Destination must be a drive and/or path name such as A: or B:\ or \NEWDIR. After copying, the new file is given the same date and time as the original. Note that periods are displayed as files are skipped, so you can see that something is happening.

PDQKEY.BAS is a TSR keyboard macro program that also expands the keyboard buffer to more than 15 characters. Of course, it is nowhere near as sophisticated as, say, Borland's SuperKey which allows the user to modify macros on the fly. But PDQKEY does provide a solid foundation for designing programs of this type. Also see MACRO.BAS which does less but is much smaller.

PDQMAKE.BAS is a "poor man's" MAKE program you are bound to find useful. Unlike MAKE and NMAKE from Microsoft that require an elaborate script file, PDQMAKE can be run easily from a batch file. The primary purpose of a MAKE program is to automate the building of an application, whenever one or more dependent pieces have been modified. If you are working on a single-module program, then compiling and linking the file each time it is modified is reasonable. But when many modules and libraries are involved in an entire application, it is a waste of time to have your batch files recompile every file each time a small change is made. Thus, PDQMAKE lets a batch file decide which files actually need

to be recompiled. The results of PDQMAKE's date and time comparisons are returned to the calling batch file through the DOS ERRORLEVEL function. The header comments in PDQMAKE.BAS shows how it is used, and also present a simple batch file as an example.

PDQPARSE.BAS is an example showing how to use the PDQParse assembler routine.

PDQZIP.BAS is an example program that shows how to read .ZIP file header information.

PLAY.BAS plays some pretty funky (not in the good sense) tunes using the PDQRand function.

POPREQ1.BAS and **POPREQ2.BAS** are example programs that accompany the discussion in the section *Using PopRequest*.

POPUPFP.BAS shows how and where to enable and disable floating point interrupts so floating point operations can be used in a P.D.Q. TSR program.

RANDOM.BAS shows how to simulate reading and writing random access files when using the SMALLDOS.LIB library.

READFILE.BAS reads a text file, and also demonstrates using the P.D.Q. critical error trapping routines. (Critical errors are those that DOS reports when you try to access a disk with the door open.) **READFILE.MAK** is its accompanying .MAK file.

REBOOT.BAS is a simple example program that shows how to reboot a PC. You can use it as is from within a batch file, or add the code to programs of your own. REBOOT can be particularly handy when run from a batch, since the batch file could also rename your AUTOEXEC.BAT or CONFIG.SYS files letting you boot using different configurations.

REDIMABS.BAS demonstrates the P.D.Q. RedimAbsolute routine.

SCRNCAP.BAS is a complete text-mode TSR screen capture utility that lets you capture text screen images from within any application. The screens are saved to disk as normal BASIC BLOAD files, so they may be loaded into your own programs if you wish. We developed SCRNCAP to accompany our QuickScreen screen designer, however it is also quite useful in its own right. SCRNCAP recognizes the 25-, 43-, and 50-line text modes automatically, and saves the appropriate number of bytes.

SCRNCAP may be deinstalled by running it again with a /U command line switch.

SCRNSHOW.BAS is a slide show program that is meant to be used with the PDQCAP.BAS utility described earlier.

SETUP.BAS lets you send control codes to an Epson or compatible printer, to enable compressed, enhanced, or tiny printing. SETUP also shows how to print characters by calling the BIOS routines directly.

SETUPTSR.BAS is similar to SETUP.BAS (above), but it has been designed as a TSR allowing it to be used even when another program is running.

SHELL.BAS is a TSR utility that provides a shell-to-DOS feature from within any application. Be sure to read the header comments carefully before using SHELL, and before adding the techniques it shows to TSR programs of your own.

SMALLDOS.BAS provides a simple example of sequential file reading when using the SMALLDOS.LIB library.

SPEEDUP.BAS is a keyboard speed-up utility for use with AT-class computers. Unlike the original PC and XT models, AT computers contain additional logic in their keyboard hardware to vary the initial delay and key repeat rate.

SYSINFO.BAS is yet another Norton Utilities clone, and it displays all of the pertinent information about the PC it is running on. **SYSINFO.MAK** lets you run it within the QuickBASIC editor. SysInfo was written by Crescent friend Jonathan Zuck.

TEMPLATE.BAS is a "template" program that shows the minimum steps necessary to create a simplified TSR program. It is intended to be used as a starting point—or program skeleton—for writing simplified pop-up TSR programs of your own. That is, you load the source file into QuickBASIC, and then add your own code to it. All of the pieces needed to set up a TSR and test for prior installation are already in place. Comments throughout give step-by-step explanations of the program's operation, and also show how to test a TSR program within the QB and QBX editor.

TIMER.BAS shows how to simulate BASIC's TIMER function using the PDQTimer replacement routine. **TIMER.MAK** is the associated .MAK file.

TRAP3.BAS gives an example of trapping three separate interrupts manually, and shows how flags are needed to arbitrate different portions of the program to prevent runtime reentrance. Because hardware interrupts can occur at any time, it is up to you to ensure that more than one interrupt handler does not try to use the same BASIC or P.D.Q. routines at the same time. The program's header comments explain the principles of operation.

WAITTIL.BAS is a silly little utility that is also quite useful. It accepts a command line argument specifying a time of day, and pauses the PC until then. It is meant to be run from a batch file, perhaps to delay a modem transmission until late at night when the phone rates are lower or to start a network backup when nobody will be inconvenienced.

The syntax is **WAITTIL Time**, where Time is in 24-hour military format with the seconds being optional. That is, **WAITTIL 01:30** pauses until 1:30 am, and **WAITTIL 14:29:02** pauses until 2:29:02 in the afternoon. Note that you must provide either five or eight characters. WAITTIL was inspired by a utility of the same name in the January 1991 issue of PC Computing.

WHEREIS.BAS searches a hard disk for all files that match a given file specification. Note that WHEREIS relies on several routines from our QuickPak Professional. Therefore, you must have QuickPak Professional to compile and link it.

WMTELL.BAS is a transcription of the entire William Tell Overture. This one crossed our path by way of a friend, though we'd sure like to thank whomever was responsible for creating it!

DEMOEGA.BAS shows the P.D.Q. EGA/VGA graphing routines in context. **DEMOEGA.MAK** is the necessary .MAK file that loads the actual routines into the QuickBASIC editor.

EGABOX.BAS contains a subroutine of the same name that draws boxes.

EGADOT.BAS contains a dot-plotting routine.

EGAELIPS.BAS contains the EGAEllipse routine that draws circles and ellipses.

EGALINE.BAS contains the EGALine line-drawing graphics routine.

EGAPRINT.BAS contains a subroutine that lets you print in the SCREEN 9 or SCREEN 12 graphics mode, and also specify a display color.

MAKEEGA.BAT is a batch file that builds DEMOEGA.EXE automatically.

STR00256.OBJ through **STR49152.OBJ** are replacement string pool stub files which let you control how much string memory will be available to your P.D.Q. applications. Please see the section entitled *Linking With Stub Files* for a complete description of these alternate data files.

POPSWAP.OBJ is a special stub file that is needed to allow popup TSR programs to be swapped to disk or expanded memory.

DOSVRQLB.OBJ is a modified version of the P.D.Q. DOSVer extension, but written for use within the QB and QBX environments. Note that DOSVRQLB is not meant to be used by you. Rather, it is accessed by the QUICKLIB.BAT and QUICK7.BAT batch files.

FLUSHQLB.OBJ, like DOSVRQLB discussed above, is meant to be used by the Quick Library batch files. In truth, FLUSHQLB is an "empty" file that doesn't actually do anything. But it is needed since the Flush extension can accept a varying number of parameters.

VALQLB.OBJ, like DOSVRQLB.OBJ and FLUSHQLB.OBJ above, is a variant of the PDQVAL function for inclusion in a Quick Library only.

_xxxxxxx.OBJ The remaining object files whose names begin with a leading underscore are stub files containing reduced-capability versions of several language statements and P.D.Q. extensions. These are all described in the section entitled *Linking With Stub Files*.

There are also a number of files that are intended for use by assembly language programmers, and these are described separately in the section of the manual that describes using P.D.Q. as an assembler toolbox.

Chapter 2: Compiling And Linking

■ ■ ■


Compiling &
Linking

Overview

All P.D.Q. programs will be compiled using the BC.EXE compiler that comes with QuickBASIC version 4.*xx*, BASIC 6.*xx*, or BASIC PDS version 7.*xx*. P.D.Q. supports only stand-alone programs that have been compiled using the /o option. Therefore, you must use /o when you are compiling your programs:

```
BC program /o ;
```

When linking, you must tell LINK not to use the default BCOM library by adding the /nod (No Default library) option. Here's the minimum LINK command line you will use:

```
LINK program /nod , , nul, pdq
```

The /nod option tells LINK to use only the library stated on the command line, which in this case is PDQ.LIB. Notice the use of the DOS NUL device in the LINK command line. If omitted, LINK will create a mostly useless .MAP file that merely clutters up your disk.

You could also rename the PDQ.LIB file to BCOM45.LIB or whatever is appropriate for your version of BASIC, though we recommend against doing that. However, this would be necessary if you intend to compile from within the QuickBASIC editing environment, rather than from the DOS command line. We have provided a sample COMPILE.BAT batch file that will compile and link your programs using the correct options.

BC.EXE options supported by P.D.Q. are /a (Assembly listing), /ah (Huge Array), /d (Debug), /e (Error), /mfb, /s (String), and /zi (CodeView). If you have BASIC 7 PDS you may also use the options /g2 (Generate 80286 code), and /ot (Optimize procedure calls).

Supported LINK options are /co (CodeView), /ex (EXE pack), /far (Far call translate), /nod, /noe (No External lookup), /packc (Pack Code), and /stack. Many of these compile and link options will be discussed later.

P.D.Q. is provided as several libraries: PDQ.LIB, PDQ386.LIB, BASIC7.LIB, and SMALLDOS.LIB. PDQ.LIB is the main P.D.Q. library intended for use with any IBM PC or compatible; PDQ386.LIB is slightly more efficient for some BASIC statements, but it requires an 80386- or 80486-equipped PC to run; BASIC7.LIB is required when using BASIC 7 PDS; and, SMALLDOS.LIB contains reduced-capability versions of BASIC's file commands.

PDQ386.LIB takes advantage of the expanded instruction set available on machines so equipped, and it is useful when multiplying, dividing, and

comparing long integer values. It is also useful with the MOD function when either operand is a long integer. However, you should not use it unless you are absolutely certain that the target computer is equipped with an 80386 or newer CPU.

When you use any of the three alternate libraries you must also use PDQ.LIB, because the alternate libraries replace only selected statements. Therefore, these libraries must be listed on the LINK command line *before* PDQ.LIB. For example, to use PDQ386.LIB you will link as follows:

```
LINK program /nod/noe , , nul, pdq386 pdq ;
```

Because the 386 version contains only those BASIC language routines that are different, it must be listed first. Thus, LINK will find the routines contained therein, and not continue looking. The remaining routines will be taken from the regular PDQ.LIB library file. The /noe switch is needed to prevent LINK from being confused by the presence of the same routine names in more than one library file.

The two other libraries—SMALLDOS.LIB and BASIC7.LIB—are also meant to be used in conjunction with the main PDQ.LIB library. SMALLDOS.LIB is described separately in this manual, because the differences affect many routines.

BASIC7.LIB is needed if you are creating programs using that version of Microsoft BASIC. In either case, the replacement libraries must be listed before PDQ.LIB on the LINK command line. If you intend to use both SMALLDOS.LIB and BASIC7.LIB at the same time, the order is not important as long as you use /noe and PDQ.LIB is listed last:

```
LINK program /nod/noe , , nul, smalldos basic7 pdq ;
```

Finally, you can also link P.D.Q. programs with third-party assembly language libraries such as QuickPak Professional, by simply listing their names at the end of the LINK command line:

```
LINK program /nod/noe , , nul , pdq pro ;
```

When linking with routines in QuickPak Professional, be sure to use PRO.LIB and not PRO7.LIB. Although P.D.Q. supports compiling with BASIC 7 PDS, it does not work with programs that are compiled using the /fs (far strings) option.

Other LINK Options

The QuickBASIC environment by default uses the /ex LINK switch, which creates a "packed" .EXE file. This is similar to the various ARC programs, that compress a file to take up less disk space. Very small programs you write with P.D.Q. may in fact be larger when linked with

/ex, because the unpacking code that LINK adds to the file could be larger than the saving it affords!

One exception, though, is when you have static arrays. Arrays are stored as a contiguous group of zero bytes in the file, and thus may be packed quite effectively. We recommend that you try linking your programs both ways, and use whichever method produces the smaller file. (With very tiny programs LINK issues an error message that the file is unsuitable for use with /ex.) To reduce the program's size even further you should also use the supplied EXE2COM utility. EXE2COM is described in the section *Files On The P.D.Q. Disk.*

While we're on the subject of linking, there may be situations where the number of code and data segments being linked to your program exceeds LINK's default capacity of 128. This would happen if more than 120 or so different BASIC statements are used, or when the combination of BASIC statements plus external P.D.Q. routines exceeds that amount. If LINK issues a "Too many segments" error, you should add the /seg:*nnn* switch, where *nnn* is the number of segments needed. (Internally, LINK dimensions an array to hold all of the segment names. If more than 128 are required, then the /seg: option tells LINK to reserve more memory for the names.) The sample COMPILE.BAT file includes /seg:250 to avoid that error message.

Another LINK switch you may find useful is /stack, which lets you change the size of the system stack when a program is linked. By default, P.D.Q. provides a stack that is only 1,024 bytes (1K) in size. QuickBASIC lets you use the CLEAR command to increase the stack size when necessary, and BASIC 7 PDS provides the more useful STACK statement. To eliminate the added code overhead required to support these statements, there is no provision in P.D.Q. for changing the stack size at runtime.

Using /stack tells LINK to alter the amount of memory that is allocated for a system stack when your .EXE program is created. The general syntax is as follows:

```
LINK /stack:nnn program
```

Here, *nnn* is the number of bytes you want allocated for a stack. You may specify any reasonable number; however, bear in mind that the stack shares the same near memory segment that is used by strings and other variables. Therefore, increasing the size of the stack makes that much less available for near DGROUP data. Be sure to specify a stack size that is a multiple of 2. That is, **/stack:2000** is okay, but **/stack:2001** is not.

If you specify a stack value that is too high you will receive the LINK error message "Stack plus data exceeds 64k". See the section entitled *The Stack* in Section I, Appendix H, *Miscellaneous Considerations* which discusses selecting an appropriate stack size. Also see the section *String Memory Considerations* for additional information.

Finally, there are two additional LINK options that you should be aware of. The /far and /packc options are intended to be used together, and they will reduce the size of your final .EXE program and increase its speed in most cases. However, Microsoft places the following warning in their Macro Assembler 5.1 owner's manual (Microsoft CodeView and Utilities, page 277; MASM 6.0 uses similar wording on page 349):

> *"There is a small risk involved with the /far option; the linker may mistakenly translate a byte in a code segment that happens to have the far-call opcode (&H9A). If a program linked with /far inexplicably fails, then you may want to try linking with this option off. However, object modules produced by Microsoft high-level languages should be safe from this problem, because relatively little immediate data is stored in code segments."*

In our experience, using /far has never caused a problem. Further, our contact at Microsoft tells us that the problem described in the MASM manual could never happen. However, an error is theoretically possible if you are using ON GOTO, and the target address of one of the labels happens to have an &H9A as one of the bytes. Our suggestion is that you try it to see how much improvement you realize. We also suggest that you carefully test all of your program's functions if you are using ON GOTO or ON GOSUB.

Creating A Quick Library

We have provided two Quick Libraries for use when developing P.D.Q. programs in the QuickBASIC environment. PDQ.QLB is meant for use with QuickBASIC 4.0 and 4.5 and BASIC 6, and PDQ7.QLB is for use with QBX.EXE that comes with BASIC 7.*xx* versions. If you need to combine other routines with these libraries, you will have to recreate them from scratch.

There are several batch and response files on the P.D.Q. disk that we use to create these Quick Libraries. To create a custom version you must edit these files, and run the batch files.

QUICKLIB.BAT extracts all of the necessary .OBJ files from the PDQ.LIB file, and then creates PDQ.QLB from those files. The two response files are EXTRACT.RSP and QUICKLIB.RSP, and QUICK-

LIB.BAT uses these to create the Quick Library. Notice that the name BQLB45.LIB is specified within the QUICKLIB.RSP file. If you are using a version of QuickBASIC other than 4.5 you may either edit the file, or wait for LINK to prompt you for the correct name.

QUICK7.BAT is nearly identical to QUICKLIB.BAT, except it creates a Quick Library named PDQ7.QLB which is meant for use with QBX. QUICK7.BAT relies on two other response files—EXTRACT7.RSP and QUICK7.RSP—to extract and then combine the various .OBJ files into a Quick Library. Similar to QUICKLIB.RSP, QUICK7.RSP uses the name QBXQLB.LIB as the default for obtaining the "Quick Library support" routines necessary for use in the QBX environment.

Because so many of our customers own both P.D.Q. and QuickPak Professional, we have provided a pair of batch files that will combine all of the P.D.Q. extensions with the QuickPak libraries. PDQPRO.BAT and PDQPRO7.BAS are meant for use with QuickBASIC and QBX respectively. Note that you may need to edit these files slightly, to specify the correct drive and path names for your particular system. Also note that the MAKEQLB utility from QuickPak Professional can create custom Quick Libraries automatically. Please see the description for that program in the QuickPak Professional manual.

Linking With Stub Files

A stub file is an object module that contains an alternate version of a BASIC language statement, P.D.Q. extension, or data item. The primary purpose of a stub file is to let you replace one or more statements or extensions with others having a reduced capability and hence smaller code. For example, the KILL command accepts wild cards (* and ?) to let you specify more than one file to be deleted. The _KILL.OBJ stub file provided with P.D.Q. does not accept wild cards, and as a result is approximately one-third the size of the default KILL routine in the PDQ.LIB library. In other cases, a stub file may provide additional capabilities such as _CPRINT.OBJ.

Linking with stub files is quite easy—you simply specify the name of the replacement .OBJ file on the LINK command line after your main BASIC program. The example below links a BASIC program with the reduced-capability LOCATE statement:

```
LINK /noe/nod program _locate , , nul, pdq ;
```

Likewise, to eliminate support for READ you would link with _NOREAD.OBJ like this:

```
LINK /noe/nod program _noread , , nul, pdq ;
```

Note that more than one stub file may be used at a time. This next example uses both of the stub files just mentioned, as well as the less capable KILL replacement:

```
LINK /noe/nod program _noread _locate _kill , , nul, pdq ;
```

As with the alternate P.D.Q. libraries, stub files are listed on the link command line before the libraries. The routines they contain are therefore found before others with the same name in the libraries listed at the end of the line. As you can see, /noe is needed with stub files too, for the same reason.

Because many of the BASIC and P.D.Q. subroutines call each other it is not possible to create your own stub library comprised of selected stub object modules.

String Pool Stub Files

Besides the reduced-capability language statements we provide as stub files, there are several other important stub files you should be aware of. The object files whose names begin with *STR* let you control the amount of string memory that is available to your programs.

Several of the P.D.Q. routines and functions access a data area we call the *string pool*. By default, the string pool is comprised of 32,768 bytes of memory; however, several alternate object files are provided which contain other sizes. For example, STR04096.OBJ is a replacement string pool with a size of only 4096 bytes. Therefore, to create a program that uses only 4K of string memory, you would link it with your program like this:

```
LINK /noe/nod program str04096 , , nul, pdq ;
```

For most programs, there is no harm in using the default string pool size of 32K, since all PCs have at least that much memory available. Because the actual string data area is not added to your .EXE program, using a smaller string pool will usually provide no advantage. (But see the section *Initialized Vs. Uninitialized Data* for more on this.)

The STR49152.OBJ stub file provides more string space than the default, but at the expense of impinging on variable space. Since this file sets aside 48K for strings, only 16K is left for your variables and the stack.

One situation in which it is important to reduce memory usage to the absolute minimum is when the program will be installed as a TSR. Another is when the program will be run under a multi-tasking operating system such as Microsoft Windows or Quarterdeck's DESQview. These programs let you create partitions in which your programs will run, and

being able to create a program that uses less memory at run-time is a very desirable feature.

Finally, there may be situations where you need to control the size of the string pool more precisely than the supplied stub files allow. The MAKESTR.BAS program, when compiled and run, lets you create a custom STR#####.OBJ file with nearly any amount of string space. A complete discussion of how to determine the required amount of string space is in the section *String Memory Considerations* later in this manual.

The POPSWAP Stub File

The POPSWAP.OBJ stub file is a special stub file that is needed to allow simplified TSR programs to be swapped out of memory when they are not active. This stub file is used in conjunction with the Swap2Disk and Swap2EMS functions, and it must be listed first on the link command line if either of those routines are used:

```
LINK /noe/nod popswap program , , nul, pdq;
```

Of course, you may use additional LINK options and other stub files as necessary. See the description for Swap2Disk and Swap2EMS for information about creating a swapping TSR program.

Other Stub Files

All of the remaining object modules that can be used as stub files have a leading underscore in their name to identify them as such. Notice that some of the stub files completely remove a particular feature (such as _NOREAD.OBJ), while others replace a statement with a different version which has less functionality and thus smaller code. Object files that totally remove support for a BASIC language statement always begin with the letters *NO*.

It is important to understand that a stub file will save code only if you are using the statement it is designed to replace. Normally, LINK adds only those routines that are actually called for by a program. So if you do not use, say, the KILL command, the library routine that handles KILL is not added to your program. In that case, if you link with the alternate _KILL.OBJ stub file you are telling link to explicitly add that code to your program.

Stub File Details

This section describes in detail each stub file provided with P.D.Q. and explains what they are used for. All of the stub files are summarized in Table II-2 at the end of this section, and the list that follows describes each

in detail. Stub files that always reduce the size of the program are identified in this table; all of the others should be used only when the statement they replace is also being used.

_87ONLY.OBJ

_87ONLY.OBJ avoids adding the floating point emulator routines to a program, and therefore requires an 8087 math coprocessor if floating point math is used. This is similar to the 87.LIB stub library that Microsoft provides with BASIC 7 PDS. When a program is linked with _87ONLY.OBJ and a coprocessor is not present at runtime, the message "Math coprocessor required" is displayed and the program ends. _87ONLY.OBJ is needed and useful only with programs that actually use floating point math.

_CPRINT.OBJ

By default, P.D.Q. PRINT statements send their output through DOS using its built-in console I/O services. Since DOS performs the bulk of the work this results in the smallest code size possible, though at the expense of several important features. The most significant limitation is that the DOS print services do not accept a color argument. Although P.D.Q. does support the COLOR statement, it is normally used only by CLS when clearing the screen.

When you link with _CPRINT.OBJ, printing is instead sent through the BIOS, using a service that does honor color. This lets you print using COLOR, without having to change your programs. This also prevents Ctrl-C and Ctrl-Break from terminating the program if those keys are pressed during (or prior to) a PRINT statement. Be aware, however, that using _CPRINT.OBJ prevents your program's output from being redirected by the user with the " > " DOS command modifier.

Note that _CPRINT.OBJ is not compatible with the P.D.Q. SMALLDOS stub library, because _CPRINT.OBJ honors the current TAB position which SMALLDOS doesn't support.

Also see the PDQPrint and PDQCPrint routines listed in the reference portion of this manual. These routines use a different syntax than PRINT, but they are substantially faster.

_DEBUGFP.OBJ

Unless you truly need floating point math in a program, you can realize a substantial improvement in code size and execution speed by using integers

and long integers only. In past versions of P.D.Q. attempting to use a floating point command resulted in an "Unresolved External" error message when linking. To add insult to injury, LINK does not say where in the program an unsupported command is being used.

Now that P.D.Q. does support many floating point operations, it is possible—even likely—that you could inadvertently use a floating point command and not know it. LINK simply brings in the requisite routine from the P.D.Q. library, and you will be none the wiser. Worse, if you link with _NOVAL.OBJ to completely eliminate all floating point support, your program will crash without warning when that statement is executed.

To determine if a program is using floating point operations you can link with _DEBUGFP.OBJ and then run it. If a floating point command is encountered the program is terminated with the message "Floating Point Required at ####:####". Here, ####:#### indicates the segment and address in your program where the floating point command was used. You can then use CodeView to examine your program, to identify the exact place where floating point math was used.

You can also recompile the program using the /a option, to create an assembly language source listing. Although using CodeView is a more accurate method, /a will get you within a statement or two of the problem. When compiling with /a you must also provide the name of a list file like this:

```
BC program /o /a [/other options] , , listfile.lst ;
```

This creates a file named LISTFILE.LST containing both the BASIC source statements and the equivalent assembly language commands that BC.EXE generated. Simply look in the address column of the listing for the address reported, and the offending BASIC command will be in that vicinity.

Note that the segment reported by _DEBUGFP.OBJ is irrelevant, and only the address value to the right of the colon is significant. Also note that when using _DEBUGFP.OBJ you should not link with _NOVAL.OBJ. That is, _DEBUGFP.OBJ is to be used on a test basis to see if floating point operations are being used.

_DIM.OBJ

Although BASIC limits dynamic arrays to 32,767 elements per dimension, you can create arrays larger than that by using more than one dimension. For example, **REDIM Array%(1 TO 10000, 1 TO 8)** creates an integer array with 80,000 elements. To be as compatible with Microsoft BASIC

as possible, P.D.Q. follows the same rules and allows as many elements as memory can hold.

When _DIM.OBJ is used, the number of elements in an array is limited to 65,535 or less. Very few programs actually need arrays larger than that, and using _DIM.OBJ will reduce the size of any program that creates dynamic numeric, TYPE, or fixed-length string arrays.

_EMONLY.OBJ

_EMONLY.OBJ is the opposite of _87ONLY.OBJ, and it avoids adding the code that supports a coprocessor to your programs. In truth, the savings is much less than for _87ONLY.OBJ, because the code needed to handle a coprocessor is much smaller than the code that emulates it. However, if you are certain that a coprocessor will not be present when the program runs, _EMONLY.OBJ will make your program slightly smaller. And if a coprocessor is present, your program will not use it.

_FLUSH.OBJ

Unlike the FLUSH routine described in the P.D.Q. extensions section of this manual, this version flushes all files only, as opposed to individual files. Because it does not need the added code to accept a varying number of arguments, it will reduce the size of programs that use the P.D.Q. Flush extension.

_GET1BYT.OBJ

This stub file contains an alternate version of Get1Byte that treats the returned values as being unsigned, having values between 0 and 255 instead of -128 to 127. Note that an alternate version of Set1Byte is not needed, because it accepts either signed or unsigned values.

_INKEY$.OBJ

By default, INKEY$ in a P.D.Q. program uses a DOS service to read the keyboard. This lets your program honor redirection using the "<" less-than symbol on the DOS command line. But that particular service cannot be used in a P.D.Q. TSR, as described in the section *TSR Programming With P.D.Q.*

_INKEY$.OBJ replaces INKEY$, and it uses the BIOS rather than DOS to read the keyboard. This lets you use INKEY$ in a TSR program, without having to modify your program's logic to use BIOSInkey.

_KILL.OBJ

The alternate version of KILL contained in _KILL.OBJ does not support the DOS wild cards "*" or "?". It is therefore useful when a program must be as small as possible, for example in a TSR utility. Note, however, that this version of KILL must not be used with an argument comprised of concatenated strings. That is, code such as the example shown below should be avoided:

```
KILL Drive$ + ":" + FileName$      'do not do this

Temp$ = Drive$ + ":" + FileName$   'do this instead
KILL Temp$
```

_LOCATE.OBJ

This stub file supports only the Row and Column parameters, and it requires both to function correctly. Using any syntax other than **LOCATE X, Y** (or **LOCATE 12, 2** or the like) is guaranteed to cause a crash. We also provide a pair of complementary routines to turn the cursor on and off (CursorOn and CursorOff), as well as a routine to set the cursor size (CursorSize). Even if you do need to turn the cursor on and off, using _LOCATE.OBJ and those routines will result in less code.

When BASIC generates a call to its own LOCATE routine, it uses a complicated system of flags and values, and passes a varying number of parameters. The approach we have taken in this stub file adds much less code to your program. The examples in Table II-1 following show how to effectively use only one argument with this version of LOCATE.

Table II-1
LOCATE Syntax Differences

BASIC	P.D.Q.
LOCATE , X	LOCATE CSRLIN, X
LOCATE Y	LOCATE Y, POS(0)

_NOERROR.OBJ

Unless you are using ON ERROR in your programs, we recommend that you link with the _NOERROR.OBJ stub file. By default, all of the P.D.Q. routines call a central error handler if an error occurs, and that handler checks to see if ON ERROR is in effect. If so, it jumps to the ON ERROR GOTO address that was saved when ON ERROR was used. Otherwise,

the error is simply saved for the next time ERR is examined. This _NOERROR.OBJ version of the error handler eliminates the additional code to check for ON ERROR being in effect.

_NONET.OBJ

Like regular Microsoft BASIC, P.D.Q. fully supports shared network file access. However, the code to implement all of the possible OPEN options adds to the size of any program that uses OPEN. _NONET.OBJ contains an alternate version of the OPEN statement, but without support for the SHARED or ACCESS options.

_NOREAD.OBJ

By default, all P.D.Q. programs include the READ routine, even when READ is not being used. Therefore, linking with _NOREAD.OBJ will save nearly 600 bytes from your programs that do not require READ. It is important to understand that you *must* use _NOREAD.OBJ if you are linking with the SMALLDOS.LIB alternate library.

_NOVAL.OBJ

_NOVAL.OBJ is used to exclude all floating point support from a P.D.Q. program. Even if no floating point commands are being used, code is added to take over the floating point interrupt vectors at startup, and release them upon termination. _NOVAL.OBJ instead replaces that code with a single Return command, and does nothing else. Thus, it adds only one byte, as opposed to more than 3000 that are added when the emulator and interrupt code is linked to your program. Therefore, you should use _NOVAL.OBJ if your program is not using floating point math.

When using _NOVAL.OBJ, it is essential that your program not use any floating point commands. If you do the program will crash without any warning, and without displaying any messages. Please note that using the Currency data type with BASIC 7 PDS require floating point support, and you cannot use _NOVAL.OBJ with programs that also use Currency variables. See the _DEBUGFP.OBJ stub file described earlier for more information on avoiding floating point code in your programs.

_PDQVAL.OBJ

This file contains versions of PDQValI and PDQValL that are nearly identical to the default versions in the PDQ.LIB library, except they do not recognize "&H" or a leading plus sign (+) in the strings being passed to them. They are consequently smaller than the default PDQValI and

PDQValL routines. Notice that none of the P.D.Q. *VAL* replacement routines recognize "&O" to specify Octal notation.

_SKIPEOF.OBJ

_SKIPEOF.OBJ eliminates support for recognizing a CHR$(26) EOF (end of file) character when a file has been opened for APPEND. Although P.D.Q. never adds an EOF byte to the files it writes, some older applications do (and so does COPY CON). Prior to adding this stub file, P.D.Q. never looked for an EOF byte. Therefore, if you opened a file that had one for APPEND, anything written to the file would be added after that byte which is incorrect.

In this version of P.D.Q. OPEN checks to see if the last byte (or group of bytes) in the file is an EOF marker, and backs up so the next PRINT # statement will overwrite it. But we just hate to see P.D.Q. program size grow, so the _SKIPEOF.OBJ stub file was added to let you remove that feature if you don't need it.

Do not use _SKIPEOF.OBJ when linking with the SMALLDOS library.

_SORT.OBJ

The _SORT.OBJ stub file is approximately one-half the size of the default string sort routine, however it is only one-third as fast. Therefore, you should use this version when code size is paramount, or the number of strings being sorted is small.

_STR$.OBJ

One of BASIC's most irritating quirks is the leading blank space that STR$ returns when used with positive numbers. PRINT too displays an extra leading space when numbers are printed, because internally it calls upon STR$ to format the value into ASCII digits suitable for display. To avoid this many programmers use:

```
LTRIM$(STR$(Number))
```

or worse:

```
MID$(STR$(Number), 2)
```

or even worse still:

```
RIGHT$(STR$(Number), LEN(STR$(Number)) - 2)
```

If you link with _STR$.OBJ an alternate version of STR$ that does not add a leading blank is used. Using _STR$.OBJ also avoids the added

blank when numbers are printed using PRINT. However, a trailing blank is always printed.

Note that _STR$.OBJ affects only integers and long integers. To obtain the same result with single and double precision values use the _STR$FP.OBJ stub file.

_STR$FP.OBJ

Like _STR.OBJ, _STR$FP.OBJ is an alternate version of STR$ but for use with single and double precision and Currency values only. (See _STR$.OBJ above.) If you are using the STR$ function with both integer (or long integer) values and also with single or double precision numbers, then both stub files would be used.

_TIME$.OBJ

The default TIME$ function does not use DOS to get the time, allowing it to be used safely in any type of TSR program. However, the _TIME$.OBJ stub file contains a slightly smaller version that does use DOS, and should not be used in a manual timer interrupt handler. See CLOCK.BAS for an example of an on-screen clock TSR that uses the default TIME$ function. CLOCK.BAS also uses some clever P.D.Q. programming tricks that show how to avoid adding the string managing routines to your programs.

TABLE II-2
P.D.Q. Stub Files

STUB FILE	COMMENTS
_87ONLY	Avoids adding the floating point emulator. Programs that use floating point math will require a coprocessor. Saves about 2,750 bytes.
_CPRINT	Redirects PRINT statements to use the BIOS instead of DOS, and honor the current COLOR settings. Does not support redirection. Is immune to pressing Ctrl-C and Ctrl-Break. Adds about 70 bytes. Do not use with SMALLDOS.
_DEBUGFP	Causes a runtime error if any floating point commands are used. The address of the offending statement is shown. For debugging purposes only.
_DIM	Excludes support for arrays having more than 65,535 elements. Saves about 50 bytes.
_EMONLY	Forces a program to use the floating point emulator, even if a coprocessor is installed. Saves about 40 bytes.

TABLE II-2 *(Continued)*
P.D.Q. Stub Files

STUB FILE	COMMENTS
_FLUSH	Flushes all open files only, as opposed to selectively. Saves about 40 bytes.
_INKEY$	Forces INKEY$ to accept input from the BIOS rather than DOS. Allows using INKEY$ in a TSR program. No appreciable difference in size.
_KILL	Does not allow DOS wild cards or concatenated strings. Saves about 90 bytes.
_LOCATE	Expects two and only two arguments (row and column). Saves about 250 bytes.
_NOERROR *	Excludes support for ON ERROR. Saves about 50 bytes.
_NONET	Excludes support for network file operations. Saves about 90 bytes.
_NOREAD *	Excludes support for READ. Saves about 600 bytes. Must be used with SMALLDOS.
_NOVAL *	Excludes support for floating point math. Saves more than 3000 bytes.
_PDQVAL	Does not recognize "&H" or a leading plus sign (+). Saves about 230 bytes.
_SKIPEOF	Eliminates support for having OPEN FOR APPEND detect an EOF mark. Saves about 70 bytes. Should not be used with SMALLDOS.LIB.
_SORT	One third the speed of the default SORT but about half the size.
_STR$	Tells STR$ not to include a leading blank with positive numbers. Affects integer and long integers only.
_STR$FP	Tells STR$ not to include a leading blank with positive numbers. Affects floating point and Currency only.
_TIME$	Smaller version of BASIC's TIME$ function. Do not use in an Interrupt 8 handler. Saves about 30 bytes.

* Always useful for reducing the size of your programs.

Stub files not marked are useful only when you are using the statements they replace. Adding a stub file for a command that is not used makes a program larger.

The SMALLDOS Library File

As a complement to the individual stub files described in the preceding section, the SMALLDOS.LIB library contains reduced-size versions of several BASIC file-related statements. As much as possible, we have isolated individual routines into separate stub files, to let you mix and match those statements that require full or limited capabilities. However, several of the BASIC DOS routines are inseparable, and must be grouped together.

For example, if you use the SMALLDOS version of OPEN which does not support random access, then you must also use the SMALLDOS version of LOC. In regular BASIC, LOC returns either a byte offset into a binary or sequential file, or a record number if the file had been opened for random access. Therefore, the SMALLDOS version of LOC does not contain the additional code that is needed to return both types of information. In fact, the primary limitation of the statements and functions contained in SMALLDOS is the lack of support for random files and network operation. Also, OPEN FOR APPEND is not supported, and there are a few minor limitations with PRINT. These will be described momentarily.

The brief summary in Table II-3 shows which BASIC language statements are contained in the SMALLDOS.LIB library.

TABLE II-3
BASIC Statements Contained In SMALLDOS.LIB

STATEMENT	DESCRIPTION
CLOSE	Accepts one and only one file number.
INPUT #	Allows reading strings only.
LOC	No support for RANDOM files (always returns a byte offset).
OPEN	No support for the ACCESS, APPEND, LOCK, RANDOM, or SHARED options.
PRINT [#]	No support for TAB.
PRINT [#],	Simply sends a Tab character to the screen or file instead of padding with the appropriate number of blanks.
RESET	No support for RANDOM files.
SEEK	No support for RANDOM files.

Linking with SMALLDOS is similar to linking with the PDQ386 library. That is, you must list the SMALLDOS library on your LINK command line before the PDQ.LIB library file. This is shown below.

```
LINK /noe/nod program _noread , , nul, smalldos pdq ;
```

As you can see, you must also link with the _NOREAD.OBJ stub file when using SMALLDOS. We have provided the SMALLDOS.BAT batch file to assist you, and you could use that as a model for other batch files as well.

IMPORTANT:

You must use the _NOREAD.OBJ stub file when linking with the SMALLDOS library.

You may not use the _CPRINT.OBJ stub file with SMALLDOS.LIB.

You may not use _SKIPEOF.OBJ with SMALLDOS.LIB.

APPEND

OPEN FOR APPEND is not supported in the SMALLDOS library, merely to reduce the amount of code that is added to programs that do not need this feature. However, it is a simple matter to first open a file for BINARY access, and then SEEK one byte past the end, as shown below.

```
OPEN FileName$ FOR BINARY AS #1          'open the file
SEEK #1, LOF(1) + 1                      'seek just past the end
PRINT #1, whatever                       'do whatcha gotta do
  ...
  ...                                    'program continues
```

CLOSE

Regular BASIC and P.D.Q. allow the CLOSE command to be given with a single file number, multiple numbers separated by commas, or no number to close all open files. However, the SMALLDOS version requires one—and only one—file number. To close multiple files you must use CLOSE once for each, or use the RESET statement to close all files in one operation. Notice that DOS itself will close all open files when a program ends, so it is not really necessary to close files before ending a program. Not using CLOSE will of course save a few bytes in your programs, because that routine won't be included by LINK.

DATA

The DATA statement is supported in SMALLDOS, but not READ. Therefore, DATA is useful only for storing text such as a copyright notice in the .EXE file. However, you may simulate READ and DATA with the PDQParse routine. Related to PDQParse is PDQRestore, and SetDelimit-Char, and these are described separately in the reference portion of this manual. Notice that you must use the _NOREAD.OBJ stub file when linking with the SMALLDOS library.

GET

The SMALLDOS library does not directly support random access file operations. Therefore, GET must specify bytes on a binary basis, and it may not be used with record numbers. However, it is easy to simulate using GET for random access records, as demonstrated in the RANDOM.BAS example program. The SeekLoc function is provided to help you calculate the binary offset based on the record length and desired record number.

INPUT

BASIC's INPUT command is not directly supported when the SMALLDOS library is used. We have therefore provided the PDQInput routine which uses the built-in DOS command-line editor for entering strings. As with INKEY$, PDQInput may not be used within a "simplified" TSR program. You must instead use the BIOSInput or BIOSInput2 routines which are similar, and also offer several additional features.

INPUT

INPUT # may be used to input only strings from a disk file, and may not be used with numeric values. Thus, you must input numbers as strings, and then use VAL (or PDQValI or PDQValL) to obtain their value. Notice that the SMALLDOS version of INPUT # does not recognize a comma or colon as a text delimiter, and is therefore functionally equivalent to LINE INPUT #. Also notice that the length of a line being read from disk is limited to 128 characters. If a line of text in a disk file exceeds this, then only the first 128 characters will be read and ERR is set to 83. Error 83 is the special P.D.Q. "Buffer too small" error. The remainder of the line must then be read using a subsequent INPUT # statement. This is illustrated in the SMALLDOS.BAS example program.

LINE INPUT

LINE INPUT is not supported with the SMALLDOS library. However, we have provided a similar routine called PDQInput that you may use to obtain the same functionality.

LINE INPUT

As with LINE INPUT, the SMALLDOS version does not support the file version of that statement either—you must use the regular INPUT #

statement instead. However, INPUT # does not recognize commas or colons as delimiters in a file, and is therefore functionally equivalent to LINE INPUT #.

LOCK and UNLOCK

LOCK and UNLOCK are not supported, primarily because the SMALLDOS version of OPEN does not honor the SHARE or LOCK READ/WRITE arguments.

OPEN

When using SMALLDOS.LIB, files may be opened for INPUT, OUTPUT, and BINARY modes only, and the older BASICA-style syntax is not supported. For example, to open a file for input you must use the first example shown below, but not the second.

```
OPEN FileName$ FOR INPUT AS #1 'this is supported in SMALLDOS
OPEN "i", 1, FileName$         'this is not supported
```

Finally, the SMALLDOS version of OPEN should not be used with a file name comprised of concatenated strings, as shown below:

```
OPEN Drive$ + ":" + FileName$  'don't do this
```

Instead you must create a temporary variable first, and then use that with OPEN. Concatenated strings passed to OPEN are never deleted, and will thus take string memory permanently. Further, the SMALLDOS version of OPEN does not trap against inadvertently using the same file number more than once. If you use **OPEN "XYZ" FOR OUTPUT AS #1** and then **OPEN "ABC" FOR INPUT AS #1**, your program will not function correctly and no error will be reported.

The network options ACCESS, LOCK READ/WRITE, and SHARED are not supported by the SMALLDOS version of OPEN.

Please see the RANDOM.BAS program for an example of manipulating random access files by record number using TYPE variables.

PRINT and PRINT

When linking with SMALLDOS.LIB you may not use the TAB function to advance to a specified column on the display screen or in a file. Further, using a trailing comma (as in PRINT X,) is useful only when printing to the screen, because the SMALLDOS version of PRINT merely sends a

CHR$(9) Tab character after printing the variable or data. Although DOS will expand the Tab to advance the cursor to the next eighth column on the screen, it does not do this when printing to disk files. Also, be aware that the regular BASIC and P.D.Q. PRINT statements consider Tab "print zones" to be fourteen columns wide rather than only eight.

TAB

TAB is not supported when linking with the SMALLDOS.LIB library, and attempting to use TAB will result in LINK errors. Also see the preceding section on PRINT.

Chapter 3: File And Error Handling

File and Error Handling

File Handling In P.D.Q.

For the most part, file operations in a P.D.Q. program are identical to those in a conventional QuickBASIC program. However, there are a few important differences which are outlined in this section.

Error Handling

Perhaps most important, normal DOS errors such as "File not found" and "Bad file mode" do not end the program abruptly. Rather, the P.D.Q. routines that open a file, change directories, and so forth merely set BASIC's ERR function to the appropriate value. This lets you test the success or failure of the most recent operation by examining ERR. Some typical examples are shown below.

```
OPEN FileName$ FOR INPUT AS #1
IF ERR = 53 THEN
  PRINT "File not found"
  END
END IF

OPEN FileName$ FOR OUTPUT AS #1
FOR X = 1 To NumLines
  PRINT #1, Lines$(X)
  IF ERR = 61 THEN
    PRINT "Disk full"
    END
  END IF
NEXT
```

In practice, you would probably use PDQMessage$ to print the errors. This way you do not have to test for explicit error values and have many separate PRINT statements. PDQMessage contains the text of all possible messages stored in the code segment, which frees up that much more string memory for your program. You could also use a GOSUB statement after selected file operations to invoke a central error checking routine. A typical example would be as follows:

```
OPEN FileName$ FOR OUTPUT AS #1
GOSUB CheckErr
FOR X = 1 TO NumLines
  PRINT #1, Array$(X)
  GOSUB CheckErr
NEXT
CLOSE #1
```

Regular BASIC requires you to use ON ERROR prior to any file operations that may result in an error. Unfortunately, using ON ERROR makes your programs larger and slower. Worse still, it is up to you to design a central error handler that receives control when an error occurs, and from there figure out what part of the program was active at the time the error

occurred. The method P.D.Q. uses is the same as that of C, Pascal, and indeed, DOS itself.

Critical errors, however, normally result in the infamous "Abort, Retry, Ignore" message unless special precautions are taken. For many DOS utility programs this is sensible, and is generally preferable to what regular BASIC does when ON ERROR is not in effect. That is, rather than simply print "Disk not ready in module xyz" and then end the program, P.D.Q. at least gives the user a chance to retry the operation.

Critical errors in a P.D.Q. program may be prevented by using CritErrOff, and then reenabled again afterward with CritErrOn. CritErrOff should be called prior to performing a DOS operation that may fail due to an open drive door. Similarly, CritErrOn reenables the DOS critical error handler afterward. If an error does occur while CritErrOff is in effect, BASIC's ERR function will be set to indicate which error. If you use CritErrOff, it is essential that CritErrOn be called to restore the critical error handler to its original state before the program ends.

Other DOS errors may also be trapped by examining BASIC's ERR function. All DOS-related errors are represented using the 11 codes shown in Table III-1.

Note that error number 71 will never occur unless critical error trapping has been activated using the CritErrOff routine. Also note that error 83 is reported only when linking with the SMALLDOS library.

Table III-1
P.D.Q. DOS-Related Error Codes

NUMBER	MEANING	POSSIBLE CAUSE
52	Bad file number	PRINT #n, where "n" hasn't been opened
		OPEN "xxx" AS #n, where n is greater than 15.
53	File not found	OPEN "xxx" FOR INPUT where "xxx" doesn't exist.
		BLOAD "xxx" where "xxx" doesn't exist.
		NAME "xxx" AS "yyy" where "xxx" doesn't exist.
54	Bad file mode	PRINT #1, where the file was opened for INPUT.
55	File already open	OPEN #1, where #1 is already in use.
61	Disk full	PRINT #1, PUT #1, or BSAVE to a full disk.
62	Input past end	INPUT # past the end of the file.
67	Too many files	All 15 DOS file handles are already in use.
71	Disk not ready	(CritErrOff only) Any DOS operation when the drive door is open.
		LPRINT or PRINT # to a printer that is turned off or off-line.
75	Path/File access	OPEN "xxx" FOR OUTPUT where "xxx" is read-only.
		NAME "xxx" AS "yyy" where "yyy" already exists.
76	Path not found	OPEN a file in a non-existent directory.
83	Buffer too small	(SMALLDOS only) INPUT # when the string is longer than 128 characters.

Understand that P.D.Q. does not attempt to totally decipher all possible error conditions. For example, trying to rename a file to a new name that already exists results in a Path/File Access error. But this error is also generated by DOS if your program tries to open a read-only file for output. Also, notice that a successful DOS operation will clear a previous ERR setting, so you must save the ERR variable if you do not intend to test it until later. This is shown in the example that follows.

```
OPEN "xyz" FOR INPUT AS #1        'let's suppose the file isn't there
SaveErr = ERR                     'save the error code
PRINT #2, "Have a nice day"       'a successful PRINT to a different
                                  '  file clears the previous error
CHDIR "\"                         'this will also be successful
IF SaveErr THEN ...               'now act on the possible OPEN error
```

File Numbers

Valid numbers for OPEN and CLOSE are 1 through 15 inclusive. Even though regular BASIC lets you specify any number between 1 and 255, a translation table must be maintained that holds each possible file number. DOS allows only 15 files to be opened at one time in most situations anyway, so this limitation is insignificant.

Although BASIC allows you to decide which numbers will be used to refer to the various files that are opened, internally it is really DOS that issues the file handles. Therefore, BASIC (and thus P.D.Q.) must translate the numbers you choose to the values that DOS assigned when the file was first opened.

Legal File Operations

Unlike QuickBASIC, P.D.Q. lets you perform nearly any file operation on any file, regardless of how it was opened. The only exception is when a file has been opened for INPUT, DOS itself prevents writing to it. In this case, P.D.Q. sets the ERR function to error 75, "Path/file access error". This is different from BASIC which issues a "Bad file mode" error.

As with regular BASIC, if you open a file for OUTPUT it is either created if it didn't already exist, or truncated to a length of zero if it did. However, if a file has been opened for OUTPUT you may freely write to *or read from* the file. Opening a file for BINARY also gives you both read and write access, but without truncating the file to a length of zero.

DOS Devices

Like regular BASIC, P.D.Q. lets you open any of the DOS logical devices, as well as normal disk files. For example, OPEN "LPT1" FOR OUTPUT

is perfectly legal, although you must not include a colon after the 1 (or 2) as you would with regular BASIC. One reason for opening the printer rather than using LPRINT is to defer selection of the print destination until the program runs.

For example, many programs offer an option to output to the screen, the printer, or a disk file. Thus, you could assign a string variable to "CON", "LPT1", or a file name, and then use the same block of code to open whatever was specified and send its output there.

Table III-2 shows the device names you can specify in a P.D.Q. OPEN statement.

Table III-2
Valid DOS Devices In P.D.Q.

OPEN FOR OUTPUT	OPEN FOR INPUT
AUX	AUX
CON	CON
LPT1	
LPT2	
LPT3	
PRN	

Finally, the reserved PRINT number 255 sends its output to the DOS STDERR (standard error) device. You do not have to OPEN #255 before printing to it, and doing so will set ERR to 52, which is the "Bad file number" error. STDERR is always the console (screen), even if a program's output has been redirected. Thus, you can use PRINT #255 to print error or other messages, confident that they will be visible.

File and Error Handling

Chapter 4: TSR Programming

TSR
Programming

One of the most powerful and exciting capabilities of P.D.Q. is its built-in support for TSR programming. This section describes all of the steps necessary to create any type of TSR program, from simple pop-up utility programs to full-blown interrupt handlers. TSR programs written using P.D.Q. are as safe and reliable as any commercially available, as long as you follow a few simple rules. Writing a reliable TSR program is normally very difficult; however, we have done all of the hard parts for you!

There are two basic types of TSR programs that may be written using P.D.Q. The first is the easiest to implement—you simply specify the hot key to use for popping up, and the location in your program to execute each time that key is detected. The second type of TSR requires only a little more effort, however any interrupt or combination of interrupts may be intercepted within a single program. A third type of TSR combines the two methods letting you take over interrupts manually, and also perform safe file operations. We'll discuss the simplified method first.

Notice that this section provides an overview of the concepts involved in creating TSR programs using P.D.Q. You should refer to the individual routine descriptions for the exact calling syntax. Many working example programs are included on the P.D.Q. distribution disk, and you should examine these as well. Also, be sure to see the section entitled *Linking With Stub Files*, which describes how to reduce the runtime memory requirements of a TSR program.

Simplified Pop-Ups

TSR programs that use a single hot key as a trigger may be written using what we call the simplified TSR method. That is, you issue a call indicating which hot key to detect and where to go when that key is pressed, and the appropriate P.D.Q. routines do the rest. For this type of TSR program, three different routines are used.

The first is PopUpHere, which requires a numeric code that represents the desired hot key and a unique identifying string. The numeric key code is comprised of two parts—a *scan code* which specifies the hot key itself, and a *shift mask* that indicates which combination of Alt, Ctrl, and Shift keys are recognized. A table of keyboard scan codes and shift masks is shown in the section that follows. The unique identification string is described in the section following this one.

The second routine is PopDown, and it is called when your program is ready to return control to the underlying application.

The third routine is EndTSR. It is called as part of the install sequence, and it returns control to DOS allowing other programs to be subsequently run. EndTSR also expects the unique identification string as a parameter, so it can determine if the program has already been loaded into memory. This lets you prevent a novice user from attempting to install the same TSR program twice. Notice that the call to EndTSR must be the last statement in your program.

A fourth routine, PopDeinstall, may optionally be used within a simplified TSR program, and it lets a program remove itself from memory.

The brief program skeleton below shows the minimum steps necessary to create a simplified pop-up TSR program.


```
DEFINT A-Z                      'all integers, please
ID$ = "My TSR program V1.0"     'every program needs a unique ID
PRINT ID$                       'we might as well use the
                                '  sign-on message
HotKey = &H81F                  'Alt-S
CALL PopUpHere(HotKey, ID$)     'specify the hot key
GOTO EndIt                      'skip over the hot key handler
 ...
 ...                            'the actual program goes here,
 ...                            '  and it is executed whenever
 ...                            '  the hot key is pressed
 ...
 CALL PopDown                   'return to the underlying application
EndIt:
 CALL EndTSR(ID$)               'install as a TSR and return to DOS
```

Each time the specified hot key is pressed, the code that immediately follows the line containing the GOTO will be executed. It is imperative that you follow this sequence exactly, using a GOTO as the only line after the call to PopUpHere. Of course, you may print a sign-on or copyright message before calling PopUpHere, and perform any other necessary initialization. This code could also be placed after the *EndIt:* label, but before the call to EndTSR.

Internally, PopUpHere retrieves the address in your BASIC program that follows the CALL statement. Then, knowing that a GOTO follows, it adds the correct number of bytes (three) to obtain the address to jump to when the hot key has been detected.

Restrictions

You may use almost any BASIC statements that are supported by P.D.Q. within the body of the TSR; however, you must call PopDown when you are finished and want to return to the underlying application. The only statements that may not be used in a simplified TSR are INKEY$,

PDQInkey and PDQInput, and the console (keyboard) versions of INPUT and LINE INPUT.

INPUT # and LINE INPUT # are allowable in a TSR, but you must use BIOSInkey and BIOSInput instead of INKEY$ and INPUT. If you prefer not to change existing source code, you may optionally use the _INKEY$.OBJ stub file, which redirects INKEY$ to use the BIOS instead of DOS.

Critical Errors

If you intend to perform file operations, we strongly suggest that you use the CritErrOn and CritErrOff routines. CritErrOn is needed to prevent an error caused by an open drive door from affecting the other program. These routines are described elsewhere in this manual.

Memory Allocation And Dynamic Arrays

One important point to be aware of is a small restriction on using dynamic arrays in a TSR program. P.D.Q. uses the standard DOS memory allocation services to set aside memory for dynamic (far) arrays. But most applications claim all available memory when they load, thereby preventing a TSR program from allocating additional memory for itself when it pops up or receives control through a system interrupt. Therefore, you must dimension all dynamic arrays before your program calls EndTSR to terminate and stay resident. The same is true if you are using the AllocMem routine to allocate DOS memory. You must use AllocMem before calling EndTSR.

Note that this restriction does not affect conventional (not fixed-length) string arrays. These arrays are stored in the P.D.Q. string pool, and that memory is contained with the program when it loads and stays resident.

TSR Programs That Swap To Disk Or EMS

One of the most powerful capabilities P.D.Q. offers is the ability to create large TSR programs that require very little memory when they are idle. Where most TSR programs remain fully in memory even while they are not being used, P.D.Q. provides a special "swapping" option that lets your programs reside on disk or in expanded memory when they are inactive. All that remains in memory between popups is a tiny "code kernel" that occupies a very small amount of memory.

This feature is called swapping because the underlying program is saved from memory each time the hot key is pressed, and your TSR program is

then loaded in its place. Thus, your program and the underlying program are exchanged, or swapped, in memory. This process is similar to using program overlays, whereby different portions of a single program occupy the same area of memory but at different times.

Please understand that swapping is supported in "simplified" popup TSR programs only. However, it is possible to cause a program to pop up using CALL INTERRUPT from another program—even from another TSR that takes over interrupts manually—and this will be described later in this section.

TSR programs may be swapped to either expanded memory (EMS) or a disk file. Since expanded memory can be accessed much more quickly than a disk drive, this is the preferred method. If you know that expanded memory will not be available when your program is run, you can save a small amount of code by using only the disk swapping routine. Likewise, if you are certain that sufficient expanded memory is available you can use that routine only, and avoid adding the code that handles swapping to a disk file. Otherwise, you will probably want your program to decide at runtime which storage method is available.

Adding the swapping capability to a TSR popup program requires only two simple steps:

- You must link with the POPSWAP.OBJ stub file, listing that as the first object file on the LINK command line.
- You must call either Swap2Disk or Swap2EMS, before calling EndTSR to end your program and leave it resident in memory.

Again, you will probably want to swap to EMS if it is available, and use a disk file as a second choice. However, it is also possible that there will be insufficient disk space to allow swapping to disk. In that case swapping cannot be used at all. Therefore, you should request each method in turn just before calling EndTSR, like this:

```
    ...
    ...

EndIt:
  IF Swap2EMS%(ProgramID%) THEN
    PRINT "Program is installed, swapping is to expanded memory."
  ELSEIF Swap2Disk%(SwapFile$, ProgramID%) THEN
    PRINT "Program is installed, swapping is to a disk file."
  ELSE
    PRINT "Program is installed, swapping not enabled."
  END IF
```

```
CALL EndTSR(ID$)
```

When your program is first run the swapping routine is installed as a TSR, leaving only that code in memory. Then, when the hot key is pressed the current program is swapped out of memory and your TSR is loaded in its place. When your program calls PopDown the process is reversed.

Swap2Disk requires you to provide the name of the swap file, and a program ID number which is needed to let the TSR be accessed from another program. (The program ID number can usually be set to zero, and it will be described momentarily.) If the file does not exist Swap2Disk creates it, using a size large enough to hold the in-memory image of your TSR program's code and data. If the file does exist Swap2Disk ensures that it is large enough, and extends it if necessary. This would be necessary when developing a program, and new features have been added since the last time it was run.

Swap2EMS is similar, except it does not require a file name. Instead of allocating disk space it tries to locate sufficient expanded memory, returning either -1 if enough EMS is present or 0 if EMS is not usable.

Naming The Swap File

When swapping to disk we recommend that you give the swap file the same name as your main program, but with a .SWP extension. By using the same name as the main program rather than a fixed name such as *SWAPFILE.DAT*, different swapping TSR programs may be installed simultaneously. You should also specify that the file reside on a local (non-network) hard disk for best performance. The FNRemovable and FNRemote functions in the SYSINFO.BAS program show how to determine which disk drives are floppy (removable) and which are on a network (remote).

If you prefer you may let the user specify the swap file name as a command line option, and read it using COMMAND$. This would let the person running the program decide not only the name of the file, but also the drive and directory. In either case, the file name you pass to Swap2Disk can include a drive letter and colon, a directory path, or both.

When Swap2Disk is called it also sets BASIC's ERR function to indicate success or failure. This way you can tell not only if something went wrong, but also what. If Swap2Disk returns 0 to indicate failure, then ERR will be set to one of the three BASIC error codes shown in Table IV-1. Otherwise ERR is cleared to zero.

TABLE IV-1
BASIC ERR Codes Set By Swap2Disk.

61	Disk full
75	Path/file access error
76	Path not found

Be aware that for programs that are swapped to disk, rebooting or turning off the PC without first deinstalling the program leaves the swap file on disk. Of course, the same file will be used the next time the program is run, assuming you specify the same name.

Deinstallation

Deinstalling a swapping TSR is handled the same as for a TSR that doesn't use swapping, except a program can deinstall itself only while popped up—you cannot deinstall by running a second copy. PopDeinstall knows if disk or EMS swapping has been employed, and if so erases the disk file or frees the expanded memory respectively. Either way, deinstallation is transparent to your program and you don't have to consider which, if either, swapping method is in use.

Dynamic Memory Allocation

As with non-swapping TSR programs, you may not allocate DOS memory once the program has been installed. This applies both to memory that is allocated manually with the P.D.Q. AllocMem function, and memory that is claimed implicitly using REDIM with numeric, TYPE, or fixed-length string arrays. Therefore, it is essential that your program create any such dynamic arrays it needs, before calling EndTSR to end and stay resident. Dynamic conventional (not fixed-length) string arrays reside in the P.D.Q. string pool and do not have this restriction.

Handling Interrupts In A Swapping TSR

Swap2Disk and Swap2EMS are suitable for use only with simplified popup TSR programs. Further, you may not take over interrupts manually in a TSR that employs swapping, unless you hook the interrupts within the popup handler and unhook them again before calling PopDown. However, you can combine manual interrupt handling and swapping by using a second TSR program that communicates with the first one. By requiring two programs for this special case we are able to keep the resident code kernel as small as possible.

To tell a swapping TSR that it is to pop up, use CALL INTERRUPT specifying interrupt &HAA with a *program ID number* in the AX register and the number of ticks to try to pop up for in CX. This number is equivalent to the NumTicks argument used with PopRequest. Since all P.D.Q. swapping programs use this interrupt, you need to identify which is to be popped up in case more than one is loaded. This is the purpose of the ProgramID% parameter that is passed to Swap2Disk and Swap2EMS, and it must agree with the number in Registers.AX when you use CALL INTERRUPT from another program.

If only one swapping TSR will be accessed with CALL INTERRUPT, you can use a value of zero for the program ID variable both when calling Swap2Disk or Swap2EMS and when assigning Registers.AX. But it is also possible that you will want to have more than one swapping TSR dormant, and be able to specify which is to be activated with CALL INTERRUPT.

For example, you could have a communications program that needs to be popped up at midnight when phone rates are lower, and another program that pops up in response to an incoming RING command from a modem. In this case the communications program and the timer interrupt handler would use one program ID number, and the phone answering program and the modem handler using another. As with programs that use PopUpHere, you may optionally specify a hot key of zero to prevent someone from popping up the program that way.

Communicating With A Swapped TSR

Besides specifying the program ID number in AX and the number of system timer ticks to try to pop up for in CX, you may also pass an additional integer parameter to a swapping TSR in BX. You can use this parameter for any purpose; for example, to tell the TSR program to deinstall itself. Whatever value was in the BX register when CALL INTERRUPT was used is available within the TSR via the SwapCode function. The following examples show this in context.

To invoke a swapping TSR from another program use CALL INTERRUPT as follows:

```
DIM Registers AS RegType
Registers.AX = 0          'pop up the program whose ID is 0
Registers.BX = 1          'pass it a parameter value of 1
Registers.CX = 18         'try to pop up for one second
CALL INTERRUPT(&HAA, Registers)
```

Then within the swapping TSR you can retrieve the parameter passed in BX using the SwapCode function. This example uses a code value of 1 to indicate that the program is to deinstall itself:

```
Parameter = SwapCode%
IF Parameter = 1 THEN
  Success = PopDeinstall%(DGroup%, ID$)
  CALL PopDown
END IF
```

IMPORTANT:

Be careful not to call Interrupt &HAA if no swapping TSR programs are loaded. Since this interrupt is normally uninitialized (the segment is set to zero by default), calling it will likely hang the PC. You can easily test if the segment pointed to by Interrupt &HAA is zero like this:

```
DEF SEG = 0
IF PDQPeek2%(&HAA * 4 + 2) THEN
   'it is okay to call this interrupt
 ELSE
   'the interrupt is invalid, do not call it!
END IF
```

Please see the DEMOSWAP.BAS demonstration program for a complete working example of a popup TSR program that swaps to either EMS or disk.

The Unique Identification String

Every P.D.Q. TSR program must define a unique identification string, which is used by PopUpHere, EndTSR, and TSRInstalled to determine if the program is already loaded into memory. This string must be at least 8 characters long, simply to ensure that the same bytes won't be found in memory by coincidence.

The simplest way for a program to know for certain if it is already installed is to search all of memory for a copy of itself. The routines that do this in P.D.Q. begin at the bottom of memory that immediately follows DOS, and search successively higher addresses until the ID string is located.

If the code segment and address where the ID string is found is the same as the current code segment, then the program has "found" itself, and no other copies are already resident. (This is why the string must not be allowed to move. Instead of having to search all memory address, all that is needed is to search a single address in all possible segments.) If the ID$ is found in a lower segment, then that copy of the program was loaded first, and this is an attempt at re-installation.

We recommend that you use the program name and version number when defining the ID string. For example, **ID$ = "My TSR version 2.03"**. This greatly minimizes the likelihood of inadvertently creating two different programs with the same ID string. Note that only the first 16 characters of ID$ are used when searching for a match in memory.

You should not use the identification string after it has been passed to EndTSR. As part of its own initialization, EndTSR modifies the first character in the string. Therefore, you must not change it once your program has become resident. If the string were not modified, TSRInstalled could possibly find it in a DOS file buffer. All programs that are loaded into memory pass through these buffers, and it is possible that a copy of the string is still present there. This would confuse TSRInstalled into thinking that the program was already resident.

Of course, it is not necessary for you to understand how the ID$ is processed internally, and this information is provided solely for those who are interested. All you need to remember are the following three rules:

1. Create a unique string that is at least eight characters long and is not fixed length.

2. Define the string before any other string variables or functions are referenced.

3. Do not use the string after calling EndTSR.

Specifying The Hot Key

The hot key is specified in two portions—a shift mask and a scan code. The easiest way to indicate the hot key is by using an "&H" value, since the high and low byte portions of an integer variable can be manipulated separately. The code fragment below uses Alt-S as the hot key.

```
HotKey = &H081F
```

In this example, 08 is the shift mask for Alt, and 1F is the scan code for the "S" key.

The shift mask portion of the key code is derived as shown in Table IV-2.

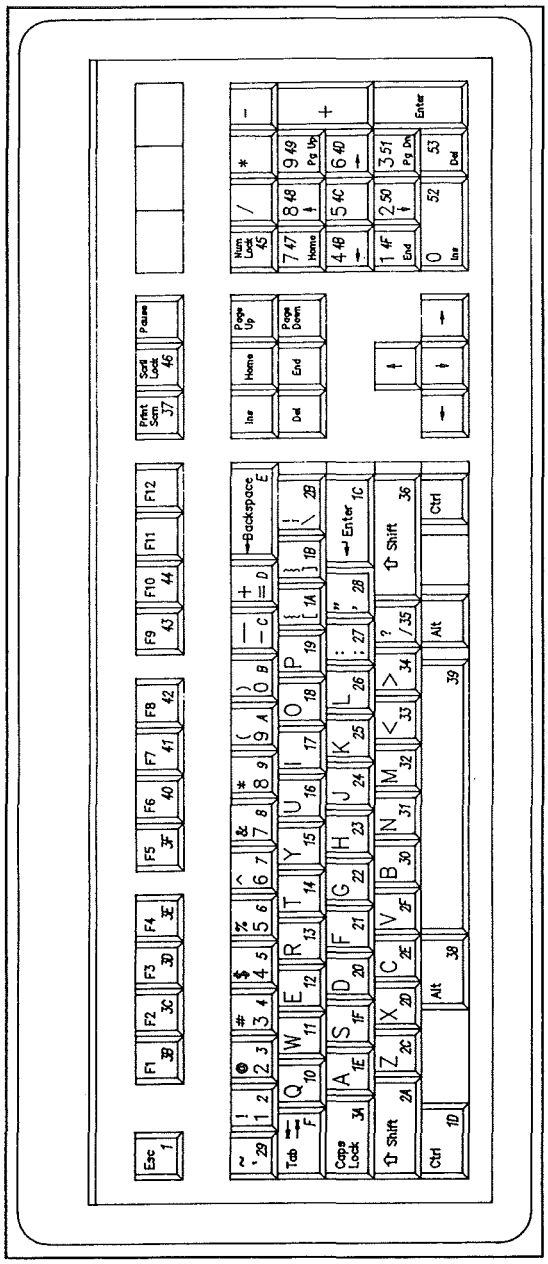


Figure 1: Hexadecimal Keyboard Scan Codes

Table IV-2
Shift Mask Values

8 = Alt
4 = Ctrl
2 = Left Shift
1 = Right Shift

You may also use OR to combine the various components. For example, to detect Ctrl-Alt-S the upper portion of the key code would be 8 OR 4 which equals 12 decimal, or 0C Hex. Therefore, the correct key code for Ctrl-Alt-S is &H0C1F.

A listing of keyboard scan codes is shown in Figure 1. Notice that in most cases, the scan codes are identical for all variations of the IBM PC keyboard and its clones. However, some keyboards such as the 101-key models have slight differences on some keys. The scan codes shown below apply to all keyboards, and we have purposely omitted showing those codes that are not the same on all keyboards. We recommend using only those keys that produce the same scan codes regardless of the type of keyboard being used.

Hexadecimal notation is used, because that is the simplest way to specify the scan code when calling PopUpHere and TestHotKey.

Detecting Installation And Deinstalling

One of the most useful features we have provided with P.D.Q. is the ability to remove a TSR program from memory. Two options are provided—the first allows one copy of a program to deinstall another, earlier copy. The second lets a program remove itself from memory.

Many TSR programs accept a command-line parameter such as /U, to indicate that the program is to be uninstalled. In that case, you are actually running a second copy of the program, and asking it to remove the copy that was installed earlier. However, it is also possible to have a TSR program remove itself from memory. For example, this could be a menu option selected from within the program. These two methods obviously require very different code internally to implement. Fortunately, the P.D.Q. routines shield you from much of the messy details.

Any P.D.Q. TSR program that will be deinstalled must invoke the TSRInstalled function early in its execution, and save the value it returns

before calling EndTSR. If TSRInstalled returns zero, then the program has not yet been installed and it is safe to do so. If, however, TSRInstalled returns any other value, then that value is the DGROUP data segment being used by the resident copy of the program. The TEMPLATE.BAS example program shows how to detect prior installation and also how to remove a previously loaded copy.

With TSR programs that use a command line switch to indicate deinstallation, you would use the non-zero segment value that TSRInstalled returns to indicate that the previous copy is to be removed from memory. If a program is going to remove itself from memory, then you would instead use a value of zero for DGROUP. The appropriate logic for installing, detecting installation, and de-installing both types of TSR is shown next.

Remove a previous copy via a command-line switch:

```
DGroup% = TSRInstalled%(ID$)        'see if we're already installed
IF INSTR(COMMAND$, "/u") THEN       'see if they want to deinstall
  IF DGroup% THEN                   'we're already installed
    Okay% = PopDeinstall%(DGroup%, ID$)  'try to deinstall
    IF NOT Okay% THEN               'if not successful then say so
      PRINT "Unable to deinstall.  Reboot now!"
    ELSE                            'otherwise report success
      PRINT "Program successfully removed."
    END IF
  ELSE                              'DGroup% was 0, not installed
    PRINT "Program is not resident, try again without /u."
  END IF
  END                               'either way end the program
ELSE                                'they're not deinstalling
  IF DGroup% THEN                   'if we're already installed
    PRINT "Program already installed, press the hot key."
    END                             'say so and end
  END IF
END IF
CALL PopUpHere(HotKey%, ID$)        'install the program
GOTO EndIt                          'skip to the end to terminate
  ...
  ...                               'the pop-up handler goes here
  ...
EndIt:
  CALL EndTSR(ID$)                  'terminate and stay resident
```

Remove the current copy via a menu choice:

```
DGroup% = TSRInstalled%(ID$)        'see if we're already resident
IF DGroup% THEN                     '  early in the program
  PRINT "Already installed!"        'yes, say so and end
  END
ELSE
  CALL PopUpHere(HotKey%, ID$)      'no, install ourselves here
  GOTO EndIt                        'skip to the end to terminate
```

```
    GOTO Main                          'jump into the pop-up handler
  END IF
    ...
    ...                                'show your menu here
    ...
  IF MenuChoice% = Quit% THEN          'do they want to deinstall?
    Okay% = PopDeinstall%(0, ID$)      'yes, try to do that
    IF NOT Okay% THEN                  'if not successful say so
      PRINT "Unable to deinstall, reboot now!"
    ELSE                               'otherwise report success
      PRINT "Program successfully removed."
    END IF
  END IF
  CALL PopDown                         'either way, pop down

Main:
  ...                                  'the pop-up handler goes here
  ...
  ...
EndIt:
  CALL EndTSR(ID$)                     'terminate and stay resident
```


Advanced TSR Applications

Besides the simplified pop-up method, P.D.Q. TSR programs may also intercept one or more interrupts directly. This is slightly more difficult than using the simplified method, because it is up to you to determine which DOS and BIOS services are "safe" to call. This is especially true for programs that intercept a hardware interrupt such as keyboard Interrupt 9, because that interrupt could occur at any time—even when DOS is in the middle of an operation. Because DOS is not reentrant, you are not allowed to call a DOS service when DOS is already busy servicing another request.

For example, if a DOS service called by the underlying program is currently in progress and your program is invoked by someone pressing a hot-key, using PRINT or OPEN is guaranteed to cause a crash. Likewise, when a TSR program interrupts a BIOS service while it is executing, that same service may not be called again by your program.

Unless your program is extremely simple, or does not use any BASIC statements that call DOS or the BIOS, we strongly recommend that you use the simplified method when writing pop-up keyboard handlers. However, several example programs are provided with P.D.Q. that show the steps necessary to create completely safe "manual" TSR programs.

It is important to point out that the various P.D.Q. interrupt handling routines may also be used in non-TSR programs. For example, you could

create a subroutine that receives control each time the system timer interrupt is generated, thus simulating BASIC's ON TIMER feature. Of course, it is imperative that the interrupt be deinstalled with the UnhookInt routine before your program is allowed to terminate. This technique is illustrated in the ONTIMER.BAS example program. Also see the section *P.D.Q. Runtime Reentrance* for a discussion of potential problems when handling interrupts manually this way.

P.D.Q. Interrupt Handling Services

Several routines are provided with P.D.Q. to support interrupt handling, and each of these will be described in turn. For every interrupt that your program will handle, an 18-element TYPE variable must be defined. This TYPE variable is used to hold the contents of the processor's registers, as well as the address and segment to use if the original interrupt is accessed. Let's begin by examining each of the P.D.Q. interrupt routines.

The first is PointIntHere, and like PopUpHere it indicates where in your program execution is to go when the specified interrupt occurs. Also like PopUpHere, a call to PointIntHere must be immediately followed by a GOTO. The next program statement then receives control each time the specified interrupt occurs. Please understand that any interrupts you intercept with PointIntHere must be unhooked later, if you intend to remove the TSR program from memory. For interrupt handlers that are not resident, you must unhook the interrupts before ending. The UnhookInt routine is meant for this purpose, and it will be described in a moment.

The next two routines are IntEntry1 and IntEntry2, and these must be the very first two statements in the body of the BASIC interrupt handler code. These routines copy the current register values into the Registers TYPE variable, so they may be examined and set by your program. Each time a program receives control through an interrupt, it must immediately call IntEntry1 and IntEntry2.

IntEntry1 simply saves the current value of the AX register and establishes "DGROUP addressability", so variables within the TSR program may be accessed correctly. IntEntry2 copies the remaining register values into the TYPE variable, and jumps to the correct location in the BASIC program.

IntEntry2 also expects an "Action" parameter, which tells it what to do if another interrupt occurs before you have finished processing the first one. There are two options: pass control on to the original interrupt handler, or ignore the interrupt entirely and simply return to the caller. The only situation in which a second interrupt could occur like this is when trapping

hardware interrupts such as the timer tick or the keyboard or communications interrupts.

While your program is handling the interrupt, it may call one of the three P.D.Q. routines. The first is GotoOldInt, which passes control to the original interrupt handler. GotoOldInt is used when you are intercepting only certain services, and the current service is not one of those. In that case you would want to pass control on to the original (or subsequent) handler. For example, if you are intercepting DOS interrupt 21h and care only about, say, service 4Eh, then you would call GotoOldInt for all of the other services so DOS will handle them. Notice that a call to GotoOldInt does not return to your program.

The second routine is CallOldInt, and it lets you call the original interrupt as a subroutine and then receive control again when it has finished. This would be useful when intercepting the printer interrupt, perhaps to test the success of the last print attempt.

The last interrupt handling routine is ReturnFromInt, which returns control to the underlying application. ReturnFromInt would be used when your program has processed the interrupt entirely by itself, and no further action is needed by the original handler. That is, you will use ReturnFromInt when you do not call GotoOldInt.

Related Routines

Besides the three core interrupt handling routines, there are several other related routines. One of these is UnhookInt, which lets you remove a P.D.Q. program from an interrupt chain. UnhookInt is designed as a function and it returns a flag that indicates if it was successful.

Please understand that if another TSR program has been loaded after yours, it is possible—even likely—that it has taken over the same interrupts that you have. In that event there is no reasonable way to remove your program from the interrupt chain, unless the other program is removed first. Likewise, if you intend to load and deinstall multiple TSR programs yourself, you must remove them in the reverse order from which they were installed. This has nothing to do with P.D.Q. and the same rules apply to all TSR programs that take over interrupts.

The next two related routines are TestHotKey and ResetKeyboard, and these are used in programs that handle the keyboard Interrupt 9 directly. TestHotKey lets your program quickly determine if the current key press is one your program plans to act upon. If it is, then ResetKeyboard should

also be called, to reset the keyboard hardware and interrupt controller chip.

Again, you don't need to know the actual details within these routines, only the sequence in which they must be called. TestHotKey and ResetKeyboard are shown in the HIGUY.BAS demonstration program. Also see the section in the manual that describes TestHotKey for an example showing how to read the key from the keyboard hardware directly.

The next two P.D.Q. TSR routines are TSRInstalled and DeinstallTSR. TSRInstalled, described previously, lets you determine if a TSR program is already resident, and it was already described. DeinstallTSR is similar to PopDeinstall, and it is meant to remove a TSR that does not use PopUpHere from memory. As with the "simplified" P.D.Q. TSR programs, if you intend to remove the TSR from memory you must invoke TSRInstalled as one of the first steps in your program before calling EndTSR. TSRInstalled returns the current DGROUP segment value, which allows DeinstallTSR to locate the program to remove. This is exactly the same as when detecting and removing simplified TSR programs, as described earlier.

Unlike the simplified method, though, it is up to you to determine when it is safe to call DeinstallTSR. This is the purpose of the last function, DOSBusy. DOSBusy reports if any DOS interrupts are currently in progress, which would preclude DeinstallTSR from being used. Please notice that like TSRInstalled, you must invoke DOSBusy once in your program before it calls EndTSR to terminate and stay resident. Also notice that programs that are deinstalled by running a second copy do not need to bother with DOSBusy. This is shown in the DOSWATCH example program. For programs that will be deinstalled from the DOS command line, it is not necessary to use DOSBusy.

Resetting The 8259 PIC

Finally, if you have a program that intercepts a hardware interrupt and it handles the interrupt entirely by itself, you must reset the 8259 Programmable Interrupt Controller chip (PIC) at some point in your interrupt handling code. This is easily done with an OUT statement as shown below.

```
OUT &H20, &H20
```

This is not needed if the program subsequently uses CallOldInt or GotoOldInt, because the original interrupt handler undoubtedly has code to do this already.

The Registers TYPE Variable

For every interrupt you intend to process, an 18-element TYPE variable is needed. This variable specifies the interrupt number, and also holds the current values of the processor registers. Thus, your program may read these registers when it gets control, and then optionally set them prior to calling or jumping to the original interrupt handler. This TYPE variable is shown in Figure 2.

```
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
    DS      AS INTEGER
    ES      AS INTEGER
    SS      AS INTEGER
    SP      AS INTEGER
    Busy    AS INTEGER
    Address AS INTEGER
    Segment AS INTEGER
    ProcAdr AS INTEGER
    ProcSeg AS INTEGER
    IntNum  AS INTEGER
END TYPE
DIM Registers AS RegType
```

Figure 2: The Registers TYPE varible.

The Busy element is set automatically by IntEntry2, to prevent your program from entering an endless loop if the same interrupt comes along before you have finished processing the first one. This could happen only with hardware interrupts; how the Busy flag is handled depends on the value you use for Action when IntEntry2 is called. Your programs can also test the status of the busy flag to prevent reentrance problems, and this is described in the section *P.D.Q. Runtime Reentrance*.

The Address and Segment portions of the TYPE variable contain the address and segment of the original interrupt handler. The ProcAdr and ProcSeg components hold the address and segment to jump to within the P.D.Q. interrupt handling program. These are used internally by the P.D.Q. routines, so GotoOldInt and CallOldInt can access the replacement

interrupt handler. None of these components are intended to be altered by your programs, and they are explained solely for completeness.

The IntNum element is meant to be assigned by you, and it specifies which interrupt is being intercepted. Simply assign Registers.IntNum to the interrupt number you wish to trap, and then call PointIntHere.

Floating Point Considerations

When writing TSR programs with P.D.Q. it is best to link them with the _NOVAL.OBJ stub file, and avoid using BASIC's floating point commands and functions if possible. Including the floating point library increases the size of a program; therefore, less memory will be needed for the TSR if floating point math can be avoided altogether. In many cases this is easy. For example, if you need to obtain the value of a string such as a numeric parameter from COMMAND$, you can use PDQValI or PDQ-ValL instead of VAL. Likewise, you can often use long integers and the P.D.Q. Dollar$ function to simulate fractional values. This is shown in the PDQCALC.BAS demonstration program.

But when floating point math really is necessary in a P.D.Q. TSR program, only a few simple steps are needed. However, it is imperative that you heed these instructions! The discussion that follows explains how floating point math is handled generally by P.D.Q. (and indeed, by most high-level languages). First, some background information is presented. Then, specific instructions show how to safely include floating point operations in a P.D.Q. TSR program.

Floating Point Interrupts

Floating point operations in most high-level languages are handled through a system of interrupts. Interrupts provide a simple yet effective way to invoke code whose address is not known when the program is created, or whose address changes. This method is also used for accessing DOS and BIOS services, and is described in the section *Using CALL Interrupt* elsewhere in this manual.

Interrupts are also ideal for handling floating point math—a program can generate an interrupt whenever it needs to perform a floating point assignment or calculation, and the interrupt will be directed as needed. If a numeric coprocessor is present, the interrupts are directed to code that uses it. Otherwise, the same interrupts will instead invoke routines in a software emulation library.

When a P.D.Q. program is first run, code in the startup module calls a routine named P$HookFP. This routine first checks to see if a coprocessor is installed in the host PC. If so, it saves the current contents of the floating point interrupt vectors, and then points those vectors to the P.D.Q. floating point library routines that use a coprocessor. Otherwise, it directs the interrupt vectors to the P.D.Q. floating point emulator. This emulator mimics the behavior of an 80*x*87 chip with routines written using conventional 8088 instructions only. Either way, a public flag variable is set or cleared so other P.D.Q. routines can know if a coprocessor is present.

Because interrupts are used to invoke floating point operations, there is a potential for conflict between a P.D.Q. TSR program and a conventional program running in the foreground. More specifically, to which code do the interrupts point at any given time? While the foreground program is active the interrupts must be available, so it can use them if necessary. But when your TSR program pops up it requires the interrupts to point to itself. Otherwise, your program's interrupt calls will invoke the routines in the underlying program's emulator. This will of course destroy any calculations that were pending when your TSR received control.

Therefore, it is up to you to manually take over and release the floating point interrupts as part of your program's operation. In fact, besides saving the interrupts, you must also save the state of the coprocessor if one is present. Just as your use of another program's emulator will overwrite its data, so too will interrupting another program's use of a coprocessor. Fortunately, the 80*x*87 family includes instructions to save and restore the entire current context.

Using Floating Point In A TSR

Four routines are provided to let you control where the various floating point interrupts point to, and when. HookFP and UnhookFP merely claim and release the floating point interrupt vectors (interrupts &H34 through &H3C). EnableFP and DisableFP call HookFP and UnhookFP respectively, and they also save the state of the coprocessor if one is present. It is not usually necessary to call HookFP yourself. (If you have looked at the startup code in PDQ.ASM, note that these routines refer to the same code as the routines whose names begin with "P$".)

Because the P.D.Q. startup code calls HookFP initially, you must call UnhookFP manually before calling EndTSR to terminate and stay resident. This releases the interrupts for use by other programs that are subsequently run from the DOS command line. But you must also call EnableFP as the first action when you pop up, and DisableFP just before calling PopDown. This directs the interrupts to your program only while it is popped up.

The code fragment that follows is excerpted from the POPUPFP.BAS example program supplied on the P.D.Q. disk.

```
CALL PopUpHere(&H819, ID$)      'pop up on Alt-P
GOTO EndIt                      'skip over the pop up handler

CALL EnableFP                   'enable floating point interrupts
X! = 1.2                        'F.P. assignments are now okay
Y! = X! * 3.4                   'and so are calculations
PRINT "You pressed Alt-P"       'say hello

CALL DisableFP                  'always disable before popping down
CALL PopDown                    'pop down

EndIt:
  CALL UnHookFP                 'unhook floating point interrupts
  CALL EndTSR(ID$)              'then end as a TSR
```


When this program begins the floating point interrupts have already been taken over. But they are immediately released at the *EndIt:* label before the program ends and stays resident. Then when the program pops up, EnableFP is called. Again, EnableFP takes over the interrupts, and also saves the current state of the 80*x*87 if one is present. When the program is ready to pop down, DisableFP restores the interrupts so the underlying program can use them, and also restores the state of the coprocessor.

Another reason for manually releasing the interrupts when they are not being used is to allow for a complete deinstallation. If the floating point interrupts are pointing to the TSR when PopDeinstall is called, there is no way for PopDeinstall to know this and also release those interrupts. Since the interrupts are not taken over except while the TSR is active, a user can safely deinstall your program at any time.

Floating Point Stub Files

If you link a P.D.Q. TSR program with the _EMONLY.OBJ stub file, the version of HookFP in that file does not check to see if a coprocessor is installed. However, interrupt handing code is employed so you must use UnhookFP, EnableFP, and DisableFP. In truth, you can replace EnableFP and DisableFP with HookFP and UnhookFP respectively. Since you are certain that a coprocessor will not be used, there is no need to add the code and data storage to save and restore its state.

When you use _87ONLY.OBJ the version of HookFP contained therein assumes that a coprocessor is installed, and the emulator code is not added to your program. But interrupts are still used even when a coprocessor is present, so you again need UnhookFP, EnableFP, and DisableFP. Further, EnableFP and DisableFP are required to save and restore the context of the coprocessor.

Finally, when you link with _NOVAL.OBJ, the empty versions of HookFP and UnhookFP it contains simply return immediately without doing anything at all. Therefore, TSR programs that avoid floating point math and are linked with _NOVAL.OBJ do not have to call UnhookInt, EnableFP, or DisableFP, nor should they.

Accessing A Resident Program

There may be situations where you want to access data or code in a TSR program that is already resident. For example, some TSR programs let you modify their current parameters by running them again with a new command line. This technique is used in the ENVELOPE.BAS and PDQBLANK.BAS example programs.

Likewise, you may have code in a TSR that you want to invoke either from a non-resident program, or from another TSR. For instance, Novell's Btrieve includes a TSR that is activated by other programs through CALL INTERRUPT. If you plan to write a TSR that other programs can call in the same way, you will need to establish a custom interrupt handler in the TSR, and then use CALL INTERRUPT to access it.

Because TSRInstalled returns the DGROUP segment of a TSR that is already resident, poking new values into that segment is trivial. The following example is taken from PDQBLANK.BAS, and shows how a subsequent copy of that program assigns a new delay time to the currently resident copy:

```
Delay = PDQVal%(COMMAND$)                'get the delay value
DGroup = TSRInstalled%(ID$)              'already resident?
IF DGroup THEN                           'yes
  DEF SEG = DGroup                       'access the resident data
  CALL PDQPoke2(VARPTR(Delay), Delay)    'assign the new value
  END                                    'end this second copy
END IF
```

The beauty and simplicity of this approach is made possible by the fact that the address of Delay is the same for both the resident copy being modified, and the currently executing subsequent copy. Therefore, although VARPTR(Delay) refers to the address of Delay in the currently running copy, it is the same as the address for Delay in the resident copy. Only the segments are different.

Strings may also be assigned, though it is easier if they are fixed-length. Since conventional strings move around in memory as a program executes, extra steps are needed to first find the descriptor in the resident copy, and then read the string data address from the descriptor. Therefore, ENVELOPE.BAS simply uses a fixed-length string for the file (or device)

name that is to be reassigned. Here's a code snippet adapted from ENVELOPE.BAS that shows how to assign new data into a fixed-length string:

```
DIM SHARED FileName AS STRING * 40  'FileName$ never moves
   ...
   ...
IF DGroup THEN                      'we're already resident
  DEF SEG = DGroup                  'access resident DGROUP
  Temp = VARPTR(FileName$)          'get address of FileName$
  FOR X = 1 TO LEN(FileName$)
    Char = MidChar%(FileName$, X)   'read Char from this copy
    POKE Temp + X - 1, Char         'poke into resident copy
  NEXT
END IF
```


Where SADD is used to get the address of a conventional string's data, VARPTR is meant for use with fixed-length strings. Although MidChar and POKE are quite efficient, you could also use the P.D.Q. BlockCopy routine when many bytes of data must be copied.

Accessing code in a TSR is also quite easy, and the only tricky part is finding an interrupt that is not already in use. The IBM technical reference manuals list Interrupts &H80 through &HF0 as being reserved for the BASIC interpreter, so they are probably safe to use. This is especially true now that GW-BASIC has been replaced by QBasic in DOS 5.

To determine for certain if an interrupt is available you can peek at the interrupt vector table, to see if a given entry points to an empty handler. An empty interrupt handler is one that has an Iret (Interrupt Return) instruction as its first (and thus only) byte of code. A segment and address value of zero in the interrupt vector table also identifies an unused interrupt. The FREEINTS.BAS utility program examines every interrupt in a PC, and reports which are available using these techniques.

P.D.Q. Runtime Reentrance

There are several potential problems that can occur when taking over interrupts in a P.D.Q. program. As you have already seen, DOS is not reentrant, and you may not invoke a DOS service when another one may already be in progress. In a similar vein, the BASIC language routines in the P.D.Q. runtime library are also not reentrant.

Imagine the situation where a P.D.Q. program is in the middle of using, say, LEFT$, and a hardware interrupt occurs passing control to another place in the same program. In this case, the currently executing portion of the program may not use LEFT$, or any of the internal routines that

LEFT$ uses. This seriously restricts what you can and cannot do within a program that takes over hardware interrupts.

The TRAP3.BAS example program shows how to avoid this problem, using a system of flags. TRAP3.BAS takes over three interrupts, and each interrupt handler calls PDQPrint to show that it has received control. But since PDQPrint is not reentrant either, each interrupt handler must test the *other* handlers' busy flags, to be sure that PDQPrint wasn't interrupted. Please understand that this potential problem affects only hardware interrupts, because software interrupts do not occur asynchronously.

See the header comments in TRAP3.BAS for a more complete discussion of this issue. Also see the section *Using PopRequest* that follows, which discusses interrupt conflicts in terms of the real-world example programs that are included with P.D.Q.

Using PopRequest

PopRequest is a major and important feature that lets you perform nearly any DOS or BIOS service from within a manual interrupt handler. Unlike the simplified pop-up TSR method P.D.Q. provides, manual interrupt handlers are limited as to what they can do within an Interrupt Service Routine (ISR).

To use PopRequest you will set up both a simplified PopUpHere hot key handler, and one or more manual interrupt handlers. The hot key handler ensures that it is safe to use any DOS service from within your ISR. When the manual interrupt handler receives control and determines that a file operation or some other usually forbidden service is needed, it uses PopRequest to do the actual work of creating a safe environment for those operations. If you do not want a hot key capability simply specify a key code of 0, and the user will not be able to invoke the pop-up handling code.

Understand that invoking PopRequest does not actually send the program to the hot key handler. Rather, to use protected-mode terminology, it *spawns a thread* that launches an independent, asynchronous action. This thread examines the state of the machine at each timer tick (18.2 times per second). If and when P.D.Q. determines that it is safe, it jumps into the hot key handling code. Therefore, PopRequest always returns to your program immediately, and at some later point in time (usually just a few milliseconds) the hot key handler will receive control.

This is not unlike BASIC's ON TIMER GOSUB and other ON *Event* GOSUB statements, because you in effect say, "*Later on, when the time*

is right, GOSUB to that block of code." A special flag variable is also passed to PopRequest, so your program can determine if it arrived in the hot key code from someone actually pressing the hot key or through a call to PopRequest. We'll get to that in a moment.

Bear in mind that PopRequest simply triggers PopUpHere in the same way that pressing the hot key does in a simplified TSR. When you use PopUpHere and the hot key is pressed, all that really happens is that a flag is set. Then at every timer tick, code in PopUpHere is invoked which checks the various system interrupts to see if it is safe to pass control to the simplified handler portion of the program. Thus, when you call PopRequest the same flag is set, and it is PopUpHere that checks the system interrupts at each timer tick to see if it is safe to pop up. Where PopUpHere continues to try to pop up for one second, PopRequest lets you specify for how long to keep checking.

Because PopRequest launches an asynchronous action, the strategy to implement it may appear a bit tricky at first. Therefore, we'll first show a simple program that traps a single interrupt and also accepts a hot key. A second example shows how to trap two interrupts, and determine which one triggered the call to the simplified hot key handler. Several example programs are also provided that show PopRequest in context, and you should study them carefully. These are APPOINT.BAS, DEMOINT8.BAS, KEY2FILE.BAS, LPT2FILE.BAS, and possibly others that have been added since the printing of this manual.

IMPORTANT:

1. When using PopRequest be sure to install the manual interrupt handlers first (in any order), and then call PopUpHere.

2. You may not call PopRequest from within an interrupt handler that takes over DOS Interrupt &H21.

The following program takes over timer Interrupt 8 and also sets up Alt-J as a hot key. When either event occurs an appropriate message is displayed. This program is supplied on the P.D.Q. disk in the POPREQ1.BAS file.

```
DEFINT A-Z
'$INCLUDE: 'PDQDECL.BAS'

DIM Regs AS RegType
ID$ = "Pop up this program with Alt-J"
PRINT ID$

'-- Set up the manual interrupt 8 handler.
```

```
  Regs.IntNum = 8                 'specify timer interrupt 8
  CALL PointIntHere(Regs)         'trap the interrupt
  GOTO PopUp                      'skip past the Int 8 handler

  CALL IntEntry1                  'we arrive here at each timer tick
  CALL IntEntry2(Regs, 0)         'these are the two mandatory calls
  CALL CallOldInt(Regs)           'first defer to original handler

  Ticks = Ticks + 1               'another 1/18th second has passed
  IF Ticks > 182 THEN             'but has ten seconds passed yet?
    Dummy = PopRequest%(Flag, 18) 'yes, try to pop up for 1 second
  END IF

  CALL ReturnFromInt(Regs)        'all done with this timer tick

  '-- Set up the simplified pop-up handler.
PopUp:
  CALL PopUpHere(&H824, ID$)      '&H824 = Alt-J
  GOTO EndIt                      'skip over and end as a TSR

  '-- The following block of code is executed each time Alt-J
  '   is pressed, and also every 10 seconds.
  CLS
  IF Flag THEN                    'PopRequest sent us here
    PRINT "The timer handler sent me here."
    Flag = 0                      'in case Alt-J is pressed later
    Ticks = 0                     'start a new 10-second period
  ELSE                            'Alt-J was pressed
    PRINT "You pressed Alt-J."
  END IF

  CALL PopDown                   'either way, pop down

EndIt:
  CALL EndTSR(ID$)               'terminate and stay resident
```

The first half of this program looks much like any other P.D.Q. manual interrupt handling TSR, using PointIntHere and ReturnFromInt to define the portion of code that will receive control at each interrupt. However, after calling PointIntHere this program also jumps to install a simplified popup handler, before calling EndTSR to end the program and stay resident.

The interrupt 8 handler counts the number of times it received control from each timer tick. Once ten seconds have passed (182 ticks), it tells PopRequest to try to jump into the simplified hot key handler. Two parameters are passed to PopRequest, and a success code is returned as the PopRequest function output. Now let's look at what the parameters and return code mean.

The first parameter is a flag which is cleared to zero by PopRequest. When PopRequest determines that it is safe to jump to the hot key code, it sets

the flag to -1 (True) just before doing that. Thus, the hot key handler can tell how control arrived there. That is, if the flag is zero then the hot key was pressed. Otherwise, a call to PopRequest was made and PopRequest has set the flag to -1 to show that. Notice that the flag must be cleared manually within the simplified handler. If this is not done, you won't be able to tell how execution arrived there the next time. Although PopRequest does clear the Flag variable automatically, pressing a hot key does not. Therefore, if the flag has been set but never cleared, it will still be set when the hot key is pressed.

Arbitrating Multiple Requests

If more than one manual interrupt handler is using PopRequest, you should use additional variables so that multiple requests can be arbitrated within the hot key handler. For example, you might have variables named Flag1 and Flag2 which are assigned to 0 or -1 within the different interrupt handlers. The example below is expanded from the previous one, and it shows how to trap both the timer interrupt and the keyboard interrupt, so a program can accommodate multiple hot keys. In this case we will detect the Alt-A, Alt-B, and Alt-C keys.

Because the timer and keyboard interrupts can occur at any time and in any order, this next example must also examine the "success" return status of PopRequest. Imagine the following scenario: let's say the 182nd timer tick has just happened, so the Interrupt 8 handler asks PopRequest to invoke the hot key code. But before PopRequest can do that, one of the hot keys the Int 9 handler recognizes is pressed, and that handler also calls PopRequest.

In such a case, which service should PopRequst honor? We have not designed PopRequest to support multiple, nested calls and service each in turn. Therefore, if the Interrupt 8 timer handler calls PopRequest first, then PopRequest will return zero (no success) to the Interrupt 9 handler when it asks for control. This way the Interrupt 9 handler can know that another request is already in the queue, and its own request cannot be processed.

Notice that PopRequest can be executed directly within an IF test, and using a Dummy or Success variable is not really necessary. That is, you could use **IF PopRequest%(Flag9, 18) THEN** or whatever is appropriate.

Also notice that the Flag variable used by the earlier example is not needed, since 0 was used for the hot key. Thus, it is impossible for the pop-up handler to receive control via a hotkey. However, a different set of flags (Flag8 and Flag9) are used to indicate which interrupt passed execution

to the simplified handler. In truth, Flag9 is not really necessary since only two handlers are being used. That is, if Flag8 is not true then the request must have come from Flag9. However, both flags are tested in the next example for clarity. This program is supplied on the P.D.Q. disk in the POPREQ2.BAS file.

```
DEFINT A-Z
'$INCLUDE: 'PDQDECL.BAS'

DIM Reg8 AS RegType
DIM Reg9 AS RegType
ID$ = "Pop me up with Alt-A, Alt-B, or Alt-C"
PRINT ID$


'-- Set up the manual interrupt 8 handler.
Reg8.IntNum = 8                 'specify timer interrupt 8
CALL PointIntHere(Reg8)         'trap the interrupt
GOTO Trap9                      'go install the Int 9 handler
CALL IntEntry1                  'arrive here at each timer tick
CALL IntEntry2(Reg8, Action)
CALL CallOldInt(Reg8)           'first defer to original handler

Ticks = Ticks + 1               'another 1/18th second has passed
IF Ticks > 182 THEN             'but has ten seconds passed yet?
  Success = PopRequest(Flag8, 18)'try to pop up for one second
END IF

CALL ReturnFromInt(Reg8)        'all done with this timer tick

Trap9:
  '-- Set up the manual interrupt 9 handler.
  Reg9.IntNum = 9               'specify keyboard interrupt 9
  CALL PointIntHere(Reg9)       'trap the interrupt
  GOTO PopUp                    'go set up the popup handler

  CALL IntEntry1                'arrive here at each timer tick
  CALL IntEntry2(Reg9, Action)

  IF TestHotKey%(&H81E) THEN    'test for each possible key, and
     KeyHit$ = "Alt-A"          '  assign a string based on which
  ELSEIF TestHotKey%(&H830) THEN '  key it was
     KeyHit$ = "Alt-B"
  ELSEIF TestHotKey%(&H82E) THEN
     KeyHit$ = "Alt-C"
  ELSE
     CALL GotoOldInt(Reg9)      'not our key, defer to the BIOS
  END IF

  CALL ResetKeyboard            'eat the key, clear the hardware
  Success = PopRequest(Flag9, 18) 'try to pop up for one second
  IF NOT Success THEN
     'If PopRequest failed because the timer handler's PopRequest
     '  was already in progress, then ignore it.  But you could
```

```
    ' set other flags here and perhaps try again later.
  END IF

  CALL ReturnFromInt(Reg9)          'all done with keyboard handler


PopUp:
  '-- Set up the simplified pop-up handler.
  CALL PopUpHere(0, ID$)            '0 disables hot key detection
  GOTO EndIt                        'skip over and end as a TSR

  '-- We get here if any of the recognized hot keys are pressed,
  '   and also every ten seconds.
  IF Flag8 THEN                     'if non-zero the timer got us here
    PRINT "The timer handler sent me here."
    Flag8 = 0                       'clear the service flag for later
    Ticks = 0                       'and start a new 10-second period
  ELSEIF Flag9 THEN                 'otherwise Int9 sent us here
    PRINT "You pressed hot key "; KeyHit$
    Flag9 = 0
  END IF

  CALL PopDown                      'either way, pop down


EndIt:
    CALL EndTSR(ID$)                'terminate and stay resident
```

Deinstalling and Unhooking Interrupts

There are two final points you must understand that relate to TSR installation and deinstallation. First, because a program that uses PopRequest is in fact a simplified TSR, you must deinstall it using PopDeinstall, and not DeinstallTSR. DeinstallTSR is meant for use only in programs that do not use PopUpHere. However, any program that takes over interrupts manually with PointIntHere must of course release those interrupts using UnhookInt before removing itself from memory.

Second, programs that take over interrupts manually and also use PopRequest must do these in the correct sequence. When installing the program, call PointIntHere to take over all of the interrupts and then call PopUpHere. When deinstalling, call PopDeinstall first, and then call UnhookInt to release the interrupts taken over earlier with PointIntHere. If you try to release the interrupts before calling PopDeinstall, UnhookInt will fail because those interrupts have since been taken over by PopUpHere. Of course, this applies only when you take over one or more of the interrupts that PopUpHere (and thus PopRequest) also use. These interrupts are 8, 9, 10, 13, 14, 16, 17, 21, 25, 26, and 28.

The DOSWATCH Example Program

The supplied DOSWATCH program shows how to intercept DOS Interrupt 21h, and it also serves as a good example of writing a general purpose interrupt handler. DOSWATCH is a TSR program that monitors every call to Interrupt 21h, and then displays information about the current service that is being requested. The service number is printed in the upper left corner of the screen, and in many cases additional information is also shown. Watching DOS as it works can be very enlightening, and DOSWATCH lets you view not only the activity of your application programs, but also DOS itself. We'll assume that you have a copy of the DOSWATCH.BAS source listing in front of you, or are viewing it on the screen as you read this.

As with most P.D.Q. programs, DOSWATCH begins by including PDQDECL.BAS, which declares all of the available P.D.Q. extensions and defines the Registers TYPE variable. Next, a unique ID string is defined, which in this case is also the sign-on message. The next block of code checks for a previous installation, or a request to remove DOSWatch from memory. After that, the sign-on message is displayed using PDQPrint.

Because DOSWATCH may receive control at any time, it is essential that the regular BASIC PRINT statement is not used. Unlike the simplified TSR method which allows nearly any BASIC statement to be used freely without regard to the current state of DOS or the BIOS, DOSWATCH must use PDQPrint which writes directly to video memory.

PDQPrint uses the BIOS to know what type of monitor is installed, however it does this only once the first time it is called. Because PDQPrint is also used to print the sign-on message, this ensures that the BIOS call is performed when DOSWATCH is installed. Where BASIC's PRINT always displays at the current cursor location, PDQPrint expects the row and column as passed parameters. Therefore, we must use CSRLIN and POS(0) to know where that is.

The next two statements define the strings that will receive the information message later, and the file or directory name when appropriate. The statements that follow establish several variables that are used by the program. Note that using variables as parameters is always faster than constants, because BC.EXE adds code to store constants in memory each time they are used. Likewise, assigning Zero$ once eliminates repeated references to CHR$() each time INSTR is used later on in the program.

Although it may not be obvious, using **INSTR(DOSName$, Zero$)** is faster than **INSTR(DOSName$, CHR$(0))**, because CHR$() is actually a called routine. A few microseconds either way is unlikely to matter in most programs, but in a TSR that steals interrupt processing time, speed is generally more important than program size. (Defining and assigning Zero$ adds a dozen or so bytes.)

The last preparatory step is to assign Registers.IntNum to the value &H21, to specify which interrupt to monitor. Finally, PointIntHere is called, followed by a GOTO to the very end of the source listing where the program is installed as a TSR.

The remaining program statements are executed every time Interrupt &H21 is invoked. The first two steps call IntEntry1 and IntEntry2, and this is mandatory in all TSR programs that do not use the simplified method. These routines simply save the current processor registers values, so they can be restored when DOSWATCH passes control to DOS later on.

The next two steps clear the Ticks variable, and derive the current service number from the AX register. For DOS services that process a file or directory name, DOSWATCH pauses for a half-second allowing time to read the name. Ticks specifies the number of timer ticks in 18ths of a second. For other services, only the service number is displayed, and there is no added delay.

Fifteen different DOS services are reported, and these are filtered through a SELECT/CASE block. All of the supported services assign Message$ to the appropriate text, and those services that process a file or directory name also use the GetDOSName subroutine. GetDOSName copies the current name into the DOSName$ variable, and also sets Ticks to 8 to allow time to read it.

In some cases, additional information is assigned to Message$, for example when reporting a drive letter or handle number. This information is taken from the appropriate registers as necessary. If a particular service is not supported, then Message$ is simply cleared to blanks in the CASE ELSE code.

Because DOSWATCH is not actually processing any of the DOS services, the final step is to call GotoOldInt, which in turn jumps to the internal DOS function dispatcher. Thus, DOSWATCH displays what is about to happen, before DOS actually receives control.

The remaining statements comprise the GetDOSName subroutine, which copies the current file or directory name into the DOSName$ variable. In this case, BlockCopy copies the first 50 characters pointed to by DS:DX into DOSName$, and then INSTR is used to locate the CHR$(0) which marks the end of the name. Only those characters that precede the zero byte are retained, and the LEFT$ assignment clears any characters that follow the name. (The unused characters are cleared because assignments to a fixed-length string are treated as if LSET were used.)

Running DOSWATCH once installs it as a TSR, and all subsequent DOS operations are then displayed on the top line of the screen. Observing DOSWATCH as it works can provide much insight into DOS' internal operation. For example, you will no doubt find it enlightening to start QuickBASIC and load a program such as DOSWATCH itself. This shows DOS at work as it loads and executes QB.EXE, and then you can watch QuickBASIC as it loads the main program and the PDQDECL.BAS $INCLUDE file. Equally interesting is all the unnecessary DOS activity QuickBASIC performs to obtain a list of file names when you select Alt-F-L from the pull-down menu.

You could also modify DOSWATCH to pause briefly for all of the DOS services rather than just some of them. Even though the operation of your PC will be slowed dramatically, many varied and interesting facets of DOS become apparent. For example, each time you use the DIR command, DOS (actually COMMAND.COM) changes to the current directory, searches for all names that match "*.*", and then writes the directory information to Handle 1. (Handle 1 is the standard console output device.) An added delay also lets you observe DOS as it closes all available handles each time a program terminates.

As you can see, besides providing an excellent way to learn more about TSR programming, DOSWATCH is also a very useful utility program in its own right.


TSR
Programming

Section II, Chapter 5: P.D.Q. Extensions


P.D.Q.
Extensions

Overview

This section provides a brief overview of the P.D.Q. language additions; a more detailed description of each is given in the reference portion that follows. Besides the actual extension routines, P.D.Q. also comes with a number of useful example programs including a pop-up calculator, a directory changing utility that features a scrolling menu and highlight bar, and a unique program called DOSWATCH that lets you see DOS as it works. Many other useful and informative examples are included as well.

DOS Extensions

BufIn is a very fast LINE INPUT replacement for use with sequential text files. By default, P.D.Q. programs perform no file buffering when reading, which makes INPUT # and LINE INPUT # relatively slow. BufIn more than compensates for that, and is approximately four times faster than BASIC's LINE INPUT.

CritErrOn and CritErrOff let you enable and disable critical errors during DOS operations. Internally, DOS handles critical errors separately from normal errors, reporting them with the familiar "Abort, Retry, Fail" message. Some examples include attempting to read an unformatted disk, or writing to a printer that is not available.

DOSVer allows your program to determine the version of DOS that is currently running on the host PC. This is useful for programs that perform network operations, or use other services which require a particular DOS version.

Calling the EndLevel routine is an optional way to end a P.D.Q. program, and set the DOS ERRORLEVEL value at the same time. This lets you communicate with batch files, which can then make decisions based on the success or failure of the program. However, if you have BASIC 7 PDS you can simply use END *level*.

The EnvOption routine allows your programs to manipulate either their own environment, the environment of their parent, or the environment of the current application from a TSR. EnvOption also lets you selectively honor or ignore capitalization. These are powerful capabilities which are sorely lacking in regular BASIC, as well as in other high-level languages.

Flush is a new "command"" that lets you flush the DOS file buffers to disk without having to close and then re-open them. This lets you ensure that data which has been written is safely on the disk.

The PDQExist function quickly reports if a file exists, to eliminate errors in programs that manipulate files.

SeekLoc is meant for use with the SMALLDOS.LIB stub library when accessing random files, and it calculates the appropriate byte offset based on a specified record number and record length.

Dynamic Memory Allocation

Dynamic memory allocation is supported with the AllocMem and ReleaseMem routines, and these are based on standard DOS memory services. Memory may then be used for any purpose, such as storing the contents of the display screen. Memory that has been allocated manually may also be treated as an array using the following routines.

Get1Byte, Get1Word, Get1Long, and Get1Type all accept a segment and element number, and return the appropriate value from memory. The complementary routines Set1Byte, Set1Word, Set1Long, and Set1Type assign byte, integer, long integer, and TYPE variables.

These routines are useful for two reasons:

- They can be used to treat any arbitrary block of memory as an array.
- They generally require less code than conventional array accesses.

If you dimension a dynamic array in a P.D.Q. program, approximately 850 bytes of code is added to the .EXE file. REDIM must be able to accommodate all of BASIC's data types, which adds to its complexity. Further, using REDIM also brings in ERASE, since REDIM must first erase the array before recreating it. In contrast, AllocMem, ReleaseMem, and the various Get1/Set1 routines occupy only a few dozen bytes each.

Input And Keyboard Routines

Several INKEY$ and INPUT replacements are provided, to improve the efficiency of your programs. Where INKEY$ returns a string with different lengths to tell what type of key was pressed, PDQInkey instead returns an integer value. Using an integer function not only reduces the amount of code added to your programs, but it is also more flexible.

BIOSInkey is similar to PDQInkey, but it uses the BIOS rather than DOS which is safer within a TSR program. BIOSInput is a small input/editing routine which also lets you control the length of the string being entered.

BIOSInput2 is an enhanced version that adds capability in exchange for an increase in code size.

The StuffBuf routine is used to insert characters into the keyboard buffer. This lets your programs run batch files and perform other tricks that conventional BASIC cannot. StuffBuf is provided in lieu of BASIC's RUN command.

Miscellaneous Routines

The BlockCopy routine lets you move a block of memory anywhere within the PC's address space. It is ideal for copying screens to far (DOS) memory, as well as for other general purpose memory moves.

BreakOn and BreakOff allow you to disable and enable the Ctrl-Break and Ctrl-C keys. BreakHit can then be used to report how many times Ctrl-Break or Ctrl-C were pressed while they were disabled.

GetCPU returns the type of CPU that is installed on the host PC. It was originally written for our QuickPak Professional product, but we thought P.D.Q. users would also find it beneficial.

GetSeg lets you retrieve the current DEF SEG setting in a program. This lets you write reusable modules that are "well behaved", by restoring the original DEF SEG setting to what it had been.

HookInt0 is an interrupt handler designed to prevent "Divide by zero" errors from crashing your programs. Although most errors that occur in a P.D.Q. program are simply ignored, dividing by zero generates a CPU exception that tells DOS to end your program. But if that program is a TSR, then the result is a hung computer.

The Pause routine provides a way to add delays to a P.D.Q. program with a resolution of approximately 1/18th second.

PDQCompare reports if two blocks of memory are the same.

PDQMessage is a string function that returns an error message for any of the error numbers that are supported by P.D.Q. For example, given the value 53 it will return the message "File not found".

PDQPeek2 and PDQPoke2 let your programs PEEK and POKE words (two bytes) in one operation, resulting in much less code than using BASIC commands alone.

PDQRand is an integer-only random number function, and PDQRandomize lets you seed the initial value. These are provided as an alterative to the BASIC versions, which add the floating point library routines to your programs.

PDQShl and PDQShr let you shift integer bits left and right.

PDQSound is a direct replacement for BASIC's SOUND command, except it does not require the use of BASIC's floating point library.

PDQTimer returns the current setting of the system timer double-word kept in low memory. It is similar to BASIC's TIMER function but does not require floating point support.

PDQValI and PDQValL are for obtaining the value of a BASIC string, for integer and long integer values respectively.

Power and Power2 let you raise integer numbers to a power, without requiring floating point math as BASIC's exponentiation operator (^) does.

RedimAbsolute is used to reassign a BASIC array to and arbitrary segment in memory.

String Handling Routines

To accommodate dollar amounts in a P.D.Q. program the Dollar$ function accepts a long integer that holds the number of pennies, and returns a string formatted in dollars and cents. That is, the value 12345 is returned as the string "123.45"".

FUsing is a numeric formatting function that provides most of the services of BASIC's PRINT USING statement. However, it is more versatile because it returns the result as a string that you can then manipulate in any way necessary.

MidChar and MidCharS are integer-only replacements for BASIC's MID$ function and statement respectively. When you need to access only a single character in a string, these routines are much faster and add less code than MID$.

PDQParse is a small-code replacement for BASIC's READ and DATA statements. It is also useful for breaking up a string such as COMMAND$ into separate components. PDQRestore is meant to be used with PDQParse, and it mimics BASIC's RESTORE command. Another related

routine, SetDelimitChar, lets you set the recognized delimiting character that separates each item.

PoolOkay provides a simple way to test the integrity of string memory. Since a P.D.Q. program does not halt when an error occurs, this function lets you detect if string memory has become corrupted. PoolOkay is intended primarily as a debugging aid.

Two functions are provided to help you assess the string memory needs of your programs. StringUsed returns the number of bytes currently in use by string variables, and StringShort tells if your program ever needed more string memory than is available. Most programs do not have to deal with string memory, unless they must take as little as possible such as in a TSR.

An assembler string sort routine, Sort, is included for sorting all or part of a conventional (not fixed-length) string array in either ascending or descending order.

TSR And Interrupt Support Routines

Interrupt and InterruptX let you access system interrupts from within a P.D.Q. program. These are similar, but not identical, to the routines with the same name that are provided with Microsoft BASIC.

A number of subroutines and functions are provided for creating TSR pop-up programs and interrupt handlers. PopUpHere and PopDown provide a simplified method for creating a TSR that pops up via a hot key. Other routines allow you to gain complete control over any system interrupt, and these include PointIntHere, CallOldInt, GotoOldInt, ReturnFromInt, and PopRequest.

DeinstallTSR and PopDeinstall are used to remove a TSR program from memory, and TSRInstalled reports if a program is already resident. A related routine, UnhookInt, lets you manually remove an interrupt handler from the PC's interrupt chain.

DOSBusy lets you determine if DOS is currently active, to prevent a conflict from accessing a DOS service when it is not safe to do so.

EndTSR is used to end a program and leave it resident in memory.

EnableFP, DisableFP, HookFP, and UnhookFP are needed in P.D.Q. TSR programs that use floating point math, to prevent a conflict with the foreground program.

Swap2Disk and Swap2EMS let a TSR program remove itself from memory when idle. SwapCode provides a way for a swapping TSR to receive information from another application.

TestHotKey and ResetKeyboard let a TSR program manipulate the keyboard hardware port and PIC chip (programmable interrupt controller) directly and easily.

TSRFileOn and TSRFileOff are used to make DOS file operations safe from within a P.D.Q. TSR program.

Video Routines

ColorSave and ColorRest let you easily save and restore the current foreground and background colors. They are ideal when writing reusable modules that change the current colors, but must be able to restore them to their previous, but unknown, values.

CursorOn and CursorOff turn the cursor on and off using less code than LOCATE.

CursorSave and CursorRest let you save and restore the current cursor position and shape in a single operation. Like ColorSave and ColorRest above, they are ideal when your program has changed the cursor, and must restore it again later.

CursorSize lets you control the cursor size using less code than LOCATE.

HercMode lets you switch to Hercules graphics mode and back, and is equivalent to using SCREEN 3 and then SCREEN 0 in BASIC.

NoSnow disables CGA "snow suppression" for faster access with CGA adapters.

PDQMonitor reports the type of video adapter installed in the PC.

PDQPrint is a "quick printing"" routine that accepts a string, a row and column, and a print color. PDQPrint is extremely fast and doesn't access the BIOS, making it especially suitable for use in TSR programs.

PDQCPrint is another quick printing routine, but it uses the current COLOR setting. This eliminates an extra passed parameter, thus creating less code when it is called many times with the same color value.

PDQSetMonSeg lets you set any arbitrary video segment for use by PDQPrint and PDQCPrint. This makes it easy to write well-behaved applications that run under Quarterdeck's DESQview. Being able to set a new segment also provides the ability to create virtual screens stored in an array.

PDQSetWidth lets PDQPrint and the other routines that access video memory directly accommodate screen widths other than 80 characters.

Extensions Details

All of the subprograms and functions described herein are contained in the PDQ.LIB, PDQ.QLB, and PDQ7.QLB library files. They are also declared in the PDQDECL.BAS include file. BASIC equivalents of some of the extensions are in the PDQSUBS.BAS program module. You should therefore include the PDQDECL.BAS file in your programs that will use these extensions, and load PDQSUBS.BAS as a module.

Note that PDQSUBS.BAS is needed only during development in the QuickBASIC environment. If you plan to compile from within the QuickBASIC environment, you must unload PDQSUBS.BAS first. All of the P.D.Q. extension routines contained in PDQSUBS.BAS are listed in Table V-1. Some of the routines are empty, such as those for TSR and interrupt handling services.

To access routines in a Quick Library you need to tell QuickBASIC to load it by using the /l option on the QB command line:

```
QB [program] /l pdq
```

If you are using BASIC 7 and QBX then you will instead specify PDQ7.QLB like this:

```
QBX [program] /l pdq7
```

If you are already using another Quick Library, you will need to combine those routines with the P.D.Q. routines into a new Quick Library. Although Quick Libraries may not be combined directly, you can easily create a new Quick Library. This requires access to the original object modules or linking library (with an .LIB extension). See the section entitled *Creating A Quick Library* elsewhere in this manual. Also see the documentation that comes with Microsoft BASIC for further information about creating Quick Libraries.

TABLE V-1
Routines Contained In PDQSUBS.BAS

AllocMem	BreakHit	BreakOff	BreakOn
BufIn$	CallOldInt	CritErrOff	CritErrOn
DeInstallTSR	DisableFP	Dollar$	DOSBusy
EnableFP	EnableRead	EndLevel	EndTSR
EnvOption	GotoOldInt	HookFP	HookInt0
IntEntry1	IntEntry2	PDQCPrint	PointIntHere
PoolOkay	PopDeinstall	PopDown	PopUpHere
ReleaseMem	ResetKeyboard	ReturnFromInt	StringShort
StringUsed	TestHotKey	TSRFileOff	TSRFileOn
TSRInstalled	UnhookFP	UnhookInt	UnhookInt0

AllocMem function

Purpose

AllocMem calls on the standard DOS services to allocate up to 64K of memory when a program is running.

Syntax

```
Segment = AllocMem%(NumBytes%)
```

Where

NumBytes% is the number of bytes to be allocated, and Segment receives the segment where that memory begins. The first available address within that segment is always 0. If DOS cannot allocate the requested amount of memory, Segment will receive a value of zero. Notice that if NumBytes% is larger than 32K (32,767), it must be specified as an equivalent negative value. A long integer argument may also be used if that is more convenient.

Comments

Because AllocMem has been designed as a function, it must be declared before it may be used.

One important advantage of allocating memory at run time is to reduce the size of the .EXE program file. Even though DIM could be used within a program to reserve memory for screens and other data, that memory would be included within the .EXE file. Although REDIM (or DIM with a dynamic array) serves a similar purpose, AllocMem comprises less code than REDIM. Further, when an undefined number of memory blocks is needed, AllocMem avoids having to create multiple named arrays.

Memory that has been allocated may be treated as either a byte, integer, long integer, fixed-length string, or TYPE array. The example below requests 4,000 bytes for saving an entire text screen image, and then uses the P.D.Q. BlockCopy routine to copy the screen there.

```
Segment% = AllocMem%(4000)        'request 4000 bytes
IF Segment% = 0 THEN              'see if there was an error
  PRINT "Error allocating memory"
   END
END IF

DEF SEG = 0                       'get the monitor type in low
                                  '  memory
IF PEEK(&H463) = &HB4 THEN
  ScreenSeg% = &HB000             'mono monitor
 ELSE
  ScreenSeg% = &HB800             'color monitor
END IF
                                  'copy the screen
CALL BlockCopy(ScreenSeg%, 0, Segment%, 0, 4000)
```

Memory that is allocated using AllocMem does not have to be released before a program is ended. DOS remembers all memory allocations, and will release the memory automatically. Further, any memory that is allocated by a TSR program before it ends and remains resident will be protected by DOS and thus be available when the TSR gets control. However, it is imperative that you request any memory your program will need before calling EndTSR. Most DOS applications claim all available memory when they load, thereby preventing a TSR program from allocating additional memory for itself when it pops up or receives control through a system interrupt.

Even though a single call to AllocMem may specify no more than 64K (65,535) bytes, AllocMem may be called more than once.

IMPORTANT:

If AllocMem is unable to allocate the requested amount of memory, it sets ERR to 7 (Out of memory), and returns a segment value of zero. Do not attempt to write data to segment zero under any circumstances, since this will overwrite the PC's interrupt vector table.

Also see the ReleaseMem function which releases memory allocated by AllocMem.

BIOSInkey function

Purpose

BIOSInkey is similar to BASIC's native INKEY$ function, except it returns an integer result.

Syntax

```
KeyHit = BIOSInkey%
```

Where

KeyHit receives 0 if no key is pending in the keyboard buffer, a positive number that represents a normal key's ASCII code, or a negative value that corresponds to an extended key code.

Comments

Because BIOSInkey has been designed as a function, it must be declared before it may be used.

Unlike INKEY$ and the PDQInkey function, BIOSInkey does not support DOS redirection. However, it may be used safely in a "simplified" TSR program.

In general, integer functions such as BIOSInkey require less setup and processing by the BASIC compiler than do string functions. Also, integer comparisons are much faster and require less code than string comparisons.

Please see the PDQInkey routine which also returns an integer result, but supports DOS redirection.

BIOSInput subroutine

Purpose

BIOSInput is a general purpose text input routine which also allows editing an existing string.

Syntax

```
CALL BIOSInput(Work$, EditColor%)
```

Where

Work$ is the string being input or edited, and EditColor% is the field color to use. Work$ must have already been assigned before BIOSInput may be called. The left-most portion of the input field is placed at the current cursor location.

Comments

BIOSInput provides simple editing capabilities using the left and right cursor arrow keys only. (The backspace key is handled as a left arrow.) Pressing either Enter or Escape terminates editing.

Like BIOSInkey, BIOSInput does not support redirection. However, it may be safely used at any time within a "simplified" TSR program.

The maximum string length that may be entered using BIOSInput is dictated by the length of the string that is passed to it. For example, to input a string and limit its length to, say, fifteen characters, you must first assign a string to that length. This is shown in the example below.

```
Work$ = "               "     '15 spaces
CALL BIOSInput(Work$, 112)    '112 is black on white (inverse)
```

or

```
Work$ = "This is a default response"
CALL BIOSInput(Work$, 112)
```

Because BIOSInput uses PDQPrint to display the string being input, you must first call the PDQSetWidth routine if your program has selected the 40-column video mode using WIDTH.

Note that BIOSInput can also be declared as an integer function. When used this way the return value is either 13 if Enter was pressed, or 27 if Escape was used:

```
DECLARE FUNCTION BIOSInput%(Work$, EditColor%)
LastKey = BIOSInput%(Work$, EditColor%)
```

Also see the PDQInput routine which supports redirection, but not an editing color or the ability to edit an existing string.

BIOSInput2 function

Purpose

BIOSInput2 is an enhanced version of BIOSInput that adds more capability, but also more code to your programs. In particular, BIOSInput2 also recognizes the Home, End, Ins, and Del keys.

Syntax

```
LastKey = BIOSInput2%(Work$, BYVAL Row%, BYVAL Column%, _
  BYVAL EditColor%)
```

Where

Work$ is the string being input or edited, Row% and Column% specify the left edge of the input field, and EditColor% is the field color to use. LastKey then receives either 13 if the Enter key was pressed to terminate editing, or 27 if Escape was used. Work$ must have already been assigned before BIOSInput2 may be invoked.

Comments

Because BIOSInput2 has been designed as a function, it must be declared before it may be used.

Like BIOSInput, BIOSInput2 does not support DOS command-line redirection. However, it may be safely used at any time within a "simplified" TSR program.

The maximum string length that may be entered using BIOSInput2 is dictated by the length of the string that is passed to it. You can use either SPACE$ to initialize the string, a quoted string constant, or both. The example below lets the user edit default text and increase its length by up to 10 characters:

```
Work$ = "This is a default response" + SPACE$(10)
LastKey = BIOSInput(Work$, 1, 10, 112)
```

Like BASIC's INPUT statement, BIOSInput2 saves the current cursor state, turns the cursor on if necessary, and restores it to what it had been afterward.

BIOSInput2 may also be declared as a subprogram if the value of the last key pressed is not important in your program. PDQDECL.BAS includes an additional DECLARE SUB statement as a comment showing BIOSInput2 declared that way.

Comments in the assembler source code show how to have BIOSInput2 exit when an extended key is pressed, instead of ignoring the key. This will let the program exit if the user presses, say, the F1 key for help. If you make that modification, extended key codes are represented as negative values.

Because BIOSInput2 uses PDQPrint to display the string being input, you must first call PDQSetWidth if your program has selected the 40-column video mode using WIDTH.

Also see the BIOSInput routine which has fewer capabilities but adds less code to your program.

BlockCopy subroutine

Purpose

BlockCopy copies a block of memory between any locations within the PC's address space.

Syntax

```
CALL BlockCopy(FromSegment%, FromAddress%, ToSegment%, _
  ToAddress%, NumBytes%)
```

Where

FromSegment% and FromAddress% indicate the source memory to copy, and ToSegment% and ToAddress% indicate where it is to be copied to. NumBytes% is the number of bytes to copy, and may not exceed 64K (65,535).

Comments

BlockCopy is useful in a variety of situations, for example when saving or restoring portions of the display screen. Other uses include moving data from "far" memory into a conventional string for printing, and copying the contents of one entire numeric or TYPE array to another very quickly.

For values of NumBytes% larger than 32K (32,767), an equivalent negative value must be used. A long integer may also be used for NumBytes% if that is more convenient.

BlockCopy is not suitable for copying ranges of memory that overlap one another.

BreakHit function

Purpose

BreakHit returns the number of times Ctrl-Break or Ctrl-C were pressed since BreakOff was last called, or since the last time BreakHit was queried.

Syntax

```
NumTimes = BreakHit%
```

Where

NumTimes receives the number of times Ctrl-Break or Ctrl-C were pressed.

Comments

Because BreakHit has been designed as a function, it must be declared before it may be used.

There are many occasions when a program needs to disable the Ctrl-Break key, for example when using the PDQInput routine to prompt for, say, a file name. If Break is not disabled, then the user would be able to terminate the program inadvertently. But if someone really wants to break out of the program, you will certainly want to know that. Therefore, BreakHit lets you see if the Ctrl-Break or Ctrl-C keys were pressed while they were disabled.

Also see the BreakOff and BreakOn routines which disable and reenable Break.

BreakOff subroutine

Purpose

BreakOff disables the action of the Ctrl-Break and Ctrl-C keys until they are re-enabled with a call to BreakOn.

Syntax

```
CALL BreakOff
```

Where

Ctrl-Break and Ctrl-C are disabled until a call is made to the BreakOn subroutine.

Comments

Normal break operation must be explicitly restored by calling the BreakOn subroutine before the program terminates, or the PC will crash.

Each time BreakOff is called, it resets the BreakHit function count to zero.

To intercept Break from inside a TSR, BreakOff must be called each time the TSR becomes active, and BreakOn must be called before returning control to the foreground program.

Also see the comments that accompany the BreakHit function.

BreakOn subroutine

Purpose

BreakOn re-enables the Ctrl-Break and Ctrl-C keys after a call to BreakOff.

Syntax

```
CALL BreakOn
```

Where

Ctrl-Break and Ctrl-C are re-enabled.

Comments

Calling BreakOn when BreakOff has not been called has no effect.

Also see the comments that accompany the BreakHit function.

BufIn function

Purpose

BufIn is a very fast LINE INPUT replacement for reading text data from a sequential file.

Syntax

```
Text$ = BufIn$(FileName$, Done%)
```

Where

FileName$ is the name of the file to read, Done% is returned as either 0 or -1 to indicate if all text in the file has been read, and Text$ receives the current line from the file.

Comments

Because BufIn$ has been designed as a function, it must be declared before it may be used.

BufIn is much faster than the default P.D.Q. LINE INPUT routine, and also about four times faster than QuickBASIC's LINE INPUT. It is extremely easy to use, since it opens and closes the file automatically. However, only one file may be processed using BufIn at a time, and it is not intended for returning strings that are longer than 4095 bytes.

Declare and use BufIn as shown in the short, complete program below.

```
DECLARE FUNCTION BufIn$ (FileName$, Done%)
FileName$ = "WHATEVER.DAT"
DO
```

```
    This$ = BufIn$(FileName$, Done%)        'read the file
    PRINT This$                             'optionally print the data
  LOOP UNTIL Done%                          'until BufIn$ says it's done
```

Normally you will use BufIn to read a file in its entirety. You can tell BufIn to close the file and set Done% to -1 by calling it with a null string as the file name.

Note that BufIn does not set the Done% flag until it is called one more time after the file has been completely read. Therefore, it always returns an extra false null string at the end. You can avoid this by placing the test within the loop as follows:

```
DO
  This$ = BufIn$(FileName$, Done%)
  IF Done% THEN EXIT DO
  PRINT This$
LOOP
```

You can also use the following, somewhat tricky method of declaring and using BufIn, to reduce the amount of compiler-generated code and minimize string memory usage:

```
DECLARE FUNCTION BufIn% (FileName$, Done%)
DECLARE SUB Assign ALIAS "B$SASS" (BYVAL Source%, Dest$)
DO
  CALL Assign(BufIn%(FileName$, Done%), This$)
  PRINT This$
LOOP UNTIL Done%
```

Normally, the output of all external string functions is assigned to a temporary string, before being assigned to the final destination variable. By declaring BufIn as an integer function you are telling BASIC not to add code to create the additional copy. BASIC's internal string assignment routine, B$SASS, is then declared using ALIAS. ALIAS is needed to allow the otherwise illegal dollar sign in a procedure name.

When BufIn is declared as an integer function BASIC generates 29 bytes of code, compared to 36 when it is declared and invoked as a string function. An important side benefit is that only as much string space as is really needed is taken, rather than twice as much for an extra copy.

A BASIC equivalent of BufIn is provided in PDQSUBS.BAS for use in the QuickBASIC and QBX editors. In fact, the BASIC version also works with regular QuickBASIC and BASIC PDS, although the assembly language version contained in PDQ.LIB does not.

CallOldInt subroutine

Purpose

CallOldInt is used within a P.D.Q. TSR or interrupt handler program to call the original interrupt as a subroutine.

Syntax

```
CALL CallOldInt(Registers)
```

Where

Registers is the TYPE variable being used to service the specified interrupt.

Comments

Depending on what a TSR is intended for, there may be situations where you need to call the original interrupt handler. For example, a printer driver might be designed to intercept the printer stream, looking for a special code that indicates a macro to be expanded. In that case, the program couldn't simply use Int 17h to print the replacement string, because that would invoke itself again! Therefore, CallOldInt would be called repeatedly for each character to be sent to the printer, before the TSR returns control to the underlying application.

ColorRest subroutine

Purpose

ColorRest restores the current foreground and background colors that were obtained earlier using ColorSave, as a single operation.

Syntax

```
CALL ColorRest(SavedColor%)
```

Where

SavedColor% is the value that was returned by ColorSave earlier.

Comments

ColorSave and ColorRest allow you to retrieve the current COLOR settings as a single parameter, and then restore them again later. This is useful for writing reusable modules that need to change the colors and restore them, when the original colors are not known. Because regular BASIC provides no way to determine the current COLOR values, this routine is not available in the QuickBASIC editing environment.

Also see the complementary ColorSave function.

ColorSave function

Purpose

ColorSave obtains the current foreground and background colors that were set the last time COLOR was used.

Syntax

```
SavedColor% = ColorSave%
```

Where

SavedColor% receives the current foreground and background colors combined in a single integer word.

Comments

Because ColorSave has been designed as a function, it must be declared before it may be used.

BASIC provides no way to determine the current COLOR values; therefore, this routine is not available in the QuickBASIC editing environment.

Also see the comments that accompany the ColorRest function.

CritErrOff subroutine

Purpose

CritErrOff lets you disable critical error handling, to avoid the DOS "Abort, Retry, Ignore" error message.

Syntax

```
CALL CritErrOff
```

Where

Critical errors will be trapped until a subsequent call is made to the P.D.Q. CritErrOn routine.

Comments

DOS handles critical errors separately from normal errors such as "File not found". Critical errors are generated by hard I/O faults such as attempting to read an unformatted disk, or accessing a floppy drive when the door is open. Critical errors are also caused by trying to write to a write-protected disk, as well as attempting to access a printer that is off-line or otherwise unavailable.

CritErrOff establishes a replacement critical error handler, which takes control of the critical error vector (Interrupt &H24). When a critical error occurs and CritErrOff is in effect, the error is not passed on to DOS. Rather, CritErrOff assigns BASIC's ERR function to a value of 71 (disk not ready) which you may then test.

Calling CritErrOff when it has already been activated has no effect.

CritErrOff should be used in TSR programs before performing any disk or device I/O, to prevent an "Abort" response from terminating both the TSR and the foreground program. Further, if the underlying application has its own critical error handler (most do), then a critical error in your TSR would hopelessly confuse both programs and undoubtedly crash the PC.

If CritErrOff is used in a TSR, CritErrOn must be called before returning control to the foreground program.

Neither CritErrOff nor CritErrOn may be used within the QuickBASIC editor, because they access the internal P.D.Q. error handler.

Also see the section *File Handling In P.D.Q.*, for more information about critical error.

CritErrOn subroutine

Purpose

CritErrOn re-enables critical error handling that had been disabled by a call to CritErrOff.

Syntax

```
CALL CritErrOn
```

Where

Critical errors are reinstated, causing DOS to generate the familiar "Abort, Retry, Ignore" message. In a TSR, CritErrOn returns control to the underlying program's error handler.

Comments

Calling CritErrOn when CritErrOff has not been called has no effect.

Please see the comments that accompany the CritErrOff routine.

CursorOff subroutine

Purpose

CursorOff turns off the cursor making it invisible.

Syntax

```
CALL CursorOff
```

Where

The cursor is hidden until turned on again with a call to the CursorOn routine.

Comments

Calling CursorOff is equivalent to using **LOCATE , , 0**.

CursorOff is necessary when you are linking with the limited-functionality _LOCATE stub file. Using that version of LOCATE reduces the size of your programs, but it does not allow the options to turn the cursor on and off, or to change its size. By using CursorOff in conjunction with the _LOCATE stub file, only the code that is actually needed will be added to your program.

Please see the section entitled *Linking With Stub Files* for more information about this and other alternate language statements. Also see the CursorOn routine which turns the cursor on.

CursorOn subroutine

Purpose

CursorOn turns on the cursor making it visible.

Syntax

```
CALL CursorOn
```

Where

The cursor is made visible after a previous call to the CursorOff routine.

Comments

Calling CursorOn is equivalent to using **LOCATE , , 1**.

Please see the comments that accompany the CursorOff routine.

CursorRest subroutine

Purpose

CursorRest restores the current cursor location and size that were obtained earlier using CursorSave.

Syntax

```
CALL CursorRest(SavedCursor&)
```

Where

SavedCursor& is the value that was returned earlier by CursorSave.

Comments

CursorSave and CursorRest allow you to retrieve the current cursor location and size as a single parameter, and then restore them again later. This is useful when writing reusable modules that need to change the cursor parameters and restore them, when the original values are not known. Although BASIC provides the CSRLIN and POS(0) function to obtain the current location, there is no way to obtain the cursor shape or on/off state. Further, using these routines results in much less code because only one call is needed and only one parameter is used.

Also see the complementary CursorSave function.

CursorSave function

Purpose

CursorSave obtains the current cursor location and size in a single operation.

Syntax

```
SavedCursor& = CursorSave&
```

Where

SavedCursor& receives the current cursor location and size combined into a single long integer.

Comments

Because CursorSave has been designed as a function, it must be declared before it may be used.

Please see the comments that accompany the CursorRest routine.

CursorSize subroutine

Purpose

CursorSize lets you control the size (in scan lines) of the cursor.

Syntax

```
CALL CursorSize(TopLine%, BottomLine%)
```

Where

TopLine% and BottomLine% indicate the start and stop cursor scan lines respectively. As with BASIC's LOCATE command, the scan line values may range from zero to the highest legal value supported by the installed display adapter.

Comments

CursorSize is equivalent to **LOCATE , , , TopLine%, BottomLine%.**

Please see the comments that accompany the CursorOff routine.

DeinstallTSR function

Purpose

DeinstallTSR is called by a non-simplified TSR program to remove itself from memory.

Syntax

```
Success = DeinstallTSR%(DGroup%, ID$)
```

Where

DGroup% was returned by TSRInstalled when the program was first run, and ID$ is the unique identification string for this program. Success then receives either -1 if the deinstallation was successful, or 0 if it was not. If DGroup% is set to 0, then the current copy of the program is removed from memory. That is, the program deinstalls itself rather than another, already resident copy.

Comments

Because DeinstallTSR has been designed as a function, it must be declared before it may be used.

DeinstallTSR is not meant for use with "simplified" P.D.Q. TSR programs. Use PopDeinstall to remove those from memory. A program that manually handles interrupts but also uses PopRequest is considered a simplified TSR, and must also use PopDeinstall instead of DeinstallTSR.

Before calling DeinstallTSR, all active interrupt vectors must have been successfully unhooked. If this is not done and the TSR is deinstalled, the PC will surely crash because the interrupt points to code that is no longer present in memory (or may subsequently be overwritten by another program).

DeinstallTSR uses several DOS functions, so conditions in the TSR must be favorable. In other words, deinstalling should be considered a DOS service, which requires the appropriate assurance that such a service is safe to invoke.

If this function returns 0, then the memory control block chain has probably been corrupted. The safest course of action is to print a message advising the user to reboot the PC.

Please see TSRInstalled, which allows a program to determine if it has already been installed.

TSR installation and de-installation are discussed in depth in the section *TSR Programming With P.D.Q.* elsewhere in this manual.

DisableFP subroutine

Purpose

DisableFP is called within a P.D.Q. simplified TSR program that uses floating point math. It releases the floating point interrupt vectors, and also restores the previous state of the coprocessor if one is present.

Syntax

```
CALL DisableFP
```

Where

The floating point interrupt vectors are restored to what they had been before EnableFP was called. If a coprocessor is installed its entire state is also restored, so computations in your program do not disturb the underlying program.

Comments

If DisableFP is called twice in a row or when EnableFP has not been called, the second request is ignored.

See the topic *Floating Point Considerations* in the section *TSR Programming*.

Also see the description for EnableFP, and the POPUPFP.BAS sample program for an example of using floating point math in a P.D.Q. simplified TSR.

Dollar$ function

Purpose

Dollar$ accepts an incoming long integer value, and returns an equivalent string formatted as a dollar amount.

Syntax

```
Amount$ = Dollar$(Cents&)
```

Where

Cents& is a long integer representing a dollar amount as pennies, and Amount$ receives a string formatted to two decimal places but without a dollar sign. That is, the value 12345 is returned as "123.45".

Comments

Because Dollar$ has been designed as a function, it must be declared before it may be used.

Even though P.D.Q. programs may manipulate floating point values, there may be occasions where you need only to accommodate dollar amounts. By treating a value as the number of pennies and formatting it with Dollar$ when needed, a wide range of values may be represented (approximately plus or minus 21 million dollars) without requiring the added overhead of floating point math.

Also see the FUsing function which is more powerful than Dollar$, but at the expense of increased code size.

DOSBusy function

Purpose

DOSBusy lets a non-simplified P.D.Q. TSR program determine when it is safe to access DOS interrupt services.

Syntax

```
Busy = DOSBusy%
```

Where

Busy receives -1 if DOS is busy, or 0 if it is not, thus allowing any BASIC statement (except INKEY$, PDQInkey, or PDQInput) to be used.

Comments

Because DOSBusy has been designed as a function, it must be declared before it may be used.

The internal DOS "busy" flag is set to a non-zero value whenever DOS is servicing an interrupt and may not be interrupted. If the flag is zero, then DOS is not currently servicing an interrupt. This flag is used by DOS to prevent nested calls to its own services which are not supported.

Note that COMMAND.COM uses a DOS function to read input from the command line. Thus, when at the command line, the DOS flag will always indicate non-zero, or busy.

This function is normally used in a TSR to determine if the TSR may safely execute a DOS interrupt. Because of the command line problem (as described above), the call is of limited value unless it is used in conjunction with other techniques. Further, in many cases it is not sufficient to know that DOS is not busy. For example, a TSR that intercepts the system timer interrupt could receive control when DOS is free but a video BIOS routine is in progress. If the TSR then calls DOS to print a string and DOS in turn calls the BIOS—whammo!

If you plan to use DOSBusy you must use it once early in your program, before calling EndTSR:

```
Dummy = DOSBusy%
```

Please see the PopRequest function which allows manual interrupt handlers to access DOS and BIOS services with the same level of safety that P.D.Q. "simplified" TSR programs enjoy.

DOSVer function

Purpose

DOSVer returns the DOS version that is currently running on the host PC.

Syntax

```
Version = DOSVer%
```

Where

Version receives the DOS version *times 100.* For example, if a PC is using DOS 3.20, then Version will receive the value 320.

Comments

Because DOSVer has been designed as a function, it must be declared before it may be used.

Internally, the DOS service that reports the version number returns two separate values—the major version and the minor version. In the example above, the major version component would be 3, and the minor would be 20. These may be easily isolated as follows:

```
Major = DOSVer% \ 100
Minor = DOSVer% MOD 100
```

There are a number of situations where DOSVer will come in handy. One would be when you are writing an application for use on a network, which of course requires DOS version 3.0 or later. Also, a bug in some DOS 2.*x* versions prevents the SHELL statement from working reliably. By knowing the version of DOS that is running, you can avoid potential problems.

EnableFP subroutine

Purpose

EnableFP is used within P.D.Q. simplified TSR programs that perform floating point math.

Syntax

```
CALL EnableFP
```

Where

The floating point interrupt vectors are set to point to the P.D.Q. emulator library if appropriate. If a coprocessor is installed its entire state is also saved, so computations in your program do not disturb the underlying program.

Comments:

You must call EnableFP as the first action in your popup handler, before performing any floating point calculations. You must also call the complementary routine, DisableFP, just before calling PopDown to return to the foreground program or DOS prompt.

If EnableFP is called twice without an intervening call to DisableFP the second request is ignored.

See the topic *Floating Point Considerations* in the section *TSR Programming*.

Also see the description for DisableFP, and the POPUPFP.BAS sample program for an example of using floating point math in a P.D.Q. simplified TSR.

EndLevel subroutine

Purpose

EndLevel is an alternate END routine, which allows your programs to set the DOS ERRORLEVEL value from a BASIC program.

Syntax

```
CALL EndLevel(ErrorLevel%)
```

Where

ErrorLevel% is the desired error level, between 0 and 255.

Comments

The DOS ERRORLEVEL is a convenient way for your programs to interact with a batch file. One example is to indicate the success or failure of a program's actions. Another is when writing an "ASK" program to create a DOS-only menu using batch files.

See the ASK.BAS example program, and its corresponding MENU.BAT batch file which illustrates this in context.

If you are using Microsoft BASIC version 7, the END statement can be used with an exit code directly, hence calling EndLevel is not necessary.

EndTSR subroutine

Purpose

EndTSR is called by a P.D.Q. TSR program when it has finished its installation sequence and will relinquish control to DOS.

Syntax

```
CALL EndTSR(ID$)
```

Where

ID$ is a unique program identifier that may be used later on in the program to determine if it is already resident.

Comments

The ID$ string variable is used to trap against multiple installations, and must be unique for every TSR program. Please note that ID$ is altered

by EndTSR, and should not be used after EndTSR is called. ID$ must contain at least eight characters, and we suggest using your sign-on or copyright notice.

The call to EndTSR should be the last statement in your program.

Please see the complete discussion about writing TSR programs elsewhere in this manual. In particular, read the section entitled *The Unique Identification String*.

EnvOption subroutine

Purpose

EnvOption allows a program to switch between accessing its own environment, that of its parent, or that of the currently active process from within a TSR. EnvOption also provides an option to honor the capitalization of strings being added to or retrieved from the DOS environment.

Syntax

```
CALL EnvOption(Option%)
```

Where

Option% is bit-coded as described below.

Comments

The environment handling routines in P.D.Q. have been significantly enhanced over regular BASIC to improve their usefulness.

Every program in memory, starting with COMMAND.COM, has access to an environment. If the program is run from the DOS command line, COMMAND.COM creates a copy of its environment, and makes that available to the program. In this case, COMMAND.COM is referred to as the *parent*, and the program being run is referred to as the *child*.

Any changes made by the child to its own copy of the environment disappear when the child program terminates, although those changes will be reflected in programs that it subsequently shells to. Unfortunately, there is no normal means for a child to change its parent's environment.

When a P.D.Q. program is running as a child of COMMAND.COM, the ENVIRON statement and the ENVIRON$ function act on the P.D.Q. program's environment. If the EnvOption routine is used, however, several additional options become available that let you use the environment more effectively. These options are assigned using bit-coding as shown in Table V-2.

The equivalent Decimal values are shown in Table V-3, with bit 0 in the rightmost binary column.

Notice that if you set bit 1 to 1 when assigning variables into the environment and then use lower case letters, you will not be able to access them later using the DOS SET command.

Also notice that bit 2 allows a P.D.Q. TSR program to access the environment of the underlying application. Further, this may be combined with bit 0 to access the environment of the parent of the currently active program.

TABLE V-2
Bit-Coded Values For EnvOption

BIT POSITION	BIT VALUE	DESCRIPTION
0	0	Access the current program's environment.
	1	Access the current program's parent's environment.
1	0	Capitalize variables before adding or retrieving them.
	1	Don't capitalize the variables.
2	0	Access the current program's environment.
	1	Access the environment of the currently active process.

TABLE V-3
Binary/Decimal Values For Use With EnvOption

BINARY	DECIMAL
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

If EnvOption is not called (or is called with an argument of zero), all environment options use the normal BASIC defaults.

These environment options may not work with certain 2.*xx* versions of DOS.

Please see the section entitled *The Environment* for more information on this subject. Also, that section lists the error codes related to the ENVIRON and ENVIRON$ BASIC language statements.

Flush subroutine

Purpose

Flush lets you flush all or selected open DOS file buffers to disk without having to close and then re-open the files.

Syntax

```
CALL Flush ([File1%] [, File2%] [, File3%] [, ...])
```

Where

File1%, File2%, and so forth specify the BASIC file numbers to be flushed.

Comments

Flush lets you flush the DOS file buffers to disk without having to close and then re-open the files. We have designed Flush to accept any number of parameters, to emulate a new BASIC command. If Flush is used with no arguments, then all open files are flushed to disk. You may also specify individual files like this:

```
CALL Flush(1, 3, 4)
```

When data is read from or written to disk, it is always passed first through an area of memory called a file buffer. The total size of the buffer is determined by the setting of the BUFFERS= statement in your CONFIG.SYS file. If CONFIG.SYS is not present or there is no BUFFERS= statement, then the number of buffers defaults to either 2 for a PC or XT, or 3 for an AT. Some 286 and 386 computers have a larger default. Each buffer comprises 512 bytes of memory, which is the size of one disk sector.

Buffers are an important factor in speeding up the operation of a PC, because they allow information that has previously been read or written to be accessed again later, but without having to actually read it from the disk. Further, by always reading an entire sector rather than only the number of bytes an application requests, subsequent sequential reads will not require DOS to physically access the disk again.

However, one problem with buffering disk writing is that the information is not written to disk at the time the write is performed. Rather, the data sits there in memory until the buffer becomes full, or the file is closed. If your program has just used a PRINT or PUT to write data to a disk file and the power goes out, the data will never be transferred to the file. Worse, the file's directory entry will not show the correct size, making it nearly impossible to retrieve what has been written even if you use DEBUG or the Norton Utilities. Flush therefore allows you to force DOS to write the file buffer's contents to disk, but without having to close the file and then re-open it again.

The _FLUSH.OBJ stub file flushes all files only and expects no arguments. _FLUSH.OBJ is described in the section *Linking With Stub Files*.

FUsing function

Purpose

FUsing accepts a number and formatting image string, and returns the number as a formatted string much like BASIC's PRINT USING does.

Syntax

```
Formatted$ = FUsing$(STR$(Number), Image$)
```

Where

Number is any number—integer, long integer, single, or double precision—and Image$ indicates how the result is to be formatted. If the result cannot fit within the allotted space, the first digit of the returned string is replaced with a percent sign (%).

Comments

Because FUsing has been designed as a function, it must be declared before it may be used.

By using the STR$ function, FUsing lets BASIC do most of the dirty work to interpret floating point numbers. This also lets FUsing accept any type of numeric variable. Normally, assembly language routines must be written to expect one type of variable only.

Note that FUsing$ requires a leading blank space when passed a positive number. Therefore, if you use the _STR$.OBJ or _STR$FP.OBJ stub files you must add the leading blank manually:

```
IF SGN(Number) > 0 THEN                    'if zero or positive
   Number$ = " " + STR$(Number)            'add a leading blank
ELSE
   Number$ = STR$(Number)                  'use as is if negative
```

```
END IF
Result$ = FUsing$(Number$, "####.##")
```

Most of the formatting options that PRINT USING recognizes are supported, including commas, dollar signs, and leading asterisks.

Although regular BASIC accepts two different ways to specify commas in the result number, FUsing honors only one of them. To specify commas use a single comma immediately to the left of the decimal point like this:

```
Result$ = FUsing$(STR$(Number), "####,.##")
```

If there is no decimal point place the comma at the end of the formatting string.

Table V-4 summaries FUsing's capabilities.

Also see the P.D.Q. Dollar$ function which requires much less code than FUsing$ when its limitations are acceptable.

TABLE V-4
FUsing Image Codes

#	Represents a digit position.
.	Specifies a decimal point.
+	Used as the first character in the image string causes the sign of the number (+ or -) to be added.
**	Replace leading spaces with asterisks.
$$	Adds a dollar sign to the left of the number.
**$	Combines the effects of ** and $$.
,	Placed to the left of the decimal point (or at the end of the string if there is no decimal point) causes commas to be added at every third position.

Get1Byte function

Purpose

Get1Byte retrieves a single byte from the specified segment and element.

Syntax

```
Value = Get1Byte%(Segment%, Element%)
```

Where

Segment% and Element% indicate where in memory the byte being returned is located, and Value receives its value.

Comments

Because Get1Byte has been designed as a function, it must be declared before it may be used.

Element numbers start at one; there is no element zero.

Get1Byte is intended primarily for accessing memory that was allocated using the P.D.Q. AllocMem function. This effectively adds a new "byte" variable type, which of course occupies less memory than a conventional integer.

The values returned by Get1Byte range from -128 to 127, as opposed to 0 to 255. However, you can link with the alternate GET1BYT.OBJ stub file to have Get1Byte return values between 0 and 255.

Because Get1Byte will accept any segment value, it may also be used to fake Pascal's "array of absolute" feature. For example, by specifying the segment as &H40, any of the internal BIOS data variables may be accessed without requiring DEF SEG and PEEK.

Also see the related Get1Long, Get1Word, and Get1Type functions.

Get1Long function

Purpose

Get1Long retrieves a long integer from the specified segment and element.

Syntax

```
Value& = Get1Long&(Segment%, Element%)
```

Where

Segment% and Element% indicate where in memory the long integer being returned is located, and Value& receives its value.

Comments

Because Get1Long has been designed as a function, it must be declared before it may be used.

Element numbers start at one; there is no element zero.

Get1Long is intended primarily for accessing memory that was allocated using the P.D.Q. AllocMem function. AllocMem lets you simulate dynamic arrays, which generates somewhat less code than using such arrays directly.

Because Get1Long accepts any segment value, it may be used to retrieve data from any arbitrary address. For example, by specifying the segment as 0, any interrupt vector may be accessed directly. (Though in that case interrupt 0 is accessed as element 1, interrupt 1 as element 2, and so forth.)

Also see the related Get1Byte, Get1Word, and Get1Type functions.

Get1Type subroutine

Purpose

Get1Type retrieves a TYPE variable from the specified segment and element.

Syntax

```
CALL Get1Type(Segment%, Element%, Length%, TypeVar)
```

Where

Segment% and Element% indicate where in memory the type element being retrieved is located, Length% is its length in bytes, and TypeVar is the TYPE variable in near memory that is to receive its contents.

Comments

Element numbers start at one; there is no element zero.

Get1Type is intended primarily for accessing memory that was allocated using the P.D.Q. AllocMem function. AllocMem lets you simulate dynamic arrays, which generates somewhat less code than using such arrays directly.

Using **LEN(*TypeVar*)** as the length parameter causes BASIC to use the correct value even if the TYPE structure is changed.

Also see the related Get1Byte, Get1Long, and Get1Word functions.

Get1Word function

Purpose

Get1Word retrieves an integer word (two bytes) from the specified segment and element.

Syntax

```
Value = Get1Word%(Segment%, Element%)
```

Where

Segment% and Element% indicate where in memory the word being returned is located, and Value receives its value.

Comments

Because Get1Word has been designed as a function, it must be declared before it may be used.

Element numbers start at one; there is no element zero.

Get1Word is intended primarily for accessing memory that was allocated using the P.D.Q. AllocMem function. AllocMem lets you simulate dynamic arrays, which generates somewhat less code than using such arrays directly.

Because Get1Word accepts any segment value, it may be used to retrieve data from any arbitrary address. For example, by specifying the segment as &HB000 (for a monochrome display), the entire screen may be treated as if it were a data array.

Also see the related Get1Byte, Get1Long, and Get1Type functions.

GetCPU function

Purpose

GetCPU returns an integer value that indicates the type of CPU installed in the host PC.

Syntax

CPU = GetCPU%

Where

CPU receives either 86, 286, or 386 to indicate the presence of an 8086/88 (or NEC V20/30), an 80286, or an 80386 respectively.

Comments

Because GetCPU has been designed as a function, it must be declared before it may be used.

GetSeg function

Purpose

GetSeg returns BASIC's current DEF SEG setting.

Syntax

```
CurrentSeg = GetSeg%
```

Where

CurrentSeg receives the value used in the most recent DEF SEG command, or BASIC's default data segment if DEF SEG hasn't been used.

Comments

Because GetSeg has been designed as a function, it must be declared before it may be used.

In most cases, you will know the current DEF SEG value because your program set it. But when writing reusable modules that are added to more than one program, the current setting may not be known. Of course, any well-behaved routine should always clean up after itself and return things to the way they were. Therefore, you would use GetSeg before changing the current DEF SEG segment, and then use DEF SEG with the original value when you are finished:

```
SaveSeg = GetSeg%          'save the current segment
DEF SEG = 0                'look at an address in low memory
MonAddress = PEEK(&H463)   'get the info from segment 0
DEF SEG = SaveSeg          'restore it to what it had been
```


GotoOldInt subroutine

Purpose

GotoOldInt is used in a P.D.Q. TSR or interrupt handler program to pass control on to the original interrupt routine.

Syntax

```
CALL GotoOldInt(Registers)
```

Where

Registers is the P.D.Q. TYPE variable being used to service the interrupt.

Comments

In many TSR programs, control must be passed on to the original interrupt handler. For example, a program that has intercepted the keyboard

interrupt will probably act on only one key. Thus, it will defer to the original interrupt routine for all keys except its own hot key.

HercMode subroutine

Purpose

HercMode switches a Hercules display into either graphics mode or text mode.

Syntax

```
CALL HercMode(Action%)
```

Where

Action% is non-zero to switch to graphics mode, or zero to switch back to text mode.

Comments

HercMode is a substitute for both SCREEN 3 and SCREEN 0 afterward.

We decided not to add SCREEN 3 to the SCREEN statement, because it would add that code to programs that do not need Hercules support.

Note that HercMode does not require MSHERC.COM or QBHERC.COM to be loaded as does regular BASIC. Use HercMode as follows:

```
CALL HercMode(1)        'enable graphics mode
CALL HercMode(0)        'go back to text mode
```

Please understand that HercMode does *not* allow you to use PRINT or any of BASIC's graphics commands. Rather, it simply puts the display adapter into graphics mode. It is then up to you to POKE pixels or whatever into display memory, or use an assembly language routine to do that.

HookFP subroutine

Purpose

HookFP takes over the floating point interrupts in a P.D.Q. TSR program.

Syntax

```
CALL HookFP
```

Where

The original floating point interrupt vectors (&H34 through &H3C) are saved, and the vectors are then directed to point to the P.D.Q. TSR program.

Comments

You do not normally need to call HookFP manually, and it is documented here in the interest of completeness only.

See the topic *Floating Point Considerations* in the section *TSR Programming* for information about taking over and releasing the floating point interrupt vectors.

Also see the description for UnhookFP, and the POPUPFP.BAS sample program for an example of using floating point math in a P.D.Q. simplified TSR.

HookInt0 subroutine

Purpose

HookInt0 takes over the "Divide by zero" interrupt (Interrupt 0), to prevent that condition from crashing a program.

Syntax

```
CALL HookInt0(Action%)
```

Where

Action% is zero to take over Interrupt 0 for the duration of the program, or non-zero to take it over only until UnhookInt0 is called.

Comments

When the CPU performs an integer or long integer divide (or MOD) operation that results in division by zero it generates hardware Interrupt 0. This interrupt may not be prevented using normal methods. The default Interrupt 0 handler is inside DOS, and it simply ends the program and displays an error message. To avoid this problem you can call HookInt0 and UnhookInt0, and they are used in a manner similar to the CritErrOn and CritErrOff routines.

An enhancement to HookInt0 lets you specify that you will be turning this trapping on and off manually, rather than simply leaving it on all the time. In most cases you can simply call HookInt0 once and forget it. Unlike the CritErr routines, HookInt0 ties itself into the P.D.Q. termination procedure (B_ONEXIT) and restores the interrupt vector automatically when your program ends. This behavior is specified with an argument of zero:

```
CALL HookInt0(0)
```

A non-zero argument instead tells HookInt0 that you need full control over when Interrupt 0 is trapped, and you will therefore be calling UnhookInt0 manually to disable it. This is necessary when writing a TSR application. In a TSR that might possibly cause a "Division by zero" error you would want to trap the interrupt only while the TSR is active, and then disable trapping before returning to the underlying program. This is shown below.

```
    Call PopUpHere(HotKey%, ID$)
    GOTO EndIt

    CALL HookInt0(1)          'first disable Interrupt 0
    ...                       'do whatever you need here
    ...
    CALL UnhookInt0           'reenable the original handler

    CALL PopDown              'return to the underlying program
EndIt:
```

If HookInt0 were never called, dividing by zero in the TSR would either crash the underlying program or jump into its error handler. And if the interrupt trapping were not removed with UnhookInt0, division by zero in the underlying program would jump into the TSR's HookInt0 handler. In either case, a hung computer is guaranteed.

Again, HookInt0 affects integer and long integer operations only. Floating point math that results in division by zero never halts a running program—regardless of whether an 8087 coprocessor or the emulator is being used. When division by zero does occur, both the floating point emulator and HookInt0 set BASIC's ERR function to 11, "Division by zero".

IntEntry1 and IntEntry2 subroutines

Purpose

IntEntry1 and IntEntry2 are used by P.D.Q. TSR programs to copy the calling program's context to a TYPE variable before processing the interrupt.

Syntax

```
CALL IntEntry1
CALL IntEntry2(Registers, Action%)
```

Where

Registers is the P.D.Q. TYPE variable being used to service this interrupt, and Action% tells IntEntry2 what to do if another interrupt comes along before processing of the first one has been completed.

Comments

Every interrupt handler written using P.D.Q. must immediately call two routines each time it gets control. The first routine is IntEntry1, which saves the AX register and then sets DS to the correct value for the program to access its variables. Next, IntEntry2 must be called before the BASIC program does anything that might destroy the remaining register contents.

The Action% parameter lets you specify what action to take if another interrupt occurs before the first one has been processed. Because P.D.Q. TSR programs are not re-entrant, it is important to prevent a routine from being invoked more than once. However, this is possible only with hardware interrupts such as the timer, keyboard, or communications interrupts.

Action% also lets you indicate whether the interrupt handler is to become active immediately, or is not to receive control until the call to EndTSR has been completed. Because it is possible to write interrupt handlers that are not necessarily TSR programs, you would set the second bit to a value of 1 in that case.

In most situations Action% will be left unassigned, holding a default value of zero.

A complete, detailed discussion of each bit in the Action% variable follows in Table V-5, followed by decimal equivalents for each value in Table V-6.

TABLE V-5
Bit-Coded Values For Action%

BIT POSITION	BIT VALUE	DESCRIPTION
0	0	Jump to old interrupt
	1	Ignore the interrupt
1	0	Wait until resident
	1	Work immediately

TABLE V-6
Binary/Decimal Values For Use With Action%

BINARY	DECIMAL
00	0
01	1
10	2
11	3

Bit 0, handling subsequent interrupts:

An interrupt can occur at any time in the DOS environment. This causes a problem for P.D.Q. programs that intercept interrupts, such as TSR, filter, and system programs. If, while servicing the first occurrence of an interrupt, a second interrupt of the same type occurs, three possible options exist.

First, the interrupt service routine (ISR) may possibly be reentrant. This means that it can be invoked at any time without losing track of previous interrupt levels. However, neither DOS nor P.D.Q. support this type of code.

Second, the ISR can simply terminate the interrupt immediately by performing an IRET. This treatment can have dire consequences since the action that was to be performed by the interrupt never occurred. If the interrupt was the hardware timer tick, for example, then the internal PC clock will begin to lose time. And if the keyboard interrupt were used, an unserviced interrupt would disable the keyboard, effectively disabling the computer. This option is therefore provided mostly for completeness, although it could have application when servicing software interrupts.

Third, the ISR can pass the interrupt request on to the original ISR. So, for example, an ISR that intercepted the printer interrupt could itself call the printer interrupt, which would in turn go directly to the BIOS.

Because the code that is executed when an interrupt is redirected by P.D.Q. is not reentrant, the decision of how to handle nested interrupts must be made at the assembly code level.

Bit 1, the option flag:

This second option determines whether P.D.Q. should intercept the interrupt immediately, or only after becoming memory resident.

This option is provided to allow a TSR to install all interrupt hooks before allowing any action to take place. For example, if an ISR that performed disk access was allowed to pop up on a hotkey before it established an intercept to the disk interrupt, it might crash the machine.

Non-TSR programs will never become resident, so they should use the *activate immediately* option when trapping interrupts.

In general, a TSR would normally use Action% = 0, jump to old interrupt, wait until resident to activate. A non-TSR would normally use Action% = 2, jump to old interrupt, and become active immediately.

Interrupt subroutine

Purpose

Interrupt is the P.D.Q. replacement for BASIC's CALL INTERRUPT routine, and it allows a program to access system interrupts.

Syntax

```
CALL Interrupt(IntNum%, Registers)
```

Where

IntNum% is the interrupt number to invoke, and Registers is the TYPE variable that holds the register values for this interrupt.

Comments

Interrupt requires a TYPE variable to access each of the processor's registers. This variable should be designed as follows:

```
TYPE Registers
    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
```

This is similar to the examples shown in the BASIC manuals, and our version shares the same problem—it is difficult to access the individual halves of those registers that can be split. The best solution is to use integer division with the BASIC AND operator, to isolate the low and high portions. The variables AL and AH below receive the low and high components respectively:

```
AL = AX AND 255          'AL gets just the low part
AH = AX \ 256            'and AH gets the high part
```

Likewise, you could use multiplication and addition to assign a register from the low and high halves as follows:

```
AX = AL + 256 * AH
```

Since most DOS and BIOS services require you to assign the AH portion only, this may also be simplified:

```
AX = 256 * AH
```

Using Hex notation is equally appropriate, when the register values are known in advance and numbers are being used instead of variables:

```
AX = &H4E01              'assign &H4E to AH, 1 to AL
```

Interrupt uses the current setting of the DS and ES registers when calling an interrupt, and the values returned in DS and ES are not available. Also notice that the Registers variable defined in the PDQDECL.BAS file contains additional components for use with the various TSR interrupt handling routines. The additional elements in that type definition will cause no problems when used with Interrupt or InterruptX. Of course, you could also define a smaller, subset TYPE for use only with the Interrupt routine.

The DS and ES portions of this TYPE variable are not used by Interrupt, and may be omitted. However, they are used by the InterruptX routine.

Interrupt differs from Microsoft BASIC's implementation in that the same TYPE variable is used both when calling the interrupt, and when examining the register values that are returned. This reduces the amount of code that is added to your programs, and (usually) the amount of work you must do as well.

The P.D.Q. program's stack is used by the interrupt being called.

Interrupt may be used from within a P.D.Q. TSR program.

Do not use Interrupt to access Interrupts &H25 or &H26. These are the DOS interrupts that directly read and write disk sectors, and they leave extra information on the stack that cannot be removed by a P.D.Q. BASIC program. If you really do need to access these interrupts, use the INTERRUPT routine that comes with later versions of BASIC 7 PDS, because it contains extra code to accommodate the stack problem.

InterruptX subroutine

Purpose

InterruptX is the P.D.Q. replacement for BASIC's CALL INTERRUPTX routine, and it allows a program to access system interrupts.

Syntax

```
CALL InterruptX(IntNum%, Registers)
```

Where

IntNum% is the interrupt number to invoke, and Registers is the TYPE variable that holds the register values for this interrupt.

Comments

InterruptX requires a TYPE variable to access each of the processor's registers. See the comments that accompany the Interrupt routine for a description of this TYPE variable.

If the DS portion of the TYPE variable is set to -1 when InterruptX is called, the current value of BASIC's data segment is sent to the interrupt in the DS register. That is, DS is passed unchanged. If the ES portion is set to -1, then ES is set to the current value of DS. Regardless of which values you use for DS and ES when the interrupt is called, the register contents returned by the interrupt are assigned to those components of the TYPE variable for access by your BASIC program.

InterruptX differs from regular BASIC's implementation in that the same TYPE variable is used both when calling the interrupt, and when examining the register values that are returned.

The P.D.Q. program's stack is used by the interrupt being called.

InterruptX may be used from within a P.D.Q. TSR program.

Do not use InterruptX to access Interrupts &H25 or &H26. These are the DOS interrupts that directly read and write disk sectors, and they leave extra information on the stack that cannot be removed by a P.D.Q. BASIC program. If you really do need to access these interrupts, use the INTERRUPTX routine that comes with later versions of BASIC 7 PDS, because it contains extra code to accommodate the stack problem.

MidChar function

Purpose

MidChar returns the ASCII value for a single character within a string.

Syntax

```
Char = MidChar%(Work$, Position%)
```

Where

Char receives the ASCII value for the specified character, or -1 if Position% is less than 1 or past the end of Work$.

Comments

Because MidChar has been designed as a function, it must be declared before it may be used.

Every time you use the MID$ function in either regular BASIC or P.D.Q. a copy of that portion of the string is extracted, and new memory is allocated to receive the copy. This takes a lot of time and adds a fair amount of code to your program each time MID$ is used. Because MidChar is an integer function, no string operations are required. Further, BASIC can compare integers much faster than it can compare strings. Therefore, MidChar is most useful when you are walking through a string looking for a particular single character.

The first example below shows a typical use for MidChar, by searching for the backslash that separates a file name from its path. The line that compares MidChar to 92 compiles to 26 bytes of code, not counting the MidChar routine itself:

```
DEFINT A-Z
FileName$ = "C:\SUBDIR1\SUBDIR2\FILENAME.EXT"

FOR X = LEN(FileName$) TO 1 STEP -1
  IF MidChar%(FileName$, X) = 92 THEN  '92 = ASC("\")
    Path$ = LEFT$(FileName$, X)        'isolate the path
    FileName$ = MID$(FileName$, X + 1) 'and the name
   END IF
NEXT
```

Contrast that to this next example which uses MID$ to do the same thing, but with a compiled code size of 32 bytes (not counting the actual MID$ routine):

```
FOR X = LEN(FileName$) TO 1 STEP -1
  IF MID$(FileName$, X, 1) = "\" THEN
    Path$ = LEFT$(FileName$, X)
    FileName$ = MID$(FileName$, X + 1)
```

```
    END IF
  NEXT
```

Although the code size savings may seem small, it quickly adds up when MID$ is used many times in a program. More important, the MidChar routine is at least five times faster than MID$—and that's where the real benefit becomes obvious!

MidCharS subroutine

Purpose

MidCharS inserts a single character into a string much faster than using the MID$ statement.

Syntax

```
CALL MidCharS(Work$, BYVAL Position%, BYVAL Char%)
```

Where

CHR$(Char%) is assigned into Work$ at the indicated position. If Position% is less than 1 or greater than the length of Work$ the request is ignored.

Comments

MidCharS (the "S" stands for statement, as opposed to function) complements the MidChar function described earlier, and it is much more efficient than using the statement form of MID$ when inserting a single character into an existing string.

A typical example of using MidCharS is shown below, along with the equivalent using MID$ in BASIC:

BASIC:

```
MID$(Work$, Start, 1) = Char$
```

MidCharS:

```
CALL MidCharS(Work$, Start, Char)
```

One reason MidCharS is so efficient is because MID$ must be able to handle varying lengths in both the source and destination strings. Because MidCharS works with only single characters it is much faster. Another factor is that fewer passed parameters are required. In the example above, MID$ requires 25 bytes of compiler-generated code, compared to only 17 for MidCharS. More important, however, MidCharS is more than five times faster than equivalent code using the MID$ statement.

NoSnow subroutine

Purpose

NoSnow lets you disable CGA "snow suppression" in PDQPrint, PDQCPrint, BIOSInput, and BIOSInput2, to achieve the fastest display speed possible when using a CGA display adapter.

Syntax

```
CALL NoSnow(Flag%)
```

Where

Flag% is either 0 to disable snow suppression, or non-zero to reenable it again later.

Comments

Most CGA displays require using a special timing loop within "quick print" routines, to prevent interference when the routines write directly to display RAM. If this is not done, a disturbing burst of static appears on the screen each time display memory is read from or written to. (However, this static causes no harm to the PC or display hardware.)

Unfortunately, the timing loop slows down the printing process considerably. Many newer CGA boards do not require such special treatment, but there's no way for PDQPrint and PDQCPrint to know that a newer adapter is present. NoSnow therefore lets you manually override the assumptions these routines make, by either allowing or preventing the suppression timing. You can also use NoSnow with older CGA adapters to sacrifice esthetics for speed.

Pause subroutine

Purpose

Pause delays a program for a specified number of 18ths of a second.

Syntax

```
CALL Pause(Ticks%)
```

Where

Ticks% is the number of system timer ticks (18ths of a second) to delay.

Comments

Pause provides an easy way to add short delays to your programs, while providing a finer resolution than the SLEEP command. Pause was adapted from our QuickPak Professional package, and it is used internally by

several of the BASIC language statements in P.D.Q. Thus, it seems only sensible to include and document it as a separate subroutine.

Although P.D.Q. supports the BASIC TIMER function, you should avoid that when possible because TIMER requires floating point math. Even if you do not otherwise need floating point math, using TIMER even once adds the entire P.D.Q. floating point emulator to your program.

Note that the system timer event occurs 18.206481 times per second. Therefore, you may want to use 18.2 instead of 18 in your calculations when more accuracy is necessary. For example, to delay 10 seconds you would use 182 instead of 180.

PDQCompare function

Purpose

PDQCompare compares two TYPE variables and reports if they are the same.

Syntax

```
IF PDQCompare%(SEG Type1, SEG Type2, NumBytes%) THEN
 'they match
END IF
```

Where

Type1 and Type2 are TYPE variables, and NumBytes% is the number of bytes to compare (usually the full length of one of the variables).

Comments

Because PDQCompare has been designed as a function, it must be declared before it may be used.

PDQCompare lets you compare any two TYPE variables or blocks of memory (up to 64K) to see if they are identical. If the TYPE variables are equal PDQCompare returns -1 (True); otherwise 0 is returned (False). This lets you use NOT with the function result:

```
IF NOT PDQCompare%(SEG Type1, SEG Type2, NumBytes%) THEN
  'they are different
END IF
```

PDQCompare is especially valuable with TYPE variables, because comparing them using purely BASIC statements requires a separate test for each component:

```
IF Type1.LastName = Type2.LastName AND _
  Type1.FirstName = Type2.FirstName AND _
  Type1.DatePaid = Type2.DatePaid AND ......
```

PDQCompare may be declared and invoked in either of two ways: with the source and target segments and addresses passed individually by value, or as SEG addresses for each variable.

We used the BYVAL method in PDQDECL.BAS, however a second DECLARE using SEG is also shown there as a remark. The examples below show both methods in context.

```
1. DECLARE FUNCTION PDQCompare%(SEG Source AS ANY, _
     SEG Dest AS ANY, NumBytes AS ANY)

   IF PDQCompare%(Type1, Type2, LEN(Type1)) THEN
     'the two TYPE variables are the same
   END IF

2. DECLARE FUNCTION PDQCompare%(BYVAL Seg1%, BYVAL Addr1%, _
     BYVAL Seg2%, BYVAL Addr1%, NumBytes AS ANY)

   IF PDQCompare%(Seg1%, Addr1%, Seg2%, Addr2%, NumBytes%) THEN
      'the two blocks of memory are the same
   END IF
```

Note that you must use a long integer (or equivalent negative value) to specify NumBytes values larger than 32,767. NumBytes is declared using the AS ANY option, so you can use either type of variable interchangeably in the same program.

PDQCPrint subroutine

Purpose

PDQCPrint is a "quick print" routine that bypasses DOS and writes directly to screen memory using the current COLOR setting.

Syntax

```
CALL PDQCPrint(Work$, Row%, Column%)
```

Where

Work$ is the string to be printed, and Row% and Column% specify where on the screen to print.

Comments

PDQCPrint is an alternate version of PDQPrint. Unlike PDQPrint, PDQCPrint uses the current COLOR settings to avoid an extra passed parameter when the same color will be used repeatedly. This saves a few bytes of code each time PDQCPrint is called.

PDQCPrint may be directed to print to any arbitrary segment, by using the PDQSetMonSeg routine.

Please see the discussion that accompanies PDQPrint elsewhere in this section.

PDQExist function

Purpose

PDQExist provides a simple way to determine if a file exists.

Syntax

```
There = PDQExist%(FileName$)
```

Where

FileName$ is either a file name or a file specification such as "*.BAS", and There receives -1 if the file exists, or zero if it does not.

Comments

Because PDQExist has been designed as a function, it must be declared before it may be used.

A drive and path specification may be optionally used to specify other than the current defaults. For example, you could use:

```
FileName$ = "C:\BASIC7\MYFILE.DAT"
IF PDQExist%(FileName$) THEN ...
```

Because of the way P.D.Q. programs handle DOS errors, PDQExist is not as necessary as when writing in regular BASIC alone. That is, attempting to open a file that doesn't exist will not halt your program.

PDQInkey function

Purpose

PDQInkey serves the same purpose as BASIC's INKEY$ function, except it returns an integer value rather than a string.

Syntax

```
KeyHit = PDQInkey%
```

Where

KeyHit receives zero if no keys are pending in the keyboard buffer, a positive number equal to a normal key's ASCII value, or a negative value corresponding to an extended key code.

Comments

Because PDQInkey has been designed as a function, it must be declared before it may be used.

Even though P.D.Q. fully supports BASIC's INKEY$ function, using PDQInkey adds less code and is faster. The example below shows this in context.

```
DO
   KeyHit = PDQInkey%
LOOP UNTIL KeyHit

IF KeyHit > 0 THEN
   'they pressed a regular key
ELSE
   'it was an extended key
END IF
```

Like BASIC's INKEY$ function, PDQInkey fully supports DOS command line redirection using the " < " symbol.

PDQInkey is not intended for use within a P.D.Q. TSR program. You must instead use BIOSInkey, which does not support redirection.

PDQInput subroutine

Purpose

PDQInput is a P.D.Q. substitute for BASIC's LINE INPUT command.

Syntax

```
CALL PDQInput(Work$)
```

Where

Work$ is the string that is to receive the entered text.

Comments

Although P.D.Q. supports the normal INPUT command, it is not available when linking with the SMALLDOS.LIB library. Further, each time BASIC's INPUT is used, many bytes of assembler instructions are added to your program which is avoided when PDQInput is used. PDQInput calls on the built-in DOS command-line editor, which is already present in RAM. The maximum string length that may be input is 127 characters.

PDQInput supports redirection from the DOS command line.

PDQInput is not intended for use within a P.D.Q. TSR program. You must instead use the BIOSInput or BIOSInput2 routines which do not support redirection, but provide several additional features.

If the operator presses Ctrl-C or Ctrl-Break during input the program will be ended, unless BreakOff has been called.

See the BreakOff, BreakOn, and BreakHit routines, which let you disable the Ctrl-Break and Ctrl-C keys. Also see the BIOSInput and BIOSInput2 routines which allow editing an existing string, and specifying an input field color.

PDQMessage function

Purpose

PDQMessage accepts a BASIC error number, and returns an appropriate text message as a string.

Syntax

```
ErrMsg$ = PDQMessage$(ErrNumber%)
```

Where

ErrNumber% is one of the BASIC error numbers supported by P.D.Q., and ErrMsg$ receives the equivalent text message.

Comments

Because PDQMessage has been designed as a function, it must be declared before it may be used.

You may also print the output of PDQMessage directly, as shown below.

```
PRINT PDQMessage$(ErrNumber%)
```

Supported error numbers are listed in Table I-1 on Page 1-7. Any error number that is not represented in the table will return the message "Undefined error".

If you have an assembler you may also add new messages, or remove those that are not necessary to reduce your program's size. Complete instructions are provided in the PDQMSG.ASM source file.

PDQMonitor function

Purpose

PDQMonitor reports the type of display adapter that is installed in the host PC.

Syntax

```
MonType = PDQMonitor%
```

Where

MonType receives a code number that corresponds to the monitor type.

Comments

Because PDQMonitor has been designed as a function, it must be declared before it may be used.

If two monitors are installed, the one that is currently active is reported. Note that PDQMonitor remembers the type of monitor internally, to increase the speed for subsequent invocations. Since PDQMonitor is used by PDQPrint and PDQCPrint, having to assess the monitor type at each call would slow those routines considerably. Therefore, if the active monitor is changed on a two-monitor system after PDQMonitor has already been used, the original monitor code number is returned. However, PDQSetMonSeg lets you force PDQMonitor to reevaluate the display adapter.

You can easily determine if the current monitor is monochrome or color using a simple PEEK as follows:

```
DEF SEG = 0
IF PEEK(&H463) = &HB4 THEN
   'it is a monochrome monitor using video segment &HB000
ELSE
   'it is a color monitor using video segment &HB800
END IF
```

PDQMonitor recognizes all of the popular display adapter types; however, it does not report which screen mode is currently active. The type of monitor detected is returned using the codes shown in Table V-7.

TABLE V-7
Values Returned By Monitor

RETURNED VALUE	DESCRIPTION
1	Monochrome adapter
2	Hercules monochrome adapter
3	CGA adapter
4	EGA adapter w/mono monitor
5	EGA adapter w/color monitor
6	VGA adapter w/mono monitor
7	VGA adapter w/color monitor
8	MCGA adapter w/mono monitor
9	MCGA adapter w/color monitor
10	EGA adapter w/CGA monitor
11	IBM 8514/A

PDQParse function

Purpose

PDQParse extracts individual portions from a delimited string.

Syntax

```
ThisItem$ = PDQParse$(Work$)
```

Where

ThisItem$ receives the next delimited portion of Work$.

Comments

Because PDQParse has been designed as a function, it must be declared before it may be used.

PDQParse serves two important purposes in a P.D.Q. program. First, it can be used to parse delimited information in a string such as the current DOS PATH. It also provides an easy way to simulate READ and DATA, to reduce the amount of code that is added to your programs. Further, READ is in not supported when you are using the SMALLDOS library provided with P.D.Q.

There are actually three related routines in this group. The first is PDQParse, which is designed as a string function with a single string argument. Each time PDQParse is invoked, it returns the next successive item in the string. Once the last item in the string has been returned, subsequent calls to PDQParse return a null string.

PDQRestore is used to reset reading at the beginning again—either with the same string or with a new one.

The last routine is SetDelimitChar, and it lets you establish any character as a delimiter.

The short program below shows PDQParse in action.

```
Work$ = "One; Two; Three"
FOR X = 1 TO 3
  PRINT PDQParse$(Work$);
NEXT
```

And this is the result on the screen:

```
OneTwoThree
```

By default, PDQParse uses a semicolon (;) as a delimiter, since parsing the DOS PATH is a fairly common operation. But you may also change the delimiter to, say, a comma, which is what BASIC uses when reading DATA items. If your data has a comma in it, then you could change the delimiter to some other character, for example a "|" or even a Ctrl-A. The delimiter can even be changed mid-read if necessary.

Like BASIC's READ statement, PDQParse also strips leading blanks and Tab characters from each item.

PDQParse and the other related routines mentioned here are demonstrated in the PDQPARSE.BAS example program.

PDQPeek2 function

Purpose

PDQPeek2 peeks one word (two bytes) in a single operation.

Syntax

```
Value = PDQPeek2%(Address%)
```

Where

Address% is the address of the first of two bytes to be examined, and Value receives their contents.

Comments

Because PDQPeek2 has been designed as a function, it must be declared before it may be used.

The current DEF SEG setting is used to identify the segment PDQPeek2 will access.

Reading two bytes from memory has always been tedious in BASIC. For example, without PDQPeek2 the following steps are necessary:

```
Value = PEEK(Address) + 256 * PEEK(Address + 1)
```

PDQPeek2 lets you do the same thing in a single operation:

```
Value = PDQPeek2%(Address%)
```

This not only requires you to do less typing, but it also adds much less code to the final .EXE program.

Also see the PDQPoke2 routine which assigns two bytes at once.

PDQPoke2 statement

Purpose

PDQPoke2 pokes one word (two bytes) in a single operation.

Syntax

```
CALL PDQPoke2(Address%, Value%)
```

Where

Address% is the address of the first of two bytes to be assigned, and Value% is placed there.

Comments

The current DEF SEG setting is used to identify the segment PDQPoke2 will access.

Writing two bytes into memory has always been tedious in BASIC. For example, without PDQPoke2 the following steps are needed:

```
POKE Address%, Value% AND 255
POKE Address% + 1, Value% \ 256
```

PDQPoke2 lets you do the same thing in a single operation:

```
CALL PDQPoke2(Address%, Value%)
```

This not only requires you to do less typing, but it also adds much less code to the final .EXE program.

Also see the PDQPeek2 routine which retrieves two bytes at once.

PDQPrint subroutine

Purpose

PDQPrint is a "quick print" routine that bypasses DOS and writes directly to screen memory.

Syntax

```
CALL PDQPrint(Work$, Row%, Column%, Colr%)
```

Where

Work$ is the string to be printed, Row% and Column% specify where on the screen to begin printing, and Colr% is the combined foreground and background color to use.

Comments

PDQPrint assumes an 80 column display using text page zero.

PDQPrint fully supports the 25-, 43-, and 50-line modes available with EGA and VGA adapters.

Because PDQPrint accepts row and column parameters, your program should use BASIC's built-in CSRLIN and POS(0) functions if you want to print at the current cursor location. This is shown below.

```
CALL PDQPrint(Work$, CSRLIN, POS(0), Colr%)
```

The foreground and background colors must be combined into a single value using the following formula:

```
Colr% = (FG AND 16) * 8 + ((BG AND 7) * 16) + (FG AND 15)
```

The simplified formula below does not accommodate flashing:

```
Colr% = FG + 16 * BG
```

The first time PDQPrint is called, it examines the type of display adapter installed using a BIOS service, and saves that information internally. Thus, subsequent calls to PDQPrint will be extremely fast. This also makes PDQPrint ideal for use within a TSR program. As long as it is called once before the program becomes resident, it may be used at any time without regard to whether the BIOS is in an "interruptable" state.

See the PDQSetMonSeg routine which lets you direct PDQPrint to write to any arbitrary segment, and PDQSetWidth which is used to accommodate column widths other than 80. Also see PDQCPrint which uses the current COLOR setting instead of requiring a color parameter, and COLORS.BAS which displays a chart of color values.

PDQRand function

Purpose

PDQRand is an integer-only replacement for BASIC's RND function.

Syntax

```
Value = PDQRand%(Limit%)
```

Where

Limit% is any positive integer value, and Value receives a random integer number between 0 and Limit%.

Comments

Because PDQRand has been designed as a function, it must be declared before it may be used.

BASIC's RND function requires floating point support, which adds a substantial amount of code to programs that do not otherwise need it. Therefore, PDQRand is preferable in many cases. PDQRand further expands on BASIC's RND because it lets you specify an upper limit for the returned number.

Like BASIC's RND, PDQRand produces the same sequence of numbers each time it is used. Please see the PDQRandomize routine, which lets you seed the random number function with a new starting value.

PDQRandomize subroutine

Purpose

PDQRandomize assigns a new seed value that affects the numbers returned by the PDQRand function.

Syntax

```
CALL PDQRandomize(Seed%)
```

Where

Seed% is used to start a new sequence of random numbers.

Comments

Any non-zero integer value may be used as a seed.

See the PDQRand function described elsewhere in this section.

PDQRestore subroutine

Purpose

PDQRestore is used with PDQParse to force that routine to begin reading from the beginning of whichever string is used in the next call to PDQParse.

Syntax

```
CALL PDQRestore
```

Where

The internal pointer PDQParse uses is reset to 1.

Comments

See the PDQParse routine elsewhere in this section for more information about using PDQRestore.

PDQSetMonSeg subroutine

Purpose

PDQSetMonSeg directs the PDQPrint and PDQCPrint routines to write to a specified segment.

Syntax

```
CALL PDQSetMonSeg(NewSegment%)
```

Where

NewSegment% is the new segment that PDQPrint and PDQCPrint will use. If NewSegment is 0, the next time PDQPrint or PDQCPrint are called they will reevaluate the type of monitor installed and use the appropriate video segment.

Comments

PDQSetMonSeg has two important uses. The first, and perhaps most useful, is to create well-behaved programs that run under DESQview. Because PDQPrint writes directly to video RAM, it is not otherwise compatible with DESQview. However, DESQview provides services that report a partition's video segment, and these services may be accessed from BASIC using CALL INTERRUPT. Once the correct segment is known and assigned using PDQSetMonSeg, your program may then use PDQPrint.

The second use is to create "virtual screens", by directing PDQPrint to write to an array, or a block of memory that has been claimed using the

P.D.Q. AllocMem routine. This opens up all sorts of possibilities, such as printing in the background. Once a "phantom" screen has been created, it is a simple matter to copy it to the actual screen video segment:

```
'--- create an 80 x 25 screen, and make that the new segment
REDIM Array(1 TO 2000)
ArraySeg% = VARSEG(Array(1))
CALL PDQSetMonSeg(ArraySeg%)

'--- write some test messages there
FOR X = 1 TO 25
   CALL PDQPrint("Test on line " + STR$(X), X, 1, 112)
NEXT

'--- copy screen from the array to color text video RAM
VideoSeg = &HB800
CALL BlockCopy(ArraySeg, 0, VideoSeg, 0, 4000)

CALL PDQSetMonSeg(0)              'restore to normal for later
```

See the MULTPAGE.BAS example program, which shows how to simulate BASIC's **SCREEN , , apage, vpage** capability. This program selects both the video page that is written to, and also the page that is currently being displayed.

PDQSetWidth subroutine

Purpose

PDQSetWidth lets PDQPrint, PDQCPrint, BIOSInput, and BIOSInput2 use a screen width other than 80 columns.

Syntax

```
CALL PDQSetWidth(NewWidth%)
```

Where

NewWidth% is the new number of columns (40, 80, 132, or whatever).

Comments

Unlike the video routines in our QuickPak Professional product that accommodate any screen size automatically, it is important to keep code size (and thus features) to a minimum in P.D.Q. To support this feature added only two bytes to PDQPrint and PDQCPrint, and ten or so more are added only if you actually call PDQSetWidth.

If you plan to use 40 columns you also must use the WIDTH 40 command in your program. PDQSetWidth merely tells the print routines how many columns to use in their internal calculations.

PDQShl and PDQShr functions

Purpose

PDQShl and PDQShr return an integer value with the bits shifted left or right a specified number of places.

Syntax

```
Shifted = PDQShl%(BYVAL Value%, BYVAL NumBits%)
Shifted = PDQShr%(BYVAL Value%, BYVAL NumBits%)
```

Where

Shifted receives the result of shifting the bits in Value% by NumBits% places. Value% itself is not changed.

Comments

Because PDQShl and PDQShr have been designed as functions, they must be declared before they may be used.

Shifting bits is one area where BASIC is particularly weak, and these functions can replace a substantial amount of code. In most cases bits may be shifted by multiplying or diving by a power of 2. For example, to shift the bits in a variable right one position you would divide by 2 like this:

```
Shifted = Value% \ 2
```

The problem is that BASIC treats all integer and long integer values as being signed, which produces incorrect results if the highest bit is set. PDQShr and PDQShl actually use the assembly language Shr and Shl instructions, to shift the bits directly.

The arguments to these functions are passed by value, to let them operate as quickly as possible.

PDQSound subroutine

Purpose

PDQSound is a small-code replacement for the BASIC SOUND command.

Syntax

```
CALL PDQSound(Frequency%, Duration%)
```

Where

Frequency% is the desired frequency in Hz. (cycles per second), and Duration% is the number of 18ths of a second to sustain the tone for.

Comments

Although P.D.Q. supports BASIC's SOUND command, SOUND requires the floating point library. For programs that do not otherwise require floating point math, using PDQSound can provide a considerable reduction in code size.

PDQSound is nearly identical to the SOUND command from regular BASIC, except it operates in the foreground only. Where BASIC's SOUND returns to your program immediately and continues to play the tone in the background, PDQSound does not return until the sound has completed.

You may also use a *negative* value for Duration%, which tells PDQSound not to turn off the PC's speaker when it is finished. This lets you create smoother glissandos and other effects by using negative values for all but the last call. Otherwise, having the speaker turned on and off between calls creates a slight clicking sound. To see this in action run the following program:

```
FOR X = 500 TO 1000 STEP 10                'make a siren
   CALL PDQSound(X, 1)                     'use a positive duration
NEXT

FOR X = 500 TO 1000 STEP 10                'this sounds smoother
   CALL PDQSound(X, -1)                    'using a negative duration
NEXT

CALL PDQSound(1010, 1)                     'now turn the speaker off
```


PDQTimer function

Purpose

PDQTimer returns the number of timer ticks stored in the BIOS data area in low memory.

Syntax

```
NumTicks& = PDQTimer&
```

Where

NumTicks& receives the contents of the four-byte system timer.

Comments

Because PDQTimer has been designed as a function, it must be declared before it may be used.

Even though P.D.Q. supports BASIC's TIMER command, TIMER requires the floating point library. For programs that do not otherwise require floating point math, using PDQTimer will reduce their size considerably.

As with TIMER, when the clock passes midnight the system time is reset to zero:

```
Start& = PDQTimer&          'start the timer
FOR X = 1 TO 10000          'we want to time how long this takes
  ...
  ...
NEXT
Done& = PDQTimer&           'done timing

IF Done& < Start& THEN      'we passed midnight
   Done& = Done& + 1573040
END IF
PRINT Done& - Start&; "clock ticks have elapsed"
```

Please see the TIMER.BAS demonstration program for an example of simulating BASIC's TIMER resolution and results.

PDQVall and PDQValL functions

Purpose

PDQVall returns an integer that represents the value of a string, and PDQValL returns a long integer.

Syntax

```
Value = PDQVallI%(Work$)
Value = PDQVallL&(Work$)
```

Where

Work$ is a string containing a number such as "1234", and Value receives its value.

Comments

Because PDQVall and PDQValL have been designed as functions, they must be declared before they may be used.

Although P.D.Q. supports BASIC's VAL function, it requires the floating point library. For programs that do not otherwise require floating point math, using PDQVall or PDQValL can reduce their size considerably. Because floating point operations are not required, PDQVall and PDQValL are also extremely fast.

The default PDQValI and PDQValL functions recognize a leading "&H" in the string to specify Hex values, but not a leading "&O" for Octal. The _PDQVAL.OBJ stub file does not recognize "&H" values or a leading plus sign (+), and also must not be used with concatenated strings. That is, the following code should be avoided if you are linking with _PDQVAL.OBJ:

```
Value = PDQValI%(First$ + Second$)
```

Also see the section entitled *Linking With Stub Files* for information on other reduced-capability routines provided with P.D.Q.

PointIntHere subroutine

Purpose

PointIntHere is used to show where in a BASIC program control is to go when the specified interrupt occurs.

Syntax

```
CALL PointIntHere(Registers)
```

Where

Registers is the TYPE variable that holds the register values for this interrupt.

Comments

PointIntHere is intended for use in P.D.Q. TSR and interrupt handler programs. It indicates where in your BASIC program control should be passed when the specified interrupt occurs. The statement following a call to PointIntHere must be a GOTO, and the line that follows the GOTO is the one that receives control each time the interrupt occurs.

Do not use PointIntHere to intercept Interrupts &H25 or &H26. These are the DOS interrupts that directly read and write disk sectors. Due to a "design decision" at Microsoft, these interrupts leave extra information on the stack that cannot be removed by a P.D.Q. BASIC program.

PointIntHere may not be called from within a SUB, FUNCTION, or DEF FN function.

Interrupts that have been intercepted using PointIntHere may be removed from the interrupt chain with the complementary routine UnhookInt.

If you are writing a manual interrupt handler that intercepts timer interrupt 8, be sure to use CallOldInt as the very first action in your handler code, so the original BIOS routines will be executed immediately. When you

are finished servicing the interrupt, call ReturnFromInt to return to the underlying application.

In most cases, the order in which interrupts are serviced does not matter. Therefore, you could just as easily do whatever work is needed, and then use GotoOldInt to continue on to the original handler. But in the case of the timer interrupt it is essential that the original BIOS services be executed as soon as possible, especially if there is a chance that your code will take longer than one timer tick to complete. All of the examples that trap interrupt 8 show this in context, as do the examples shown in the section entitled *Using PopRequest* elsewhere in this manual.

Due to a bug in DOS 2.*xx* versions (so what else is new) there is an unavoidable interaction between PointIntHere and the use of ENVIRON and ENVIRON$. Programs that manually trap one or more interrupts may *not* use ENVIRON or ENVIRON$ in conjunction with an EnvOption of 4. Attempting to do so results in an "Environment not found" error. See EnvOption in this section for more information on setting environment options in P.D.Q.

TSR programming is described in depth elsewhere in this manual.

PoolOkay function

Purpose

PoolOkay tests if string memory has been corrupted.

Syntax

```
Okay = PoolOkay%
```

Where

Okay receives -1 if string memory is intact, or 0 if it is not.

Comments

Because PoolOkay has been designed as a function, it must be declared before it may be used.

PoolOkay provides a simple way to test the integrity of string memory. Since a P.D.Q. program does not halt when an error occurs, this function lets you detect if string memory has become corrupted. PoolOkay is intended primarily as a debugging aid.

In practice, you would probably test PoolOkay in an IF statement like this:

```
IF PoolOkay% THEN PRINT "No errors so far!"
```

or

```
IF NOT PoolOkay% THEN PRINT "String space corrupt"
```

PopDeinstall function

Purpose

PopDeinstall is used by a P.D.Q. "simplified" pop-up program to remove itself from memory.

Syntax

```
Success = PopDeinstall%(DGroup%, ID$)
```

Where

DGroup% is the value returned by TSRInstalled when it was invoked earlier, and it identifies the data segment of the program to deinstall. If zero is used for DGroup%, then the current copy of the program is deinstalled.

ID$ is the unique identification string that has been defined for this program, and it must be at least 8 characters long.

Success is assigned either -1 or 0, to indicate the success or failure of the deinstallation respectively.

Comments

Because PopDeinstall has been designed as a function, it must be declared before it may be used.

PopDeinstall is intended only for use with P.D.Q. "simplified" pop-up programs that use the PopUpHere routine. For other P.D.Q. TSR programs you should instead use DeinstallTSR.

If your program accepts a command line switch as a flag to deinstall its resident copy, then you must invoke the TSRInstalled function before calling EndTSR.

If PopDeinstall is successful, you must immediately end the program. If a program is removing itself from memory—as opposed to removing a previously installed copy—you must call PopDown to terminate the program.

The only likely reason for PopDeinstall to fail is when another program has intercepted one or more of the same interrupts that are used by a P.D.Q. pop-up program. The original version of Borland's SideKick is notorious

for stealing interrupts. Therefore, attempting to deinstall a P.D.Q. TSR program when SideKick is resident is sure to fail.

Deinstallation will also fail if PopDeinstall is unable to release the program's memory to DOS. However, this is unlikely to happen.

PopDeinstall is typically used like this:

```
Success = PopDeinstall%(DGroup%, ID$)
IF NOT Success THEN
   PRINT "Sorry, unable to deinstall.  Reboot now!"
ELSE
   PRINT "Program successfully removed."
END IF
CALL PopDown
```

Please see the section entitled *TSR Programming With P.D.Q.* Also see the description for the TSRInstalled function elsewhere in this section. ID$ is discussed in the section *The Unique Identification String*.

PopDown subroutine

Purpose

PopDown is used by a P.D.Q. TSR program to return control to the underlying application.

Syntax

```
CALL PopDown
```

Where

Control is returned to the underlying application.

Comments

PopDown is intended for use in P.D.Q. TSR programs that use the simplified PopUpHere method of detecting a hot key. Please see the complete discussion on writing TSR programs elsewhere in this manual.

PopRequest function

Purpose

PopRequest lets a manual interrupt handling TSR program enjoy the same safe DOS and BIOS access as "simplified" TSR programs.

Syntax

```
Success = PopRequest%(Flag%, NumTicks%)
```

Where

Flag% is cleared to zero by PopRequest when it is invoked, and then set to -1 when control is passed to the simplified portion of the program. NumTicks% tells PopRequest for how many system timer ticks (18ths of a second) it is to try to find DOS and the BIOS not busy. Success is assigned either -1 (True) if no other interrupt handlers currently have a PopRequest pending, or 0 (False) if PopRequest is already servicing another handler's request.

Comments

Because PopRequest has been designed as a function, it must be declared before it may be used.

PopRequest is a major and important feature that lets you perform nearly any service from within a manual interrupt handler. The explanation and examples are fairly lengthy, so PopRequest is described separately in the section entitled *Using PopRequest*.

PopUpHere subroutine

Purpose

PopUpHere is used in "simplified" P.D.Q. TSR programs, to show where in the BASIC program control is to go when the specified hot key is pressed.

Syntax

```
CALL PopUpHere(HotKey%, ID$)
```

Where

HotKey% indicates which key is being used as the hot key, and the code that begins two lines after the call to PopUpHere will receive control when that key is pressed. ID$ is the unique identification string that is required by all P.D.Q. TSR programs.

Comments

PopUpHere is intended for creating pop-up TSR programs using P.D.Q. It indicates where in your BASIC program control is to go when the specified hot key has been pressed. The statement following the call to PopUpHere must be a GOTO, and the line that follows the GOTO is the one that receives control each time the specified key is pressed. Programs that use PopUpHere to gain control when a hot key is pressed must use the complementary PopDown routine to return to the underlying application.

PopUpHere should be called only once, since a simplified TSR can only have one entry point. Subsequent calls to PopUpHere are ignored.

PopUpHere will also fail if another copy of the program is already resident in memory. A copy is defined as another program having an identical ID string. Other, previously installed copies may be detected with the TSRInstalled function.

When the resident section of the P.D.Q. program gets control, you must not use BASIC's INKEY$ function, the PDQInkey function, PDQInput, or any DOS interrupt service lower than &H0D (13).

PopUpHere intercepts the interrupts listed in Table V-8.

If the TSR cannot gain control safely when the hot key is pressed, it will continue to try for approximately one second before giving up. This time may be altered by changing an Equate constant in the assembly language source code.

TABLE V-8
Interrupts Intercepted By PopUpHere

8, 9, 10, 13, 14, 16, 17, 21, 25, 26, 28

All preparations for file and device I/O are made by this routine, and there is no need to call the TsrFileOn and TsrFileOff routines.

TSR programming and defining the hot key and ID string are described in depth elsewhere in this manual.

Power and Power2 functions

Purpose

Power and Power2 raise any number to a power, or the value 2 to a power respectively.

Syntax

```
X = Power(Y%, Number%)
X = Power2(Number%)
```

Where

The first example assigns X equal to Y% ^ Number% and the second example assigns X equal to 2 ^ Number%.

Comments

Because Power and Power2 have been designed as functions, they must be declared before they may be used.

Power and Power2 may be declared as either integer or long integer functions, depending on the expected range of return values. The PDQDECL.BAS file declares them as long integer functions, to accommodate either situation. If you declare Power or Power2 as integer functions and the result exceeds 32,767, the number returned may be negative.

Whenever you use BASIC's exponentiation operator (^) the numbers are converted to floating point. Although P.D.Q. does support floating point math, raising a number to a power is not supported in this version. Therefore, these functions let you perform integer-only exponentiation in your P.D.Q. programs.

RedimAbsolute subroutine

Purpose

RedimAbsolute lets you assign any arbitrary segment to an existing dynamic array.

Syntax

```
CALL RedimAbsolute(Array(), LoBound%, UpBound%, NewSeg%)
```

Where

Array() is an existing dynamic array, LoBound% and UpBound% indicate the new upper and lower element numbers, and NewSeg% specifies the new segment that references to elements in Array() will access.

Comments

RedimAbsolute is useful in a variety of situations. For example, you can use it to treat video memory or the interrupt vector table as a BASIC array. It can also be used to mimic a C-style Union whereby different data items overlap the same area of memory.

It is essential that the array already exist, and also that it is dynamic. RedimAbsolute can be used with any type of dynamic arrays except conventional (not fixed-length) string arrays.

To establish an integer array with elements numbered 1 through 2000 using color text-mode display memory you would do this:

```
REDIM Array%(1 TO 1)
CALL RedimAbsolute(Array%(), 1, 2000, &HB800)
```

Because the original array contents remain in memory, you should use only one element initially to minimize memory waste. Note that there is *no way* to make this routine work in the QB or QBX editor.

Once you have used RedimAbsolute, do not attempt to use ERASE or REDIM with the array. You can, however, use RedimAbsolute again any number of times with the same array.

Do not call RedimAbsolute within a non-static SUB or FUNCTION procedure if the array was first dimensioned within that procedure.

See REDIMABS.BAS for more information and some examples.

ReleaseMem function

Purpose

ReleaseMem releases a block of memory that had previously been allocated with the AllocMem routine.

Syntax

```
ErrorFlag = ReleaseMem%(Segment%)
```

Where

Segment% is the segment value that was originally returned by AllocMem. ErrorFlag receives either zero if the memory was successfully released, or -1 if it was not.

Comments

Because ReleaseMem has been designed as a function, it must be declared before it may be used.

The only error that is likely to occur would be caused by specifying an invalid segment.

Also see the complementary AllocMem function, which allocates memory.

ResetKeyboard statement

Purpose

ResetKeyboard is needed by any P.D.Q. TSR program that intercepts the keyboard interrupt (Interrupt 9) directly. It is not needed by programs that use the simplified PopUpHere/PopDown method.

Syntax

```
CALL ResetKeyboard
```

Where

The keyboard and 8259 programmable interrupt controller (PIC) are reset.

Comments

Each time a key is pressed, an Interrupt 9 is generated by the keyboard hardware. Any program that is inserted in the Interrupt 9 chain will then receive control. If the key pressed is not going to be acted upon, then the TSR should use GotoOldInt to pass control on to the original (or subsequent) keyboard handler. However, if the key that was pressed is the correct hot key, and the original BIOS routine will not be called, then the keyboard hardware must be reset manually.

TSR programming is described in depth elsewhere in this manual.

ReturnFromInt subroutine

Purpose

ReturnFromInt is used to return control to the currently executing program when an interrupt handler has finished servicing an interrupt.

Syntax

```
CALL ReturnFromInt(Registers)
```

Where

Registers is the TYPE variable that holds the register values for this interrupt.

Comments

ReturnFromInt is primarily intended for use in P.D.Q. TSR programs, and it provides a way to return to the underlying application after an interrupt has been processed. However, interrupts may also be handled in non-TSR programs.

TSR programming and interrupt handling are described in depth elsewhere in this manual.

SeekLoc function

Purpose

SeekLoc is for use with the P.D.Q. SMALLDOS library, and it calculates a binary offset given a record number and record length.

Syntax

```
Offset& = SeekLoc&(RecNumber%, RecLength%)
```

Where

RecNumber% is the desired 1-based record number to be read or written, RecLength% is its length in bytes, and Offset& receives the offset into the file where the record begins.

Comments

Because SeekLoc has been designed as a function, it must be declared before it may be used.

When you use the PUT statement to write a record to a random access file, BASIC calculates the offset to seek to based on the record length it saved when the file was opened. But if you are linking with the SMALLDOS library to reduce a program's size, random access operations are not permitted. SeekLoc can therefore be used to quickly calculate the necessary offset for an equivalent operation using the binary form of PUT.

The formula SeekLoc uses is as follows:

```
SeekLoc& = ((RecNumber% - 1) * RecLength%) + 1
```

SeekLoc is demonstrated in the RANDOM.BAS example program.

Set1Byte subroutine

Purpose

Set1Byte assigns a single byte to the specified segment and element.

Syntax

```
CALL Set1Byte(Segment%, Element%, Value%)
```

Where

Segment% and Element% indicate where in memory the byte is to be assigned, and Value% ranges either from -128 to 127, or 0 to 255.

Comments

Element numbers start at one; there is no element zero.

Set1Byte is intended primarily for accessing memory that was allocated using the P.D.Q. AllocMem function. This effectively adds a new "byte" variable type, which of course occupies less memory than a conventional integer.

Please see the comments that accompany the Get1Byte function.

Also see the related Set1Word, Set1Long, and Set1Type routines.

Set1Long subroutine

- **Purpose**

  Set1Long assigns a long integer to a specified segment and element.

- **Syntax**

```
CALL Set1Long(Segment%, Element%, Value&)
```

- **Where**

  Segment% and Element% indicate where in memory the value is to be assigned, and Value& is any long integer value.

Comments

Element numbers start at one; there is no element zero.

Set1Long is intended primarily for accessing memory that was allocated using the P.D.Q. AllocMem function.

Please see the comments that accompany the Get1Byte function.

Also see the related Set1Byte, Set1Word, and Set1Type routines.

Set1Type subroutine

- **Purpose**

  Set1Type assigns a TYPE variable to a specified segment and element.

- **Syntax**

```
CALL Set1Type(Segment%, Element%, Length%, TypeVar)
```

Where

Segment% and Element% indicate where in memory the element is to be assigned, Length% is its length in bytes, and TypeVar is the TYPE variable in near memory that will be copied there.

Comments

Element numbers start at one; there is no element zero.

Set1Type is intended primarily for accessing memory that was allocated using the P.D.Q. AllocMem function.

Because the P.D.Q. memory allocation routines are limited to allocating 64K (65,536) bytes at one time, the maximum number of elements that may be assigned will vary, depending on their length. For example, if you have created a TYPE variable that is 100 bytes long, then only 655 elements may be stored in a sine array.

Using LEN(*TypeVar*) as the length parameter causes BASIC to use the correct value even if the TYPE structure is changed.

Please see the comments that accompany the Get1Byte function.

Also see the related Set1Byte, Set1Long, and Set1Word routines.

Set1Word subroutine

Purpose

Set1Word assigns a word (two bytes) to the specified segment and element.

Syntax

```
CALL Set1Word(Segment%, Element%, Value%)
```

Where

Segment% and Element% indicate where in memory the word is to be assigned, and Value% is the integer value.

Comments

Element numbers start at one; there is no element zero.

Set1Word is intended primarily for accessing memory that was allocated using the P.D.Q. AllocMem function.

Please see the comments that accompany the Get1Byte function.

Also see the Set1Byte, Set1Long, and Set1Type routines.

SetDelimitChar subroutine

Purpose

SetDelimitChar lets you change the default delimiter recognized by the PDQParse function.

Syntax

```
CALL SetDelimitChar(NewChar%)
```

Where

NewChar% is the ASCII value of the new delimiting character.

Comments

See the PDQParse routine elsewhere in this section for a discussion of using SetDelimitChar.

Sort subroutine

Purpose

Sort is a general purpose assembly language Quick Sort for string arrays.

Syntax

```
CALL Sort(BYVAL VARPTR(Array$(Start)), NumEls%, Direction%)
```

Where

Array$(Start) is the first element in the portion of the array being sorted, NumEls% is the total number of elements to include, and Direction% is 0 to sort ascending, or anything else to sort descending.

Comments

Sort is intended for use with conventional (not fixed-length) string arrays only.

The alternate _SORT.OBJ stub file contains a much smaller version of this routine, but at the expense of sorting speed. Please see the section entitled *Linking With Stub Files* for more information on this and other reduced-capability versions of P.D.Q. statements and extensions.

StringShort function

Purpose

StringShort reports the number of bytes, if any, that were requested but not available in the P.D.Q. string pool.

Syntax

```
BytesShort = StringShort%
```

Where

BytesShort receives the number of additional bytes that are needed by your program.

Comments

Because StringShort has been designed as a function, it must be declared before it may be used.

You may also use the BASIC FRE(0) and FRE("") functions, which return the number of bytes that are free.

See the section *String Memory Considerations* for more information about the P.D.Q. string pool.

StringUsed function

Purpose

StringUsed reports the number of bytes that are currently in use within the P.D.Q. string pool.

Syntax

```
BytesUsed = StringUsed%
```

Where

BytesUsed receives the current number of bytes in use.

Comments

Because StringUsed has been designed as a function, it must be declared before it may be used.

StringUsed can tell you how many bytes of string memory are actually needed by your program. However, you should first use the FRE("") function to compact string space. If a given program needs, say, only 4000 bytes, then you could link it with the STR04096.OBJ file to reduce the amount of memory required when it runs.

Using the alternate STR*xxxxx*.OBJ string pools is discussed in the section entitled *Linking With Stub Files*. Also see the section *String Memory Considerations* elsewhere in this manual.

The BASIC FRE(0) and FRE("") functions return the number of bytes that are free.

StuffBuf subroutine

Purpose

StuffBuf places characters into the PC's keyboard buffer, as if they had been entered by the user.

Syntax

```
CALL StuffBuf(Argument$)
```

Where

Argument$ is a string up to fifteen characters long. Argument$ may exceed fifteen characters, but only when used to specify extended keys.

Comments

StuffBuf is useful in a variety of applications, for example to run batch files from a BASIC program, or to insert keystrokes from a TSR program into an underlying application. To run a batch file you would call StuffBuf specifying its name, and then immediately end the program.

Because StuffBuf is limited to only fifteen keystrokes, you may not be able to run programs located in another directory if the complete path name exceeds this limit. In that case you should create a batch file in the current directory from within the program, and then run that batch file.

To "stuff" an extended key you must specify it as a two-character combination, where the first character is CHR$(0), and the second is the ASCII equivalent of the key's extended code. The example below specifies the F3 key, followed by the string "TEST", followed by an Enter key press (for a total of six keystrokes).

```
CALL StuffBuf(CHR$(0) + CHR$(61) + "TEST" + CHR$(13))
```

Please note that any characters which are already present in the keyboard buffer are discarded when StuffBuf is called.

Specifying more than fifteen keys or using CHR$(0) for other than the first of a valid two-character single-key sequence is guaranteed to cause a crash.

Swap2Disk function

Purpose

Swap2Disk lets you create "swapping" TSR programs whose code and data are removed from memory when they are not active.

Syntax

```
Success = Swap2Disk%(FileName$, ProgramID%)
```

Where

FileName$ is the name of the swap file to use, and ProgramID% is either 0 or the program number when more than one TSR program will be invoked using CALL INTERRUPT.

Success receives either -1 if sufficient free disk space was available, or 0 if swapping to disk cannot be performed.

Comments

Because Swap2Disk has been designed as a function, it must be declared before it may be used.

See the section entitled *TSR Programs That Swap To Disk Or EMS* for information on naming the swap file and accessing a swapping TSR with CALL INTERRUPT.

Swap2EMS function

Purpose

Swap2EMS lets you create "swapping" TSR programs whose code and data are removed from memory when they are not active.

Syntax

```
Success = Swap2EMS%(ProgramID%)
```

Where

ProgramID% is either 0, or the program number when more than one TSR program will be invoked using CALL INTERRUPT.

Success receives either -1 if sufficient expanded memory is available on the host PC, or 0 if swapping to EMS cannot be performed.

Comments

Because Swap2EMS has been designed as a function, it must be declared before it may be used.

See the section entitled *TSR Programs That Swap To Disk Or EMS* for information on assigning the program ID value.

SwapCode function

Purpose

SwapCode lets a swapping TSR program receive information from another application that caused it to pop up using CALL INTERRUPT.

Syntax

```
Parameter = SwapCode%
```

Where

Parameter receives whatever was in the BX register when CALL INTERRUPT was used to pop up this swapping TSR program.

Comments

Because SwapCode has been designed as a function, it must be declared before it may be used.

See the section *TSR Programs That Swap To Disk Or EMS* elsewhere in this manual for information on using SwapCode.

TestHotKey function

Purpose

TestHotKey is used to test the key that was just pressed, as part of an Interrupt 9 keyboard handler.

Syntax

```
IF TestHotKey%(KeyCode%) THEN ...   'this is our key
```

Where

KeyCode% is a special code that indicates which key is being tested for.

Comments

Because TestHotKey has been designed as a function, it must be declared before it may be used.

KeyCode% is comprised of two elements, a *shift mask* and a *scan code*. The shift mask lets you specify Alt, Ctrl, either Shift key, or any combination. The scan code is the keyboard scan code for the key being tested. These are described in detail in the section that discusses TSR programming elsewhere in this manual.

Although TestHotKey will check for a single specified key, there is no way to have it return which key was just pressed. The following code fragment shows how to do this:

```
ScanCode = INP(&H60)      'first get the current key manually
DEF SEG = 0               'then look in low memory at the shift
                          '  status byte
ShiftMask = PEEK(&H417) AND &HF
```

Then to combine these two bytes into a single word using the same method that PopUpHere and TestHotKey uses, add and multiply as follows:

```
KeyCode = ScanCode + (256 * ShiftMask)
```

TSRFileOff subroutine

Purpose

TSRFileOff works in conjunction with the TSRFileOn routine to ensure that file operations performed by a P.D.Q. TSR program do not fail because of conflicts with the underlying application.

Syntax

```
CALL TSRFileOff
```

Where

The underlying application's PSP and DTA address are restored.

Comments

TSRFileOff restores the program state that was saved by TSRFileOn. It uses DOS functions 1AH (Set DTA) and 50H (Set Active PSP).

It is not necessary to use TSRFileOff in a "simplified" TSR program.

Please see the discussion that accompanies the TSRFileOn routine.

TSRFileOn subroutine

Purpose

TSRFileOn is meant for use in a P.D.Q. TSR program, and it saves the portion of the current machine state that is affected by DOS-based file or device I/O in a TSR.

Syntax

```
CALL TSRFileOn
```

Where

The underlying application's PSP and DTA address are saved, and then a new, local PSP address is assigned for subsequent use by DOS.

Comments

In order for a TSR to function properly, it must exactly restore the state of the PC and DOS to what they were when it gained control. TSRFileOn, and its companion TSRFileOff, perform the task of saving and restoring respectively two important conditions. TSRFileOn should be called before performing any file or device I/O through DOS.

Note that TSRFileOn itself uses three DOS functions, 2Fh (Get Current DTA), 50h (Set Active PSP) and 51h (Get Active PSP). The assumption is that if the P.D.Q. environment is such that it is safe to perform DOS disk activity, then these functions will also execute safely.

If you call TSRFileOn, it is imperative that you also call TSRFileOff before returning to the underlying application.

Also note that a TSR that has gone resident and uses this call may open files when activated, and leave them open between activations. This is because the file information is stored in the PSP of the TSR, not the currently active process. DOS will not close these files.

It is not necessary to use TSRFileOff in a "simplified" TSR program. We recommend using the PopRequest function if you need to perform file operations in a manual interrupt handling TSR. Because PopRequest ties into the simplified PopUpHere routine, TSRFileOn and TSRFileOff are not needed.

TSRInstalled function

Purpose

TSRInstalled is used inside a P.D.Q. TSR program to determine if another copy of the same program already exists in memory.

Syntax

```
DGroup% = TSRInstalled%(ID$)
```

Where

ID$ is the unique identification string required by every P.D.Q. TSR program, and DGroup% receives either 0 if the program is not already resident, or the DGROUP data segment of the previously installed copy if it is.

Comments

Because TSRInstalled has been designed as a function, it must be declared before it may be used.

If your program accepts a command-line switch as a flag to deinstall its resident copy, then you must invoke TSRInstalled before calling EndTSR.

Knowing if a TSR program has already been installed is important in a number of situations. For example, many TSR programs use a command line switch such as /U to indicate that the user wants to remove it from memory. By knowing if it is already installed, a program could either deinstall the resident copy, or display an appropriate error message when such a switch is used.

While the DGROUP address of the resident copy does not need to be known from the point of view of the P.D.Q. program, it identifies the data segment of the resident version that contains all of the information regarding the program. This value must be known to perform certain functions, such as removing the resident copy or unhooking an interrupt.

Knowing this segment also allows a subsequent copy of a program to modify data in the resident copy of the same program. This is shown in ENVELOPE.BAS whereby running the program additional times lets you change parameters.

When writing a TSR program that is designed to be loaded only once, it is appropriate to detect an attempt by the user to load multiple copies and prevent it. Reference TSRInstalled early in your program, before you call EndTSR to terminate and stay resident. If TSRInstalled returns a value of zero, no other copy of the program is installed and the program can proceed to install itself normally. If the returned value is non-zero, then another copy of the program is already resident. A message such as "Program already resident" should be displayed. Then end the program, but without calling EndTSR.

Please see the section *TSR Programming With P.D.Q.* Also see the PopDeinstall and DeinstallTSR functions described elsewhere in this manual, and the TEMPLATE.BAS program skeleton you can use as a starting point for your own TSR programs.

UnhookFP subroutine

Purpose

UnhookFP restores the floating point interrupts in a P.D.Q. TSR program to their original contents after they had been hooked by a previous call to UnhookFP, or automatically as part of the P.D.Q. startup process.

Syntax

```
CALL UnhookFP
```

Where

The original floating point interrupt vectors (&H34 through &H3C) are restored.

Comments

See the topic *Floating Point Considerations* in the section *TSR Programming* for information about using this routine.

Also see the description for HookFP and the POPUPFP.BAS sample program for an example of using floating point math in a P.D.Q. simplified TSR.

UnhookInt function

Purpose

UnhookInt lets a P.D.Q. program that intercepts interrupts remove itself from the interrupt chain, thus reversing the action of PointIntHere.

Syntax

```
Success = UnhookInt%(Registers, DGroup%)
```

Where

Registers is the TYPE variable that was established for this interrupt, and DGroup% is the value returned by TSRInstalled when the program was first run. Success then receives either -1 if everything went as planned, or 0 if the interrupt was not able to be unhooked.

Comments

Because UnhookInt has been designed as a function, it must be declared before it may be used.

UnhookInt attempts to restore the interrupt vector that was in effect when PointIntHere redirected it. If another process has since changed that

vector, it cannot be restored. Therefore, programs that chain into an interrupt vector must be unchained in reverse order.

The DGroup% parameter is normally set to the value returned by TSRInstalled when this routine is used to deinstall a manually installed Interrupt Service Routine (ISR). If you are handling interrupts in a non-TSR program, use a value of zero for DGroup%.

UnhookInt0 subroutine

Purpose

UnhookInt0 restores the "Divide by zero" interrupt (Interrupt 0) after using HookInt0 to disable that condition.

Syntax

```
CALL UnhookInt0
```

Where

The interrupt vector for Interrupt 0 is restored.

Comments

See the comments that accompany HookInt0 earlier in this section.

P.D.Q.
Extensions

Chapter 6: Using P.D.Q. With Assembly Language Programs

Using P.D.Q. with Assembler

Introduction

If P.D.Q. is a tool for BASIC, why should assembly language programmers be interested? Because P.D.Q. is, at heart, a collection of assembly language routines. You can use those routines to add BASIC-like power to your assembly language programs. Instead of struggling over the parts that would be easy to write in BASIC, you can let P.D.Q. take care of the details while you focus on the important parts of your program.

If you are familiar with both BASIC and assembly language, you'll soon feel at home working with the P.D.Q. library. You will still have the control that assembly language programmers insist on, but you will also have the capabilities of a tested, flexible set of library routines that let you write high quality programs very quickly.

When you add P.D.Q. to your arsenal of assembly language tools, you get:

- BASIC-like string handling including management of dynamic string memory
- Simple and powerful screen and keyboard I/O
- BASIC-like file handling
- A powerful method of allocating and managing arrays
- A floating point emulation library that will let you add real number calculations to your program painlessly
- A simple method for writing TSR programs and interrupt handlers

P.D.Q. also includes the complete, commented source code for all of its library routines. If you want to see how we handle dynamic strings or floating point emulation, all you need to do is study the code. If you want to change the way that a P.D.Q. routine works, you can do so easily.

This section of the P.D.Q. manual is divided into two parts. The first chapter explains the mechanics of working with the P.D.Q. library in an assembly language program. Please read that chapter carefully. If you forget an important step or use the wrong assembler directives, your program may not run at all.

You'll find a discussion of assemblers, memory models, segment names, and calling conventions in this chapter, along with a description of P.D.Q.'s method of handling strings, arrays, errors, and floating-point emulation.

The second chapter is a programmer's reference. It contains detailed information about each of the P.D.Q. memory variables and routines you may want to use in your own programs. Look up the information you need for your present project, and browse through the chapter often to see what is available.

Although a lot of information is presented in this section, we have not duplicated the discussions that appear elsewhere in the P.D.Q. manual. For more information about what a particular routine does or why you might want to use, you should refer to the reference portion of this manual.

Assembly Language Details

Please read this chapter carefully. P.D.Q. has a lot to offer assembly language programmers, but you have to abide by its rules. If you just start calling the routines that are described in the next chapter without a little preparation, you will almost certainly cause a system crash and end up spending a lot of time chasing bugs. This chapter has the information you will need to make the best use of the P.D.Q. library in your own programs.

To use P.D.Q. routines in your own assembly language programs, you will have to follow many of the same rules as the BC.EXE BASIC compiler. Those rules include the way your program initializes itself, the way you use segments in your program, and the way you set up to call the routines. In effect, the assembly language code you write will be similar to the code that BASIC compilers create—but more efficient.

If you are confused about how to do something—perhaps how to code a floating point multiplication—write a short test program with QuickBASIC or BASIC 7. Then, from the command line, compile it with the /a switch. The compiler will create an assembly language listing file which you can examine. You can also compile your test program to include CodeView debugging information, and look at the details with CodeView or a compatible debugger. In either case, you will see the way that BASIC writes the assembly code and you can use the same techniques in your own program.

Throughout this section of the manual, we assume that you have some experience with assembly language and some experience with QuickBASIC or BASIC 7 PDS. We assume that you have (or will) read the applicable sections in the rest of this manual about P.D.Q. and its extensions to the BASIC language. You will also need to read the sections about TSR programming if you want to use the P.D.Q. library to write memory-resident programs.

The remainder of this chapter explains the general techniques of using P.D.Q. with your assembly language programs. The next chapter contains detailed discussions of each of the P.D.Q. routines and variables that you may want to use.

Choice Of Assembler

Before you can write an assembly language program, you need an assembler. To call the P.D.Q. library procedures you will need an assembler that can create Microsoft-compatible object (.OBJ) files. You cannot use the DOS DEBUG utility because it does not create object files at all. Some shareware or freeware assemblers are unsatisfactory because they cannot create the appropriate object files either.

Specifically, your assembler must be able to create object files that contain the same segment and naming information that are included in Microsoft's object files. You don't necessarily need to create CodeView-compatible object files unless you want to use Microsoft's CodeView debugger.

If you want to use the P.D.Q. floating point emulator or any of the library routines that support floating point operations, your assembler must also be able to create the correct LINK fixup information.

We suggest that you use Microsoft's MASM version 5.1 or later or a compatible assembler. MASM 5.1 meets all of the criteria above. The examples in this and the next chapter make use of several of MASM's "high-level language" features, which were introduced in that version of MASM. We relied on many of those features to write the source code for the P.D.Q. library and the examples in this manual. With the introduction of MASM 6.0 even more "helper" features have been added to the language.

We have not tested the P.D.Q. library with any assemblers other than MASM 5.1 and 6.0. However, Borland International's TASM and SLR Systems' OPTASM assemblers should both be satisfactory. Other assemblers may work as well. If you are using an assembler like TASM that includes options for MASM 5.1 compatibility, we strongly suggest that you use those options.

This chapter includes information about using an assembler that can create appropriate object files, but which doesn't support MASM's high-level language or simplified directive features. If you are using such an assembler, it is your responsibility to apply those techniques to the examples in both this chapter and the next.

Memory Models

The segmented memory architecture used by the 8086 CPU (and the 80286, 386, and 486 in real, or DOS mode) gives every location in memory both a segment and an offset address. The memory model that you choose for a program establishes what items will require explicit segment addresses and what items will have implied segment addresses.

The P.D.Q. library routines have been written to work with the *Medium* memory model. In this memory arrangement, procedure calls are far by default. This means that procedures must be called by specifying both a segment and offset address, and that the called procedures must end with a far return, usually to a different segment.

The Medium memory model also assumes that all data objects are in one segment and that the segment part of their addresses is always in both the DS and SS segment registers. This doesn't mean that your program is limited to 64K bytes of data, but it does mean that you will have to do some extra work to access memory that is not normally addressable via the DS segment register.

To write a program that uses the P.D.Q. library you should specify the Medium memory model at the beginning of each of your assembly language modules. If you are using MASM 5.1, the correct memory directive is as follows:

```
.Model Medium, BASIC
```

The *Medium* directive word tells the assembler to make far calls to procedures (using both a segment and an offset address) by default, and to use far returns as a default. Most assemblers use the same command to specify the medium memory model.

The *BASIC* directive word tells the assembler how to write the prologue and epilogue for each procedure, which sets up and releases the stack frame. It also tells the assembler how arguments will be arranged on the stack for calls from one procedure to another. Finally, when you add the BASIC directive, MASM 5.1 and compatible assemblers will correctly interpret the OFFSET assembler directive in your programs. Adding BASIC also makes all of your procedures public automatically.

You can override the default settings of the medium memory model quite easily when you need to do so. Generally, procedure calls from one object module to another must be far, but calls entirely within a module may be either near or far at your discretion. If you don't do anything other than use the PROC and ENDP directives at the beginning and end of each

procedure, calls to those procedures will use far addressing by default. If you define a procedure with the directive PROC NEAR, calls to that procedure will require an offset address only and the procedure will end with a "near" return. But be careful! You will be able to call near procedures only from within their own .OBJ modules, and not from other modules.

Programs written in the Medium model can access more than 64K bytes of data memory. All you need to do is to use a DOS call to allocate a block of memory for your program and then remember where that block is. The block is yours to use as you wish and will be released back to DOS when your program ends.

If your assembler does not support the *Model Medium, BASIC* directive described above, you will have to do some extra work. When you call P.D.Q. library routines, be sure to tell your assembler that they are in far memory, using whatever mechanism your assembler supports. For some assemblers, you may define the location (near or far) of a procedure when you declare it as external to the current module. Such a declaration line usually looks something like this:

```
Extrn PDQProcedure:FAR
```

You should put this declaration outside of all segments in order to avoid a possible fixup overflow error from the linker if the procedure happens to be physically more than 64K bytes from the calling address.

When you use the *.Model Medium, BASIC* directive and incoming parameters have been defined, MASM 5.1 creates a procedure prologue and epilogue automatically. Each procedure begins with the following instructions:

```
PUSH BP                 ;Save caller's BP
MOV  BP,SP              ;Establish stack frame
```

This creates a local stack frame that is addressable via the BP register. The prologue may also include space for local variables and may save some registers on the stack automatically. The epilogue retrieves any saved registers from the stack and then performs these instructions:

```
MOV  SP,BP              ;Restore original SP
POP  BP                 ;Restore caller's BP
RET
```

If the procedure declaration includes the names of values received on the stack, the RET command is modified to remove them from the stack before returning to the calling procedure. For example, if there are two incoming word-sized arguments, MASM creates the instruction RETF 4 when you use RET alone.

If you are using an assembler which is not compatible with MASM 5.1, you will have to write any necessary prologue and epilogue yourself. You will also have to make other changes to your program, which will be explained in the following sections.

Finally, you will have to decide how you want to pass parameters and values between your procedures, and how you will set up stack frames (if you need them) in each of your own procedures.

Segments, Segment Names, And DGROUP

When programmers talk about an 80x86 memory segment, they usually mean the 64K bytes of memory that can be addressed without changing a particular segment register. For example, if the value in DS is 1000h, then a program can access memory from 10000h to 1FFFFh (1000:000 through 1000:FFFF) without altering DS.

But to assembly language programmers, the word segment also means a block of named memory. The block can be any length from 0 bytes to 64K bytes. You begin one of these named segments with a SEGMENT directive and end it with an ENDS directive in your assembly language program. MASM 5.1 and compatible assemblers also let you use simplified commands for starting and stopping segments with standard names.

Code Segments

In the Medium memory model, the source code in every object file may be in a different named segment. By tradition, programmers use the filename of the module plus "_TEXT" as the name of the code segment. For example, if you have a source code file called CLEANUP.ASM, the code in that module would be placed in a segment called CLEANUP_TEXT. The code segment will be just large enough to hold the code in that particular source code module.

MASM 5.1 and compatible assemblers will create the correct code segment name for you if you begin each section of code with the directive .CODE. For example, you would begin your source code module this way:

```
.Model Medium, BASIC
.CODE
  MyProc Proc
```

If you prefer to write the segment directives yourself, you can do so this way:

```
filename_TEXT    SEGMENT WORD PUBLIC 'CODE'
...              ;your code goes here
filename_TEXT    ENDS
```

The result will be the same, but you will have to do more typing. In exchange, you will be able to put the procedures from multiple source code files into the same code segment, if you wish. However, remember that almost every P.D.Q. library routine is in its own code segment.

Data Segments

In the Medium memory model, all data by default must fit within a single 64K memory segment. You could use just one named segment for the entire data area, but you will give up some of the assembler's and linker's power by doing so.

Instead, by convention, several named data segments are combined with a GROUP directive. The resulting group still cannot exceed 64K bytes in length, but you can control data placement much more easily by using individual named segments for each kind of data. By convention, the entire data group is called DGROUP, and that convention is followed in the P.D.Q. library.

The size of DGROUP can range from 2 bytes to 64K bytes in length. In BASIC programs, DGROUP contains all scalar (non-array) variables, all strings and string descriptors, all static arrays, all array descriptors (whether static or dynamic), DATA statements, quoted strings, and the program stack. Arrays of numeric values, fixed-length strings, and TYPE variables may optionally be stored outside of DGROUP by establishing the arrays as dynamic.

Many of the routines in the P.D.Q. library assume that you have followed this standard organization. If you haven't, you will have to study the P.D.Q. source code carefully to make sure that it is compatible with the data organization that you want to use.

Inside DGROUP, there are three named segments that you will be concerned with the most. These segments contain initialized data, uninitialized data, and your program's stack.

Initialized Data

The difference between initialized and uninitialized data is suggested by their names. You give an initial value to each item of initialized data when

you assemble the program. If you use an initial value of "?" then the assembler will assume that you want to initialize the data to 0.

The initial values are stored in your .EXE executable file, along with your program code. If you are using MASM 5.1's simplified naming conventions, the initialized data segment is called .DATA. If you would rather name segments yourself, use the following definition for the segment:

```
_DATA SEGMENT WORD PUBLIC 'DATA'
```

Uninitialized Data

When you use uninitialized data, the assembler and linker reserve space for your data items but they do not store any initial values in the .EXE executable file. Your programs will be shorter and load faster if you make sure to put as many data items as possible into the uninitialized data segment.

The starting value of uninitialized data items is undefined. Normally, it is whatever bytes happen to be sitting around in memory from the last program. If you use uninitialized data, you must be sure that your code assigns a value to each item before it tries to read from that item. Note that many of the syntax examples that will follow show data in the uninitialized data segment. It is assumed that you have already assigned that data, or at least cleared it to zeros. Of course, you may also store data in the initialized segment.

If you use simplified segment names, use .DATA? to start each uninitialized data segment. If you use full names, use _BSS as the name of the segment:

```
_BSS SEGMENT WORD PUBLIC 'BSS'
```

In order to prevent the assembler and linker from putting uninitialized data into your executable file, you have to make sure that you and the assembler agree that you don't care about the starting values of these data items. The only way to do so with MASM is to define every uninitialized data item using a ***name count*** **dup (?)** construction. For example, to define one uninitialized integer value, you can use these commands:

```
.DATA?
  my_integer dw 1 dup (?)
```

You may, of course, replace the count of 1 with any other number if you are defining an array. But if you omit the *dup (?)* part of the definition and use a specific value or even a question mark without the *dup* directive, the assembler and linker will decide that all data items in _BSS should have initial values. And all of the values from *all* modules will be placed

into your .EXE file. This won't hurt the program at all, but it will make the .EXE file much larger than it needs to be.

If the size of your program suddenly seems to balloon from one version to another, the likely reason is that you added an uninitialized data item without using *dup (?)*. Please understand that your program will be the same size in memory no matter what mix of initialized and uninitialized data it contains. The only difference is the size of the .EXE executable file.

The Stack

Your program's runtime stack is also part of DGROUP. The runtime stack is defined in STARTUP.ASM, which sets the stack size to 1,024 bytes. This is usually sufficient, but if you need a larger stack, you can edit STARTUP.ASM and compile a new STARTUP.OBJ file to link with your program. Easier still, you can use the /stack: linker directive to change the stack size when you link your program.

The stack is used for procedure return addresses, for passing parameters from one procedure to another, and as temporary variable space by several routines in the P.D.Q. library. You probably won't have to increase the size of the stack unless your own procedures are very deeply nested, call themselves recursively, or use large blocks of stack space for temporary variables.

One way to check your program's stack usage is to edit STARTUP.ASM and put a known ASCII value in each byte of the stack. Run your program inside CodeView or another debugger, and put a break point just before the end of the program. Run your program and exercise as much of it as you can. When you hit the breakpoint, look at the stack area and see how much of your ASCII text has been overwritten. You can also compare the current stack pointer to the public variable PDQ_Stack_Foot, as shown in the FRE.ASM source code. In fact, this is a much simpler method.

You should not, however, limit the program's stack size to exactly those bytes which it has used. Give it a little leeway because some device drivers and memory-resident programs also use your stack, and you may not have exercised all possible paths through your code. You'll have to learn what the optimum stack size is by experimenting and experience. In general, we recommend leaving at least 256 bytes for system resources.

Variable References

When you use the OFFSET directive to find the address of a variable the assembler can resolve the address immediately, or let the linker find the address of your variable once you are ready to combine all .OBJ and .LIB modules into a complete program.

If the assembler does the work itself, it finds the variable's address from the beginning of the named segment in the current source file. This is usually not the address you intended, especially if the variable is in a segment which spans several source code modules and .OBJ files. Neither you nor the assembler have any idea of how much space other modules will use in the segment before the current module's variables are placed in memory.

Normally, what you really want to know is the variable's offset address from the beginning of DGROUP, because the DS segment register will contain the address of DGROUP and not the portion of the segment in the current module. If the offset address is calculated by the linker—as opposed to by the assembler—then it will be the offset from the beginning of the group or named segment, which is what you probably intended.

You can find the correct offset address in two ways. If you are using MASM 5.1 or a compatible assembler, you can use the **.Model Medium, BASIC** directive at the beginning of each source file. By including *,BASIC* you are telling the assembler how to address parameters on the stack. This also tells the assembler to defer all Offset calculations to the linker.

If you omit *,BASIC* from the .Model directive, you will have to explicitly defer every offset calculation manually. To do so, you must tell the assembler what reference point to use for each and every Offset command. Generally, that reference point is the beginning of DGROUP, so you will write a line that looks like this:

```
MOV AX,Offset DGROUP:my_variable
```

If your assembler is not compatible with MASM 5.1 and requires you to do that extra typing, you might want to use a text macro to change all *Offset* commands to *Offset DGROUP.*

Omitting *,BASIC* also tells MASM not to make your procedure names public by default.

Assembly Specifics

There are only a few simple rules that you have to follow when you want to add parts of the P.D.Q. library to your assembly language programs. The easiest way to describe the rules is to go through the process of writing a short program, step-by-step.

At the beginning of your program, after any opening comments, you'll need a .Model directive. Assuming that you are using MASM 5.1 or later and simplified segments, your first line should look like this:

```
.Model Medium, BASIC
```

Next, you'll probably want to include the set of simple macros that will help you use some of P.D.Q.'s features. To do so, the next line of your program will look like this:

```
Include MACROS.ASM
```

You can place the .DATA, .DATA?, and .CODE segments in any order you choose, and start and stop segments as often as you like in your source code file. Some programmers like to place all of the data at the beginning of the file; others like to place each data item immediately above the first reference to it. Assuming that you agree with the first group you'll want to define your data items next in the .DATA and .DATA? segments. Remember to use ***name count*** **dup (?)** to create space for all items in the uninitialized data segment.

Your program's code goes into the .CODE segment. You *must* have a procedure called MAIN somewhere in your program; that procedure will be the first part of your program that runs, and will usually contain or call your initialization code.

When your program is loaded by DOS, the P.D.Q. startup code (in the file called STARTUP.OBJ) will get control before any of your code. STARTUP is responsible for doing a small amount of housekeeping to get the P.D.Q. library ready to run. When the initialization code is done, it jumps to MAIN to give your program control of the computer.

At the end of your source code module, close any open procedures, and also any open segments if you're not using simplified segment conventions. Then end the file with the END directive. Do *not* specify the optional starting address as part of the END statement, because the program must begin in STARTUP.OBJ, not in your code.

When you are ready to assemble your program you can use the batch file called C.BAT that is on the distribution diskette. If you prefer you can

simply run MASM directly and assemble your program as you always have. You must use the /e assembler switch if you are using the P.D.Q. floating point library, and you may use the /zi assembler switch if you want to include CodeView debugging information.

To link your program, you must include STARTUP.OBJ as well as the PDQ.LIB library (and perhaps the SMALLDOS library) to make a complete program. You *must* use the /noe link option (No extended dictionary search). You may also use /co to link for CodeView, or /ex, /far, and /packc to get the smallest possible program when you link a final copy. To create a program from a source code module called MYPROG.ASM, you could use the following MASM and LINK command lines:

```
MASM myprog;
LINK /ex/noe/far/packc myprog startup , , nul, \pdq\pdq;
```

This line assumes that STARTUP.OBJ is in the current directory and that PDQ.LIB is in the \PDQ directory.

Calling Conventions In The P.D.Q. Library

The P.D.Q. library has three kinds of procedures that you can call from an assembly language program. Almost all procedures, regardless of their type, receive their arguments on the stack. The routines that return a 2-byte value place that value in AX. A few routines return a 4-byte long integer value in the DX:AX register pair. A few routines receive one or more arguments in registers instead of on the stack.

Each of the extensions that P.D.Q. adds to BASIC is the name of a procedure. These extension procedures normally receive arguments by *reference*, which means that you pass the address of the arguments to the procedure, and not the argument values themselves. For example, suppose you want to call NoSnow to disable CGA snow checking. NoSnow expects to receive a reference, or pointer, to the actual data value that you send it. Therefore, you must place the data into a 2-byte integer, get the offset of the integer, and pass that offset to NoSnow on the stack.

The second class of routines in the P.D.Q. library are internal, "helper" procedures that P.D.Q. uses to get other work done. The names of these routines typically start with *P$*—like P$Delay. These routines normally receive their arguments by value, so to delay 20 timer ticks you would push a 20 onto the stack and then call P$Delay.

The third class of routines are the substitutes for items in the BASIC library. Each of these routines has a name that begins with *B$* and ends with 4 letters, which often seem unintelligible at first glance. The routines have the same names in the P.D.Q. library as they do in the BASIC libraries that Microsoft ships with QuickBASIC and BASIC 7. The names were chosen by Microsoft; please don't blame us for the strange, cryptic nomenclature!

All library routines expect SS and DS to point to DGROUP when you call them. The routines preserve the DS, SS, BP, DI, and SI registers, but are free to change ES, AX, BX, CX, and DX. If you have important data in any of those registers, you should either push them onto the stack or save them in a data variable before you call a library routine. The direction flag should always be cleared when you call a library routine, and it will remain cleared when the routine returns. Any data that you pass to a routine on the stack will be removed before the routine returns.

Table I-1 will help you find the BASIC emulation procedures that you need. It lists each of the BASIC key words that are supported in the library, followed by the corresponding procedure name. Most of the procedures are discussed in detail in the following chapter. Those marked with an asterisk are not generally useful to assembly language programmers and are discussed very briefly at the end of the next chapter.

TABLE VI-1

BASIC Keywords Cross-Referenced By Call Name.

BASIC NAME	LIBRARY ROUTINE(S)
ASC()	B$FASC
BLOAD	B$BLOD
BSAVE	B$BSAV
CHDIR	B$CDIR
CHDRIVE	B$CHDR *
CHR$()	B$FCHR
CLOSE	B$CLOS
CLS	B$SCLS
COLOR	B$COLR
COMMAND$	B$FCMD
CSRLIN	B$CSRL
CURDIR$	B$FCD0, B$FCD1
CVD()	B$FCVD
CVI()	B$FCVI
CVL	B$FCVL
CVS()	B$FCVS
DATE$ function	B$FDAT
DATE$ statement	B$SDAT
DEF SEG	B$DSEG *
DIM (dynamic)	B$DDIM

TABLE VI-1 *(Continued)*
BASIC Keywords Cross-Referenced By Call Name

BASIC NAME	LIBRARY ROUTINE(S)
DIR$	B$FDR0, B$FDR1
END()	B$CENC *
ENVIRON statement	B$SENV
ENVIRON$(Environ$)	B$FEVS
ENVIRON$(n)	B$FEV1
EOF()	B$FEOF
ERASE	B$ERAS
ERR	B$FERR
ERROR	B$SERR
FILEATTR()	B$FATR
FILES	B$FILS
FIX()	B$FIX4, B$FIX8 *
FRE("")	B$FRSD
FRE()	B$FRI2
FREEFILE	B$FREF
GET	B$GET3, B$GET4
HEX$()	B$FHEX
INKEY$	B$INKY
INPUT #n	B$DSKI
INPUT (from keyboard)	B$INPP
INPUT$	B$FINP
INSTR	B$INS2, B$INS3
INT()	B$INT4, B$INT8 *
IOCTL	B$SICT
IOCTL$	B$FICT
KILL	B$KILL
LBOUND	B$LBND
LCASE$()	B$LCAS
LEFT$	B$LEFT
LEN	B$FLEN *
LINE INPUT	B$LNIN
LOC	B$FLOC
LOCATE	B$LOCT
LOCK and UNLOCK	B$LOCK
LOF()	B$FLOF
LPRINT	B$LPRT *
LSET	B$LSET
LTRIM$	B$LTRM
MID$ function	B$FMID
MID$ statement	B$SMID
MKD$	B$FMKD
MKDIR	B$MDIR
MKI$	B$FMKI
NAME	B$NAME
OCT$()	B$FOCT
ON ERROR GOTO	B$OEGA *
OPEN ("wordy" syntax)	B$OPEN
OPEN ("terse" syntax)	B$OOPN
PLAY	B$SPLY

TABLE VI-1 *(Continued)*
BASIC Keywords Cross-Referenced By Call Name

BASIC NAME	LIBRARY ROUTINE(S)
PRINT	B$PESD
PRINT #n	B$CHOU *
PRINT ;	B$PSSD
PRINT X!	B$PER4
PRINT X!,	B$PCR4
PRINT X!;	B$PSR4
PRINT X#	B$PER8
PRINT X#,	B$PCR8
PRINT X#;	B$PSR8
PRINT X$	B$PESD
PRINT X$,	B$PCSD
PRINT X$;	B$PSSD
PRINT X%	B$PEI2
PRINT X%,	B$PCI2
PRINT X%;	B$PSI2
PRINT X&	B$PEI4
PRINT X&,	B$PCI4
PRINT X&;	B$PSI4
PUT	B$PUT3, B$PUT4
RANDOMIZE	B$RNZP
REDIM	B$RDIM
RESET	B$REST
RESTORE	B$RSTA, B$RSTB *
RESUME	B$RESA, B$RESN *
RIGHT$	B$RGHT
RMDIR	B$RDIR
RND	B$RND0
RSET	B$RSET
RTRIM$	B$RTRM
SADD	B$SADD *
SCREEN function	B$FSCN
SCREEN statement	B$CSCN
SEEK function	B$FSEK
SEEK statement	B$SSEK
SETMEM	B$SETM *
SGN	B$SGN4
SHELL	B$SSHL
SLEEP	B$SLEP
SOUND	B$SOND
SPACE$	B$SPAC
SPC	B$FSPC
SSEG	B$SSEG *
STOP()	B$STP1 *
STR$ double-precision	B$STR8
STR$ single-precision	B$STR4
STR$ long integer	B$STI4
STR$ integer	B$STI2
STRING$	B$STRI, B$STRS
SWAP	B$SWPN, B$SWP2, B$SWSD

TABLE VI-1 *(Continued)*
BASIC Keywords Cross-Referenced By Call Name

BASIC NAME	LIBRARY ROUTINE(S)
TAB()	B$FTAB
TIME$ function	B$FTIM
TIME$ statement	B$STIM
TIMER function	B$TIMR
UBOUND	B$UBND
UCASE$ function	B$UCAS
VAL()	B$FVAL
WIDTH (video)	B$WIDT
WIDTH (device or file)	B$DWID, B$FWID *
WIDTH (LPRINT)	B$LWID*

Using P.D.Q. String Routines

The P.D.Q. library contains routines which let your programs manipulate string data as easily in assembly language as you can in BASIC. But you will need to understand how the string routines work and what they expect from you.

Both BASIC and P.D.Q. make a distinction between fixed-length strings and variable-length or "normal" strings. Fixed-length strings are nothing more than a block of memory set aside to hold some data. You can manipulate a fixed-length string just as you would any other data block or buffer. However, you cannot normally perform string operations (like LEFT$, UCASE$, and so forth) on a fixed-length string unless you first copy it into a variable-length string first (using B$ASSN, for example).

The string data normally is inside a block of memory that P.D.Q. maintains called the *string pool*. By default, the string pool is 32,768 bytes long, but you can choose a different size by linking your program to one of the alternate STR*nnnnn*.OBJ files. The *nnnnn* value indicates the number of bytes allocated to the string pool. When you are writing a TSR program, you will probably want to pick the smallest string pool possible to keep the memory requirements of your program low. For other programs, the default will probably be sufficient.

Every variable-length string is stored as three data items: a 4-byte string descriptor, a 2-byte back pointer, and the actual string data. The first word of the descriptor contains the string's length. The second word is the offset of the string data within DGROUP.

String descriptors are usually stored in the .DATA segment just like any other data item, because they should be initialized to zero. However, there is one restriction: it is essential that every string descriptor begin on an even-numbered address.

Each string inside the string pool is preceded by a 2-byte back pointer that contains the address of the string's descriptor. When the address in the back pointer is an even number, P.D.Q. knows that the string data is in use. When it is odd, P.D.Q. assumes that the string data has been discarded and can be erased during string pool compaction. If your string descriptor is stored at an odd address, the entire string pool may be corrupted the next time P.D.Q. compacts the string pool.

The 4-byte string descriptor exists in your normal data space inside DGROUP. It never moves, and its information remains up to date unless your program explicitly changes it, which it should never do. The string data, however, is in the string pool that P.D.Q. maintains, and it may move during almost any P.D.Q. routine call that involves strings.

If P.D.Q. is asked to create a new string, it looks for space in the string pool. The request for a new string can come explicitly from your program or from a P.D.Q. library routine that needs to create a temporary string.

If there is insufficient memory available to create a new string, P.D.Q. calls its string compaction, or "garbage collection" routine. This routine examines the string pool and, when possible, moves strings in memory to overwrite string data that is marked as abandoned. The compaction routine updates the string descriptors associated with valid string data every time it moves a string in memory. After compaction, all of the free space in the string pool will be in one contiguous block.

If there is sufficient space for the requested string after compaction P.D.Q. allocates the space and returns normally. If there is still insufficient space, P.D.Q. allocates as much space as possible for the string.

You can often save significant string pool space by keeping literal text separate from string variables. Normally, a BASIC statement like:

```
KeyPress$ = "Press any key"
```

forces the program to store the text twice, once as a literal value and once again in the string pool. You don't need to copy literal text to the string pool in an assembly language program, even if you want to use the string with P.D.Q. string routines. You can simply store the string in your data list and build your own string descriptor for it. You don't need a back

pointer because the string is outside the string pool and will never be moved by the P.D.Q. heap compaction garbage collection.

To create a literal string and its own permanent string descriptor, you can use the DefStr macro in the MACROS.ASM file. If you do, the code that creates a literal string will look like this:

```
DefStr KeyPress$, "Press any key"
```

In this example, the DefStr macro will create a string descriptor and attach KeyPress$ to it as its label. The descriptor will contain the necessary length word and the address of the literal text, which is stored in your .DATA segment. Other useful macros are provided in MACROS.ASM, and you should look at those as well.

You can set aside space for string descriptors in either the .DATA or .DATA? segments. To do so, use code like this:

```
.DATA?
Even
StringDesc  dd  1 dup (?)
```

or like this:

```
.DATA
Even
StringDesc  dd  0
```


IMPORTANT:

If a string is initialized with constant data using either DefStr or DB it is considered to be a constant, and you must not reassign it later with the P.D.Q. string memory routines.

Temporary Strings

Many of the string functions in the P.D.Q. library return what is called a *temporary string* as their result. For example, the P.D.Q. routines that implement LEFT$, UCASE$, and RTRIM$ all create such a temporary string.

P.D.Q. contains a pool of string descriptors for its internal use. A temporary string is simply a normal string that is attached to one of these descriptors. A back-pointer and the string text are stored in the string pool, and the descriptor is in P.D.Q.'s array of 20 temporary descriptors.

It is your responsibility to release these descriptors for further internal use by P.D.Q. as soon as is practical. If you don't and more than 20 temporary

strings have been assigned, the results will be unpredictable and your program will probably crash.

If you want to keep the contents of a temporary string, use P.D.Q.'s B$SASS routine to copy it to your own string descriptor. B$SASS will then delete the temporary string for you. All other P.D.Q. "B$" routines that accept a string as an argument also delete the string if it is temporary. However, the P.D.Q. extensions do not do this, because BASIC normally adds code to delete temporary strings passed to an external procedure. Note that none of the P.D.Q. routines change any arguments that have descriptors outside of the P.D.Q. temporary string pool.

If you want to delete a temporary string yourself, you have two choices. The safest way is to pass the string to the P.D.Q. routine P$FreeTemp, which examines the descriptor and deletes the string only if the descriptor is in the temporary string pool. A more direct method is to pass the string to B$STDL which deletes any string that it receives.

You can also use B$STDL to delete other strings from the string pool when you no longer have a use for them. By doing so, you release string pool memory for future strings. However, you must not pass a string defined using DefStr (or any other string constant you have defined) to B$STDL. As when assigning, B$STDL is for use with strings whose data resides in the string pool only.

Using Arrays

An array is nothing more than a block of contiguous memory space and some rules for addressing that space. In many cases you will find it easier to manipulate arrays manually using normal assembly language techniques. However, the P.D.Q. routines can be very helpful when you need to manipulate multi-dimensional arrays, or arrays larger than 64k.

P.D.Q. will help you use arrays by creating array descriptors and allocating the necessary memory space, and it will also find the address of individual array items for you. But you can address and use array memory in any way that you wish. If you want to see how BASIC does it, write a short test program and view the results with CodeView.

If you create an array of integers, long integers, single- or double-precision floating point values, fixed-length strings, or TYPE variables, the values themselves are stored in the array. The array can either be stored inside DGROUP, or outside of DGROUP in a memory block allocated with a DOS memory service. Because of the way the DOS memory services work, all dynamic arrays that are stored outside of DGROUP (that is, all

but variable-length string arrays) always start at address 0 in a particular segment.

If you create an array of variable-length strings, the 4-byte string descriptors are stored in the array, which must be inside DGROUP. The text of each string and the corresponding back pointer are stored in the string pool. Please understand that P.D.Q. does all the work of maintaining the string data and back pointers. We describe how dynamic string management works in P.D.Q. for your interest only.

BASIC recognizes two kinds of arrays: static and dynamic. A static array is simply a data block inside DGROUP that is set aside for the array of values. The values can be integers, long integers, string descriptors, floating point values, TYPE variables, or fixed-length strings.

A dynamic array is referenced through an array descriptor that is stored inside DGROUP. The values themselves are stored in a memory block which can either be inside or outside of DGROUP. A dynamic array of variable-length strings has an array descriptor like any other array. But the block of string descriptors must be inside DGROUP and the string text must be stored inside the string pool.

You can create the array descriptor and allocate the memory for a dynamic array with the B$DDIM (or B$RDIM) routine in the P.D.Q. library. Before you do, you must set aside memory inside DGROUP for the array descriptor. The descriptor requires 12 bytes plus 4 bytes for each dimension of the array. The descriptor for a single-dimension array will contain 16 bytes (12 + 1 * 4), 20 bytes for a 2-dimensional array (12 + 2 * 4), and so forth.

The organization of the array descriptor is shown in Table VI-2.

TABLE VI-2
The Organization Of A BASIC Array Descriptor.

OFFSET	SIZE	DESCRIPTION
00	4	Offset and segment of the data block in memory.
04	4	Far heap descriptor and array size. Not used or set by P.D.Q. routines.
08	1	Number of dimensions in the array.
09	1	Array type, stored as a bit record: Bit 0 set means a far (non-DGROUP) array. Bit 1 set means a huge (/ah) array. Bit 6 set means a static array. Bit 7 set means a string array.
0A	2	Adjusted Array Offset for optimized access.
0C	2	Length of each array element in bytes.
0E	2	Number of elements in the *last* subscript. This number is calculated as UBOUND - LBOUND + 1.
10	2	Number of the first element in the *last* subscript (LBOUND).
12	2	Number of elements in the next-to-last subscript.
14	2	Number of the first element in that subscript.
.	2	Repeat as necessary,
.	2	until the *first* subscript is reached.

The Adjusted Array Offset deserves some explanation because it can significantly reduce the amount of time required to find a location in an array. BASIC calculates the location of an array element by assuming that all dimensions start at 0 instead of their LBOUND subscript. Then, instead of finding the location of an item as an offset from the first element of the array, BASIC adds the Adjusted Array Offset to the element's location within the array.

For example, suppose you dimension a dynamic array with the statement **DIM Array%(1 to 10)** and the array data block (reflected in bytes 0 to 3 in the descriptor) is placed at 8000:0000. Because the array contains integer values, each array item requires two bytes of storage. To find Array%(5), BASIC could subtract the LBOUND from the subscript of 5 to get 4, multiply 4 * 2 to find the number of preceding bytes in the array, and then add 8000:0000 to the result to find the location of item 5. But it is faster (especially for multi-dimensioned arrays), to store 0FFFEh as the Adjusted Array Offset. Now item 5 can be found by multiplying 5 times the item size and adding that to the Adjusted Array Offset. The result is the same but the process is faster.

BASIC stores items in multi-dimension arrays in such a way that the first subscript changes the fastest and the last subscript changes the slowest as you move from one data item to another. For example, suppose you dimension an array with this statement:

```
DIM Array%(0 TO 9, 0 TO 9)
```

and that the array data block is placed at 8000:0000. The memory address for representative elements of the array is shown in this table:

```
Array%(0,0)  8000:0000
Array%(1,0)  8000:0002
Array%(2,0)  8000:0004
  ...
  ...
Array%(0,1)  8000:0014
Array%(1,1)  8000:0016
Array%(2,1)  8000:0018
  ...
  ...
Array%(7,9)  8000:005E
Array%(8,9)  8000:0060
Array%(9,9)  8000:0062
```

Of course, there is no requirement that you follow BASIC's method of numbering array items in a program that you write completely in assembly language. You can consider an array's data block as simply one large chunk of memory to work with as you see fit. If you do not follow BASIC's method of addressing array items, however, you cannot use the routine called B$HARY which can calculate the address of an array item for you.

Error Handling

No matter how you try to guard against them, errors happen. Any program may face a "File not found" or "Access denied" error. A well-written program knows how to handle such errors and take appropriate action when they occur.

Many of the P.D.Q. library routines can report errors, which range in severity from inconvenient to catastrophic. It is up to your program to handle the errors gracefully.

The P.D.Q. word variable P$PDQErr holds the number of the last error or zero if no error occurred. Many library routines that can report an error condition begin by setting this variable to zero to clear any prior error. When these routines report an error, they do so by loading the error number into the AX register and then calling P$DoError.

The P$DoError routine stores the error number in P$PDQErr and then checks to see if ON ERROR GOTO is in effect. If so, P$DoError sets up some internal variables for a possible Resume, and then jumps to the designated error handler. If ON ERROR GOTO is not in effect, P$DoError simply stores the error number in P$PDQErr and then returns.

From a purely assembly language perspective, it is unlikely that you will be using BASIC's ON ERROR mechanism. We therefore suggest that you always link with the _NOERROR.OBJ stub file, and examine P$PDQErr manually when you want to see if the most recent action caused an error. You could also create a custom version of P$DoError, and this will be described in a moment.

Critical errors (errors caused by hardware failures such as an open drive door or an off-line printer) can also set P$PDQErr if you have invoked the routine called CritErrOff. If CritErrOff has been called, a critical error will set the error number but not halt program flow. At its next invocation, P$DoError will recognize that a critical error has occurred, and report the critical error instead of the new error that invoked P$DoError.

Assembly language programs can use ON ERROR GOTO but not RESUME, which depends on BASIC's system of module-level code with subprogram and function procedures. The lack of an effective RESUME command makes ON ERROR GOTO almost useless. However, your assembly language programs can use several different methods to recognize and report errors.

First, you can set up your code with a system of polling. As soon as you return from any routine which *could* cause an error, your program could check P$PDQErr to see if an error has been reported. If so, it can look at the error number, perhaps call PDQMessage to print an error message, and then decide what to do next. Polling like this works well in small and simple programs, but can grow to be too complex in larger programs.

Another simple method of detecting errors is to write your own P$DoError routine. Then your code will be called whenever an error occurs and you can handle the error in whatever way makes sense for your program. If you take this approach, you should look carefully at the error handling routines that are built into P.D.Q. In particular, you should look at ERR.ASM, ERROR.ASM, ERRDATA.ASM, and PDQMSG.ASM. P.D.Q. has to do some translation to get DOS and BASIC error numbers to jibe. The details of those translations and of other parts of the error system are in those four files.

The following routines in the P.D.Q. library call P$DoError to report errors:

AllocMem	B$BLOD	B$BSAV	B$CDIR
B$CLOS	B$CSCN	B$DDIM	B$FCD0
B$FCD1	B$FEOF	B$FICT	B$FILS

B$FINP	B$FLOC	B$FLOF	B$GET3
B$HARY	B$INPP	B$KILL	B$LNIN
B$LOCK	B$MDIR	B$NAME	B$OOPN
B$OPEN	B$PC*xx*	B$PE*xx*	B$PS*xx*
B$PUT3	B$PUT4	B$RDIM	B$REST
B$SICT	B$SSEK	B$SSHL	B$WIDT
BufIn	HookInt0	P$TempStr	_FLUSH

Using The P.D.Q. Floating Point Emulator

The fastest and most efficient way to perform floating point arithmetic is with a numeric coprocessor chip like the 8087, 80287, 80387, or the coprocessor built into the 486DX processor. Writing a program that uses a math coprocessor is no more difficult than writing any other kind of assembly language program. However, you and everyone who runs your program must have a coprocessor installed. If a coprocessor is not installed, programs written for a math coprocessor will normally hang or crash the system.

P.D.Q. includes a coprocessor emulation library that lets you write coprocessor instructions as part of your assembly language routines and run those programs on a computer that doesn't have a coprocessor chip. Once you have linked the P.D.Q. emulation library with your program, floating point instructions will perform as you expect on any computer. Of course, the calculations will be much faster if a coprocessor is available.

The P.D.Q. emulation library supports all of the operations that the BASIC compiler generates, which is most of the 8087 and 80287 instruction set. The tables at the end of this section shows which instructions are supported and which are not.

Using the floating point library requires three separate steps:

1. You must begin your program with a call to the P.D.Q. routine P$HookFP.

2. You must end your program with a call to the P.D.Q. routine P$UnhookFP.

3. You must use the MASM switch /e (assemble for emulator) when you assemble each source code module that contains floating point instructions.

The /e switch tells MASM to include fixup information for LINK so it can convert the floating point operations in your program into equivalent interrupt calls. The emulator routines intercept those interrupts so that

the appropriate part of the P.D.Q. floating point library can carry out each operation. P$UnhookFP releases those interrupts before your program ends so that any future calls to those interrupts won't hang your system.

If P$HookFP detects that you have a coprocessor installed, it directs the interrupts to a routine that patches the calling code back into the corresponding coprocessor instructions. This happens only the first time the interrupt is executed. Once the code is back-patched, control is returned to that instruction so it can execute. The first time a given block of code executes additional time is needed to patch the code. Thus, a coprocessor provides the most benefit when a section of floating point code is executed in a loop.

You can use floating point operations with 4-byte single-precision values and 8-byte double-precision values. You can also use the floating point operations with 2-byte integers and 4-byte long integers. Conversions between one numeric type and another is a simple matter of loading the value in one form onto the coprocessor stack, and then storing it in a different form to another variable.

If you are sure that your program will never be run without a coprocessor and won't need the floating point library at all, you can link your program with the _87ONLY.OBJ stub file. If you are sure that your program will never run on a computer that has a coprocessor—or if you want to force the use of the emulator library, perhaps for testing purposes—you can link your program with the _EMONLY.OBJ stub file. You can also dispense with all FWAIT instructions if you link with _EMONLY.OBJ. In either case, you still need to call P$HookFP and P$UnhookFP to handle the floating point interrupts that have been assembled into the library routines.

It is beyond the scope of this manual to teach you how to write coprocessor instructions. The MASM manual contains a useful introduction to using the coprocessor. You can also compile short test programs with the BC compiler and view them with CodeView or another debugger to see how BC compiles the instructions that you are interested in.

For example, if you compile the one-line BASIC program

```
A! = A! * Y#
```

the compiler creates the following floating point instructions:

```
FLD  DWORD PTR [A!]      ;Load A! onto the FP stack
FMUL QWORD PTR [Y#]      ;Multiply it times Y#
FSTP DWORD PTR [A!]      ;Store result in A! and clear the stack
FWAIT                    ;Wait for the 8087 to finish
```

All coprocessor instructions begin with the letter F. The assembler adds a memory type to the instructions that access memory, so that the coprocessor knows how to interpret the bits it receives from memory, and how to translate them to or from its own, internal 80-bit format. Inside the coprocessor there is no distinction between integers and real numbers, or between single precision and double precision numbers.

Using Floating Point Math In A TSR

You may use floating point operations safely inside a simplified TSR by following a few simple rules. If floating point operations are needed within the popup handling code, you must call EnableFP as the first action, and then call DisableFP just before popping down again. These routines hook and unhook the floating point interrupts, and also save the state of the 80*x*87 coprocessor if one is installed. If you are certain that a coprocessor will not be present when the program runs, you can replace EnableFP and DisableFP with P$HookFP and P$UnhookFP respectively.

If you need to execute floating point instructions during the initialization portion of your program, you must call P$HookFP before using those instructions. However, you must be sure to then call P$UnhookFP before calling EndTSR. This way you will release the interrupts back for use by a subsequent program that may be run. Many high-level languages use the same system of interrupts to implement floating point emulation, and it is important that your use of floating point math does not call an underlying program's emulator—especially if that program is in the middle of a computation!

Supported Coprocessor Instructions

Table VI-3 lists all of the coprocessor instructions that are supported by the P.D.Q. library at this time, and Table VI-4 shows which are not now supported. You may, of course, use the unsupported instructions if you are sure that a coprocessor will be installed, and if you link with the _87ONLY.OBJ stub file. In that case you should *not* use the /e (emulator) switch when assembling your main program.

TABLE VI-3
Coprocessor Instructions Supported By The P.D.Q. Emulator.

FABS	FADD	FADDP	FCHS
FCOM	FCOMP	FCOMP	FCOMPP
FDIV	FDIVP	FDIVR	FDIVRP
FIADD	FICOM	FICOMP	FIDIV
FIDIVR	FILD	FIMUL	FINIT
FIST	FISTP	FISUB	FISUBR
FLD	FLD1	FLDCW	FLDZ
FMUL	FMULP	FST	FSTCW
FSTP	FSTSW	FSUB	FSUBP
FSUBR	FSUBRP	FTST	FXCH

TABLE VI-3:
Coprocessor Instructions *Not* Supported By The P.D.Q. Emulator.

F2XM1	FBLD	FBSTP	FCLEX
FDECSTP	FDISI	FENI	FFREE
FFREE	FINCSTP	FLD L2E	FLD L2T
FLD LG2	FLD LN2	FLD PI	FLDENV
FPATAN	FPREM	FPTAN	FRNDINT
FRSTOR	FSAVE	FSCALE	FSQRT
FSTENV	FXAM	FXTRACT	FYL2X
FYL2XP1			

Using P.D.Q. with Assembler

Chapter 7: Programmer's Reference

Programmer's Reference

This chapter contains the details that you will need to use the P.D.Q. library with your own assembly language programs. You'll find details here about the routines and internal data variables that you can use, along with hundreds of examples that will help you learn to use the routines.

There are three sections in this chapter. The first explains the P.D.Q. data areas that you may find useful. Many of these internal P.D.Q. variables contain information that your program may need. You can change the values in some of these variables to change the way that some of the library routines act.

The second and longest section in this chapter is a detailed explanation of more than 200 library routines that you can call from your own programs. Each explanation includes an example to show you how the routine works.

The chapter concludes with a short description of another 75 routines in the P.D.Q. library which have limited usefulness for assembly language programmers. Many of these descriptions include a short explanation of how you can accomplish the same task in fewer bytes or fewer clock cycles.

Browse through this chapter occasionally. Besides the routines that directly implement BASIC statements and functions and the P.D.Q. extensions to BASIC, there are several powerful "helper" routines which could make your next programming project a lot easier.

All of the source code for the library is on the P.D.Q. diskettes. If you need to see exactly what happens in a library routine, or if you need to tweak a routine to make it work just the way you want, you should have no problem altering and re-assembling the routines.

If you make extensive changes to the library, please test them carefully. Several of the library routines interact or depend on the data set by other routines. If you do change and re-assemble some source code, use a text search utility such as the P.D.Q. FINDTEXT.BAS program to see whether any other library routines call the ones that you have changed or share data with it. Searching for cross references between source code files is often a lot easier than trying to swat obscure bugs with CodeView or Turbo Debugger.

External Variables

This section documents 21 variables that P.D.Q. maintains and which you may want to access directly from your own programs.

Some of these variables contain data which alters the behavior of P.D.Q. library routines. Directly changing the data in these variables is usually much faster than calling a routine to make the change for you. For example, you can set the character color and attribute byte used by PDQCPrint and CLS with a single MOV instruction. Doing so is much easier than setting up and calling B$COLR.

Many of these variables contain information that you may want to read but which you should not change directly. They are marked *Read Only* in the list below. If you set one of these variables directly, you may confuse some of the P.D.Q. library routines. For the sake of your own sanity during debugging, please do not alter the data in these *Read Only* variables unless you are positive that you understand all of the possible side effects.

Each of the items below starts with a variable name followed by the variable size in parentheses and then the name of the source file which defines the variable. Then the variable's use is explained.

All of these variables are in the initialized data segment, called _DATA or, if you use simplified segment names, .DATA. To use a variable in your programs, you must declare it "Extrn" in the .DATA segment of the source code module that includes a reference to it. For example, if you want to set P$Color directly, you would declare it this way:

```
.DATA
Extrn P$Color:Word
```

You can then access the P$Color variable just as you would any other named data space in your program.

B$SEG (Word) in \SOURCE\PDQDATA.ASM. The segment value which is set with DEF SEG in BASIC, and which PEEK, BLOAD, and so forth use.

P$1Space (String descriptor) in \SOURCE\PRINTDAT.ASM. A descriptor for a string that contains a single space. *Read only.*

**P$87Used** (Byte) in \FPSOURCE\P$HOOKFP.ASM, and also in \FPSOURCE_87ONLY.ASM and \FPSOURCE_EMONLY.ASM. This value is 0 if the emulator library is being used, or -1 if an 80x87 coprocessor is installed and operating.

P$BytesFree (Word) in \SOURCE\STR#####.ASM. The number of bytes free in the string pool. May be increased by garbage collection. *Read only.*

P$BytesShort (Word) in \SOURCE\STR#####.ASM. The maximum number of bytes requested but not available in the string pool. *Read only.*

P$BytesUsed (Word) in \SOURCE\STR#####.ASM. The number of bytes used in the string pool. May be decreased by garbage collection. *Read only.*

P$CGAPort (Word) in \SOURCE\MONSETUP.ASM. If non-zero, video memory access routines, including screen printing, will be slowed down to avoid snow on a CGA screen. This *must* be set to 3DAh (the CGA port address) to enable snow checking. Affects P.D.Q. extensions only.

P$Color (Byte) in \SOURCE\COLORDAT.ASM. The attribute byte used by the screen display routines which support color.

P$DelimitChar (Byte) in \SOURCE\PDQPARSE.ASM. The character used by the PDQParse routine as an item delimiter.

P$Descr (Word) in \SOURCE\PRINTDAT.ASM. A descriptor for a string that contains a carriage return and line feed. *Read only.*

P$DirtyFlag (Word) in \SOURCE\PDQDATA.ASM. When this word is non-zero, there is space in the string pool that can be reclaimed by compaction. *Read only.*

P$DOSVer (Word) in \SOURCE\PDQDATA.ASM. This contains the current DOS version that P.D.Q. reads during startup. It is the value returned in AX from an Int 21h, service 30h call. *Read only.*

P$HandleTbl (15 words) in \SOURCE\FHANDLES.ASM. This table contains the DOS handle number for each of the open BASIC files. See the comments in the file. *Read only.*

P$MonSeg (Word) in \SOURCE\MONSETUP.ASM. Contains the current video segment address. You can change it for compatibility with DESQview and to "print" to memory instead of the screen as long as you use the output routines which write directly to memory.

P$NullDesc (String descriptor) in \SOURCE\PRINTDAT.ASM. A descriptor for a null (0-length) string. *Read only.*

P$PDQErr (Word) in \SOURCE\ERRDATA.ASM. This holds the most recent error number. You can check it to see if an error has occurred.

P$PrintHandle (Word) in \SOURCE\PHANDLE.ASM. This holds the DOS handle for the current PRINT device and defaults to 1 (screen). LPT1 (the device used for LPRINT statements) is print handle 4.

P$PrintWidth (Word) in \SOURCE\PDQPWIDE.ASM. This is the screen width *times 2* and is used by PDQPrint and PDQCPrint.

P$PSPSeg (Word) in \SOURCE\PDQDATA.ASM. This is the segment address of the current program's PSP. *Read only.*

P$RecordTbl (15 words) in \SOURCE\FHANDLES.ASM. This has the record length for each file that has been opened in random-access mode. *Read only.*

P$TabTable (20 bytes) in \SOURCE\FHANDLES.ASM. This table contains the current tab position for every open file and device.

P$TermCode (Byte) in \SOURCE\PDQ.ASM. This is the termination code (ERRORLEVEL value) that the program will return to DOS when it exits.

Procedure Details

What can you do with the P.D.Q. library? This section has the answer, with a detailed explanation of most of the library routines. In the interest of completeness, several less-useful routines are briefly described in the last section of this chapter.

Each of the descriptions below begins with a procedure's name and the source file (or files) that define the procedure. The name is the one you must use when you call the procedure.

A few of the procedures have synonyms which are other names for the same code. The synonyms exist because QuickBasic and BASIC7 do not always use the same naming conventions.

After the names, most of the descriptions identify the BASIC or P.D.Q. equivalent for the procedure. For example, the name B$PSR8 may not be crystal clear on first reading, but the equivalent **PRINT X#;** should help you decide whether or not you have found the right procedure.

Note that many routines use B$F*xxx* to indicate a function, and B$S*xxx* to mean statement. For example, B$FDAT is the function form of DATE$, and B$SDAT is the statement form. Routines that have both zero- and one-argument forms often end with the digits "0" or "1". For instance,

B$FCD0 stands for "Function CURDIR$ with zero arguments", while B$FCD1 means "Function CURDIR$ with one argument".

Also note that all of the PRINT services begin with B$P. The next letter is either "E" for End of line, "S" for Semicolon, or "C" for Comma. The last two characters indicate the type of data they handle. That is, "I2" stands for Integer 2-byte, and "R8" means Real 8-byte.

Next, the procedure's use is explained in a couple of sentences and then the necessary calling convention is described, along with any return value from the procedure.

The section of each description called *Notes* includes details about the procedure's use, any necessary warnings, and sometimes an explanation about how the procedure works.

The final part of each description is a code fragment which demonstrates how the procedure can be called. The fragments assume that you are using simplified segments with MASM 5.1 or later (or a compatible assembler) and that you have included the directive **.MODEL MEDIUM, BASIC** at the beginning of your source code module. Also, several of the examples use the DefStr macro from the file called MACROS.ASM on your distribution diskette (it's inside the ASM.ZIP file).

We have attempted to include enough information in each description so that you can use the procedure without any confusion. However, if you need more information about a procedure, the source code files on the P.D.Q. diskette are the final authority about how each one works.

B$ASSN \SOURCE\ASSIGN.ASM

Synonyms: B_ASSN
STRINGASSIGN

Use

Assign a fixed-length string or a TYPE variable. Either the source or destination (but not both) may be a normal, variable-length string (with a normal string descriptor). This routine moves the data from the source to the destination. If the destination is a variable-length string, it will be the same length as the original, fixed-length string or memory block.

Calling Convention

```
PUSH Segment of source
PUSH Offset of source
PUSH Length of source (or 0 if variable-length string)
PUSH Segment of destination
PUSH Offset of destination
PUSH Length of destination (or 0 if variable-length string)
CALL B$ASSN
No return value.
```

Notes

This routine calls several other P.D.Q. subroutines to do much of its work. Which routines are called depends on which string (if either) is variable-length.

When you copy data from a fixed-length string to a variable-length string, memory for the variable-length string is claimed from the string pool. The destination will be the same length as the source. If you are copying to a fixed length string or memory block, the data will be padded (with ASCII spaces) or truncated to fit the size of the destination.

To copy from one variable-length (conventional) string to another, use B$SASS. B$ASSN is also useful for copying to or from an arbitrary block of memory such as the file name portion of a DTA.

Example

If A$ is a normal, variable-length string and F$ is a fixed-length string, then the following example will copy data from A$ to F$ (F$ = A$ in BASIC):

```
    Extrn   B$ASSN:Proc

    .DATA?
```

```
EVEN
A$  dd   1 dup (0)        ;Storage for A$'s string descriptor
F$  db   50 dup (?)       ;Room for 50-byte F$

.CODE
MOV      AX,Offset A$     ;Get offset of source
PUSH     DS               ;Push segment and
PUSH     AX               ;  offset of source
SUB      AX,AX            ;AX = 0 to show source
PUSH     AX               ;  is variable-length

PUSH     DS               ;Push segment of destination
MOV      AX,Offset F$     ;  and offset of destination
PUSH     AX
MOV      AX,50            ;Length of destination
PUSH     AX
CALL     B$ASSN
```

B$BLOD \SOURCE\BLOAD.ASM

BASIC Equivalent: BLOAD

Use

Load a BLOAD-formatted file into program, data, or video memory.

Calling Convention

```
MOV  Segment of memory block into B$Seg
PUSH Offset of string descriptor of file name
PUSH Offset address of memory block
PUSH anything at all (dummy word argument)
CALL B$BLOD
```

No return value. May report an error by calling P$DoError.

Notes

This routine does no checking to make sure that the file has a legitimate BLOAD header. It simply skips the first seven bytes of the file and loads the remainder into the memory block that you specify. The file may not be more than 65542 bytes long (0FFFFh plus the header length). The memory block must either be in allocated memory or memory outside of DOS's control (video memory, an EMS memory page frame, and so forth).

Example

BLoad the file whose name is in FileName$ to 0B800:0000H. It assumes that the file name has already been placed in the string FileName$:

```
Extrn B$BLOD:Proc
```

```
.DATA
Extrn B$Seg:Word          ;Holds DEF SEG segment address

.DATA?
EVEN
FileName$  dd  1 dup (?)  ;String descriptor for file name

.CODE
MOV  [B$Seg],0B800H       ;Save segment of destination
MOV  AX,Offset FileName$
PUSH AX                   ;Pass file name
SUB  AX,AX                ;AX = 0
PUSH AX                   ;Pass offset of destination
PUSH AX                   ;Dummy argument
CALL B$BLOD
```

B$BSAV \SOURCE\BSAVE.ASM

BASIC Equivalent: BSAVE

Use

Saves a portion of memory (including video memory) in BASIC's BSAVE/BLOAD format.

Calling Convention

```
MOV  Segment of memory block into B$Seg
PUSH Offset of file name string descriptor
PUSH Offset portion of memory block address
PUSH Number of bytes to save
CALL B$BSAV
```

No return value. May report an error by calling P$DoError.

Notes

This routine opens a file, writes BASIC's 7-byte BSAVE header, and then copies the memory block to the file. Before you call this routine, you must set the segment of the memory block into B$Seg, the location where BASIC stores the current DEF SEG segment address.

Example

This example saves an 80 by 25 text screen (4000 bytes) at address 0B800:0000H. It assumes that the file name has already been placed in the string FileName$:

```
Extrn B$BSAV:Proc

.DATA
Extrn B$Seg:Word
```

```
.DATA?
EVEN
FileName$  dd  1 dup (?)  ;Space for string descriptor.

.CODE
MOV  B$Seg,0B800h         ;Save the segment address
MOV  AX,Offset FileName$  ;Get address of string descriptor
PUSH AX
SUB  AX,AX                ;AX = 0, the offset of the
PUSH AX                   ;  memory block to save
MOV  AX,4000              ;Save 4,000 bytes
PUSH AX
CALL B$BSAV
```

B$CDIR \SOURCE\CHDIR.ASM

BASIC Equivalent: CHDIR

Use

Set a new default directory on the current drive.

Calling Convention

```
PUSH Offset of string descriptor of new directory
CALL B$CDIR
```

No return value. May report an error by calling P$DoError.

Notes

You will probably want to call Int 21h, service 3Bh directly most of the time. This routine will be useful if the name of the new directory is already stored in a BASIC-style string.

Example

Implement CHDIR NewDir$:

```
Extrn B$CDIR:Proc

.DATA?
EVEN
NewDir  dd 1 dup (?)      ;Space for string descriptor

.CODE
MOV  AX,Offset NewDir     ;Get descriptor address
PUSH AX                   ;Pass it on
CALL B$CDIR               ;Let P.D.Q. do the work
```

B$CLOS

\SOURCE\CLOSE.ASM
\SMALLDOS\CLOSE.ASM

BASIC Equivalent: CLOSE

Use

Close one or more open files. The file(s) must have been opened with calls to B$OPEN or B$OOPN. Files that you open with your own calls to DOS will not be affected by this procedure.

Calling Convention

This procedure can be called to close all open files or to close specific files. For each specific file,

```
PUSH BASIC file number, then
PUSH total number of listed files
CALL B$CLOS
```

No return value. May report an error by calling P$DoError.

Notes

This procedure looks at the last parameter on the stack to see how many other parameters have been passed to it. If you want to perform the equivalent of a BASIC CLOSE statement (close all open files), PUSH a 0 (zero) on the stack and then call B$CLOS. (You can also close all open files using B$REST). If you want to close specific files, PUSH the number of each, PUSH the number of files you have listed, and then call B$CLOS. See the example below.

The SMALLDOS version of B$CLOS requires exactly one file number. If you use that version, the second argument (number of listed files) must always be 1.

Example

Perform the equivalent of CLOSE (close all open files):

```
Extrn B$CLOS:Proc

.CODE
SUB  AX,AX              ;AX = 0
PUSH AX                 ;No specific files listed
CALL B$CLOS
```

Perform the equivalent of CLOSE #3 or CLOSE 3:

```
Extrn B$CLOS:Proc
```

```
.CODE
MOV  AX,3                ;File number
PUSH AX                  ;  is on the stack
MOV  AX,1                ;One specified file
PUSH AX
CALL B$CLOS
```

Perform the equivalent of CLOSE #2, #3:

```
Extrn B$CLOS:Proc

.CODE
MOV  AX,2                ;First file number
PUSH AX
INC  AX                  ;AX = 3, the second file number
PUSH AX
DEC  AX                  ;AX = 2, two files specified
PUSH AX
CALL B$CLOS
```

B$COLR \SOURCE\COLOR.ASM

BASIC Equivalent: COLOR

- **Use**

  Set the color to be used for future calls to B$SCLS (equivalent to BASIC's CLS) and PDQCPrint.

- **Calling Convention**

  COLOR can be called to emulate any of 3 possible BASIC syntax constructions. In the list below, FG = foreground color number, BG = background color number.

  For COLOR FG:

  ```
  PUSH 1, PUSH FG, PUSH 2, Call B$COLR
  ```

  For COLOR , BG:

  ```
  PUSH 0, PUSH 1, PUSH BG, PUSH 3, Call B$COLR
  ```

  For COLOR FG, BG:

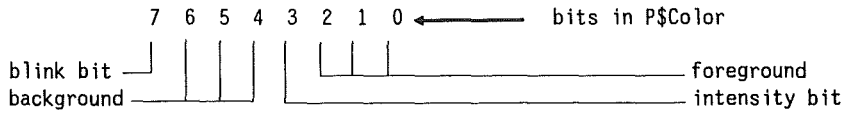
  ```
  PUSH 1, PUSH FG, PUSH 1, PUSH BG, PUSH 4, Call B$COLR
  ```

  No return value.

- **Notes**

  B$COLR recognizes four other syntax forms, but each specifies a border color, which P.D.Q.'s B$COLR ignores.

You can perform the same action as COLOR by combining the foreground and background colors yourself and storing them in the external data byte called P$Color. The figure below shows the bit arrangement of P$Color, which is the same bit arrangement used by video memory in text modes:



Example

Set color for gray text on a red background, equivalent to the BASIC statement COLOR 7, 4:

```
Extrn B$COLR
.CODE
MOV  BX,1
PUSH BX                 ;Show that FG value is coming
MOV  AX,7               ;Foreground color value
PUSH AX                 ;Pass value for gray foreground
PUSH BX                 ;Show that BG value is coming
MOV  AX,4               ;Background color value
PUSH AX                 ;Pass value for red background
PUSH AX                 ;Show 4 preceding data values
CALL B$COLR
```

or, to accomplish the same thing:

```
.DATA
Extrn P$Color: Byte

.CODE
MOV P$Color,47h         ;Combined color stored directly.
```


B$CPI4 \SOURCE\COMPARE4.ASM

Use

Compares two signed, long integers (4 bytes each) using 8088 instructions.

Calling Convention

```
PUSH high word of VAL1&
PUSH low word of VAL1&
PUSH high word of VAL2&
PUSH low word of VAL2&
CALL B$CPI4
```

Results returned in the flags register.

Notes

If you are compiling with 386 instructions enabled (.386) you can make the comparison directly, or call the special 386 version of this routine (see the source file \SOURCE\COMPAR43.ASM).

Because BASIC considers long integers as being signed, you will use Jg and Jl and so forth, but not Ja or Jb or Jbe.

Example

Implement IF A& < B& THEN do something:

```
    Extrn B$CPI4:Proc

    .DATA?
    A  dd  1 dup (?)        ;Space for A&
    B  dd  1 dup (?)        ;Space for B&
    .CODE
    PUSH Word Ptr [A+2]     ;Pass A&, high word first
    PUSH Word Ptr [A]
    PUSH Word Ptr [B+2]     ;Now pass B&
    PUSH Word Ptr [B]
    CALL B$CPI4
    JGE  @F                 ;Go if B& >= A&
    ...                     :  do something here
@:                          ;Common code again.
```

B$CSCN \SOURCE\SCREEN2.ASM

BASIC Equivalent: SCREEN statement

Use

Changes video mode.

Calling Convention

```
PUSH mode number (required)
PUSH colorswitch (optional, ignored)
PUSH apage (optional, ignored)
PUSH vpage (optional, ignored)
PUSH number of arguments above (1 - 4)
CALL B$CSCN
```

No return value. May report an error by calling P$DoError.

Notes

If you know what kind of video system your program is running under, it is faster to simply make a BIOS call to switch screen modes. The

advantage of this routine is that P.D.Q. checks to make sure that the requested mode and the current hardware are compatible.

Note that the first argument is a BASIC mode number, not a BIOS mode number.

Also notice that P.D.Q. considers only the first and last arguments. It completely ignores the colorswitch, apage, and vpage arguments.

Example

Switch into screen mode 0 (text mode):

```
Extrn B$CSCN:Proc

.CODE
SUB  AX,AX              ;AX = 0, the screen mode
PUSH AX
INC  AX                 ;AX = 1, # of arguments passed
PUSH AX
CALL B$CSCN
```

B$CSRL \SOURCE\CSRLIN.ASM

BASIC Equivalent: CSRLIN

Use

Returns the current line (row) of the cursor.

Calling Convention

```
CALL B$CSRL
```

Returns row in AX.

Notes

The return value is 1-based. B$CSRL uses Int 10h to find the current cursor position and then adds 1 to bring the value into sync with BASIC's 1-based line and column numbers.

Example

```
Extrn B$CSRL:Proc

.CODE
CALL B$CSRL             ;Now cursor line is in AX
```

B$DDIM

\SOURCE\DIM.ASM
\SOURCE_DIM.ASM

Synonyms: B$RDIM
BASIC Equivalent: DIM (dynamic) and REDIM

Use

Dimension (create) a dynamic array.

Calling Convention

```
PUSH LBOUND of first dimension
PUSH UBOUND of first dimension
PUSH LBOUND of next dimension (optional)
PUSH UBOUND of next dimension (optional)
repeat preceding two lines for each dimension
PUSH Size of each element in bytes
PUSH Features Word (see description below)
PUSH Offset of Array Descriptor
CALL B$RDIM
```

No return value. May report an error by calling P$DoError.

Notes

Dynamic arrays are addressed through an array descriptor (which is *not* the same thing as a string descriptor). The size of an array descriptor depends on the number of dimensions in the array, and can be calculated as **(number of dimensions * 4) + 12**. Therefore, the descriptor for a one-dimensional array is (1 * 4) + 12 = 16 bytes long.

B$DDIM fills in the necessary information in the array descriptor. If the array elements are anything but conventional (variable length) strings, B$DDIM will allocate the necessary data space for the array outside of DGROUP, the default data segment, by using a DOS memory allocation call. A dynamic array of conventional strings creates an array of string descriptors (4-bytes each) inside DGROUP (where the string text also resides).

The *Features Word* is really two separate bytes. The low-order byte contains the number of dimensions in the array. The high-order byte is a bit record. This is the same bit record that is stored in the array descriptor as the Array Type word at offset 9.

Bit 0 set means the array is stored outside of DGROUP.

Bit 1 set means that it may be a huge array (greater than 64K total data storage space).

Bit 6 set indicates a static array.

Bit 7 set indicates a string array.

For a complete description of how to address each element of the array, see the section *Using Arrays* in the previous chapter. Also see the source code for UBOUND.ASM which contains a table showing each component in an array descriptor.

The version of this routine in the file _DIM.ASM is similar, but excludes support for arrays that have more than 64K total elements.

Example

Implement the equivalent of REDIM Array%(1 to 10, 4 to 12):

```
Extrn B$DDIM:Proc

.DATA?
Array dw 2*4+12 dup (?)   ;Room for the descriptor

.CODE
MOV  AX,1                 ;LBOUND of first dimension
PUSH AX
MOV  AL,10                ;UBOUND of first dimension
PUSH AX
MOV  AL,4                 ;LBOUND of second dimension
PUSH AX
MOV  AL,12                ;UBOUND of second dimension
PUSH AX
MOV  AL,2                 ;Size of an integer
PUSH AX
MOV  AH,00000001B         ;Record: store outside DGROUP
MOV  AL,2                 ;Number of dimensions
PUSH AX                   ;Pass feature word
MOV  AX,Offset Array      ;Get array address
PUSH AX                   ;And pass that
CALL B$DDIM               ;Dimension the array
```


B$DSKI \SOURCE\R$DSKI.ASM

BASIC Equivalent: INPUT #n

Use

Disk initialize in preparation for reading data items from a file.

Calling Convention

```
PUSH Offset of file number
CALL B$DSKI
```

No return value.

Notes

This routine prepares a file for input into one or more variables. The actual input is done during processing of the B$RD*xx* routines (B$RDI2, B$RDI4, B$RDR4, B$RDR8, and B$RDSD) as they read and assign their variables.

After you have called the necessary B$RD*xx* routines, you must call B$PEOS before the next input or print to clean up internal flags and reset the default print handle to the console.

Example

Perform INPUT #1, Num&:

```
Extrn B$DSKI:Proc
Extrn B$RDI4:Proc          ;Read long integer
Extrn B$PEOS:Proc          ;End this input session

.DATA?
Num     dd  1  dup (?)     ;Room for result long integer
FileNum dw  1  dup (?)     ;Room for file number

.CODE
MOV  AX,[FileNum]          ;Load the file number
PUSH AX                    ;Pass it on to B$DSKI
CALL B$DSKI                ;Set input from file
MOV  AX,Offset Num         ;Get pointer to result area
PUSH DS                    ;Pass segment of result
PUSH AX                    ;  and its offset
CALL B$RDI4                ;Get a number
CALL B$PEOS                ;Then clean up
```

B$DVI4 \SOURCE\DIVLONG.ASM

Use

Long integer division: X& \ Y&

Calling Convention

```
PUSH high word of divisor   (Y&)
PUSH low word of divisor    (Y&)
PUSH high word of dividend  (X&)
PUSH low word of dividend   (X&)
CALL B$DVI4
```

Returns X& \ Y& in DX:AX.

Notes

Like all BASIC arithmetic, this routine works with signed values. If the high bit of either number is 1, the routine treats it as a negative number.

Example

Perform Z& = X& \ Y&:

```
Extrn B$DVI4:Proc

.DATA?
X&  dd  1 dup (?)          ;Memory for long integers
Y&  dd  1 dup (?)
Z&  dd  1 dup (?)

.CODE
LES  AX,[Y&]               ;Get Y& value in ES:AX
PUSH ES                    ;Send the high word
PUSH AX                    ;  and the low word
LES  AX,[X&]               ;Now get X&
PUSH ES                    ;And send it
PUSH AX
CALL B$DVI4                ;Do the division
MOV  Word Ptr [Z&],AX      ;Save the low word
MOV  Word Ptr [Z&+2],DX    ;  and the high word
```

B$ERAS \SOURCE\ERASE.ASM

BASIC Equivalent: ERASE

Use

Deallocates the memory of a dynamic array; reinitializes a static array.

Calling Convention

```
PUSH Offset of array descriptor
CALL B$ERAS
```

No return value.

Notes

See B$DDIM and *Using Arrays* in the previous chapter for information about the array descriptor and the memory allocated for an array. If you use B$ERAS on a static array, the array's memory is erased (and, for a string array, each string is deleted) but the array remains allocated. Erasing a dynamic array returns the allocated array memory to DOS.

Example

Perform ERASE Array:

```
Extrn B$ERAS:Proc

.DATA?
Array  db  16dup (?)      ;Space for the array descriptor

.CODE
MOV  AX,Offset Array      ;Pass the address of the
PUSH AX                   ;  array descriptor
CALL B$ERAS
```

B$FASC \SOURCE\ASC.ASM

BASIC Equivalent: ASC()

Use

Returns the ASCII value of the first character in a string.

Calling Convention

```
PUSH Offset of string descriptor
CALL B$FASC
```

Value returned in AX.

Notes

If you don't need to perform this task often, you can save a few bytes and cycles by finding the ASCII value directly. The source code in this routine in ASC.ASM will make the process clear. The P.D.Q. version of ASC() returns a value of -1 in AX if you request the ASCII value of a null string.

Example

Find the ASC value of the string X$:

```
Extrn B$FASC:Proc

.DATA?
EVEN
X$       dd  1 dup (?)    ;Space for string descriptor
AscValue dw  1 dup (?)

.CODE
MOV  AX,Offset X$         ;Get string descriptor address
PUSH AX                   ;Pass address on stack

CALL B$FASC               ;Get ASC(X$) in AX
OR   AH,AH                ;Was it -1?
```

```
JNZ  NullString          ;String had length of 0
MOV  AscValue,AX         ;Else save ASC value
```

B$FATR \SOURCE\FILEATTR.ASM

BASIC Equivalent: FILEATTR()

Use

Returns the DOS file handle for a BASIC file number.

Calling Convention

```
PUSH BASIC file number
PUSH anything
CALL B$FATR
```

Returns the file's equivalent DOS handle number in DX:AX.

Notes

In regular BASIC, the second argument to FILEATTR may be either 1 or 2. If it is 2, the function returns the DOS file handle for a BASIC file; if it is 1, FILEATTR returns a value that indicates the file's mode. The P.D.Q. version of B$FATR assumes that the second argument is 2—it doesn't check or use the received value at all.

Example

Find the equivalent DOS file handle for BASIC file #2:

```
Extrn B$FATR:Proc

.CODE
MOV  AX,2                ;Get the BASIC file number
PUSH AX                  ;Pass it on
PUSH AX                  ;Now pass a dummy argument
CALL B$FATR              ;Get the DOS file number
;  now AX has the equivalent DOS file handle and DX=0
```


B$FCD0 and B$FCD1 \SOURCE\CURDIR$.ASM

BASIC 7 Equivalent: CURDIR$

Use

Find the current default directory on either the default drive or on any drive in the system.

Calling Convention

For the default drive:

```
CALL B$FCD0
```

For a specific drive (like CURDIR$ "A"):

```
PUSH Offset of string descriptor holding drive name
CALL B$FCD1
```

Returns offset of temporary string descriptor in AX. May report an error by calling P$DoError.

Notes

The return value in AX is the offset of the string descriptor of a temporary string. If you want to keep the directory name for further processing, you should use B$SASS to copy it to a permanent string.

Example

Find the current directory on the default drive:

```
Extrn B$FCD0:Proc
Extrn B$SASS:Proc

.DATA?
EVEN
Direct$  dd  1 dup (?)    ;Storage for result descriptor

.CODE
CAll  B$FCD0              ;Get current directory
PUSH  AX                  ;Now pass result
MOV   AX,Offset Direct$   ; and final storage place
PUSH  AX
CALL  B$SASS              ;Put directory in Direct$
```

B$FCHR \SOURCE\CHR$.ASM

BASIC Equivalent: CHR$()

Use

Creates a 1-character string from an ASCII value.

Calling Convention

```
PUSH ASCII value (0 to 255) as a word
CALL B$FCHR
```

Returns with the offset of a temporary string descriptor in AX.

Notes

The high byte of the passed value is ignored by B$FCHR. It does not have to be a 0.

The return value in AX is the offset of the string descriptor of a temporary string. If you want to keep the character string for further processing, you should use B$SASS to copy it to a permanent string.

Example

Assign the 1-character string "A" to A$:

```
Extrn B$FCHR:Proc

.DATA?
EVEN
A$  dd  1 dup (?)          ;Room for string descriptor

.CODE
MOV  AL,'A'                ;Get character for string in AL
PUSH AX                    ;Pass it on
CALL B$FCHR                ;Create a temporary string
PUSH AX                    ;Pass on the pointer
MOV  AX,Offset A$          ;Get pointer to permanent descriptor
PUSH AX
CALL B$SASS                ;Assign string to A$
```

B$FCMD \SOURCE\COMMAND$.ASM

BASIC Equivalent: COMMAND$

Use

Returns the command line in a temporary string.

Calling Convention

```
CALL B$FCMD
```

Returns the offset of the temporary string descriptor in AX.

Notes

Unlike regular BASIC, the P.D.Q. COMMAND$ routine does not convert the command line to upper case. However, it does strip leading spaces and Tab characters. If you want to save the command line, you should follow a call to B$FCMD with a call to B$SASS to move the string to permanent memory.

Example

Implement A$ = COMMAND$:

```
Extrn B$FCMD

.DATA? EVEN
A$  dd  1 dup(?)          ;Space for A$'s descriptor

.CODE
CALL B$FCMD               ;Get the command string
PUSH AX                   ;Pass its descriptor address
MOV  AX,Offset A$         ;Get address for new string
PUSH AX
CALL B$SASS               ;Copy command line to new string
```

B$FCMP \FPSOURCE\B$FCMP.ASM

Use

Compares two real numbers, ST(0) and ST(1) on the floating point stack, and sets the CPU flags appropriately. ST(0) and ST(1) are popped from the floating point stack.

Calling Convention

```
Load the numbers onto the floating point stack
CALL B$FCMP
```

No return value, but the flags are set to show the result of the comparison.

Notes

This routine uses floating point operations. You must initialize the floating point emulator by calling P$HookFP before you call this procedure.

Floating point comparisons do not return signed information. Therefore, after a comparison you will use Ja or Jb or Jae, and so forth, instead of Jg or Jle.

B$FCMP assumes that both ST(0) and ST(1) are valid floating point numbers—not NAN (not a number) or Infinity.

Example

Jump if ST(0) > ST(1) (if Num2# > Num1!):

```
Extrn B$FCMP:Proc

.CODE
FLD  DWORD PTR [Num1!]    ;load a single-precision
FLD  QWORD PTR [Num2#]    ;then a double-precision
```

```
FWAIT                       ;wait for 80x87
;    Now Num1! is in ST(1) and Num2# is in ST(0)
CALL B$FCMP                 ;Make the comparison
JA   ItsBigger              ;Go if ST(0) > ST(1)
...                         ;(Go if Num2# > Num1!)
```

B$FCVD \SOURCE\CVS.ASM

BASIC Equivalent: CVD()

Use

Converts the first eight characters of a string into a double-precision value. This command is often used in BASIC to change a fielded string variable into a double-precision number.

Calling Convention

```
PUSH Offset of string's descriptor
CALL B$FCVD
```

Returns with AX pointing to the 8-byte double-precision value.

Example

```
Extrn B$FCVD:Proc

.DATA?
EVEN
Work$  dd  dup(?)          ;Space for string descriptor

.CODE
MOV  AX,Offset Work$       ;Pass pointer to string
PUSH AX
CALL B$FCVD                ;Pointer to value returned in AX
```


B$FCVI \SOURCE\CVI.ASM

BASIC Equivalent: CVI()

Use

Converts the first two characters of a string into an integer. This command is often used in BASIC to change a fielded string variable to an integer.

Calling Convention

```
PUSH Offset of string's descriptor
CALL B$FCVI
```

Returns the integer value in AX.

Example

```
Extrn B$FCVI:Proc

.DATA
EVEN
Work$  dd  ?              ;Space for string descriptor

.CODE
MOV  AX,Offset Work$      ;Pass pointer to string
PUSH AX
CALL B$FCVI               ;Value returned in AX
```

B$FCVL \SOURCE\CVL.ASM

BASIC Equivalent: CVL

Use

Converts the first four characters of a string into a long integer. This command is often used in BASIC to change a fielded string variable to a long integer.

Calling Convention

```
PUSH Offset of string's descriptor
CALL B$FCVL
```

Returns the long integer value in DX:AX.

Example

```
Extrn B$FCVL:Proc

.DATA?
EVEN
Work$  dd1  dup (?)       ;Space for string descriptor

.CODE
MOV  AX,Offset Work$      ;Pass pointer to string
PUSH AX
CALL B$FCVL               ;Value returned in DX:AX
```

B$FCVS \SOURCE\CVS.ASM

BASIC Equivalent: CVS()

Use

Converts the first four characters of a string into a single-precision value. This command is often used in BASIC to change a fielded string variable into a single-precision number.

Calling Convention

```
PUSH Offset of string's descriptor
CALL B$FCVS
```

Returns with AX pointing to the 4-byte single-precision value.

Example

```
Extrn B$FCVS:Proc

.DATA?
EVEN
Work$  dd  1 dup (?)      ;Space for string descriptor

.CODE
MOV  AX,Offset Work$      ;Pass pointer to string
PUSH AX
CALL B$FCVS               ;Pointer to value returned in AX
```

B$FDAT \SOURCE\DATE$.ASM

BASIC Equivalent: DATE$ function

Use

Get the current system date in a string in the format mm-dd-yyyy.

Calling Convention

```
CALL B$FDAT
```

Returns offset of the descriptor holding a date string in AX.

Notes

The date string is held in static memory. You can either copy the string with B$SASS or just save the pointer. The date string won't move in memory nor will it change unless B$FDAT is called again and the system clock has passed midnight since the last call.

Example

Get system date and store it in a normal string called Today$:

```
Extrn B$FDAT:Proc

.DATA?
EVEN
Today$  dd  1 dup (?)      ;Space for descriptor

.CODE
CALL B$FDAT                ;Get system date in ASCII
PUSH AX                    ;Pass its descriptor address
MOV  AX,Offset Today$      ;Get pointer to new string
PUSH AX
CALL B$SASS                ;Assign date to new string
```

B$FDR0 and B$FDR1 \SOURCE\DIR$.ASM

BASIC 7 Equivalent: DIR$

Use

Returns a filename that matches a specific filespec pattern. Call B$FDR1 with a filespec pattern to find the first matching file name. Then call B$FDR0 repeatedly to collect additional file names until B$FDR0 returns a null string.

Calling Convention

```
PUSH Offset of filespec string descriptor
CALL B$FDR1
```

Returns the offset of a string descriptor holding the first matching file name found in AX.

Then:

```
CALL B$FDR0
```

Returns offset of string descriptor of next matching file name in AX. When the word at address in AX = 0, there are no more matching names.

Notes

You must call B$FDR1 at least once before calling B$FDR0. Each call to B$FDR1 initiates a new search. You do not have to wait for a null string from B$FDR0 before starting a new search with B$FDR1.

This routine resets the DTA (disk transfer area). If you are using the DTA for other purposes, make sure you save the address of your DTA and then

restore yours (with DOS service Int 21h, Function 1Ah) after you call either of these services.

The address of the returned string is offset 1Eh in the DTA used for a Find First/Find Next call (Int 21h, Functions 4Eh and 4Fh). You can use the returned address to find other directory information about the file.

Example

Find and print the names of all *.ASM files in the current directory (similar to performing **DIR *.ASM** from DOS):

```
    Extrn B$FDR0:Proc
    Extrn B$FDR1:Proc

      .DATA
      DefStr DIR$, "*.ASM"
      .CODE
      MOV  AX,Offset DIR$          ;Get address of descriptor
      PUSH AX                      ;Pass it on
      CALL B$FDR1                  ;Get the first file
@@:   MOV  BX,AX                  ;Put offset in BX
      CMP  Word ptr [BX],0         ;Is it null?
      JE   @F                      ;Yes, all done
      PUSH AX                      ;Else pass on name
      CALL B$PESD                  ;  and print it
      CALL B$FDR0                  ;And look for another
      JMP  @B                      ;Then try to print that
@:    ...
      ;    code continues after all .ASM files are printed
```

B$FEOF \SOURCE\EOF.ASM

BASIC Equivalent: EOF()

Use

Test for an end-of-file condition.

Calling Convention

```
PUSH BASIC file number
CALL B$FEOF
```

Returns AX = -1 if EOF() is true, or AX = 0 if not at end of file. May report an error by calling P$DoError.

Notes

EOF() can be used with any file that you have opened by calling B$OPEN or B$OOPN. You also must have made all file accesses through P.D.Q. library routines.

The file number is a BASIC file number, not a DOS handle number.

Example

Test if EOF(#1), assuming that file number 1 is already open:

```
Extrn B$FEOF:Proc

.CODE
MOV  AX,1               ;Pass a BASIC file number
PUSH AX
CALL B$FEOF             ;At end of file
OR   AX,AX              ;Test result
JNZ  FileDone           ;Go if EOF
. . .                   ;Continue processing file here
```

B$FERR \SOURCE\ERR.ASM

BASIC Equivalent: ERR

Use

Returns the number of the last error, or zero if there was none.

Calling Convention

```
CALL B$FERR
```

Returns error number in AX.

Notes

The return value is the BASIC error code, not the DOS error code. The error number is always less than 256, so AH is always 0 on return and you can use either AX or AL as the return value. See *File Handling in P.D.Q.* and also the description for the PDQMessage function.

Example

```
Extrn B$FERR:Proc

.CODE
CALL B$FERR             ;Get the error code
                        ; which is now in AX
```

B$FEV1 \SOURCE\ENVIRON1.ASM

BASIC Equivalent: ENVIRON$(n) function

Use

Returns the *n*th string in the environment, or a null string if there are less than *n* strings.

Calling Convention

```
PUSH number of string to return
CALL B$FEV1
```

Returns offset of a temporary string descriptor in AX.

Notes

The returned string is a temporary string and is not actually part of the environment. If you want to keep the string, you should copy it to your own string variable using B$SASS.

Example

Print all of the environment strings:

```
        Extrn B$FEV1:Proc

        .CODE
        SUB  SI,SI              ;We'll use SI to count strings
@@:     INC  SI                 ;Move to next string
        PUSH SI                 ;Pass string number
        CALL B$FEV1             ;Get the string
        MOV  BX,AX              ;Copy descriptor address to BX
        CMP  Word ptr [BX],0    ;Is it null?
        JE   @F                 ;Yes -- we're done!
        PUSH AX                 ;Else pass string address
        CALL B$PESD             ;  and print it
        JMP  @B                 ;And get another one
@@:     ...
        ;    continue here when all strings are printed
```

B$FEVS \SOURCE\ENVIRON2.ASM

BASIC Equivalent: ENVIRON$(Environ String$)

Use

Searches the environment for a specific variable and returns its contents.

Calling Convention

```
PUSH Offset of EnvironString$ descriptor
CALL B$FEVS
```

Returns offset of result descriptor in AX, or of a null string if no match is found.

Notes

The returned string is a temporary string and is not actually part of the environment. If you want to keep the string you should copy it to your own string variable using B$SASS.

The string in EnvironString$ must be in the same case as it appears in the environment. Normally, this means that it must be in upper case. See the EnvOption for information on accessing mixed-case environment strings.

Example

Print the current PATH setting:

```
    Extrn B$FEVS:Proc
    Extrn B$PESD:Proc

    .DATA
    DefStr PATH$,"PATH"

    .CODE
    MOV  AX,Offset PATH$    ;Get address of descriptor
    PUSH AX                 ;  and send it on
    CALL B$FEVS             ;Look for PATH in environment
    MOV  BX,AX              ;Copy output offset
    CMP  Word ptr [BX],0    ;Is it a null string?
    JE   @F                 ;Yes -- nothing to print
    PUSH AX                 ;Pass address of result
    CALL B$PESD             ;Print the result
@@:                         ;Jump here if no string
```

B$FHEX \SOURCE\HEX$.ASM

BASIC Equivalent: HEX$()

Use

Translates a long integer into hexadecimal notation.

Calling Convention

```
PUSH High-order word of value to translate
PUSH Low-order word of value to translate
CALL B$FHEX
```

Returns offset of a temporary string descriptor in AX.

Notes

This routine returns with the result in a temporary string. If you want to keep the string for further processing, be sure to assign it to a permanent string variable using B$SASS.

Example

Translate 12345678h to hexadecimal notation:

```
Extrn B$FHEX:Proc

.DATA?
EVEN
HexStr$  dd  1 dup(?)     ;Space for a string descriptor

.CODE
MOV  AX,1234h             ;Send the high-order word
PUSH AX
MOV  AX,5678h             ; and the low-order word
PUSH AX
CALL B$FHEX               ;Translate to a string
PUSH AX                   ;Pass temporary descriptor
MOV  AX,Offset HexStr$    ; and a pointer to our
PUSH AX                   ; string descriptor
CALL B$SASS               ;Copy it to our string
```


B$FICT \SOURCE\IOCTL$.ASM

BASIC Equivalent: IOCTL$

Use

Receives a control string from an open device driver.

Calling Convention

```
PUSH BASIC file number of the device
CALL B$FICT
```

Returns AX holding the address of a temporary string descriptor. May report an error by calling P$DoError.

Notes

The returned string is part of P.D.Q.'s temporary string pool. You must copy it to a permanent string (with B$SASS) if you want to store it or use it further.

Example

Perform Control$ = IOCTL$(3):

```
Extrn B$FICT:Proc

.DATA?
EVEN
Control$  dd  1 dup(?)    ;Result string descriptor

.CODE
MOV  AX,3                 ;Get the BASIC file number
PUSH AX
CALL B$FICT               ;Get the control string
OR   AX,AX                ;Valid result?
JZ   NoControl            ;No -- handle the error
PUSH AX                   ;Else pass on the temp. string
MOV  AX,Offset Control$   ;  and result location
PUSH AX
CALL B$SASS               ;Assign string to Control$
```

B$FILS \SOURCE\FILES.ASM

BASIC Equivalent: FILES

Use

Prints the names of files matching a give filespec.

Calling Convention

```
PUSH Offset of filespec string descriptor
CALL B$FILS
```

No return value. May report an error by calling P$DoError.

Notes

This routine resets the DTA (disk transfer area) to its own buffer. If you are using the DTA for other purposes, be sure to reset it (using DOS Int 21h, function 1Ah) after you call B$FILS.

If you want B$FILS to list all files in the default directory, you can either send it a filespec of "*.*" or you can send it a pointer to a null word, which it will interpret as a zero-length string descriptor.

Example

Equivalent of BASIC's FILES:

```
.DATA
NullWord  dw  0           ;Fake null string descriptor

.CODE
MOV  AX,Offset NullWord   ;Point to fake descriptor
PUSH  AX
CALL  B$FILS              ;Print the directory
```

B$FINP \SOURCE\INPUT$.ASM

BASIC Equivalent: INPUT$

Use

Reads a string of a specific length from the keyboard or a file.

Calling Convention

```
PUSH  Number of characters
PUSH  File number or 7FFFh to read from the keyboard
CALL  B$FINP
```

Returns offset of string descriptor in AX. May report an error by calling P$DoError.

Notes

Notice that you *must* include a file number when you call INPUT$, using the file number 7FFFh for the keyboard.

Example

Perform A$ = INPUT$(1):

```
Extrn B$FINP:Proc
Extrn B$SASS:Proc         ;For string assignment

.DATA?
```

```
EVEN
A$  dd  1 dup (?)          ;Space for A$'s descriptor

.CODE
Keyboard  EQU  7fffh       ;Let the assembler remember this
MOV  AX,1                  ;Get 1 keystroke
PUSH AX
MOV  AX,Keyboard           ;From the keyboard
PUSH AX
CALL B$FINP                ;Get the keystroke
PUSH AX                    ;Pass on the address
MOV  AX,Offset A$          ;Send a pointer to the destination
PUSH AX
CALL B$SASS                ;Now it is in our string
```

B$FLOC \SOURCE\LOC.ASM

BASIC Equivalent: LOC

Use

Find the current position within a file.

Calling Convention

```
PUSH the BASIC file number
CALL B$FLOC
```

Result is returned in DX:AX. May report an error by calling P$DoError.

Notes

If the file was opened as a random file, LOC returns the current record number. If it was opened in any other mode, LOC returns the current byte position. In both cases, the result is 0-based; that is, the first byte of the file is position 0.

Notice that the result is returned as a long integer.

Example

Find LOC(#2):

```
Extrn B$FLOC:Proc

.CODE
MOV  AX,2                  ;Get the file number
PUSH AX
CALL B$FLOC                ;File position now in DX:AX
```

B$FLOF \SOURCE\LOF.ASM

BASIC Equivalent: LOF()

Use

Finds the length of an opened file in bytes.

Calling Convention

```
PUSH the BASIC file number
CALL B$LOF
```

Returns the length in DX:AX. May report an error by calling P$DoError.

Example

Find the length of File #4:

```
Extrn B$FLOF:Proc

.CODE
MOV  AX,4                ;Get file number
PUSH AX
CALL B$FLOF              ;Get the length
; the length of File #4 is now in DX:AX
```

B$FMID \SOURCE\MID$.ASM

BASIC Equivalent: MID$ function

Use

Extract a portion of a string as in **MID$(Source$, Start%, Length%).**

Calling Convention

```
PUSH Offset of source string descriptor
PUSH Start Position
PUSH Length to extract
CALL B$FMID
```

Returns offset of a temporary string descriptor in AX.

Notes

The Length is truncated if it would cause a read beyond the end of the string. If you want to extract the entire string from Start Position to the end, use 7FFFh (the highest possible value) as the Length parameter.

The result is a temporary string descriptor. Make sure you copy it to a permanent string variable with B$SASS if you want to save it.

Example

Perform A$ = MID$(A$, 3, 2):

```
Extrn B$FMID:Proc
Extrn B$SASS:Proc          ;For string assignment
.DATA?
EVEN
A$  dd  1 dup (?)          ;Room for string descriptor

.CODE
MOV  AX,Offset A$          ;Get address of source
PUSH AX
MOV  AX,3                  ;Get start position
PUSH AX
DEC  AX                    ;AX=2, the length
PUSH AX
CALL B$FMID                ;AX ==> result descriptor
PUSH AX                    ;Pass the temporary descriptor
MOV  AX,Offset A$          ;Get address of destination
PUSH AX
CALL B$SASS                ;Result now stored in A$
```

B$FMKD \SOURCE\MKD$.ASM

BASIC Equivalent: MKD$

Use

Converts a double-precision value into an 8-byte string.

Calling Convention

```
PUSH Most significant word of the value
PUSH Next-most significant word of the value
PUSH Next-most significant word of the value
PUSH Least significant word of the value.
```

Returns offset of string descriptor in AX.

Notes

All this routine does is copy eight bytes from the stack into a string. You probably will find little use for it in an assembly language program.

Notice that the double-precision number is passed by value, not by reference as you might expect.

Example

Perform MKD$(Double#):

```
Extrn B$FMKD:Proc

.DATA?
Double  dq  1 dup (?)     ;Space for 8 bytes

.CODE
PUSH Word ptr [Double+6]  ;Push high word
PUSH Word ptr [Double+4]
PUSH Word ptr [Double+2]
PUSH Word ptr [Double]    ;Push low word
CALL B$FMKD
; AX contains the offset of result string descriptor
```

B$FMKI \SOURCE\MKI$.ASM

BASIC Equivalent: MKI$

Use

Converts a two-byte integer into a two-character string.

Calling Convention

```
PUSH the integer value
CALL B$FMKI
```

Returns offset of result string descriptor in AX.

Notes

All this routine does is copy two bytes from the stack into a string. You probably will find little use for it in an assembly language program.

Example

Perform MKI$(Integer):

```
Extrn B$FMKI:Proc

.DATA?
Integer  dw  1 dup (?)    ;Space for 2 bytes

.CODE
PUSH [Integer]            ;Pass the integer value
CALL B$FMKI
; AX contains the offset of result string descriptor
```

B$FMKL \SOURCE\MKL$.ASM

Synonyms: B$FMKS

BASIC Equivalents: MKL$ and MKS$

Use

Converts a 4-byte long integer or a 4-byte single-precision value into a 4-byte string.

Calling Convention

```
PUSH High order word of the input value
PUSH Low order word of the input value
CALL B$FMKL
```

Offset of the result string's descriptor is returned in AX.

Notes

B$FMKL and B$FMKS are identical. The same block of code is used for each.

All this routine does is copy four bytes from the stack into a string. You probably will find little use for it in an assembly language program.

Notice that the input number is passed by value, not by reference as you might expect.

Example

Perform MKL$(Long&):

```
Extrn B$FMKL:Proc

.DATA?
Long  dd  1 dup (?)       ;Space for 4 bytes

.CODE
PUSH  Word ptr [Long+2]   ;Push high order word
PUSH  Word ptr [Long]     ;Push low word
CALL  B$FMKL
; AX contains the offset of result string descriptor
```

B$FOCT \SOURCE\HEX$.ASM

BASIC Equivalent: OCT$()

Use

Translates a long integer into octal notation.

Calling Convention

```
PUSH High-order word of value to translate
PUSH Low-order word of value to translate
CALL B$FOCT
```

Returns offset of a temporary string descriptor in AX.

Notes

This routine returns with the result in a temporary string. If you want to keep the string for further processing, be sure to assign it to a permanent string variable using B$SASS.

Example

Translate 12345678h to octal notation:

```
Extrn B$FOCT:Proc

.DATA?
EVEN
Octal$  dd  1 dup (?)      ;Space for a string descriptor

.CODE
MOV  AX,1234h              ;Send the high-order word
PUSH AX
MOV  AX,5678h              ; and the low-order word
PUSH AX
CALL B$FOCT                ;Translate to a string
PUSH AX                    ;Pass temporary descriptor
MOV  AX,Offset Octal$      ; and a pointer to our
PUSH AX                    ; string descriptor
CALL B$SASS                ;Copy it to our string
```


B$FREF \SOURCE\FREEFILE.ASM

BASIC Equivalent: FREEFILE

Use

Returns the number of the next free BASIC file.

Calling Convention

```
CALL B$FREF
```

AX has file number (1 - 15) or -1 if all numbers are in use.

Example

```
Extrn B$FREF:Proc

.CODE
CALL B$FREF              ;Get first free file number
OR   AX,AX               ;Test result
JS   NoFilesAvailable    ;Go if -1 (bit 15 set)
; AX has a valid and free file number
```

B$FRI2 \SOURCE\FRE.ASM

BASIC Equivalent: FRE()

Use

Returns the size of the largest available block of free DOS memory, the amount of unused stack space, or the size of the next free block of string memory.

Calling Convention

```
PUSH Action value word
CALL B$FRI2
```

Returns size in bytes in DX:AX.

Notes

The Action parameter is either -1 for DOS memory size, -2 for unused stack space, or any other value for string memory.

Unlike B$FRSD, this routine does not compact the string pool before checking its size (if the action word is other than -1 or -2).

You may prefer to compute the available stack memory directly using code like this:

```
 .DATA?
Extrn PDQ_Stack_Foot:Byte

.CODE
MOV AX,SP
SUB AX,Offset PDQ_Stack_Foot
; now AX holds the free stack memory
```

Example

Find the amount of free DOS space:

```
Extrn B$FRI2:Proc

.CODE
MOV  AX,-1              ;Select action
PUSH AX                 ;Pass it on
CALL B$FRI2             ;DX:AX holds available memory
```

B$FRSD \SOURCE\FRE$.ASM

BASIC Equivalent: FRE("")

Use

Compacts the string pool and then reports the number of bytes that are unused and thus available.

Calling Convention

```
PUSH anything
CALL B$FRSD
```

Returns number of free bytes in DX:AX.

Notes

The number of free bytes in the string pool is always less than 64K, so DX is always 0. If you want to look at the return value in AX only, be sure to treat it as an unsigned number.

If your only reason for calling B$FRSD is to force compaction of the string pool, you can save a few cycles by calling P$Compact directly.

Example

```
Extrn B$FRSD:Proc

.CODE
SUB  AX,AX              ;Create a dummy parameter
PUSH AX                 ;  of zero
CALL B$FRSD             ;Compact string pool
; number of free bytes in the string pool is in DX:AX
```

B$FSCN \SOURCE\SCREEN.ASM

BASIC Equivalent: SCREEN function

Use

Returns either the character or the color attributes of a location on the screen.

Calling Convention

```
PUSH Row (1-based)
PUSH Column (1-based)
PUSH Action word
```

The value is returned in AL, with AH = 0.

Notes

The Action word is either 0 to get character, or any other value to get the attribute.

This routine calls CursorSave to save the cursor. It then moves the cursor to the location you have specified with a BIOS call, reads the character and attribute with another BIOS call, and ends by calling CursorRest to restore the cursor to its previous location. If you know the screen mode and page, you can get the same information much faster by directly accessing video memory.

Example

Get the character in the top left corner of the screen (equivalent to SCREEN(1, 1) in BASIC):

```
Extrn:B$FSCN:Proc

.CODE
MOV  AX,1                ;Row 1
PUSH AX
PUSH AX                  ;Column 1 also
DEC  AX                  ;AX = 0: Get character
PUSH AX
CALL B$FSCN              ;Character is now in AL
```

B$FSEK \SOURCE\LOC.ASM

BASIC Equivalent: SEEK

Use

Find the current position within a file.

Calling Convention

```
PUSH the BASIC file number
CALL B$FSEK
```

File position returned in DX:AX.

Notes

If the file was opened for random access, SEEK returns the current record number. If it was opened in any other mode, SEEK returns the current byte position. In both cases, the result is 1-based; that is, the first byte or record of a file is 1, not 0.

Notice that the result is returned as a long integer.

Example

Find SEEK(4):

```
Extrn B$FSEK:Proc

.CODE
MOV  AX,4                ;Pass the file number
PUSH AX
CALL B$FSEK              ;Position is now in DX:AX
```


B$FSPC \SOURCE\SPC.ASM

BASIC Equivalent: SPC function

Use

Prints a specified number of spaces to the screen or current print device.

Calling Convention

```
PUSH number of spaces
CALL B$FSPC
```

No return value.

Notes

This routine creates a temporary string of the correct length with B$STRI (STRING$), and then prints it without a carriage return or line feed by calling B$PSSD which deletes the temporary string. See B$PSSD for information about the current print device.

Example

Perform PRINT SPC(80);

```
Extrn B$FSPC:Proc

.CODE
MOV  AX,80              ;Number of characters to print
PUSH AX
CALL B$FSPC             ;Print it
```

B$FTAB \SOURCE\TAB.ASM

BASIC Equivalent: TAB()

Use

Moves the cursor to the specified column in the current output device by printing a series of spaces.

Calling Convention

```
PUSH desired column
CALL B$FTAB
```

No return value.

Notes

If the cursor has already moved past the specified column, this routine takes no action.

This routine calls B$SPAC and B$PSSD to create and print a string of the required number of space characters.

Example

Perform PRINT TAB(20);

```
Extrn B$FTAB:Proc

.CODE
MOV  AX,20              ;Column to move to
PUSH AX
CALL B$FTAB             ;Tab to column 20
```

B$FTIM

\SOURCE\TIME$.ASM
\SOURCE_TIME$.ASM

BASIC Equivalent: TIME$ function

Use

Returns the current system time in 24-hour format.

Calling Convention

```
CALL B$FTIM
```

Returns with offset of the result's string descriptor in AX.

Notes

This routine does not use the DOS time services. It reads the number of clock ticks since midnight from memory and calculates the time from the number of clicks. Therefore, it is safe to use this routine within any TSR.

The returned string is in P.D.Q.'s temporary string space. If you want to process it further, you should copy the result to your own string with B$SASS.

The version in _TIME$.ASM uses less code, but cannot be used inside a non-simplified interrupt handler. Use the shorter version to save a few bytes in all non-TSR and simplified TSR programs.

Example

Perform A$ = TIME$:

```
Extrn B$FTIM:Proc
Extrn B$SASS:Proc

.DATA?
EVEN
A$  dd  1 dup (?)          ;Space for string descriptor

.CODE
CALL B$FTIM                ;Get the current time
PUSH AX                    ;Pass on the result
MOV  AX,Offset A$          ;Get pointer to our string desc.
PUSH AX
CALL B$SASS                ;Result now in A$
```

B$FVAL \FPSOURCE\B$FVAL.ASM

BASIC Equivalent: VAL()

Use

Converts a string into a numeric value.

Calling Convention

```
PUSH Offset of the string descriptor
CALL B$FVAL
```

Returns AX = offset of 8-byte floating point number.

Notes

This routine strips leading blanks, tabs, and line feeds. Only 16 or fewer significant digits are converted. The conversion process halts as soon as this routine encounters a character that it can't interpret as part of a number.

This routine uses floating point operations. You must initialize the floating point emulator by calling P$HookFP before you call this procedure.

Example

Convert Work$ to a number and place in ST(0):

```
Extrn B$FVAL:Proc

.DATA?
EVEN
Work$  dd  1 dup (?)      ;Room for string descriptor

.CODE
MOV  AX,Offset Work$      ;Get pointer to string
PUSH AX
CALL B$FVAL               ;Convert to floating point
MOV  BX,AX                ;Put pointer in BX
FLD  QWORD PTR [BX]       ;Put on real number stack
```


B$GET3 \SOURCE\GET.ASM

BASIC Equivalent: GET

Use

Reads from a file at the current file pointer location into a variable.

Calling Convention

If the destination is a numeric variable, a TYPE variable, or a fixed-length string then:

```
PUSH the BASIC file number
PUSH the Segment of the destination
PUSH the Offset of the destination
PUSH the number of bytes to read
CALL B$GET3
```

If the destination is a variable-length string then:

```
PUSH the BASIC file number
PUSH DS (the Segment of the string descriptor)
PUSH the Offset of the string descriptor
PUSH 0 to flag that this is a variable-length string
CALL B$GET3
```

No return value. May report an error by calling P$DoError.

Notes

If any kind of error occurs other than input past end, this routine calls the general error handler, P$DoError.

If you ask for more bytes than remains in the file, an error value of 62 (Input Past End) is posted in P$PDQErr, but no action is taken.

If the destination is a variable-length string, then the current length of the string determines how many bytes will be read by this routine.

Example

Read one long integer from File #1 (which is presumed open):

```
Extrn B$GET3:Proc

.DATA
Extrn P$PDQErr:Word             ;Space for error report
Result&  dd  ?                  ;Room for the data

.CODE
MOV  AX,1                       ;Get the file number
PUSH AX                         ;Pass it on
PUSH DS                         ;Segment of destination
MOV  AX,Offset Result           ;Get address of destination
PUSH AX                         ;  and pass it on
MOV  AX,4                       ;Number of bytes to read
PUSH AX                         ;Pass the last parameter
CALL B$GET3                     ;Read the data
TEST P$PDQErr,-1                ;Past the end?
JNZ  ReadPastEnd                ;Yes -- go handle the error
; Now the data is safely in Result&
```

B$GET4 \SOURCE\GETSEEK.ASM

BASIC Equivalent: GET

Use

B$GET4 reads data into a variable from a specific position in a file.

Calling Convention

If the destination is a numeric variable, a TYPE variable, or a fixed-length string then:

```
PUSH the BASIC file number
PUSH the high-order word of the desired file position
PUSH the low-order word of the desired file position
PUSH the Segment of the destination
PUSH the Offset of the destination
PUSH the number of bytes to read
CALL B$GET4
```

If the destination is a variable-length string then:

```
PUSH the BASIC file number
PUSH the high-order word of the desired file position
PUSH the low-order word of the desired file position
PUSH DS (the segment of the string descriptor)
PUSH the Offset of the string descriptor
PUSH 0 to flag that this is a variable-length string
CALL B$GET4
```

No return value. May report an error by calling P$DoError.

Notes

The file locations are 1-based, so the first byte in the file is at position 1 (DOS calls the first byte position 0).

This routine simply calls B$SSEK and then B$GET3. You could make those calls directly but would probably save little, if any, time by doing so.

If the file has been opened in random access mode, the file position is a record number, not a byte position.

Example

Perform the BASIC statement **GET #1, 5, MyType** assuming the file has been opened in binary mode:

```
Extrn B$GET4:Proc
```

```
.DATA
Extrn P$PDQErr:Word       ;Where the error code will be
MyType db 10 dup (?)      ;You could make this any length
MyTypeLen EQU $-MyType    ;Length of the TYPE variable

.CODE
MOV  AX,1                 ;First send the file number
PUSH AX
DEC  AX                   ;Now AX = 0
PUSH AX                   ;High word of file location
MOV  AX,5                 ;Low word of file location
PUSH AX
PUSH DS                   ;Segment of destination
MOV  AX,Offset MyType     ;Offset of destination
PUSH AX
MOV  AX,MyTypeLen         ;Number of bytes to read
PUSH AX
CALL B$GET4               ;Read the data
TEST P$PDQErr,-1          ;Did an error occur?
JNZ  ReadPastEnd          ;Yes -- go handle it
; Now the data is safely in MyType waiting for processing.
```

B$HARY \SOURCE\HUGARRAY.ASM

Synonyms: B$HAR1

Use

Finds the segment and offset address of an array element. Includes support for huge arrays.

Calling Convention

```
PUSH Leftmost subscript value
PUSH Second-to-left subscript value
. . .
PUSH Rightmost subscript value
PUSH Number of subscripts
MOV  BX,Offset of array descriptor
CALL B$HARY
```

Returns with address of array element in ES:BX. May report an error by calling P$DoError.

Notes

Notice that one of the arguments to B$HARY is passed in the BX register, not on the stack.

B$HARY contains support for huge (larger than 64K) arrays; however, it is useful for arrays of any type.

Example

Load DX:AX with the value of Array&(4, 7, 2):

```
Extrn B$HARY:Proc

.DATA
Array  db  3*4+12  dup (?)        ;Room for array descriptor

.CODE
MOV  AX,4                         ;Push left-most subscript
PUSH AX
MOV  AX,7
PUSH AX
MOV  AX,2                         ;Finish with right-most one
PUSH AX
MOV  AX,3                         ;Number of subscripts
PUSH AX
MOV  BX,Offset Array              ;Pointer to array descriptor
CALL B$HARY
MOV  AX,ES:[BX]                   ;Get low word in AX
MOV  DX,ES:[BX+2]                 ; and high word in DX
```

B$INKY

\SOURCE\INKEY$.ASM
\SOURCE_INKEY$.ASM

BASIC Equivalent: INKEY$

Use

Reads a character from the console.

Calling Convention

```
CALL B$INKY
```

Returns with offset of result string descriptor in AX.

Notes

This routine reads the keyboard with DOS calls, so redirection of input is possible (BIOSInkey doesn't allow redirection).

The result string and descriptor are internal to B$INKY. They will not be overwritten until the next call to this procedure, so you may not need to copy the result to your own string.

The version of this routine in the stub file _INKEY$.ASM uses the BIOS instead of DOS. Therefore, it does not accept redirected input.

Example

Perform the equivalent of:

```
DO
  A$ = INKEY$
LOOP UNTIL A$ <> ""

Extrn B$INKY:Proc
Extrn B$SASS:Proc      ;For string assignment

.DATA?
EVEN
A$  dd  1 dup (?)      ;Space for A$'s descriptor

.CODE
@: CALL B$INKY         ;Look for a key
   MOV  BX,AX          ;Copy the descriptor.
   CMP  [BX],0         ;Does result have 0 length?
    JE   @B            ;Yep -- try again
    PUSH AX            ;Else copy result to
    MOV  AX,Offset A$  ; our variable for
    PUSH AX            ; more processing
    CALL B$SASS
```

B$INPP \SOURCE\INPUT.ASM

BASIC Equivalent: INPUT (from the keyboard)

Use

Optionally prints a prompt and then gets the user input from the keyboard (or from a redirected input source).

Calling Convention

```
Set  flags variable (see below)
PUSH Offset of prompt string descriptor
PUSH Segment of flags variable
PUSH Offset of flags variable
CALL B$INPP
```

No return value. May report an error by calling P$DoError.

Notes

In most cases you will find the PDQInput routine much simpler to set up and use.

If you don't want to display a prompt, use a null string. You can simply pass the address of a data word of 0 instead of the address of a real string descriptor.

The flags variable is a bit record which controls the way that the prompt is displayed and what happens after the user presses Enter:

Bit 0 = 0 - Add a question mark to the prompt
1 - Do not add a question mark

Bit 1 = 0 - Go to a new line after user presses Enter
1 - Do not go to a new line

All other bits are reserved and should be set to 0.

This routine puts the user's input into a string. After you have called it, each of the B$RD*xx* routines (B$RDI2, B$RDI4, B$RDR4, B$RDR8, and B$RDSD) will get their values from that string until you call B$PEOS.

If the user does not enter enough data to fill your expected variables, the trailing variables will have a value of 0 or (for strings) a length of 0. The P.D.Q. routines never issue a warning or "Redo from start" message.

After you have called the necessary B$RD*xx* routines, you must call B$PEOS before the next INPUT from the keyboard or from a disk file.

Example

Perform INPUT "Enter a number", Num&:

```
Extrn B$INPP:Proc
Extrn B$RDI4:Proc          ;To get result
Extrn B$PEOS:Proc          ;To end this input session

.DATA?
FlagVar  dw  1 dup (?)     ;Room for flags
Num      dd  1 dup (?)     ;  and for result

.DATA
DefStr Prompt$,"Enter a number: "

.CODE
MOV.  [FlagVAr],1          ;No question mark, go to next line
MOV   AX,Offset Prompt$    ;Get pointer to string
PUSH  AX
MOV   AX,Offset FlagVar    ;Get pointer to flags
PUSH  DS                   ;Push their segment
PUSH  AX                   ;  and their offset
CALL  B$INPP               ;Get user input
MOV   AX,Offset Num        ;Get pointer to result variable
```

```
PUSH  DS                ;Push its segment
PUSH  AX                ;  and its offset
CALL  B$RDI4            ;Fill it with user's data
CALL  B$PEOS            ;Tell everyone we're done
```

B$INS2 \SOURCE\INSTR2.ASM

BASIC Equivalent:
INSTR (Source$, Search$)

Use

Finds the first occurrence of a substring in a string.

Calling Convention

```
PUSH Offset of Source$ descriptor
PUSH Offset of Search$ descriptor
CALL B$INS2
```

Returns with position in AX.

Notes

This is the two-argument form of INSTR().

If the search string is not found in the source string, AX will contain 0 on return.

Example

Find INSTR(A$, B$):

```
Extrn B$INS2:Proc

.DATA?
EVEN
A$  dd  1 dup (?)       ;Space for string descriptors
B$  dd  1 dup (?)

.CODE
MOV  AX,Offset A$       ;Pointer to source string
PUSH AX
MOV  AX,Offset B$       ;Pointer to search string
PUSH AX
CALL B$INS2             ;Returns position in AX
```

B$INS3 \SOURCE\INSTR.ASM

BASIC Equivalent:
INSTR (Start%, Source$, Search$)

Use

Finds the first occurrence of a substring in a string, with the search starting at a specified position.

Calling Convention

```
PUSH Start position
PUSH Offset of Source$ descriptor
PUSH Offset of Search$ descriptor
CALL B$INS3
```

Returns with position in AX.

Notes

This is the three-argument form of INSTR().

If the search string is not found in the source string, AX will contain 0 on return.

Example

Find INSTR(5, A$, B$):

```
Extrn B$INS3:Proc

.DATA?
EVEN
A$  dd  1 dup (?)          ;Space for string descriptors
B$  dd  1 dup (?)

.CODE
MOV  AX,5                  ;Get start position
PUSH AX
MOV  AX,Offset A$          ;Pointer to source string
PUSH AX
MOV  AX,Offset B$          ;Pointer to search string
PUSH AX
CALL B$INS3                ;Returns position in AX
```

B$KILL

\SOURCE\KILL.ASM
\SOURCE_KILL.ASM

BASIC Equivalent: KILL

Use

Deletes one or more files from disk.

Calling Convention

```
PUSH Offset of string descriptor of filespec
CALL B$KILL
```

No return value. May report an error by calling P$DoError.

Notes

This routine erases all files that match the given filespec. You may use wildcards in the filespec.

B$KILL sets the DTA (disk transfer area) to its own data area. If you are using the DTA for other purposes, make sure you reset the DTA (using Int 21h, Service 1Ah) after you have called B$KILL.

The version in the stub file _KILL.ASM is much shorter because it does not support wildcards in the filespec.

Example

Perform KILL FileSpec$:

```
Extrn B$KILL:Proc

.DATA?
FileSpec$  dd  1 dup (?)  ;Space for string descriptor

.CODE
MOV  AX,Offset Filenames$ ;Get pointer to descriptor
PUSH AX                   ;  and send it on
CALL B$KILL               ;All done
```


B$LBND

\SOURCE\UBOUND.ASM

BASIC Equivalent: LBOUND

Use

Return the lowest available subscript for a dimension of an array.

Calling Convention

```
PUSH Offset of array descriptor
PUSH Dimension number
CALL B$LBND
```

Returns lowest subscript in AX.

Notes

It may be faster to read the LBound value directly from the array descriptor. See *Using Arrays* in the preceding chapter for information about the format of the descriptor.

Example

Find LBOUND(Array, 2):

```
Extrn B$LBND:Proc

.DATA?
Array  db  20 dup (?)      ;Room for descriptor for
                           ;2-dimensional array
.CODE
MOV  AX,Offset Array       ;Get pointer to array desc.
PUSH AX
MOV  AX,2                  ;Dimension we want LBOUND of
PUSH AX
CALL B$LBND                ;Now the result is in AX
```

B$LCAS \SOURCE\LCASE$.ASM

BASIC Equivalent: LCASE$()

Use

Copies and converts all upper case characters in a string to lower case.

Calling Convention

```
PUSH Offset of string descriptor
CALL B$LCAS
```

Returns with the offset of the converted string's descriptor in AX.

Notes

The return value is a temporary string. If you want to save it for further processing, be sure to assign it to a permanent string descriptor.

Example

Perform A$ = LCASE$(A$):

```
Extrn B$LCAS:Proc
Extrn B$SASS:Proc          ;For string assignment

.DATA?
EVEN
A$  dd  1 dup (?)          ;Space for the string descriptor

.CODE
MOV  AX,Offset A$          ;Pass pointer to the string
PUSH AX
CALL B$LCAS                :Convert it to lower case
PUSH AX                    ;Pass the result string
MOV  AX,Offset A$          ; and a pointer to A$
PUSH AX
CALL B$SASS                ;Assign result to A$
```

B$LDFS \SOURCE\FLEN2STR.ASM

Use

Copies a fixed-length string or string portion of a user TYPE variable into near memory with a normal variable-length string descriptor.

Calling Convention

```
PUSH Segment of fixed-length string
PUSH Offset of fixed-length string
PUSH Number of bytes to copy
CALL B$LDFS
```

Returns offset of temporary string descriptor in AX.

Notes

The returned string is in a temporary string descriptor. If you need to keep it for further processing, you can copy it to your own data space by calling B$SASS.

B$LDFS is really a general-purpose routine that will copy any piece of memory into a normal variable-length string. You could, for example, copy the screen into a string with this routine, or use it to transfer data from an EMS page frame window into a string.

Example

Assign A$ = FarString$:

```
Extrn B$LDFS:Proc
Extrn B$SASS:Proc              ;For string assignment

.DATA?
EVEN
```

```
FarString     dd  1 dup (?)      ;Pointer to fixed-length string
FarStringLen  dw  1 dup (?)      ;Length of fixed string
A$            dd  1 dup (?)      ;Room for A$'s descriptor

.CODE
PUSH Word Ptr FarString[2]       ;Push segment of far string
PUSH Word Ptr FarString          ;  and then the offset
PUSH FarStringLen                ;Now send its length
CALL B$LDFS                      ;Copy to a temporary variable
PUSH AX                          ;Pass temporary descriptor
MOV  AX,Offset A$                ;Pointer to our descriptor
PUSH AX                          ;Send it along
CALL B$SASS                      ;Assign result to A$
```

B$LEFT \SOURCE\LEFT$.ASM

BASIC Equivalent: LEFT$

Use

Copies a specified number of characters from the left side of a string to a new, temporary string variable.

Calling Convention

```
PUSH Offset of source string's descriptor
PUSH Number of characters to copy

CALL B$LEFT
```

Returns with offset of descriptor of result string in AX.

Notes

If the number of characters requested is more than the length of the source string, the result string will be a copy of the source string.

The result is a temporary string. If you want to use it for further processing, be sure to assign it to a permanent string.

Example

Perform B$ = LEFT$(A$, 10):

```
Extrn B$LEFT:Proc
Extrn B$SASS:Proc          ;For string assignment

.DATA?
EVEN
A$  dd  1 dup (?)          ;Space for string descriptors
B$  dd  1 dup (?)
```

```
.CODE
MOV  AX,Offset A$          ;Pass pointer to source string
PUSH AX
MOV  AX,10                 ;Pass number of characters
PUSH AX
CALL B$LEFT
PUSH AX                    ;Pass temporary descriptor
MOV  AX,Offset B$          ;Pass pointer to destination
PUSH AX
CALL B$SASS                ;Assign result to B$
```

B$LNIN \SOURCE\LINEINPT.ASM

BASIC Equivalent: LINE INPUT

Use

Reads a line of text from the console or a file into a string.

Calling Convention

```
PUSH Offset of prompt string descriptor
PUSH Segment of result string or string descriptor
PUSH Offset of result string or string descriptor
PUSH Length of result string, or 0 for variable-length string
PUSH End-of-line flag
CALL B$LNIN
```

No return value. May report an error by calling P$DoError.

Notes

You will probably find the PDQInput routine easier to use in an assembly language program.

For line input from a file, precede this routine with a call to B$DSKI to start input from the appropriate file. The input device is set to the console at the end of this routine, so repeated line input from a file will also require repeated calls to B$DSKI.

The End-of-line flag is in effect only if input comes from the console. Set the flag to 0 if you want the cursor to go to a new line when the user presses Enter, or 1 if you want to cursor to remain on the current line.

The prompt string and end-of-line flag are completely ignored if input comes from a file. However, you still must push something for those arguments to keep the stack aligned properly.

Example

Perform LINE INPUT "Enter a string: "; Work$:

```
Extrn B$LNIN:Proc

.DATA
DefStr Prompt$,"Enter a string: "
DefStr Work$

.CODE
MOV  AX,Offset Prompt$          ;Point to the prompt
PUSH AX
MOV  AX,Offset Work$            ;Point to the result
PUSH DS
PUSH AX
SUB  AX,AX                      ;AX = 0: move cursor down
PUSH AX                         ;use 0 for flag too
PUSH AX
CALL B$LNIN                     ;Get a line of input
```

B$LOCK \SOURCE\LOCK.ASM

BASIC Equivalent: LOCK and UNLOCK

Use

Locks or unlocks a file or portion of a file.

Calling Convention

```
PUSH a BASIC file number
PUSH the High Word of start of region to lock
PUSH the Low Word of start of region to lock
PUSH the High Word of end of region to lock
PUSH the Low Word of end of region to lock
PUSH the Action Word (see below)
CALL B$LOCK
```

No return value. May report an error by calling P$DoError.

Notes

The Action Word is a bit record:

Bit 0: 0 to lock, 1 to unlock

Bit 1: 0 for whole file, 1 for a portion of a file

Bits 2 to 15: reserved. All should be set to 0

LOCK and UNLOCK require DOS 3.0 and later, and also that SHARE be installed.

A call to unlock a previously-locked portion of a file *must* have the same start and end values and the same action word except for bit 0.

If you want to lock the entire file (bit 1 of the Action Word is 0) then the start region and end region values will be ignored.

If the file was opened in random mode, the start and end region numbers refer to records (the first record is 1). If the file was opened in any other mode, the start and end region numbers refer to bytes (the first byte in the file is 1).

Example

LOCK and UNLOCK records 4 to 20 of file #2 (assumes that file #2 was opened in random mode):

```
Extrn B$LOCK:Proc

.CODE
SUB  DX,DX              ;DX = 0
MOV  AX,2               ;Get file number
PUSH AX
MOV  AX,4               ;Starting record
PUSH DX                 ;High word of start
PUSH AX                 ;Low word of start
MOV  AX,20              ;Ending record
PUSH DX                 ;High word of end
PUSH AX                 ;Low word of end
MOV  AX,10b             ;Action: Lock a region
PUSH AX
CALL B$LOCK             ;Now the region is locked
...                     ;process the locked region of the file
SUB  DX,DX              ;DX = 0 -- ready to unlock
MOV  AX,2               ;Get file number
PUSH AX
MOV  AX,4               ;Starting record
PUSH DX                 ;High word of start
PUSH AX                 ;Low word of start
MOV  AX,20              ;Ending record
PUSH DX                 ;High word of end
PUSH AX                 ;Low word of end
MOV  AX,11b             ;Action: Unlock a region
PUSH AX
CALL B$LOCK             ;Now the region is unlocked
```

B$LOCT

\SOURCE\LOCATE.ASM
\SOURCE_LOCATE.ASM

BASIC Equivalent: LOCATE

Use

Set the screen location and shape of the cursor.

Calling Convention

Push the arguments in this order:

```
Row
Column
Cursor
Start
Stop
```

For each of these arguments, push a 0 if you want to leave the current value, or a 1 followed by the new value:

```
PUSH 0 to show that Row stays unchanged
```

or

```
PUSH 1 to show that Row should be changed
PUSH New row setting
```

You must go through the list in order, but you can stop at any time. If you want to set the Cursor value, for example, you will need to also set Row and Column (or push 0 for each). But you won't have to pass any values for Start and Stop.

After the arguments above are on the stack, do this:

```
PUSH the number of words ahead of this argument
CALL B$LOCT
```

No return value.

To set the Row and Column only:

```
PUSH 1
PUSH New Row
PUSH 1
PUSH New Column
PUSH 4
CALL B$LOCT
```

To set cursor to 0 only:

```
PUSH 0
PUSH 0
```

```
PUSH 1
PUSH 0
PUSH 4
CALL B$LOCT
```

To set Start and Stop only:

```
PUSH 0
PUSH 0
PUSH 0
PUSH 1
PUSH New Start
PUSH 1
PUSH New Stop
PUSH 7
CALL B$LOCT
```

Notes

You may find it easier to use P.D.Q.'s Cursor routines or to use BIOS routines directly instead of B$LOCT.

The version of LOCATE in the stub file, _LOCATE.ASM, requires that you push the following arguments and no others:

```
PUSH 1
PUSH New Row
PUSH 1
PUSH New Column
PUSH 4
```

Example

Perform LOCATE 10, 1, 1 (Row 10, Column 1, Cursor On):

```
Extrn B$LOCT:Proc
.CODE
MOV  BX,1                ;To show active arguments
MOV  AX,10               ;New row value
PUSH BX                  ;Set a new row value
PUSH AX                  ;Here it is
MOV  AX,1                ;Get new column value
PUSH BX                  ;Here comes a new column
PUSH AX                  ;Move to column 1
PUSH BX                  ;Here comes a new Cursor arg.
PUSH AX                  ;Turn cursor on
MOV  AX,6                ;Number of arguments
PUSH AX
CALL B$LOCT
```

B$LSET \SOURCE\LSET.ASM

BASIC Equivalent: LSET

Use

Copies and left-justifies data from a variable-length string into either a variable-length string or a fixed block of memory (normally part of a TYPE or FIELD variable).

Calling Convention

```
PUSH Offset of source string descriptor
PUSH Segment of destination or of destination descriptor
PUSH Offset of destination or of destination descriptor
PUSH Length of destination (or zero)
CALL B$LSET
```

No return value.

Notes

If the destination is a variable-length string use a value of zero for its length.

This routine automatically pads the destination with spaces if it is longer than the source. If the source is longer, it is truncated to fit into the destination.

Example

LSET A$ = B$ (both are variable-length strings):

```
Extrn B$LSET:Proc

.DATA?
EVEN
A$  dd  1 dup (?)          ;Space for string descriptors
B$  dd  1 dup (?)

.CODE
MOV  AX,Offset B$          ;Point to source descriptor
PUSH AX
MOV  AX,Offset A$          ;Point to dest. descriptor
PUSH DS                    ;Push its segment
PUSH AX                    ;  and its offset
SUB  AX,AX                 ;AX = 0: destination is a
PUSH AX                    ;  variable-length string
CALL B$LSET
```

B$LTRM \SOURCE\LTRIM$.ASM

BASIC Equivalent: LTRIM$

Use

Copies and removes leading CHR$(32) spaces and CHR$(0) null characters from the left end of a string.

Calling Convention

```
PUSH Offset of string descriptor
CALL B$LTRM
```

Returns with offset of the result string descriptor in AX.

Notes

The result is returned in a temporary string. You must copy the temporary string to a permanent string if you want to save it for processing at a later time.

Example

Perform A$ = LTRIM$(A$):

```
Extrn B$LTRM:Proc
Extrn B$SASS:Proc          ;For string assignment

.DATA?
EVEN
A$  dd  1 dup (?)          ;Room for string descriptor

.CODE
MOV  AX,Offset A$          ;Create pointer to A$
PUSH AX
CALL B$LTRM                ;Strip leading spaces & nulls
PUSH AX                    ;Pass pointer to result descriptor
MOV  AX,Offset A$          ;Where we want the result
CALL B$SASS                ;The result is now in A$
```


B$MDIR \SOURCE\MKDIR.ASM

BASIC Equivalent: MKDIR

Use

Creates a new subdirectory.

Calling Convention

```
PUSH Offset of descriptor holding the directory name
CALL B$MDIR
```

No return value. May report an error by calling P$DoError.

Example

Perform MKDIR DirName$:

```
Extrn B$MDIR:Proc

.DATA?
EVEN
DirNam$  dd  1 dup (?)    ;Space for string descriptor

.CODE
MOV  AX,Offset DirNam$    ;Get pointer to descriptor
PUSH AX
CALL B$MDIR               ;Create the subdirectory
```

B$MUI4

\SOURCE\MULTLONG.ASM
\SOURCE\MULTLNG3.ASM

Use

Multiplies two signed long integers: X& * Y&

Calling Convention

```
PUSH High Word of Y&
PUSH Low Word of Y&
PUSH High Word of X&
PUSH Low Word of X&
CALL B$MUI4
```

Returns with result in DX:AX.

Notes

The alternate routine in MULTLNG3.ASM performs the same function for 386 and 486 CPUs using 32-bit registers. Use it, or perform the operation directly, if you know that your code will be running on a 386 or 486 computer.

Example

Perform Z& = X& * Y&:

```
Extrn B$MUI4

.DATA
X&  dd  ?                 ;Space for variables
Y&  dd  ?
```

```
Z&  dd  ?

.CODE
PUSH Word ptr [Y&+2]      ;Push high word of Y&
PUSH Word ptr [Y&]        ; and low word
PUSH Word ptr [X&+2]      ;Push high word of X&
PUSH Word ptr [X&]        ; and low word
CALL B$MUI4               ;Multiply them
MOV  Word ptr [Z&+2],DX   ;Save high word of result
MOV  Word ptr [Z&],AX     ; and low word
```

B$NAME \SOURCE\NAME.ASM

BASIC Equivalent: NAME

Use

Rename a file (and optionally move it to a new subdirectory on the same drive).

Calling Convention

```
PUSH Offset of descriptor holding the current name
PUSH Offset of descriptor holding the new name
CALL B$NAME
```

No return value. May report an error by calling P$DoError.

Notes

You can use B$NAME to move a file to a new subdirectory on the same drive by using a fully-qualified path and file specification for the new name. Note, however, that this works with DOS 3.0 or later only.

Example

Perform NAME A$ AS B$:

```
Extrn B$NAME:Proc

.DATA?
EVEN
A$  dd  1 dup (?)         ;Space for string descriptors
B$  dd  1  dup(?)

.CODE
MOV  AX,Offset A$         ;Get pointer to current name
PUSH AX
MOV  AX,Offset B$         ;Get pointer to new name
PUSH AX
CALL B$NAME
```

B$OGSA and B$OGTA \SOURCE\ONGOTO.ASM

BASIC Equivalents: ON GOSUB (B$OGSA) and ON GOTO (B$OGTA)

Use

Perform an indexed jump or call to a NEAR location.

Calling Convention

For ON Value GOSUB Label1, Label2, Label3:

```
MOV  BX,Value
CALL B$OGSA
DB   Number of labels that follow
DW   Offset of Label1
DW   Offset of Label2
DW   Offset of Label3
The next statement in sequence continues here
```

No return value.

Notes

ON GOSUB and ON GOTO can branch only to labels that are in the same code segment. Therefore, if you use B$OGSA (ON GOSUB), make sure that each routine ends with a near return.

For most assembly-language programs, a normal jump or call table, for either near or far branches, will probably be more efficient than calling these routines.

Notice that control falls through without a branch if Value is 0 or is greater than the number of labels specified.

Example

Perform ON X GOTO Loc1, Loc2, Loc3, Loc4:

```
Extrn B$OGTA:Proc

.DATA?
X  dw  1 dup (?)          ;Space for value variable

.CODE
MOV  BX,[X]               ;Get the value
CALL B$OGSA               ;Branch to a subroutine
DB   4                    ;Number of labels
DW  Offset Loc1           ;First branch address
DW  Offset Loc2
DW  Offset Loc3
```

```
DW  Offset Loc4
;Control goes here if X = 0 or X > 4
;Else control returns here when the routine returns
```

B$OOPN \SOURCE\OPENOLD.ASM

BASIC Equivalent: OPEN (older, terse syntax)

Use

OPEN a file and assign it to a BASIC file number.

Calling Convention

```
PUSH Offset of string descriptor holding the open mode
PUSH File number
PUSH Offset of string descriptor holding the filename
PUSH Record length (ignored except for RANDOM files)
```

No return value. May report an error by calling P$DoError.

Notes

This routine simply translates the old-style open mode ("I", "O", "R", "A", or "B") into a number and then calls B$OPEN to do all the real work. Capitalization of the open mode code is not important.

You must include a Record Length parameter for all calls to OOPN, regardless of the open mode you are using. For non-random files, the value is necessary but ignored.

See the comments for B$OPEN for more details about opening files.

Example

Perform OPEN "R", #3, FileName$, 212:

```
Extrn B$OOPN:Proc

.DATA
DefStr FileName$
DefStr Mode$,"r"

.CODE
MOV  AX,Offset Mode$      ;Pointer to mode descriptor
PUSH AX
MOV  AX,3                 ;Get file number
PUSH AX
MOV  AX,Offset FileName$  ;Pointer to name descriptor
PUSH AX
MOV  AX,212               ;Record length
```

```
PUSH AX
CALL B$OOPN                ;Go open the file
```

B$OPEN \SOURCE\OPEN.ASM

BASIC Equivalent: OPEN (newer, "wordy" syntax)

Use

OPEN a file and assign it to a BASIC file number.

Calling Convention

```
PUSH Offset of file name string descriptor
PUSH File number
PUSH Record length (ignored except for RANDOM files)
PUSH Open type (see below)
CALL B$OPEN
```

No return value. May report an error by calling P$DoError.

Notes

The file number must be greater than 0 and less than 16, and cannot be a number already in use.

The low byte of the Open Type defines the open mode for the file:

```
1 = INPUT mode
2 = OUTPUT mode
4 = RANDOM mode
8 = APPEND mode
32 = BINARY mode
```

The high byte of the Open Type defines the sharing modes, and indicates what access, if any, other processes have to the file:

```
0 = No sharing
1 = Deny both reading and writing
2 = Allow reading but not writing
3 = Allow writing but not reading
4 = Allow both reading and writing
```

If you are opening the file in RANDOM mode and want the default record length of 128, you can use either -1 or 128 as the record length value. You must pass some record length for all calls to B$OPEN, but the value will be ignored for files opened in modes other than RANDOM.

Example

Perform OPEN FileName$ FOR RANDOM AS #3 LEN = 212:

```
Extrn B$OPEN:Proc

.DATA?
EVEN
FileName$  dd  1 dup (?)  ;Space for string descriptor

.CODE
MOV  AX,Offset FileName$  ;Pass pointer to descriptor
PUSH AX
MOV  AX,3                 ;Pass the file number
PUSH AX
MOV  AX,212               ;Pass the record length
PUSH AX
MOV  AX,4                 ;Pass Open Type: 4 = RANDOM
PUSH AX
CALL B$OPEN               ;Open the file
```

B$PCI2 \SOURCE\PRINTINT.ASM

BASIC Equivalent: PRINT X%,

Use

Prints an integer value and then moves to the next tab stop.

Calling Convention

```
PUSH integer value
CALL B$PCI2
```

No return value. May report an error by calling P$DoError when printing to a file.

Notes

This routine converts the integer to a string, and then uses B$PSSD to do the printing and B$PCSD to tab. It updates the tab position and does not reset the print handle.

Example

Perform PRINT 9,

```
Extrn B$PCI2:Proc

.CODE
MOV  AX,9                 ;Get the value to print
PUSH AX
CALL B$PCI2
```

B$PCI4 \SOURCE\PRINTLNG.ASM

BASIC Equivalent: PRINT X&,

Use

Print a long integer value and then move to the next tab stop.

Calling Convention

```
PUSH high word of value
PUSH low word of value
CALL B$PCI4
```

No return value. May report an error by calling P$DoError when printing to a file.

Notes

This routine converts the long integer to a string, and then uses B$PSSD to do the printing and B$PCSD to tab. It updates the tab position and does not reset the print handle.

Example

Perform PRINT 12345678h,

```
ExtrnB$PCI4:Proc

.CODE
MOV  AX,1234h            ;Get high word
PUSH AX
MOV  AX,5678h            ; and the low word
PUSH AX
CALL B$PCI4
```

B$PCR4 \FPSOURCE\B$PCR4.ASM

BASIC Equivalent: PRINT X!,

Use

Print a 4-byte (single precision) real number, and then move to the next tab stop.

Calling Convention

```
PUSH Most significant word of the number
PUSH Least significant word of the number
CALL B$PCR4
```

No return value. May report an error by calling P$DoError when printing to a file.

Notes

This routine converts the number to a string by calling B$STR4, and then prints the string with B$PCSD.

This routine uses floating point operations. You must initialize the floating point emulator by calling P$HookFP before you call this procedure.

Example

Print the real number at ST(0) and then a tab:

```
Extrn B$PCR4:Proc

.DATA?
TempSingle  dd  1 dup (?)                ;Room for the number

.CODE
FSTP DWord ptr [TempSingle]              ;Pop number off fp stack
PUSH Word ptr [TempSingle+2]             :Push it onto the stack
PUSH Word ptr [TempSingle]
CALL B$PCR4                              ;And print it
```

B$PCR8 \FPSOURCE\B$PCR8.ASM

BASIC Equivalent: PRINT X#,

Use

Print an 8-byte (double precision) real number and then move to the next tab stop.

Calling Convention

```
PUSH Most significant word of the number
PUSH Next most significant word of the number
PUSH Next most significant word of the number
PUSH Least significant word of the number
CALL B$PCR8
```

No return value. May report an error by calling P$DoError when printing to a file.

Notes

This routine converts the number to a string by calling B$STR8, and then prints the string with B$PCSD.

This routine uses floating point operations. You must initialize the floating point emulator by calling P$HookFP before you call this procedure.

Example

Print the real number at ST(0) and then a tab:

```
Extrn B$PCR8:Proc

.DATA?
TempDouble  dq  1 dup (?)          ;Room for the number

.CODE
FSTP  QWord ptr [TempDouble]       ;Pop number off stack
MOV   BX,Offset TempDouble         ;Get pointer for addressing
PUSH  Word Ptr [BX+6]              ;Push in onto the stack
PUSH  Word Ptr [BX+4]
PUSH  Word Ptr [BX+2]
PUSH  Word Ptr [BX]
CALL  B$PCR8                       ;And print it
```

B$PCSD \SOURCE\PRNCOMMA.ASM

BASIC Equivalent: PRINT X$,

- **Use**

  Prints a string and then moves to the next print zone.

- **Calling Convention**

```
PUSH Offset of string descriptor
CALL B$PCSD
```

  No return value. May report an error by calling P$DoError when printing to a file.

- **Notes**

  After this routine prints a string, it updates the tab position.

Example

Perform PRINT WORK$,

```
Extrn B$PCSD:Proc

.DATA?
EVEN
Work$  dd  1 dup (?)       ;Space for descriptor

.CODE
MOV  AX,Offset Work$
```

```
PUSH AX
CALL B$PCSD
```

B$PEI2 \SOURCE\PRINTINT.ASM

BASIC Equivalent: PRINT X%

Use

Print an integer value followed by a CRLF

Calling Convention

```
PUSH the integer value
CALL B$PEI2
```

No return value. May report an error by calling P$DoError when printing to a file.

Notes

This routine calls B$PESD, which resets the tab position on the current device to 0 and resets the print handle so that the next print will be to the screen.

Example

Print 127:

```
Extrn B$PEI2:Proc

.CODE
MOV  AX,127              ;Get value to print
PUSH AX
CALL B$PEI2
```


B$PEI4 \SOURCE\PRINTLNG.ASM

BASIC Equivalent: PRINT X&

Use

Prints a long integer value followed by a CRLF.

Calling Convention

```
PUSH the high word of the value
PUSH the low word of the value
CALL B$PEI4
```

No return value. May report an error by calling P$DoError when printing to a file.

Notes

This routine calls B$PESD, which resets the tab position on the current device to 0 and resets the print handle so that the next print will be to the screen.

Example

Print 456789Ah:

```
Extrn B$PEI4:Proc

.CODE
MOV  AX,456h            ;Get high word
PUSH AX
MOV  AX,789Ah           ;Get low word
PUSH AX
CALL B$PEI4
```

B$PEOS \SOURCE\R$PEOS.ASM

Use

Cleans up after a non-console input or output.

Calling Convention

```
CALL B$PEOS
```

No return value.

Notes

The name of this routine means *Print End Of Statement* and it is used to make sure that output from each PRINT statement gets sent to the right place. Without it, a **PRINT #n, "xxx";** followed by **PRINT "yyy" would send the "yyy" to a file instead of the screen.**

Just as important, this routine cleans up after an INPUT or INPUT #n and the subsequent reads to fill variables. It ensures that future reads from the console will be handled properly.

Example

```
Extrn B$PEOS:Proc

.CODE
CALL B$PEOS             ;Everything's back to normal
```

B$PER4 \FPSOURCE\B$PER4.ASM

BASIC Equivalent: PRINT X!

Use

Print a 4-byte (single precision) real number with a trailing carriage return and line feed.

Calling Convention

```
PUSH High word of the number
PUSH Low word of the number
CALL B$PER4
```

No return value. May report an error by calling P$DoError when printing to a file.

Notes

This routine converts the number to a string by calling B$STR4 and then prints the string with B$PESD.

This routine uses floating point operations. You must initialize the floating point emulator by calling P$HookFP before you call this procedure.

Example

Print the real number TempSingle and then a carriage return and line feed:

```
Extrn B$PER4:Proc

.DATA?
TempSingle  dd  1 dup(?)          ;Room for the number

.CODE
PUSH Word Ptr [TempSingle+2]      ;Pass the high word
PUSH Word Ptr [TempSingle]        ;  then the low word
CALL B$PER4                       ;And print it
```


B$PER8 \FPSOURCE\B$PER8.ASM

BASIC Equivalent: PRINT X#

Use

Print an 8-byte (double precision) real number with a trailing carriage return and line feed.

Calling Convention

```
PUSH Most significant word of the number
PUSH Next most significant word of the number
PUSH Next most significant word of the number
PUSH Least significant word of the number
CALL B$PER8
```

No return value. May report an error by calling P$DoError when printing to a file.

Notes

This routine converts the number to a string by calling B$STR8 and then prints the string with B$PESD.

This routine uses floating point operations. You must initialize the floating point emulator by calling P$HookFP before you call this procedure.

Example

Print the real number at ST(0) and then a carriage return and line feed:

```
Extrn B$PER8:Proc

.DATA?
TempDouble  dq  1  dup (?)        ;Room for the number

.CODE
FSTP QWord ptr [TempDouble]       ;Pop number off fp stack
MOV  BX,Offset TempDouble         ;Get pointer for addressing
PUSH Word Ptr [BX+6]              ;Push in onto the CPU stack
PUSH Word Ptr [BX+4]
PUSH Word Ptr [BX+2]
PUSH Word Ptr [BX]
CALL B$PER8                       ;And print it
```

B$PESD

\SOURCE\PRINT.ASM
\SOURCE_CPRINT.ASM

BASIC Equivalent: PRINT X$

Use

Print a string with a trailing carriage return and line feed.

Calling Convention

```
PUSH Offset of string descriptor
CALL B$PESD
```

No return value. May report an error by calling P$DoError when printing to a file.

Notes

This routine resets the tab column for the current device to 0, and resets the print handle to 1 (standard output) for the next print operation.

The version in _CPRINT.ASM uses the current color attributes stored in P$Color for each character that it prints if output is to the screen

Example

Print Work$:

```
Extrn B$PESD:Proc

.DATA?
EVEN
Work$  dd  1 dup(1)        ;Room for the string descriptor

.CODE
MOV  AX,Offset Work$       ;Get address of descriptor
PUSH AX
CAL  B$PESD
```

B$PSI2 \SOURCE\PRINTINT.ASM

BASIC Equivalent: PRINT X%;

Use

Print an integer value without a trailing carriage return and line feed.

Calling Convention

```
PUSH Integer value
CALL B$PSI2
```

No return value. May report an error by calling P$DoError when printing to a file.

Notes

This routine calls B$PSSD after converting the integer into a string. It updates the tab column for the current device and does not reset the print handle.

Example

Perform PRINT A%;

```
.DATA?
A%  dw  1 dup (?)          ;Space for the integer

.CODE
```

```
MOV AX,[A%]             ;Get the value
PUSH AX
CALL B$PSI2
```

B$PSI4 \SOURCE\PRINTLNG.ASM

BASIC Equivalent: PRINT X&;

Use

Print a long integer value without a trailing carriage return and line feed.

Calling Convention

```
PUSH High word of value
PUSH Low word of value
CALL B$PSI4
```

No return value. May report an error by calling P$DoError when printing to a file.

Notes

This routine calls B$PSSD after converting the integer into a string. It updates the tab column for the current device and does not reset the print handle.

Example

Perform PRINT A&;

```
.DATA?
A&  dd  1 dup (?)       ;Space for the long integer

.CODE
LESAX,[A&]              ;Get the value
PUSH ES                 ;Push the high word
PUSH AX                 ;  and the low word
CALL B$PSI4
```

B$PSR4 \FPSOURCE\B$PSR4.ASM

BASIC Equivalent: PRINT X!;

Use

Print a 4-byte (single precision) real number and then a trailing space.

Calling Convention

```
PUSH High word of the number
PUSH Low word of the number
CALL B$PSR4
```

No return value. May report an error by calling P$DoError when printing to a file.

Notes

This routine converts the number to a string by calling B$STR4 and then prints the string with B$PSSD.

This routine uses floating point operations. You must initialize the floating point emulator by calling P$HookFP before you call this procedure.

Example

Print the real number at ST(0) and then a blank space:

```
Extrn B$PSR4:Proc

.DATA?
TempSingle  dd  1 dup (?)          ;Room for the number

.CODE
FSTP QWord Ptr [TempSingle]        ;Pop number off fp stack
PUSH Word Ptr [TempSingle+2]       :Put it onto the CPU stack
PUSH Word Ptr [TempSingle]
CALL B$PSR4                        ;And print it
```

B$PSR8 \FPSOURCE\B$PSR8.ASM

BASIC Equivalent: PRINT X#;

Use

Print an 8-byte (double precision) real number and then a trailing space.

Calling Convention

```
PUSH Most significant word of the number
PUSH Next most significant word of the number
PUSH Next most significant word of the number
PUSH Least significant word of thenumber
CALL B$PSR8
```

No return value. May report an error by calling P$DoError when printing to a file.

Notes

This routine converts the number to a string by calling B$STR8 and then prints the string with B$PSSD.

This routine uses floating point operations. You must initialize the floating point emulator by calling P$HookFP before you call this procedure.

Example

Print the real number TempDouble and then a blank space:

```
Extrn B$PSR8:Proc

.DATA?
TempDouble  dq  1 dup (?) ;Room for the number

.CODE
MOV  BX,Offset TempDouble ;Get pointer for addressing
PUSH Word Ptr [BX+6]      ;Push in onto the stack
PUSH Word Ptr [BX+4]
PUSH Word Ptr [BX+2]
PUSH Word Ptr [BX]
CALL B$PSR8               ;And print it
```

B$PSSD

\SOURCE\PRINT.ASM
\SOURCE_CPRINT.ASM

BASIC Equivalent: PRINT X$;

Use

Print a string without a trailing carriage return and line feed.

Calling Convention

```
PUSH Offset of string descriptor
CALL B$PSSD
```

No return value. May report an error by calling P$DoError when printing to a file.

Notes

Unlike B$PESD, this routine updates (but does not reset) the tab column for the current print device and does not reset the print handle.

The version in _CPRINT.ASM uses the current color attributes stored in P$Color for each character that it prints when output is to the screen.

Example

Perform PRINT Work$;

```
Extrn B$PSSD:Proc

.DATA?
EVEN
Work$  dd  1 dup(?)        ;Space for descriptor

.CODE
MOV  AX,Offset Work$
PUSH AX
CALL B$PSSD
```

B$PUT3 \SOURCE\PUT.ASM

BASIC Equivalent: PUT

Use

Outputs a series of bytes to a file using the current seek location.

Calling Convention

```
PUSH the BASIC file number
PUSH the Segment of the variable to PUT
PUSH the Offset of the variable to PUT
PUSH the number bytes to PUT (or zero)
CALL B$PUT3
```

No return value. May report an error by calling P$DoError.

Notes

This routine requires a BASIC file number, not a DOS file handle, as its first parameter.

If you want to PUT a variable-length string into a disk file, set the length word to 0. If you are using any other kind of variable (including a fixed-length string or TYPE variable), be sure to specify the correct number of bytes.

Example

Perform PUT #1, , A&:

```
Extrn B$PUT3:Proc

.DATA?
A&  dd  1 dup (?)          ;Space for the variable
```

```
.CODE
MOV  AX,1                ;Get the BASIC file number
PUSH AX
PUSH DS                  ;Segment of the variable
MOV  AX,Offset A&        ;Get the offset
PUSH AX
MOV  AX,4                ;Length of a long integer
PUSH AX
CALL B$PUT3
```

B$PUT4 \SOURCE\PUTSEEK.ASM

BASIC Equivalent: PUT #n

Use

Move the file pointer to a specified location in the file and then output a series of bytes to that file.

Calling Convention

```
PUSH BASIC file number
PUSH High word of the file location
PUSH Low word of the file location
PUSH Segment of the variable to PUT
PUSH Offset of the variable to PUT
PUSH Number of bytes to PUT, or 0 if writing a variable-length string.
```

No return value. May report an error by calling P$DoError.

Notes

This routine requires a BASIC file number, not a DOS file handle, as its first parameter.

If you want to PUT a variable-length string into a disk file be sure to set the length word to 0. If you are using any other kind of variable (including a fixed-length string or TYPE variable), be sure to specify the correct number of bytes.

It is your responsibility to calculate the (1-based) file location for the output. When you use random access record numbers in BASIC (as in PUT #1, 5, F$), the compiler determines the offset for you by calculating:

```
((RecordNumber - 1) * RecordLength) + 1)
```

This routine expects the result of that calculation, expressed as a long integer, as its second and third parameters.

Example

Assuming that file #1 has a record length of 70 bytes but the file was opened for BINARY mode, emulate PUT #1, 3, F$:

```
Extrn B$PUT4:Proc

.DATA?
EVEN
F$  dd  1 dup (?)          ;Space for a string descriptor

.CODE
MOV  BX,1                  ;Get the file number
PUSH BX                    ;Pass it on
DEC  BX                    ;BX = 0
PUSH BX                    ;High word of 0
MOV  AX,141                ;File location for record 3
PUSH AX
PUSH DS                    ;Segment of F$
MOV  AX,Offset F$          ;Offset of string descriptor
PUSH AX
PUSH BX                    ;Set length = 0 for a string
CALL B$PUT4
```

B$RDI2 \SOURCE\READINT.ASM

Use

After a call to B$INPP (INPUT) or B$DSKI (INPUT #), reads a value from the current input file or device into an integer.

Calling Convention

```
PUSH Segment of the integer variable
PUSH Offset of the integer variable
CALL B$RDI2
```

No return value. May report an error by calling P$DoError.

Notes

This procedure retrieves the next item as a string and then calls PDQVall to convert the string to an integer. If the item cannot be converted to an integer (for example, if it is text or null), the result will be 0.

Example

Perform INPUT #1, Num%:

```
Extrn B$RDI2:Proc
Extrn B$DSKI:Proc          ;Starts file input
Extrn B$PEOS:Proc          ;Ends file input

.DATA
```

```
Num      dw  ?            ;Space for the result
FileNum  dw  1            ;File to read from

.CODE
MOV  AX,Offset FileNum    ;Get pointer to file number
PUSH AX
CALL B$RDSKI              ;Start file input
MOV  AX,Offset Num        ;DS:AX points to Num
PUSH DS
PUSH AX
CALL B$RDI2               ;Get integer input
CALL B$PEOS               ;Then clean up
```

B$RDI4 \SOURCE\READLONG.ASM

Use

After a call to B$INPP (INPUT) or B$DSKI (INPUT #), reads a value from the current input file or device into a long integer.

Calling Convention

```
PUSH Segment of the long integer variable
PUSH Offset of the long integer variable
CALL B$RDI4
```

No return value. May report an error by calling P$DoError.

Notes

This procedure retrieves the next item as a string and then calls PDQValL to convert the string to a long integer. If the item cannot be converted to a long integer (for example, if it is text or null), the result will be 0.

Example

Perform INPUT #1, Num&:

```
Extrn B$RDI4:Proc
Extrn B$DSKI:Proc         ;Starts file input
Extrn B$PEOS:Proc         ;Ends file input

.DATA
Num      dd  ?            ;Space for the result
FileNum  dw  1            ;File to read from

.CODE
MOV  AX,Offset FileNum    ;Get pointer to file number
PUSH AX
CALL B$RDSKI              ;Start file input
MOV  AX,Offset Num        ;DS:AX points to Num
PUSH DS
PUSH AX
```

```
CALL B$RDI4             ;Get long integer input
CALL B$PEOS             ;Then clean up
```

B$RDIR \SOURCE\RMDIR.ASM

BASIC Equivalent: RMDIR

Use

Removes an existing, empty subdirectory.

Calling Convention

```
PUSH Offset of string descriptor holding directory name
CALL B$RDIR
```

No return value. May report an error by calling P$DoError.

Notes

If the directory name isn't already in a normal BASIC string, it is faster and easier to call DOS Int 21h, service 3Ah yourself. If the directory name is in a string, this routine does the work of extracting the name, adding a trailing NULL character, and making the call for you.

Example

Remove the directory whose name is in Dir$:

```
Extrn B$RDIR:Proc

.DATA?
EVEN
Dir$  dd  1 dup (?)       ;Room for a string descriptor

.CODE
MOV  AX,Offset Dir$       ;Make a pointer to descriptor
PUSH AX
CALL B$RDIR
```


B$RDR4 \FPSOURCE\READSNGL.ASM

Use

After a call to B$INPP (INPUT) or B$DSKI (INPUT #), reads a value from the current input file or device into a single-precision variable.

Calling Convention

```
PUSH Segment of result variable
PUSH Offset of result variable
CALL B$RDR4
```

No return value. May report an error by calling P$DoError.

Notes

This procedure retrieves the next item as a string and then calls B$FVAL to convert the string to a single precision value. If the item cannot be converted (for example, if it is text or null), the result will be 0.

This routine uses floating point operations. You must initialize the floating point emulator by calling P$HookFP before you call this procedure.

Example

Perform INPUT #1, Num!:

```
Extrn B$RDR4:Proc
Extrn B$DSKI:Proc          ;Starts file input
Extrn B$PEOS:Proc          ;Ends file input and cleans up

.DATA
Num      dd  ?            ;Location for result
FileNum  dw  1            ;File number to use

.CODE
MOV  AX,Offset FileNum     ;Pointer to file number
PUSH AX
CALL B$DSKI                ;Start file input
MOV  AX,Offset Num         ;DS:AX points to result
PUSH DS
PUSH AX
CALL B$RDR4                ;Get a single-precision value
CALL B$PEOS                ;Stop file input and tidy up
```

B$RDR8 \FPSOURCE\READDBL.ASM

Use

After a call to B$INPP (INPUT) or B$DSKI (INPUT #), reads a value from the current input file or device into a double-precision variable.

Calling Convention

```
PUSH Segment of result variable
PUSH Offset of result variable
CALL B$RDR8
```

No return value. May report an error by calling P$DoError.

Notes

This procedure retrieves the next item as a string and then calls B$FVAL to convert the string to a double precision value. If the item cannot be converted (for example, if it is text or null), the result will be 0.

This routine uses floating point operations. You must initialize the floating point emulator by calling P$HookFP before you call this procedure.

Example

Perform INPUT #1, Num# :

```
Extrn B$RDR8:Proc
Extrn B$DSKI:Proc          ;Starts file input
Extrn B$PEOS:Proc          ;Ends file input and cleans up

.DATA
Num      dq  ?            ;Location for result
FileNum  dw  1            ;File number to use

.CODE
MOV  AX,Offset FileNum     ;Pointer to file number
PUSH AX
CALL B$DSKI                ;Start file input
MOV  AX,Offset Num         ;DS:AX points to result
PUSH DS
PUSH AX
CALL B$RDR8                ;Get a double-precision value
CALL B$PEOS                ;Stop file input and tidy up
```

B$RDSD \SOURCE\READSTR.ASM

Use

After a call to B$INPP (INPUT) or B$DSKI (INPUT #), reads a value from the current input file or device into a string.

Calling Convention

```
PUSH Segment of string or descriptor
PUSH Offset of string or of variable-length string descriptor
PUSH Length of string (or 0)
CALL B$RDSD
```

No return value. May report an error by calling P$DoError.

Notes

This procedure can read into either a fixed- or variable-length string. If the destination is a fixed-length string you must specify its memory location and length. If it is a variable-length string you instead specify the segment and address of its descriptor, and use a length of 0.

If the next item does not exist (if you have read past the end of user input or past the end of a file), the result will be a null string (if it is variable-length) or padded with spaces. This routine calls B$ASSN or B$SASS to make the string assignment.

Example

Perform INPUT #1, Work$:

```
Extrn B$RDSD:Proc
Extrn B$DSKI:Proc          ;Starts file input
Extrn B$PEOS:Proc          ;Ends file input and cleans up

.DATA
DefStr Work$               ;Room for string descriptor
FileNum  dw  1             ;File to read from

.CODE
MOV  AX,Offset FileNum     ;Point to the file number
PUSH AX
CALL B$DSKI                ;Start file input
MOV  AX,Offset Work$       ;Get address of result descriptor
PUSH DS                    ;Pass its segment
PUSH AX                    ;  and its offset
SUB  AX,AX                 ;0 means a variable-length string
PUSH AX
CALL B$RDSD                ;Get one string
CALL B$PEOS                ;Then clean up
```

B$REST \SOURCE\RESET.ASM

BASIC Equivalent: RESET

Use

Closes all open files (files that were open through calls to B$OPEN or B$OOPN only).

Calling Convention

```
CALL B$REST
```

No return value. May report an error by calling P$DoError.

Notes

B$REST will *not* close files that you opened with direct calls to DOS unless you have also manipulated P.D.Q.'s list of open files, P$HandleTbl.

Example

Close all open files:

```
Extrn B$REST:Proc

.CODE
CALL B$REST              'Now all files are closed
```

B$RGHT \SOURCE\RIGHT$.ASM

BASIC Equivalent: RIGHT$

Use

Creates a new temporary string that contains the rightmost *n* characters of a given string.

Calling Convention

```
PUSH Offset of original string descriptor
PUSH Number of characters to retain
CALL B$RGHT
```

Returns the offset of a temporary string descriptor.

Notes

The maximum length of the returned string is the number of characters in the original string.

The returned string descriptor is in P.D.Q.'s temporary string pool. You should copy the string to your own string descriptor (using B$SASS) if you want to save the result.

Example

Get the rightmost 10 characters from Work$ and save the result in Saved$—Saved$ = RIGHT$(Work$, 10):

```
Extrn B$RGHT:Proc

.DATA?
EVEN
Work$   dd  1 dup (?)     ;Space for original string
Saved$  dd  1 dup (?)     ;Space for the result

.CODE
MOV  AX,Offset Work$      ;Get address of original descriptor
PUSH AX
MOV  AX,10                ;Pass the number of characters
PUSH AX
CALL B$RGHT               ;Get the right 10 characters
PUSH AX                   ;Pass along the temporary string
MOV  AX,Offset Saved$     ;Pointer to the new string
```

```
PUSH AX
CALL B$SASS                ;String is assigned
```

B$RMI4

\SOURCE\MODLONG.ASM
\SOURCE\MODLONG3.ASM

Use

Find the MOD (remainder after integer division) of two long integers: X& MOD Y&

Calling Convention

```
PUSH High Word of Divisor  (Y&)
PUSH Low Word of Divisor   (Y&)
PUSH High Word of Dividend (X&)
PUSH Low Word of Dividend  (X&)
CALL B$RMI4
```

Returns with the MOD value in DX:AX.

Notes

Note that both arguments are passed by value. This routine, like all BASIC arithmetic, treats the values as being signed. It returns the proper MOD after performing a signed long integer division (IDIV).

The alternate version of B$RMI4 in MODLONG3.ASM uses the special 32-bit registers of 386 and 486 CPUs. It is much faster than the 8088 and 80286 version, but can run only on 386 and 486 computers.

Example

Perform Z& = X& MOD Y&:

```
Extrn B$RMI4:Proc

.DATA?
X&  dd  1 dup (?)          ;Space for variables
Y&  dd  1 dup (?)
Z&  dd  1 dup (?)

.CODE
PUSH Word ptr [Y&+2]       ;High word of divisor
PUSH Word ptr [Y&]         ; and low word
PUSH Word ptr [X&+2]       ;High word of dividend
PUSH Word ptr [X&]         ;  and low word
CALL B$RMI4                ;Perform X% MOD Y%
MOV  Word ptr [Z&+2],DX    ;Store high word of result
MOV  Word ptr [Z&],AX      ;  and low word
```

B$RND0
B$RND1 \FPSOURCE\RND.ASM

BASIC Equivalent: RND

Use

Returns a single-precision random number between 0 and 1.

Calling Convention

For B$RND0:

```
CALL B$RND0
```

For B$RND1:

```
PUSH any word (a dummy argument)
CALL B$RND1 Both return with AX containing a pointer to the single-
precision result.
```

Notes

The P.D.Q. library does not support BASIC's use of RND(*n*) when *n* is less than or equal to 0, so the argument passed to B$RND1 is ignored.

This routine uses floating point operations. You must initialize the floating point emulator by calling P$HookFP before you call this procedure.

Example

Get the next random number and place it on the floating point stack:

```
Extrn B$RND0:Proc

.CODE
CALL B$RND0              ;Get the random number
MOV  BX,AX               ;Put pointer where we can use it
FLD  DWord Ptr [BX]      ;Load it into ST(0)
```


B$RNZP \FPSOURCE\RANDOMIZ.ASM

BASIC Equivalent: RANDOMIZE

Use

Sets the random number generator seed value from a single-precision number.

Calling Convention

```
PUSH Most significant word of seed number
PUSH Least significant word of seed number
CALL B$RNZP
```

No return value.

Notes

This routine uses floating point operations. You must initialize the floating point emulator by calling P$HookFP before you call this procedure.

Example

Seed the random number generator with a value of 2.0:

```
Extrn B$RNZP:Proc

.DATA
SeedNum  dd  2.0          ;Seed number

.CODE
PUSH Word ptr [SeedNum+2] ;Pass the value
PUSH Word ptr [SeedNum]   ; on the stack
CALL B$RNZP
```

B$RSET \SOURCE\RSET.ASM

BASIC Equivalent: RSET

Use

Copies and right-justifies data from a variable-length string into either a variable-length string or a fixed block of memory (normally part of a TYPE or FIELD variable).

Calling Convention

```
PUSH Offset of source string descriptor
PUSH Segment of destination or of destination descriptor
PUSH Offset of destination or of destination descriptor
PUSH Length of the destination.
```

No return value.

Notes

Use a length of zero for the destination when assigning to a variable-length string.

This routine automatically pads the destination with spaces if it is longer than the source. If the source is longer, it is truncated to fit into the destination.

Example

Perform RSET A$ = B$ when both are variable-length strings:

```
Extrn B$RSET:Proc

.DATA?
EVEN
A$  dd  1 dup (?)          ;Space for string descriptors
B$  dd  1 dup (?)

.CODE
MOV  AX,Offset B$          ;Point to source descriptor
PUSH AX
MOV  AX,Offset A$          ;Point to destination descriptor
PUSH DS                    ;Push its segment
PUSH AX                    ; and its offset
SUB  AX,AX                 ;AX = 0: destination is a
PUSH AX                    ;  variable-length string
CALL B$RSET
```

B$RTRM \SOURCE\RTRIM$.ASM

BASIC Equivalent: RTRIM$

Use

Copies and removes trailing CHR$(32) spaces and CHR$(0) nulls from the right end of a string.

Calling Convention

```
PUSH Offset of string descriptor
CALL B$RTRM
```

Returns with the offset of the result string descriptor in AX.

Notes

The result is returned in a temporary string descriptor. You must copy the temporary string to a permanent string if you want to process it further.

Example

Perform A$ = RTRIM$(A$):

```
Extrn B$RTRM:Proc
Extrn B$SASS:Proc          ;For string assignment
```

```
 .DATA?
EVEN
A$  dd  1 dup (?)          ;Room for string descriptor

.CODE
MOV  AX,Offset A$          ;Create pointer to A$
PUSH AX
CALL B$RTRM                ;Strip trailing spaces and nulls
PUSH AX                    ;Pass pointer to result descriptor
MOV  AX,Offset A$          ;Where we want the result
PUSH AX
CALL B$SASS                ;The result is now in A$
```

B$SACT \SOURCE\CONCAT2$.ASM

Use

Concatenates two strings and stores the result in a specified string location (C$ = A$ + B$).

Calling Convention

```
PUSH Offset destination string descriptor
PUSH Offset of first source string descriptor
PUSH Offset of second source string descriptor
CALL B$SACT
```

No return value.

Notes

This routine simply calls B$SCAT and then B$SASS, just as the example code in the description of B$SCAT does.

If there is insufficient space in the string pool for the result, no error will be generated but the result string will be truncated to the amount of space available.

Example

Implement C$ = A$ + B$:

```
Extrn B$SACT:Proc

.DATA?
EVEN
A$  dd  1 dup(?)           ;Space for three
B$  dd  1 dup (?)          ;  string
C$  dd  1 dup (?)          ;  descriptors

.CODE
MOV  AX,Offset C$          ;Pass destination first
PUSH AX
```

```
MOV  AX,Offset A$          ;Then pass the source
PUSH AX                    ;  strings in any order
MOV  AX,Offset B$
PUSH AX
CALL B$SACT                ;Now C$ = A$ + B$
```

B$SASS \SOURCE\ASSIGN$.ASM

Synonyms: B$SAS1
B$SASF

■ Use

Create a new variable-length string and assign it a value.

■ Calling Convention

```
PUSH Offset of source string descriptor
PUSH Offset of destination string descriptor
CALL B$SASS
```

No return value.

■ Notes

This routine takes care of all the dirty work of creating a string; it puts the correct values in the string descriptor, then finds space in the string pool for the back pointer and string text. See the section *Using Strings* for information about string descriptors, back-pointers, string compaction, and other related topics.

This is the routine that BASIC calls for assignments like **A$ = B$, C$ = "P.D.Q."**, and **D$ = ""**, as well as for creating and storing the result of string expressions like **E$ = LEFT$(F$, 1)**.

Do *not* use B$SASS to assign a string constant that was defined using the DefStr macro.

Call B$STDL to delete a string from memory when you want to release its space in the string pool.

Example

Implement A$ = B$:

```
Extrn B$SASS:Proc

.DATA?
EVEN
A$  dd  1 dup(?)          ;Space for two string descriptors
```

```
B$  dd  1 dup (?)

.CODE
MOV  AX,Offset B$          ;Assumes B$ has been initialized
PUSH AX                    ;Pass pointer to B$ descriptor
MOV  AX,Offset A$          ;Pass pointer to A$ descriptor
PUSH AX
CALL B$SASS                ;Get the work done
```

B$SCAT \SOURCE\CONCAT$.ASM

Synonyms: B$SCT1

Use

Concatenates two strings (A$ + B$).

Calling Convention

```
PUSH Offset of the first string's descriptor
PUSH Offset of the second string's descriptor
CALL B$SCAT
```

Offset of result descriptor returned in AX.

Notes

The result string will be truncated if there aren't enough bytes left in the string pool for the entire concatenated string.

Neither source string is changed and the result is stored in a temporary string. If you want to use it for future manipulation, call B$SASS to copy it to a new string. Use B$SACT if you want to concatenate and save the result all in one operation.

Example

Implement C$ = A$ + B$:

```
Extrn B$SCAT:Proc

.DATA
A$  dd  1 dup (?)          ;Space for the
B$  dd  1 dup (?)          ;   string
C$  dd  1 dup (?)          ;   descriptors

.CODE
MOV  AX,Offset A$          ;Pass A$'s descriptor address
PUSH AX
MOV  AX,Offset B$          ;And the same for B$
PUSH AX
CALL B$SCAT                ;Put them together
```

```
PUSH AX                 ;Pass result's descriptor address
MOV  AX,Offset C$       ;And the location for the result
PUSH AX
CALL B$SASS             ;Store the concatenated string in C$
```

B$SCLS \SOURCE\CLS.ASM

BASIC Equivalent: CLS

- **Use**

  Clear the text screen using the current color, and move the cursor to the top left corner.

- **Calling Convention**

```
CALL B$SCLS
```

  No return value.

- **Notes**

  The color used for CLS can be set with by calling B$COLR, or by directly setting the color value in the external data byte called P$Color.

  This routine will work properly regardless of the number of text rows on the display.

Example

Clear the screen to the current colors:

```
Extrn B$SCLS:Proc

.CODE
CALL B$SCLS
```

B$SCMP \SOURCE\COMPARE$.ASM

- **Use**

  Compares two strings and returns with the flags set appropriately.

- **Calling Convention**

```
PUSH Offset of string 1's descriptor
PUSH Offset of string 2's descriptor
CALL B$SCMP
```

  Result returned in the flags register.

Notes

Strings are compared until either a difference is found, or the end of one string is reached. If two strings are identical through to the end of one of them, then the longer string is considered to be "greater".

After calling B$SCMP you will use an unsigned conditional jump rather than one that is signed, because ASCII characters are considered to have a range of 0 through 255. That is, you will use Ja or Jbe and so forth, and not Jge or Jl.

Example

Implement IF A$ > B$ THEN do something:

```
Extrn B$SCMP:Proc

.DATA
A$  dd  1 dup (?)       ;Space for A$'s descriptor
B$  dd  1 dup (?)       ;Space for B$'s descriptor

.CODE
MOV  AX,Offset A$       ;Get address of string 1
PUSH AX
MOV  AX,Offset B$       ;  and address of string 2
PUSH AX
CALL B$SCMP
JNA  @F                 ;Go if A$ <= B$
 ;do something here
@@:                     ;Common code again
```

B$SDAT \SOURCE\DATE$$.ASM

BASIC Equivalent: DATE$ statement

Use

Sets the system date from a date in a string.

Calling Convention

```
PUSH Offset of string descriptor
CALL B$SDAT
```

No return value.

Notes

The string can be in one of two forms:

mm-dd-yy (you can use any delimiter instead of dashes)

or

mm-dd-yyyy

The second format is the only way to set a date in the 21st century.

Example

```
Extrn B$SDAT:Proc

.DATA?
EVEN
Date  dd  1 dup (?)       ;Space for string descriptor

.CODE
MOV  AX,Offset Date       ;Pass a pointer to the
PUSH AX                   ;  string descriptor
CALL B$SDAT               ;System date is now set
```

B$SENV \SOURCE\ENVIRON3.ASM

BASIC Equivalent: ENVIRON statement

Use

Modifies or creates an entry in the environment list.

Calling Convention

```
PUSH Offset of string descriptor of new variable and value
CALL B$SENV
```

No return value.

Notes

You can choose what copy of the environment to work with by using EnvOption. See *The Environment* in the main P.D.Q. manual for more information about working with the environment.

Example

Assuming that A$ contains a valid string in the form of "variable=value," set that variable into the current environment:

```
Extrn B$SENV:Proc

.DATA?
EVEN
A$  dd  1 dup (?)         ;Space for A$'s descriptor

.CODE
```

```
MOV  AX,Offset A$          ;Get address of string
PUSH AX
CALL B$SENV                ;Set it into the environment
```

B$SERR \SOURCE\ERROR.ASM

BASIC Equivalent: ERROR

Use

Simulates a BASIC error.

Calling Convention

```
PUSH Error value
CALL B$SERR
```

No return value.

Notes

The error value should be a value between 1 and 255. The high byte of the value is ignored. Use 0 to clear ERR.

This routine translates the error code it receives to an internal form, stores the error in the public external data byte P$PDQErr, and then calls P$DoError to handle possible ON ERROR calls. See the latter routine for information about error handling.

Example

Perform ERROR 5 (illegal function call):

```
Extrn B$SERR:Proc

.CODE
MOV  AX,5                  ;Get the error code
PUSH AX                    ;  and pass it on
CALL B$SERR                ;Force Error 5
```


B$SGN4
B$SGN8 \FPSOURCE\B$SGN.ASM

BASIC Equivalent: SGN

Use

Determines the sign of the real number at ST(0) on the floating point stack.

Calling Convention

```
CALL B$SGN8
```

ST(0) is replaced by -1.0, 0.0, or 1.0 and the flags are set appropriately for a conditional jump.

Notes

B$SGN4 and B$SGN8 are two names for the same routine. You may call either one. Although the names suggest that there is a different way of finding the sign for a single and double-precision number, once a number is on the floating point stack, it has no set precision.

This routine uses floating point operations. You must initialize the floating point emulator by calling P$HookFP before you call this procedure.

Example

Jump if ST(0) is zero:

```
Extrn B$SGN8

.CODE
CALL B$SGN8             ;Make the comparison
JZ   ItWasZero          ;Go if it was 0
```

B$SICT \SOURCE\IOCTL.ASM

BASIC Equivalent: IOCTL

Use

Send a control string to a device driver.

Calling Convention

```
PUSH  BASIC file number of the device
PUSH  Offset of the control string descriptor
CALL  B$SICT
```

No return value. May report an error by calling P$DoError.

Notes

This routine uses DOS service 44h, subfunction 3, to send the control string to the device. The only difference between calling B$SICT and sending the control string yourself is that this routine will translate the BASIC file number into a DOS handle for you and fill the registers correctly for the DOS service call.

Example

Send Control$ to the device opened as file number 3:

```
Extrn B$SICT:Proc

.DATA?
EVEN
Control$  dd  1 dup (?)   ;Control string descriptor

.CODE
MOV  AX,3                 ;Get BASIC file number
PUSH AX
MOV  AX,Offset Control$   ;Get descriptor address
PUSH AX
CALL B$SICT
```

B$SLEP \SOURCE\SLEEP.ASM

BASIC Equivalent: SLEEP

Use

Suspend execution for a given number of seconds.

Calling Convention

```
PUSH Any word value (a dummy word)
PUSH Number of seconds
CALL B$SLEP
```

No return value.

Notes

If the number of seconds is given as zero then B$SLEP waits indefinitely until a key is pressed.

The system time ticks 18.2 times per second. This routine multiplies the number of seconds you specify by 18, so there will be a slight inaccuracy for long pauses.

The user can exit prematurely from the pause by pressing any key. This routine begins by clearing the keyboard buffer, so any keystrokes that are waiting when B$SLEP is called are lost. The user's key is *not* retrieved from the buffer. It will be the first key read with the next keyboard input.

Example

Wait for 1 minute:

```
Extrn B$SLEP:Proc

.CODE
MOV  AX,60              ;A 60-second sleep
PUSH AX                 ;the dummy word
PUSH AX                 ;The sleep length
CALL B$SLEP
```

B$SMID \SOURCE\MID$$.ASM

BASIC Equivalent: MID$ statement

Use

Replace a portion of a string with another string:

```
MID$(Destination$, StartPos, NumChars) = Source$
```

The destination may be either a fixed-length string or the descriptor of a variable-length string.

Calling Convention

```
PUSH Segment of destination string or descriptor
PUSH Offset of destination string or descriptor
PUSH Length of destination string (or 0)
PUSH Offset of descriptor of source string
PUSH Number of characters
PUSH Start position
CALL B$SMID
```

No return value.

Notes

If the string is a conventional (not fixed-length) string use a length of zero.

This routine may create (and delete) a temporary string. It will also clip the Length parameter to the length of the source string if necessary. If you want to insert the entire source string into the destination string and not worry about how long the source really is, simply specify a Length of 7FFFh. Of course, only as many characters as will fit are assigned.

B$SMID calls several other routines including B$FMID and B$LSET.

In many cases, you will find that it is easier to overwrite part of a string directly in assembly-language rather than set up and call B$SMID.

Example

Perform MID$(B$, 5, 7) = A$:

```
Extrn B$SMID:Proc

.DATA?
EVEN
A$  dd  1 dup (?)          ;Space for string descriptors
B$  dd  1 dup (?)

.CODE
PUSH DS                    ;Segment of destination descriptor
MOV  AX,Offset B$          ;Get offset of destination descriptor
PUSH AX
SUB  AX,AX                 ;0 means B$ is variable-length
PUSH AX
MOV  AX,Offset A$          ;Get offset of source descriptor
PUSH AX
MOV  AX,7                  ;Get bytes to insert
PUSH AX
MOV  AX,5                  ;Get start position in B$
PUSH AX
CALL B$SMID                ;Get the work done
```

B$SOND \FPSOURCE\SOUND.ASM

BASIC Equivalent: SOUND

Use

Makes a tone of a specified frequency and duration through the computer's speaker.

Calling Convention

```
PUSH Frequency of tone as an integer
PUSH Duration of tone as a single-precision number
CALL B$SOND
```

No return value.

Notes

Unless you have a particular need to pass the duration as a single-precision number, it is much faster to call PDQSound directly. This procedure converts the duration to an integer and calls PDQSound for you.

This routine uses floating point operations. You must initialize the floating point emulator by calling P$HookFP before you call this procedure.

Example

Perform SOUND 440, 20:

```
Extrn B$SOND:Proc

.DATA
Frequency  dw  440              ;Frequency of tone
Duration   dd  20.0             ;And its duration

.CODE
PUSH [Frequency]                ;Pass on the frequency
PUSH Word Ptr [Duration + 2]    ;And the duration
PUSH Word Ptr [Duration]
CALL B$SOND                     ;Play the note
```

B$SPAC \SOURCE\SPACE$.ASM

BASIC Equivalent: SPACE$

Use

Create a string composed of a specific number of CHR$(32) space characters.

Calling Convention

```
PUSH Number of spaces
CALL B$SPAC
```

Returns the offset of a string descriptor in AX.

Notes

This routine merely translates the arguments into an appropriate B$STRI call (BASIC's STRING$ function).

The returned string descriptor is in P.D.Q.'s temporary string pool. If you want to work with the string further, use B$SASS to assign the results to your own string descriptor.

Example

Perform A$ = SPACE$(20):

```
Extrn B$SPAC:Proc
Extrn B$SASS:Proc          ;For string assignment

.DATA?
EVEN
A$  dd  1 dup (?)          ;Space for string descriptor

.CODE
MOV  AX,20                 ;Length of desired string
PUSH AX
CALL B$SPAC                ;Create the string
```

```
PUSH AX                 ;Pass the result's address
MOV  AX,Offset A$       ;Now pass our string descriptor
PUSH AX
CALL B$SASS             ;Assign result to A$
```

B$SPLY \SOURCE\PLAY.ASM

BASIC Equivalent: PLAY

Use

Play tones through the speaker based on a string of note values.

Calling Convention

```
PUSH Offset of descriptor of string containing notes
CALL B$SPLY
```

No return value.

Notes

This is a full implementation of the BASIC PLAY statement. See your BASIC or QuickBASIC manual for a description of how to code the tune string.

Example

Play the notes in Tune$:

```
Extrn B$SPLY:Proc

.DATA?
EVEN
DefStr Tunes, "o3 l8 edcdeee"          ;Mary had a little lamb

.CODE
MOV  AX,Offset Tune$                   ;Get address of string
PUSH AX                                ;Pass it on
CALL B$SPLY                            ;And let it play
```

B$SSEK \SOURCE\SEEK.ASM

BASIC Equivalent: SEEK statement

Use

Move the file pointer to a specific file location for the next read or write.

Calling Convention

```
PUSH BASIC file number
PUSH High word of desired location
PUSH Low word of desired location
CALL B$SSEK
```

No return value. May report an error by calling P$DoError.

Notes

If the file has been opened for random access, this routine multiplies the location you specify by the file's record length. For all other files, the location you specify is interpreted as a byte position in the file. In both cases, your location number is 1-based (as opposed to DOS's 0-based system of numbering bytes and file locations).

This routine calls B$MUI4 to perform the multiplication.

Example

Move the pointer to record 5 of file #1, which was opened as a random access file:

```
Extrn B$SSEK:Proc

.CODE
MOV  AX,1               ;Get the file number
PUSH AX
MOV  AX,5               ;Record location
CWD                     ;  now in DX:AX
PUSH DX                 ;Push the high word
PUSH AX                 ;  and the low word
CALL B$SSEK             ;Move the file pointer.
```


B$SSHL \SOURCE\SHELL.ASM

BASIC Equivalent: SHELL

Use

Runs COMMAND.COM (or any other command interpreter pointed to by the environment variable COMSPEC) and has it execute a .COM, .EXE, or .BAT program or DOS command.

Calling Convention

```
PUSH Offset of descriptor of command string
CALL B$SSHL
```

No return value. May report an error by calling P$DoError.

Notes

If the command string has a length of zero, B$SSHL shells to COMMAND.COM and waits until the user enters the DOS EXIT command. Otherwise, COMMAND.COM is told to run your command and return immediately.

If COMSPEC is not properly set, this command will fail and return immediately.

Example

Redirect a directory listing to the file DIRFILE in the default directory:

```
Extrn B$SSHL:Proc

.DATA
DefStr Cmd$,"DIR > DIRFILE"

.CODE
MOV  AX,Offset Cmd$       ;Point to string descriptor
CALL B$SSHL               ;  and do it
```

B$STDL \SOURCE\STRDELET.ASM

Synonym: StringRelease

Use

Deletes a string.

Calling Convention

```
PUSH Offset of string's descriptor
CALL B$STDL
```

No return value.

Notes

You should call this routine to delete temporary strings that you no longer need. It is also called automatically during string assignments to delete the old contents of the string being assigned.

This routine works by setting the length of the descriptor to zero, and then changing the string's back pointer to show that the data has been abandoned.

Do not *use* B$STDL to erase a string constant that was defined using the DefStr macro.

Example

Delete A$ so that its data space can be reused in the string pool. This is the equivalent of A$ = "" but with slightly less code since two arguments are not passed:

```
Extrn B$STDL:Proc

.DATA?
EVEN
A$  dd  1 dup(?)          ;Room for a string descriptor

.CODE
MOV  AX,Offset A$         ;Get pointer to descriptor
PUSH AX
CALL B$STDL
```

B$STI2

\SOURCE\STR$.ASM
\SOURCE_STR$.ASM

BASIC Equivalent: STR$ (X%)

Use

Converts an integer value into its ASCII string representation.

Calling Convention

```
PUSH Integer word to convert
CALL B$STI2
```

Returns offset of result string descriptor in AX.

Notes

This routine simply converts the integer value into a long integer and then calls B$STI4 to do the conversion. You can save a few cycles by making the conversion yourself and calling B$STI4.

The returned string is in P.D.Q.'s temporary string space. You must copy it to your own string (using B$SASS, for example) if you want to work with it further.

The version in _STR$.ASM is identical except it suppresses the leading space that is normally added before a positive value.

Example

Perform A$ = STR$(B%):

```
Extrn B$STI2:Proc
Extrn B$SASS:Proc         ;For string assignment
```

```
.DATA?
EVEN
A$  dd  1 dup(?)          ;Space for string descriptor
B   dw  1 dup (?)         ;Space for integer value

.CODE
MOV  AX,[B]               ;Get integer value
PUSH AX                   ;Send it on
CALL B$STI2               ;Convert it to an ASCII string
PUSH AX                   ;Pass on the result string
MOV  AX,Offset A$         ;Get pointer to our string
PUSH AX                   ;Pass it on also
CALL B$SASS               ;Copy result to our string
```

B$STI4

\SOURCE\STR$.ASM
\SOURCE_STR$.ASM

BASIC Equivalent: STR$ (X&)

Use

Converts a long integer value into its ASCII string representation.

Calling Convention

```
PUSH High word of value to convert
PUSH Low word of value to convert
CALL B$STI4
```

Returns offset of the result string descriptor in AX.

Notes

The returned string is in P.D.Q.'s temporary string space. You must copy it to your own string (using B$SASS, for example) if you want to work with it further.

The version in _STR$.ASM is identical except it suppresses the leading space that is normally added before a positive value.

Example

Perform A$ = STR$(B&):

```
Extrn B$STI4:Proc
Extrn B$SASS:Proc         ;For string assignment

.DATA?
EVEN
A$  dd  1 dup (?)         ;Space for string descriptor
B   dd  1 dup (?)         ;Space for long integer value
```

```
.CODE
LES  AX,[B]              ;Get long integer value
PUSH ES                  ;Pass the high word
PUSH AX                  ; and then the low word
CALL B$STI4              ;Convert it to an ASCII string
PUSH AX                  ;Pass on the result string
MOV  AX,Offset A$        ;Get pointer to our string
PUSH AX                  ;Pass it on also
CALL B$SASS              ;Copy result to our string
```

B$STIM SOURCE\TIME$$.ASM

BASIC Equivalent: TIME$ statement

Use

Sets the system clock to a specified time.

Calling Convention

```
PUSH Offset of descriptor for string holding the new time
CALL B$STIM
```

No return value.

Notes

This routine does a minimal amount of error checking, so it is your responsibility to make sure that the string contains a valid time, properly formatted.

Example

Set the system time to 9:00 am:

```
Extrn B$STIM

.DATA
DefStr Time$,"09:00:00"   ;Time to set

.CODE
MOV  AX,Offset Time$      ;Point to time descriptor
PUSH AX
CALL B$STIM               ;Set the system time
```

B$STR4 \FPSOURCE\B$STR4.ASM

BASIC Equivalent: STR$ (X!)

Use

Converts a single-precision number into its ASCII representation.

Calling Convention

```
PUSH High word of the number (bytes 2 and 3)
PUSH Low word of the number (bytes 0 and 1)
CALL B$STR4
```

Returns offset of a string descriptor in AX.

Notes

The returned string is in P.D.Q.'s temporary string space. You should copy it to your own string with B$SASS if you want to work with it further.

This routine uses floating point operations. You must initialize the floating point emulator by calling P$HookFP before you call this procedure.

Example

Perform Work$ = STR$(X!):

```
Extrn B$STR4:Proc

.DATA?
EVEN
Work$  dd  1 dup (?)      ;Room for Work$ descriptor
X      dd  1 dup (?)      ;Room for X!

.CODE
PUSH Word Ptr [X + 2]     ;Pass on the value
PUSH Word Ptr [X]
CALL B$STR4               ;Convert to a string
PUSH AX                   ;Pass on string pointer
MOV  AX,Offset Work$      ;Get pointer to our descriptor
PUSH AX                   ;Pass it on
CALL B$SASS               ;Store result in Work$
```

B$STR8 \FPSOURCE\B$STR8.ASM

BASIC Equivalent: STR$ (X#)

Use

Converts a double-precision number into its ASCII representation.

Calling Convention

```
PUSH High word of the number (bytes 6 and 7)
PUSH Next most significant word (bytes 4 and 5)
PUSH Next most significant word (bytes 2 and 3)
PUSH Least significant word of the number (bytes 0 and 1)
CALL B$STR8
```

Returns offset of string descriptor in AX.

Notes

The returned string is in P.D.Q.'s temporary string space. You should copy it to your own string (with B$SASS) if you want to work with it further.

This routine uses floating point operations. You must initialize the floating point emulator by calling P$HookFP before you call this procedure.

Example

Perform Work$ = STR$(X#):

```
Extrn B$STR8:Proc

.DATA?
EVEN
Work$  dd  1 dup (?)      ;Room for Work$ descriptor
X      dq  1 dup (?)      ;Room for X#

.CODE
MOV  BX,Offset X          ;Get pointer to number
PUSH Word Ptr [BX+6]      ;Push in onto the stack
PUSH Word Ptr [BX+4]
PUSH Word Ptr [BX+2]
PUSH Word Ptr [BX]
CALL B$STR8               ;Convert to a string
PUSH AX                   ;Pass on string pointer
MOV  AX,Offset Work$      ;Get pointer to our descriptor
PUSH AX                   ;Pass it on
CALL B$SASS               ;Store result in Work$
```

B$STRI \SOURCE\STRING$.ASM

BASIC Equivalent: STRING$ (X, Y)

Use

Creates a string composed of *n* copies of a given ASCII value.

Calling Convention

```
PUSH Number of characters in final string
PUSH ASCII value for repeated character
CALL B$STRI
```

Returns offset of string descriptor of result in AX.

Notes

Only the low byte of the character value is examined, and whatever is in AH is ignored.

The returned string is in P.D.Q.'s temporary string pool. You should copy it to your own string (using B$SASS) if you want to work with it further.

Example

Perform A$ = STRING$(80, 65):

```
Extrn B$STRI:Proc
Extrn B$SASS:Proc          ;For string assignment

.DATA?
EVEN
A$  dd  1 dup (?)          ;Room for string descriptor

.CODE
MOV  AX,80                 ;Length of final string
PUSH AX
MOV  AL,"A"                ;Get ASCII value of character
PUSH AX
CALL B$STRI                ;Create the string
PUSH AX                    ;Pass on the temporary string
MOV  AX,Offset A$          ; and a pointer to result string
PUSH AX
CALL B$SASS                ;Copy result to A$
```

B$STRS \SOURCE\STRING$$.ASM

BASIC Equivalent: STRING$ (X, Y$)

Use

Creates a string made up of *n* copies of the first letter of the specified string argument.

Calling Convention

```
PUSH Number of characters in final string
PUSH Offset of descriptor of string argument
CALL B$STRS
```

Returns the offset of the result string descriptor in AX.

Notes

You can save a little time by calling B$STRI instead of this routine, which simply translates the second argument into an integer and then calls B$STRI to do the work.

Regardless of the string argument's length, only the first character is considered. If the string argument is null the result is undefined.

The returned string is in P.D.Q.'s temporary string pool. You should copy it to your own string (using B$SASS) if you want to work with it further.

Example

Perform A$ = STRING$(50, "*"):

```
Extrn B$STRS:Proc
Extrn B$SASS:Proc          ;For string assignment
Extrn P$MakeTemp:Proc      ;To create temporary string

.DATA?
EVEN
A$  dd  1 dup (?)          ;Room for string descriptor

CODE
MOV  CX,1                  ;Length of temp. string
CALL P$MakeTemp            ;Make a temporary string
MOV  AL,"*"                ;That's the character to store
STOSB                      ;Put it in the string
MOV  AX,50                 ;Length of final string
PUSH AX                    ;Pass it on
PUSH BX                    ;Our temporary descriptor
CALL B$STRS                ;Create the new string
PUSH AX                    ;Pass on the result
MOV  AX,Offset A$          ;Pointer to final descriptor
```

```
PUSH AX
CALL B$SASS                ;Result is now in A$
```

B$SWPN \SOURCE\SWAPTYPE.ASM

BASIC Equivalent: SWAP for fixed - length strings and TYPE variables.

Use

Swaps any two non-overlapping blocks of memory as long as they have the same length and are 65,535 bytes long or less.

Calling Convention

```
PUSH Segment of first memory block
PUSH Offset of first memory block
PUSH Length of first memory block
PUSH Segment of second memory block
PUSH Offset of second memory block
PUSH Length of second memory block
```

No return value.

Notes

The two length values must be identical or B$SWPN will return without taking any action.

Example

SWAP a text screen of 4000 bytes at 0B800:0h with a memory buffer.

```
Extrn B$SWPN:Proc

.DATA?
VidBufAddr  dd  1 dup (?) ;Room for buffer address

.CODE
LES  AX,VidBufAddr        ;Get video buffer address
PUSH ES                   ;Pass the buffer segment
PUSH AX                   ;  and its offset
MOV  CX,4000              ;Bytes to swap
PUSH CX
MOV  AX,0B800h            ;Segment of video screen
PUSH AX
SUB  AX,AX                ;AX = 0; offset of video screen
PUSH AX
PUSH CX                   ;Length of segment 2
CALL B$SWPN
```

B$SWP2
B$SWP4
B$SWP8

\SOURCE\SWAPNUMS.ASM

BASIC Equivalent: SWAP X, Y

Use

Swaps two 2-byte (B$SWP2), 4-byte (B$SWP4), or 8-byte (B$SWP8) blocks anywhere in memory. Normally, the 2-byte version is used to swap integers, the 4-byte version for long integers and single-precision numbers, and the 8-byte version for double-precision numbers.

Calling Convention

```
PUSH Segment of first memory block
PUSH Offset of first memory block
PUSH Segment of second memory block
PUSH Offset of second memory block
CALL B$SWP2, B$SWP4, or B$SWP8
```

No return value.

Notes

These routines begin by placing a 2, 4, or 8 in AL (to show how many bytes should be swapped) and then call P$SwapNums. The latter routine rearranges the arguments on the stack and calls B$SWPN to do the real work. You can save time by either performing the swap yourself using Xchg or by calling either of the latter two routines directly.

Example

Perform SWAP A%, B%:

```
.DATA?
A%  dw  1 dup (?)
B%  dw  1 dup (?)

.CODE
PUSH DS                 ;Segment of first integer
MOV  AX,Offset A%       ;Offset of first integer
PUSH AX
PUSH DS                 ;Segment of second integer
MOV  AX,Offset B%       ;Offset of second integer
PUSH AX
CALL B$SWP2             ;Swap their values
```

B$SWSD \SOURCE\SWAP$.ASM

BASIC Equivalent: SWAP X$, Y$

Use

Exchanges the data associated with two string descriptors.

Calling Convention

```
MOV  SI,Offset of descriptor 1
MOV  DI,Offset of descriptor 2
CALL B$SWSD
```

No return value.

Notes

This routine switches the descriptors themselves and also fixes the back pointers in each data string. Note the unusual calling sequence whereby the string descriptor addresses are passed in SI and DI instead of on the stack.

Example

Perform SWAP A$, B$:

```
.DATA?
EVEN
A$  dd  1 dup (?)          ;Room for two string descriptors
B$  dd  1 dup (?)

.CODE
MOV  SI,Offset A$          ;Get first pointer
MOV  DI,Offset B$          ;  and second pointer
CALL B$SWSD                ;Swap the strings
```

B$TIMR \FPSOURCE\TIMER.ASM

BASIC Equivalent: TIMER function

Use

Returns the number of seconds since midnight as a single-precision number.

Calling Convention

```
CALL B$TIMR
```

Returns with AX pointing to a single-precision result.

Notes

It is generally much faster to use PDQTimer instead of this routine, in order to avoid the overhead of floating point operations.

This routine uses floating point operations. You must initialize the floating point emulator by calling P$HookFP before you call this procedure.

Example

Perform Start! = TIMER:

```
Extrn B$TIMR:Proc

.DATA?
Start  dd  1 dup (?)            ;Place for single-precision result

.CODE
CALL B$TIMR                     ;Get number of seconds
MOV  BX,AX                      ;For ease of addressing
LES  AX,[BX]                    ;Get whole number at one time
MOV  Word Ptr [Start],AX        ;Copy the results to our space
MOV  Word Ptr [Start+2],ES
```

B$UBND \SOURCE\UBOUND.ASM

BASIC Equivalent: UBOUND

Use

Return the highest available subscript for a dimension of an array.

Calling Convention

```
PUSH Offset of array descriptor
PUSH Dimension number
CALL B$UBND
```

Returns highest subscript in AX.

Notes

It may be faster to calculate the UBOUND value directly from information in the array descriptor. See *Using Arrays* in the preceding section for more information about the format of the descriptor.

Example

Find UBOUND(Array, 2):

```
Extrn B$UBND:Proc

.DATA?
Array  db  20 dup (?)     ;Room for descriptor for 2-dimensional array

.CODE
MOV  AX,Offset Array      ;Get pointer to array descriptor
PUSH AX
MOV  AX,2                 ;Dimension we want UBOUND of
PUSH AX
CALL B$UBND               ;Now the result is in AX
```

B$UCAS \SOURCE\UCASE$.ASM

BASIC Equivalent: UCASE$ function

Use

Create a copy of a string with all alphabetic characters in upper case.

Calling Convention

```
PUSH Offset of string descriptor
CALL B$UCAS
```

Returns with pointer to result string descriptor in AX.

Notes

The result string descriptor is a P.D.Q. temporary string. If you want to process the string further, you should copy it to your own string variable by using B$SASS.

Example

Perform A$ = UCASE$(A$):

```
Extrn B$UCAS:Proc
Extrn B$SASS:Proc         ;For string assignment

.DATA?
EVEN
A$  dd  1 dup (?)         ;Space for string descriptor

.CODE
MOV  AX,Offset A$         ;Get pointer to our string
CALL B$UCAS               ;Convert it to upper case
PUSH AX                   ;Pass along the result
MOV  AX,Offset A$         ;  and our string
PUSH AX
CALL B$SASS               ;Copy result to our string
```

B$WIDT \SOURCE\WIDTH.ASM

BASIC Equivalent: WIDTH (screen).

Use

Sets the number of columns and rows on a current text screen.

Calling Convention

```
PUSH Number of columns
PUSH Number of rows
CALL B$WIDT
```

No return value. May report an error by calling P$DoError.

Notes

B$WIDT supports text screen resolutions of 80x25, 40x25, 80x43, 40x43, 80x50, and 40x50 as long as the resolution is also supported by your hardware. 43-row screens can be used on EGA or VGA systems; 50-row screens on VGA systems.

Note that you must pass values for both the row and column. Use a value of -1 if you want to preserve the current setting for either value.

Example

Perform WIDTH 40:

```
Extrn B$WIDT:Proc

.CODE
MOV  AX,40              ;Number of columns
PUSH AX
MOV  AX,-1              ;Use current setting for rows
PUSH AX
CALL B$WIDT
```


BIOSInkey \SOURCE\BIOSINKY.ASM

P.D.Q. Equivalent: BIOSInkey

Use

This procedure retrieves a keystroke by calling the BIOS keyboard services directly. It does not support redirection, but it may be used in a TSR since it doesn't use any DOS services.

On return, AX = 0 if no key is waiting. If a keystroke is waiting, it is retrieved (and removed) from the keystroke buffer. If AH = 0, then the ASCII code of the keystroke is in AL. If AH = -1, then -Scan Code is in AL.

Calling Convention

```
CALL BIOSInkey
```

Returns the key value in AX if a key was waiting, or 0 in AX if no key was waiting in the keyboard type-ahead buffer.

Notes

To retrieve the keystroke, you can follow this algorithm:

```
OR   AX,AX              ;Test if any value
JZ   NO_KEY             ;Nothing was waiting for us
OR   AH,AH              ;Test type of keystroke
JZ   ASCII_KEY          ;Go if ASCII value is in AL
NEG  AX                 ;Now scan code (F-key, etc.)
EXT_KEY: (code here)    ;  is in AL
```

This routine calls Int 16H, services 1 and 0. It does not support F11, F12, and other special keys of the enhanced keyboard.

Example

```
Extrn BIOSInkey:Proc

.CODE
CALL BIOSInkey
```

BIOSInput \SOURCE\BIOSINPT.ASM

P.D.Q. Equivalent: BIOSInput

Use

BIOSInput is a simplified editing routine. It recognizes the left and right arrow keys for movement through an edit string, but does not support the Ins, Del, Home, or End keys. It gets its input from BIOSInkey, so it can be used in TSR programs.

BIOSInput supports color editing. It receives a string with the default response, and does not allow editing beyond the end of the string.

Calling Convention

```
PUSH Offset of response string descriptor
PUSH Offset of edit color
CALL BIOSInput
```

No return value.

Notes

BIOSInput calls several other routines to find the current cursor location, turn the cursor on, get keyboard input, and print the results.

Example

Implement CALL BIOSInput(Work$, EditColor):

```
Extrn BIOSInput:Proc

.DATA
EVEN
Work$      dd  ?          ;Work$ string descriptor
EditColor  dw  ?          ;Storage for the editing color

.CODE
MOV  AX,Offset Work$      ;Pass address of edit string
PUSH AX
MOV  AX,Offset EditColor  ;and address of the color
PUSH AX
CALL BIOSInput
```

BIOSInput2 \SOURCE\BIOSINP2.ASM

P.D.Q. Equivalent: BIOSInput2

Use

BIOSInput2 is an enhanced version of BIOSInput that adds support for the Home, End, Ins, and Del keys, and also returns the last key pressed (Enter or Escape) in AX.

Calling Convention

```
PUSH Offset of response string descriptor
PUSH Value of Row
PUSH Value of Column
PUSH Value of Edit color
CALL BIOSInput2
```

The ASCII value of the last key pressed is returned in AX.

Notes

BIOSInput2 calls several other routines to find the current cursor location, turn the cursor on, get keyboard input, and print the results.

Because BIOSInput2 calls upon BIOSInkey to read the keyboard, the last key pressed is coded using that routine's method. That is, if a regular key was pressed, its ASCII value will be in AX. And if an extended key was pressed and the modification described below is made, AX will hold a negative version of the extended key code.

Comments in the assembler source code (at the label *TryEnd:*) show how to modify BIOSInput2 to exit when an unrecognized extended key is pressed. This would, for example, let you exit if the user presses F1 or Alt-T. You could also change the code near line 208 to exit if a control key is pressed instead of ignoring the key. This would be needed to recognize the Tab and Ctrl-C keys.

Example

Implement CALL BIOSInput2(Work$, Row, Column, EditColor):

```
Extrn BIOSInput2:Proc

.DATA
EVEN
Work$      dd ?          ;Work$ string descriptor
EditColor  dw  ?         ;Storage for the editing color

.CODE
MOV  AX,Offset Work$     ;Pass address of edit string
PUSH AX
MOV  AX,1                ;edit on row 1
PUSH AX
MOV  AL,10               ;at column 10
PUSH AX                  ;  (AH is known zero)
PUSH EditColor           ;pass color by value
CALL BIOSInput2
```


BreakHit \SOURCE\BREAK.ASM

P.D.Q. Equivalent: BreakHit

Use

The P.D.Q. Break routines count the number of times Ctrl-Break or Ctrl-C are pressed if Break handling has been enabled with a call to BreakOff.

This routine returns the number of times Break has been hit since BreakOff was installed or since BreakHit was last called.

Calling Convention

```
CALL BreakHit
```

Returns break count in AX.

Notes

See BreakOff and BreakOn for further information about Ctrl-Break and Ctrl-C handling.

Example

```
Extrn BreakHit:Proc

.DATA?
BrkCount  dw  1 dup (?)   ;Room to hold count

.CODE
CALL BreakHit             ;Get the break count
MOV  BrkCount,AX          ;Save for future processing
```

BreakOff \SOURCE\BREAK.ASM

P.D.Q. Equivalent: BreakOff

Use

When you call BreakOff it intercepts the Ctrl-Break and Ctrl-C keys, and keeps a count of how many times they were pressed. That count is available by calling BreakHit. When BreakOff is first called, it installs itself and resets the BreakHit count to 0; if BreakOff is called when it is already installed, it simply returns without taking any action.

Calling Convention

```
CALL BreakOff
```

No return value.

Notes

BreakOff works by intercepting Int 1Bh (called when the user presses Ctrl-Break) to intercept Ctrl-Break and also Int 9 (the hardware keyboard interrupt) to look for Ctrl-C keys. If you call BreakOff, you *must* call BreakOn before your program ends, or these interrupts will be left pointing to empty code and an eventual system crash is inevitable.

Example

```
Extrn BreakOff:Proc

.CODE
Call BreakOff              ;Let P.D.Q. trap all breaks
```

BreakOn \SOURCE\BREAK.ASM

P.D.Q. Equivalent: BreakOn

- **Use**

  Reenables normal DOS and BIOS processing of Ctrl-Break and Ctrl-C after a call to BreakOff.

- **Calling Convention**

```
CALL BreakOn
```

  No return value.

- **Notes**

  If you call BreakOff to disable Ctrl-Break and Ctrl-C, you *must* call BreakOn before your program ends or you will cause a system crash.

Example

```
Extrn BreakOn:Proc

.CODE
Call BreakOn
```

BufIn \SOURCE\BUFIN.ASM

P.D.Q. Equivalent: BufIn

- **Use**

  BufIn performs very fast, buffered input from a sequential file. In QuickBASIC, it could be used this way:

```
DECLARE FUNCTION BufIn$ (FileName$, Done%)
DO
  This$ = BufIn$(FileName$, Done%)
  IF Done% THEN EXIT DO
  PRINT This$
LOOP
```

Calling Convention

```
PUSH Offset of FileName$ descriptor
PUSH Offset of word to hold Termination Flag
```

Returns the offset of a string descriptor for result string. May report an error by calling P$DoError.

Notes

Use a null string for the FileName$ parameter to tell BufIn to close the file, free up the buffer memory it uses, and return.

Each time BufIn returns a line of text it calls B$SPAC to allocate string pool memory. Therefore, you must delete the returned string before too many calls to BufIn accumulate.

Also, you *must* read to the end of the file or explicitly tell BufIn to close the file. BufIn maintains a flag that shows whether it is in the process of reading a file, and the flag is only reset on a DOS error, when the end of the file is reached, or if you have BufIn close the file explicitly.

BufIn claims a 4K buffer in DOS memory for file reads.

Example

Implements the BASIC loop shown above:

```
    Extrn BufIn:Proc
    Extrn B$PSED:Proc                ;Print-a-string routine

    .DATA?
    FileName$  dd  1 dup (?)         ;FileName$ string descriptor
    Done       dw  1 dup (?)         ;Flag to show that file is done

    .CODE
@@:                                  ;Head of the loop
    MOV  AX,Offset FileName$         ;Get address of descriptor
    PUSH AX
    MOV  AX,Offset Done              ;Get address of DONE flag
    PUSH AX
    CALL BufIn
    CMP  Word ptr[Done],0            ;Are we done reading?
    JNZ  @F                          ;Yes--leave the loop
    PUSH AX                          ;Else pass result descriptor
    CALL B$PSED                      ;Print it
    JMP  @B                          ;And do it again
@@:                                  ;Here at loop's end
```

CritErrOff \SOURCE\CRITERR.ASM

P.D.Q. Equivalent: CritErrOff

Use

Disables DOS critical error handling (the "Abort, Retry, Fail"message). This is most useful in TSR programs to prevent crashing an underlying program.

Calling Convention

```
CALL CritErrOff
```

No return value.

Notes

Make sure that you reenable critical error handling before your program ends, or you will cause a system crash. If you call CritErrOff inside a TSR when you pop up, you *must* call CritErrOn before you return control to the foreground program.

Example

```
Extrn CritErrOff:Proc

.CODE
CALL CritErrOff          ;Disable critical errors
```

CritErrOn \SOURCE\CRITERR.ASM

P.D.Q. Equivalent: CritErrOn

Use

Turns system critical error handling back on after a call to CritErrOff.

Calling Convention

```
CALL CritErrOn
```

No return value.

Notes

If you call CritErrOff in a TSR, you *must* call this procedure before you give control back to the foreground program, or you will cause a system crash.

Example

```
Extrn CritErrOn:Proc

.CODE
CALL CritErrOn
```

CursorOff \SOURCE\CURSOR.ASM

P.D.Q. Equivalent: CursorOff

BASIC Equivalent: LOCATE , , 0

- **Use**

  Hides the screen cursor, making it invisible.

- **Calling Convention**

```
CALL CursorOff
```

  No return value.

- **Notes**

  You can turn the cursor back on with CursorOn or with CursorSize.

Example

```
Extrn CursorOff:Proc

.CODE
CALL CursorOff
```


CursorOn \SOURCE\CURSOR.ASM

P.D.Q. Equivalent: CursorOn

BASIC Equivalent: LOCATE , , 1

- **Use**

  Makes the screen cursor visible.

- **Calling Convention**

```
CALL CursorOn
```

  No return value.

Notes

The cursor is displayed in the current cursor size. Use CursorSize to set the size after the cursor is visible.

Example

```
Extrn CursorOn:Proc

.CODE
CALL CursorOn              ;Turn cursor on
```

CursorRest \SOURCE\CURSORSR.ASM

P.D.Q. Equivalent: CursorRest

Use

Restore the cursor size and position determined by a previous call to CursorSave.

Calling Convention

```
PUSH Offset of saved cursor information
CALL CursorRest
```

No return value.

Notes

CursorRest expects the saved cursor information to be a 4-byte long integer in the format returned by CursorSave.

Example

```
Extrn CursorRest:Proc

.DATA?
Cursor  dd  1 dup (?)      ;Info from CursorSave

.CODE
MOV  AX,Offset Cursor      ;Get pointer to data
PUSH AX                    ;Pass it on
CALL CursorRest            ;And restore the cursor
```

CursorSave \SOURCE\CURSORSR.ASM

P.D.Q. Equivalent: CursorSave

Use

Returns the current cursor size and location packed into a 4-byte long integer in one convenient call.

Calling Convention

```
CALL CursorSave
```

Returns result in DX:AX.

Notes

CursorSave simply collects data from Int 10h, Service 3. It returns the cursor position in DX (just as Int 10H does), and the cursor size in AH and AL instead of CH and CL. You may want to save time by calling Int 10H yourself.

Example

```
Extrn CursorSave:Proc

.DATA?
Cursor  dd  1 dup (?)           ;Storage for result

.CODE
CALL CursorSave                 ;Get cursor information
MOV  Word ptr [Cursor],AX       ;Save values for later
MOV  Word ptr [Cursor+2],DX
```


CursorSize \SOURCE\CURSOR.ASM

P.D.Q Equivalent: CursorSize

BASIC Equivalent: LOCATE , , , TopLine, BottomLine

Use

Set the cursor size by specifying the top and bottom scan lines of the cursor block.

Calling Convention

```
PUSH Offset of TopLine
PUSH Offset of BottomLine
CALL CursorSize
```

No return value.

Notes

The values of TopLine and BottomLine may range from 0 to the highest legal value supported by the installed display adapter. CursorSize uses only the low byte of both values, but does no other error checking.

Example

Perform equivalent of CALL CursorSize(1, 5):

```
Extrn CursorSize:Proc

.DATA
TopLine  dw  1          ;Storage for top and
BotLine  dw  5          ; bottom lines

.CODE
MOV  AX,Offset TopLine  ;Get pointer to first argument
PUSH AX                 ;Pass it by reference
MOV  AX,Offset BotLine  ;Get pointer to second argument
PUSH AX                 ;Pass it by reference
CALL CursorSize         ;Set new cursor size
```

DeinstallTSR \SOURCE\DEINSTAL.ASM

P.D.Q. Equivalent: DeinstallTSR

Use

Removes a non-simplified P.D.Q. TSR program from memory.

Calling Convention

```
PUSH Offset of DGROUP value
PUSH Offset of ID string descriptor
CALL DeinstallTSR
```

Returns failure (0) or success (-1) in AX.

Notes

See the description for DeinstallTSR in the reference section of this manual for information about using this routine.

Example:

```
Extrn DeinstallTSR:Proc

.DATA
SavedDGROUP  dw  ?              ;Saved DGROUP from TSRInstalled
DefStr ID$                      ;Space for ID$ string descriptor

.CODE
MOV  AX,Offset SavedDGROUP      ;Pass pointer to Data
PUSH AX                         ; segment
MOV  AX,Offset ID$              ;and pointer to
PUSH AX                         ;  string descriptor
CALL DeinstallTSR               ;Remove the TSR
OR   AX,AX                      ;Test for success
JZ   Deinstall_Failed           ;Go tell user to reboot
```

Dollar \SOURCE\DOLLAR$.ASM

P.D.Q. Equivalent: Dollar$

Use

Formats a long integer into a string in dollars and cents format.

Calling Convention

```
PUSH Offset of long integer to convert
CALL Dollar
```

Returns with AX = offset of string descriptor.

Notes

This routine does not place a dollar sign in front of the result. The result contains an optional minus sign, plus the value formatted with a decimal point before the last two digits.

The result is held in a special temporary string and will not be overwritten until the next call to Dollar. If you want, you can simply save the offset returned in AX until you are ready to use the string; do not, however, modify the returned string directly unless you copy it to your own string space with B$SASS.

Example

Convert Cents& to dollars and cents format:

```
Extrn Dollar:Proc

.DATA?
```

```
EVEN
Cents&  dd  1 dup (?)     ;Space for long integer
Result$ dd  1 dup (?)     ;String descriptor for result

.CODE
MOV  AX,Offset Cents&     ;Get address of value
PUSH AX                   ;  and send it on
CALL Dollar               ;Convert it
PUSH AX                   ;Now assign string to us
MOV  AX,Offset Result$    ;This is where we want it
PUSH AX
CALL B$SASS               ;Keep result string.
```

DOSBusy \SOURCE\DOSBUSY.ASM

Use

Lets a non-simplified P.D.Q. TSR program determine when it is safe to use DOS interrupt services (Int 21h).

Calling Convention

```
CALL DosBusy
```

Returns 0 in AX if DOS interrupts can be used, or -1 if a DOS call is currently in progress.

Notes

You *must* call this routine once (and ignore the result, if you wish) *before* your TSR goes resident.

See the warnings about relying on the DOS Busy flag in the DOSBusy routine description elsewhere in this manual.

Example

```
Extrn DosBusy:Proc

.CODE
CALL DosBusy              ;Can we interrupt?
OR   AX,AX                ;Check return value
JZ   Okay_to_call_DOS     ;Yep -- we can do whatever we want

;nope -- add code here to exit and try again later.
```

EndTSR \SOURCE\ENDTSR.ASM

P.D.Q. Equivalent: EndTSR

Use

This routine is called by a TSR program when it has finished initializing and is ready to go resident and return control to DOS.

Calling Convention

```
PUSH Offset of unique ID string descriptor
CALL EndTSR
```

No return value.

Notes

See the discussions of TSR processing elsewhere in the manual for notes about using EndTSR and creating a unique ID string.

Example

```
Extrn EndTSR:Proc

.DATA?
EVEN
ID$  dd  1 dup (?)        ;String descriptor space

.CODE
MOV  AX,Offset ID$        ;Get address of descriptor
PUSH AX                   ;Pass it on
CALL EndTSR
```


EnvOption \SOURCE\ENVOPT.ASM

P.D.Q. Equivalent: EnvOption

Use

Lets a program switch between accessing its own or its parent's environment, and also determines how capitalization of environment strings is handled.

Calling Convention

```
PUSH Offset of option word
CALL EnvOption
```

No return value.

Notes

The option is stored in a data word and passed by reference. The option word is a bit record:

Bit 0:	0	Access the current program's copy of the environment.
	1	Access the current program's parent's environment.
Bit 1:	0	Capitalize variable names before adding or retrieving.
	1	Don't capitalize variable names.
Bit 2:	0	Access the current program's copy of the environment.
	1	Access the environment of the currently active process.

Bits 3-15 are reserved and should be set to zero.

Example

```
Extrn EnvOption:Proc

.DATA
Option  dw  011b          ;No capitalization, use parent's environment

.CODE
MOV  AX,Offset Option     ;Get pointer to options
PUSH AX                   ;Pass it on
CALL EnvOption            ;Reset environment options
```

FUsing \SOURCE\FUSING.ASM

P.D.Q. Equivalent: FUsing

Use

Creates a string representation of a value honoring most of the formatting conventions of BASIC's PRINT USING.

Calling Convention

```
CALL One of the STR$() routines with the value
PUSH Received pointer to a temporary string descriptor
PUSH Address of descriptor for format string
CALL FUsing$
```

Returns offset of the result string descriptor in AX.

Notes

See the explanation of FUsing$ in the reference portion of this manual for an explanation of the format string.

You must use the normal STR$() functions to create the initial string, and not the alternate STR$() function from the _STR$.ASM stub file. FUsing depends on the leading space before positive values.

The string descriptor that FUsing returns and the result string are stored in FUsing's own data space. They will be overwritten by the next call to FUsing.

Example

Perform Result$ = FUsing$(Num&, Image$)

```
Extrn FUsing:Proc
Extrn B$STI4:Proc          ;STR$() of a long integer
Extrn B$SASS:Proc          ;String assignment

.DATA?
EVEN
Result$  dd  1 dup (?)     ;Room for string descriptors
Image$   dd  1 dup (?)
Num      dd  1 dup (?)     ;Room for long integer

.CODE
LES  AX,[Num]              ;Get the value
PUSH ES                    ;Pass it on
PUSH AX
CALL B$STI4                ;Convert Num to a string
PUSH AX                    ;Pass on the result
MOV  AX,Offset Image$      ;Get pointer to the image
PUSH AX
CALL FUsing                ;Create a formatted string
PUSH AX                    ;Pass on the result
MOV  AX,Offset Result$     ;Final resting place
PUSH AX
CALL B$SASS
```


Get1Long \SOURCE\GET1LONG.ASM

P.D.Q. Equivalent: Get1Long

Use

Reads a long integer (four bytes) from anywhere in memory, given a segment and an element number.

Calling Convention

```
PUSH Offset of the variable holding the segment value
PUSH Offset of the variable holding the element number
CALL Get1Long
```

Result is returned in DX:AX.

Notes

Element numbers are considered to start at one. There is no element zero.

The offset that this routine reads from is calculated as follows:

```
Offset = (Element - 1) * 4
```

You may be able to save some bytes and cycles by making the calculation yourself and reading the memory directly.

Example

Get the fourth long integer from the segment stored in LSEG:

```
Extrn Get1Long:Proc

.DATA?
LSEG  dw  1 dup (?)       ;Space for the segment value
ELEM  dw  1 dup (?)       ;Space for the element number

.CODE
MOV  AX,Offset LSEG       ;Point to the segment
PUSH AX                   ;  and push that
MOV  ELEM,4               ;Save the element number
MOV  AX,Offset ELEM       ;Point to the element number
PUSH AX                   ;Pass it on
CALL Get1Long             ;Read the memory
; Now DX:AX has the element's value
```

Get1Type \SOURCE\GET1TYPE.ASM

P.D.Q. Equivalent: Get1Type

Use

Reads a block of memory into a TYPE variable (or any other block of bytes) in near memory.

Calling Convention

```
PUSH Offset of the variable holding the segment value
PUSH Offset of the variable holding the element number
PUSH Offset of the variable holding the length of destination
PUSH Offset of destination
```

No return value.

Notes

Element numbers are considered to start at one. There is no element zero.

The offset that this routine reads from is calculated as follows:

```
Offset = (Element - 1) * Length
```

You may be able to save a few bytes and cycles by making the calculation yourself and reading the memory directly.

Example

Get the fifth block of 20 bytes from the segment stored in TSEG:

```
Extrn Get1Type:Proc

.DATA
Result  db  20 dup (?)    ;Place for the data
TSeg  dw  ?               ;Place for segment address
Elem    dw  ?             ;Place for element number
Length  dw  TSeg - Result ;Length to read

.CODE
MOV  AX,Offset TSeg       ;Pass pointer to segment
PUSH AX
MOV  AX,Offset Elem       ;and pointer to element
PUSH AX
MOV  AX,Offset Length     ;and pointer to length
PUSH AX
MOV  AX,Offset Result     ;and pointer to destination
PUSH AX
CALL Get1Type             ;Transfer the data
```

GetCPU \SOURCE\GETCPU.ASM

P.D.Q. Equivalent: GetCPU

- **Use**

  Returns the CPU type in AX.

- **Calling Convention**

  ```
  CALL GetCPU
  ```

  The value returned in AX is 86, 286, or 386.

- **Notes**

  If you want to see how it's done, take a look at the source code. The algorithm is based on the hardware wiring of the flags register inside the CPU.

Example

```
Extrn GetCPU:Proc

.CODE
CALL GetCPU              ;Ask for the CPU type
;the result is now in AX
```

GotoOldInt \SOURCE\GOTOINT.ASM

P.D.Q. Equivalent: GotoOldInt

Use

Jumps to the previous interrupt handler.

Calling Convention

```
PUSH Offset of Registers TYPE variable
CALL GotoOldInt
```

Although this routine is called, it does not return control to your program.

Notes

The use of the Registers TYPE variable gives this routine flexibility but also adds some overhead. You may want to jump to the original interrupt service routine directly from many TSR and other interrupt handlers unless you have some reason to change the received register values via the Registers TYPE variable.

Example

```
Extrn GotoOldInt:Proc

.DATA?
Reg  db  36 dup (?)      ;Register TYPE variable

.CODE
MOV  AX,Offset Reg       ;Get near pointer to variable
PUSH AX
CALL GotoOldInt          ;And off we go
; Control never returns to this line
```

HercMode \SOURCE\HERCMODE.ASM

P.D.Q. Equivalent: HercMode

Use

Enables SCREEN 0 and SCREEN 3 on a Hercules monochrome card.

Calling Convention

```
PUSH Offset of Mode word
CALL HercMode
```

No return value.

Notes

If the Mode value is non-zero the Hercules card is switched into graphics mode. A value of zero then returns it to text mode.

By using this routine, you can switch between text and graphics on a Hercules monochrome card (or compatible) without having to load Microsoft's MSHERC.COM driver.

This routine does not check to see if a Hercules card is actually installed. If you call it on a system that doesn't have a Hercules card, the results will be, as they say, undefined. You can use the PDQMonitor function to determine the type of display adapter that is connected.

Example

Switch into Hercules graphics mode:

```
Extrn HercMode:Proc

.DATA
Mode  dw  1              ;Set for graphics mode

.CODE
MOV  AX,Offset Mode      ;Point to the mode
PUSH AX                  ;  and send the pointer
CALL HercMode            ;Switch to graphics.
```

HookInt0 \SOURCE\HOOKINT0.ASM

P.D.Q. Equivalent: HookInt0

Use

Captures any Interrupt 0 (Division by 0) calls and turns them into BASIC error 11 ("Division by zero").

Calling Convention

```
PUSH Offset of Mode variable
CALL HookInt0
```

No return value.

Notes

If Mode = 0, then UnHookInt0 will be added to the B_OnExit chain for automatic execution at the end of the program (assuming that you exit through B$CEND). If Mode < > 0 then you must call UnHookInt0 before your program exits, or you are certain to cause a system crash the next time a division by 0 error occurs.

If you happen to call HookInt0 when the P.D.Q. Int 0 interrupt is already trapped, this routine simply returns without doing anything.

If an Interrupt 0 occurs while HookInt0 is in effect, P$DoError will be called to report the error.

Example

```
Extrn HookInt0:Proc

.DATA
Mode  dw  1              ;We'll unhook it ourselves

.CODE
MOV  AX,Offset Mode      ;Point to the mode
PUSH AX                  ; and pass the pointer
CALL HookInt0
```

MidChar \SOURCE\MIDCHAR.ASM

P.D.Q. Equivalent: MidChar function

Use

Returns the ASCII value of one character within a string.

Calling Convention

```
PUSH Offset of the string's descriptor
PUSH Offset of word variable holding the position vaue
CALL MidChar
```

AX contains the ASCII value of the target character, or -1 if the specified position is past the end of the string.

Notes

Notice that the position is passed by reference, not by value. You must therefore create an integer variable to hold the position count.

If AH = 0 on return, then AL contains a valid ASCII value.

Example

Find MidChar(A$, 5):

```
Extrn MidChar:Proc

.DATA?
EVEN
A$    dd  1 dup (?)       ;Space for string descriptor
Posn  dw  1 dup (?)       ;Space for position

.CODE
MOV  [Posn],5             ;Set position value
MOV  AX,Offset A$         ;Point to descriptor
PUSH AX
MOV  AX,Offset Posn       ;Point to position variable
PUSH AX
CALL MidChar              ;Get one character
OR   AH,AH                ;Is return valid?
JNZ  BadPosition          ;No -- take care of error
; Now AL has ASCII value of character.
```

MidCharS \SOURCE\MIDCHARS.ASM

P.D.Q. Equivalent: MidCharS

Use

Replaces a character in a string with another character specified by ASCII value.

Calling Convention

```
PUSH Offset of string descriptor
PUSH Position of character to change
PUSH ASCII value of new character
```

No return value.

Notes

If the position of the character to change is past the end of the string, the original string will not be changed and MidCharS will simply return.

The high byte of the new character value is ignored.

Example

Perform MidCharS(Work$, 10, 32) to assign a CHR$(32) space at the tenth character position in Work$:

```
Extrn MidCharS:Proc

.DATA?
EVEN
Work$  dd  1 dup (?)      ;Space for a string descriptor

.CODE
MOV  AX,Offset Work$      ;Get string descriptor address
PUSH AX
MOV  AX,10                ;Position to change
PUSH AX
MOV  AL,' '               ;Space character
PUSH AX                   ;(AH will be ignored)
CALL MidCharS
```

NoSnow \SOURCE\NOSNOW.ASM

P.D.Q. Equivalent: NoSnow

Use

Enable or disable CGA snow suppression.

Calling Convention

```
PUSH Offset of SnowFlag variable
CALL NoSnow
```

No return value.

Notes

Setting Snowflag = 0 disables snow suppression; any other value enables snow suppression.

On some CGA systems, snow suppression may be necessary, but many newer CGA adapters take care of snow suppression internally and allow much faster video output if snow suppression is turned off. P.D.Q. will never turn on snow suppression automatically unless a CGA adapter is being used.

This routine calls P$MonSetup.

Example

Turn off snow suppression:

```
Extrn NoSnow:Proc

.Data
SnowFlag  dw  0          ;We want to turn it off

.CODE
MOV  AX,Offset SnowFlag  ;Point to the action flag
PUSH AX
CALL NoSnow              ;Snow suppression is turned off
```


P$Compact \SOURCE\COMPACT.ASM

Use

Compacts the string pool, deleting unused strings and moving strings that are in use so that the pool contains the largest block of contiguous string

space possible (garbage collection). Note that this routine does not delete temporary strings.

Calling Convention

```
CALL P$Compact
```

No return value.

Notes

The process of compacting the string pool is often called garbage collection. It is performed whenever space for a new string is not available, and whenever B$FRSD (FRE(X$)) is called. It is not usually necessary to compact the string pool manually, since this is done automatically as part of the P.D.Q. string management process.

P.D.Q. keeps track of the status of the string pool. If compaction is unnecessary, this routine returns very quickly.

Example

```
Extrn P$Compact:Proc

.CODE
CALL P$Compact
```

P$DelAllTemps \SOURCE\DELTEMPS.ASM

Use

Deletes all temporary strings.

Calling Convention

```
CALL P$DelAllTemps
```

No return value.

Notes

This routine goes through the temporary string list (which has room for 20 temporary string descriptors) and deletes each one that is in use.

Functions which return a pointer to a string descriptor normally point to a temporary string. You can delete such strings individually, or wait until the end of a section of code and then delete all temporary strings at once with this routine. Remember, however, that many of the string manipulation routines need temporary strings for their intermediate steps and will fail if no temporary string space is available.

Example

```
Extrn P$DelAllTemps:Proc

.CODE
CALL P$DelAllTemps        ;Delete all temporary strings
```

P$DELAY \SOURCE\P$DELAY.ASM

Use

Delay a specified number of milliseconds.

Calling Convention

```
PUSH Number of milliseconds to delay
CALL P$DELAY
```

No return value.

Notes

This routine accepts a signed integer for the number of milliseconds. If you pass it a number greater than 32,767 (7FFFh), it will see the value as a negative number, take its absolute value, and use that value for the millisecond count.

Example

```
Delay for 1000 milliseconds (1 second):

.CODE
MOV  AX,1000              ;Delay count
PUSH AX
CALL P$DELAY
```

P$FreeTemp \SOURCE\FREETEMP.ASM

Use

Frees a temporary string.

Calling Convention

```
PUSH Offset of string's descriptor
CALL P$FreeTemp
```

No return value.

Notes

This routine checks to make sure that the string descriptor points to a temporary string. If so, the string is deleted and its space is returned to the string pool. Otherwise the request is ignored.

P.D.Q. has room for 20 temporary string descriptors. Functions which return a pointer to a string descriptor almost always use one of those 20, and will fail if all 20 are in use. Be sure you free temporary strings as soon as possible after they are returned to you.

Every P.D.Q. library function which accepts a pointer to a string descriptor as an argument passes the pointer to this routine in an attempt to release temporary string space.

Example

```
Extrn P$FreeTemp:Proc

.CODE
MOV  AX,StringPointer     ;Get descriptor address
PUSH AX                   ;Pass it on
CALL P$FreeTemp           ;Erase it if it is temporary
```

P$GetTemp \SOURCE\GETTEMP.ASM

Use

Finds the next available temporary string descriptor and returns a pointer to it.

Calling Convention

```
CALL P$GetTemp
```

Returns offset of descriptor in BX (not in AX).

Notes

You will probably never need to call this routine directly—it is called for you by P$MakeTemp and B$SCAT. However, there are three lines of the source code file that have been commented out. You may want to reinstate those lines, recompile the source file, and put it in the P.D.Q. library. The three lines force P$GetTemp to check whether there are any temporary string descriptors available.

P.D.Q. has allocated room for 20 temporary string descriptors (80 bytes). Code created by the BC compiler will never require more than those 20 temporary strings. But if you forget to release temporary strings in your code, you could easily use up all 20 slots and overwrite active strings.

The three error-checking lines mentioned above can help you pinpoint bugs in your own code.

Example

```
Extrn P$GetTemp:Proc

.CODE
CALL P$GetTemp     ;Now BX has offset of a temporary string descriptor
```

P$HookFP \FPSOURCE\P$HOOKFP.ASM

Use

Determines the presence of a math coprocessor at runtime, and installs the necessary interrupt vector table to use an 80x87 or the emulator library.

Calling Convention

```
CALL P$HookFP
```

No return value.

Notes

You *must* call this procedure before your program uses any floating point operations or calls any library routines that use floating point operations.

Example

Set up to use floating point:

```
Extrn P$HookFP:Proc

.CODE
CALL P$HookFP              ;That's all we have to do.
```

P$MakeTemp \SOURCE\MAKETEMP.ASM

Use

Creates a new, temporary variable-length string.

Calling Convention

```
MOV  CX,Number of string bytes needed
CALL P$MakeTemp
```

Returns with:

```
CPU direction flag cleared
BX = Offset of temporary string descriptor
DI = Offset of first byte of new string
```

```
ES = DS = string segment
CX = Length of string allocated
```

Returns CX = 0 if insufficient string space for *any* string.

Notes

Neither this routine nor the routines it calls make any attempt to insure that there is an unused temporary string descriptor available (but see the note in P$GetTemp). The 20 temporary descriptors allocated for P.D.Q. programs are sufficient for all programs generated by the BC compiler; it is your responsibility to be ensure that they are sufficient for your programs.

Note that you may not get as much string space as you requested. Be careful not to write past the end of the allocated string space; you will likely overwrite the back pointer of another string and corrupt the integrity of all variable-length strings in your program. Because P$MakeTemp adjusts CX downward if insufficient string memory was available, you should always use that value for subsequent Rep string operations.

Example

Create a temporary string of 10 digits from 0 to 9, then copy it to a permanent string location:

```
        Extrn P$MakeTemp:Proc

        .DATA?
        EVEN
        A$  dd  1 dup (?)       ;Room for our string descriptor

        .CODE
        MOV  CX,10              ;Desired string length
        CALL P$MakeTemp         ;Create the string
        JCXZ NoRoom             ;Go if no string space
        MOV  AL,'0'             ;Put the first character in AL
    @@: STOSB                   ;Put one character in the string
        INC  AL                 ;Next character in AL
        LOOP @B                 ;Do the whole string
        ; Now string is filled and BX points to its descriptor
        PUSH BX                 ;Pass address of temp. string
        MOV  AX,Offset A$       ; and address of permanent home
        PUSH AX
        CALL B$SASS             ;String is now assigned to A$
```

P$MonSetup \SOURCE\MONSETUP.ASM

Use

Determines the correct segment for video output and sets a "snow" flag if a CGA system is in use.

Calling Convention

```
CALL P$MonSetup
```

No return value.

Notes

This routine initializes two public data words:

```
P$MonSeg holds the current (text) video segment
P$CGAPort, if non-zero, is the CGA snow flag
```

If P$MonSeg is non-zero, this routine assumes that it already holds the correct value and immediately exits. If you want to force it to check again, set P$MonSeg to 0.

P$CGAPort holds a value of either zero or the CGA adapter port address of 3DAh.

Example

Determine whether a color system is being used:

```
Extrn P$MonSetup:Proc

.DATA
Extrn P$MonSeg:Word

.CODE
CALL P$MonSetup          ;Let it do its thing
CMP  P$MonSeg,0B800h     ;Is it color?
JNE  NoColor             ;Nope -- it must be mono
; If we get here, we have to use the color segment
```


P$Num2Handle \SOURCE\NUM2HNDL.ASM

Use

Returns the DOS file handle for a BASIC file number of an opened file.

Calling Convention

```
MOV  BX, the BASIC file number
CALL P$Num2Handle
```

Returns the DOS file handle in BX, or -1 if the BASIC file handle was invalid (greater than 15 or not opened).

Notes

Note that STDERR has a BASIC file number of 255 (and returns a DOS file handle of 2).

Only files that have been opened with a call to B$OPEN or B$OOPN have BASIC file numbers. P.D.Q. maintains a table of those files and looks up the appropriate DOS handle number when P$Num2Handle is called.

Example

Find the DOS handle for BASIC File #2:

```
.CODE
MOV  BX,2                ;Put the BASIC File number in BX
CALL P$Num2Handle
OR   BH,BH               ;BH should always be 0
JNZ  BadFileNum          ;It wasn't -- take care of the error
;  Now BX has the corresponding DOS file handle.
```

P$SkipEOF \SHELL\SKIPEOF.ASM

Use

Moves a file pointer to the last position in the file. This routine is used by B$OPEN to position the file pointer when a file is opened for APPEND mode.

Calling Convention

```
MOV  DOS file handle to BX
MOV  0 to CX and DX
MOV  4202h to AX
INT  21h                      ;DOS: seek to end of file
CALL P$SkipEOF
```

No return value.

Notes

This routine moves the file pointer backward over any trailing CHR$(26) EOF characters at the end of the file, so that those characters will be overwritten by new data appended to the file. Notice that this routine requires a DOS file handle, not a BASIC file number.

Example

Prepare to append to file:

```
Extrn P$SkipEOF:Proc
.DATA
Handle  dw  1 (dup) ?

.CODE
MOV   BX,[Handle]          ;Get DOS file handle
SUB   CX,CX                ;CX = 0
MOV   DX,CX                ;CX:DX = 0
MOV   AX,4202h             ;Seek from end of file
INT   21h                  ;Let DOS do the seeking
CALL  P$SkipEOF            ;Move backwards over EOFs
```

P$SOUND \SOURCE\P$SOUND.ASM

Use

Generate a tone of a desired frequency and duration through the computer's speaker.

Calling Convention

```
PUSH Frequency in Hz.
PUSH Duration in milliseconds
CALL P$SOUND
```

No return value.

Notes

You *must* call P$Speaker once before calling this routine, to initialize channel 2 of the timer chip.

If you specify a frequency of less than 37 Hz, this routine will simply delay for the specified number of milliseconds (by calling P$Delay) instead of producing a tone.

Use the PDQSound routine to avoid having to initialize the speaker with a separate call to P$Speaker.

Example

Generate a tone of 440 Hz (concert A) for 100 milliseconds:

```
Extrn P$SOUND:Proc

.CODE
MOV   AX,440               ;Get frequency
PUSH  AX
MOV   AX,100               ;and duration
PUSH  AX
CALL  P$SOUND
```

P$Speaker \SOURCE\SPEAKER.ASM

Use

Initializes the speaker for further use.

Calling Convention

```
CALL P$Speaker
```

No return value.

Notes

The speaker only has to be initialized once for any program. If it has already been initialized, this routine returns without doing anything. You must initialize the speaker before calling P$SPKR_ON or P$SPKR_OFF yourself.

Example

Initialize the speaker:

```
Extrn P$Speaker:Proc

.CODE
CALL P$Speaker          ;Initialize the speaker
```

P$SPKR_OFF \SOURCE\SPEAKER.ASM

Use

Turn off the speaker.

Calling Convention

```
CALL P$SPKR_OFF
```

No return value.

Notes

Use the PDQSound routine to avoid having to manually turn the speaker on and off.

Use this routine only if you have called P$SPKR_ON to turn on the speaker. Do not call this routine unless you have initialized the speaker with a call to P$Speaker.

Example

Turn off the speaker:

```
Extrn P$SPKR_OFF:Proc

.CODE
CALL P$SPKR_OFF          ;Turn off the speaker
```

P$SPKR_ON \SOURCE\SPEAKER.ASM

Use

Turn on the speaker.

Calling Convention

```
CALL P$SPKR_ON
```

No return value.

Notes

Like P$Speaker and P$SPKR_OFF, this is a low-level internal routine that is really meant to be called by B$SPLY.

If you use this routine, it is up to you to call P$SPRK_OFF at the appropriate time to turn the speaker off again. How long you leave the speaker turned on determines the duration of the tone produced.

Do not call this routine unless you have initialized the speaker with a call to P$Speaker.

Example

Turn on the speaker:

```
Extrn P$SPKR_ON:Proc

.CODE
CALL P$SPKR_ON           ;Turn on the speaker
```


P$UnHookFP \FPSOURCE\UNHOOKFP.ASM

Use

Restores the floating point interrupt vectors in a program that has used floating point instructions.

Calling Convention

```
CALL P$UnHookFP
```

No return value.

Notes

To use floating point instructions you must call P$HookFP to take over the floating point interrupts, and also assemble the program with the /e switch. If you exit the program without resetting the floating point interrupts, you may cause a system crash some time in the future. Therefore, you must call P$UnHookFP before your program ends, to reset the floating point interrupts to their original values.

Example

Prepare to end a program assembled with /e:

```
Extrn P$UnHookFP:Proc

.CODE
CALL P$UnHookFP          ;Let P.D.Q. reset the interrupts
```

P$ZeroFile \SOURCE\ZEROFILE.ASM

Use

Copies a string to a buffer and adds a trailing null character.

Calling Convention

```
MOV  AX, Offset of descriptor for string
MOV  DX, Offset of buffer
CALL P$ZeroFile
```

Returns with CX = 0 and ASCIIZ string in the buffer. DX is unchanged.

Notes

This routine is used primarily for converting a BASIC string to an ASCIIZ string suitable for use as an argument to DOS.

The buffer must be long enough to hold the entire string, plus one more to hold the added CHR$(0) null byte.

This routine calls P$FreeTemp to delete the original string if its descriptor is in P.D.Q.'s temporary string space.

This routine is used internally by many P.D.Q. routines to prepare file and subdirectory names for calls to DOS services.

Example

Convert A$ to an ASCIIZ string on the stack. Assume that A$ is less than 80 bytes long:

```
Extrn P$ZeroFile:Proc

.DATA?
EVEN
A$  dd  1 dup (?)          ;Room for string descriptor

.CODE
PUSH BP                    ;Create room on stack
MOV  BP,SP                 ;Save the stack pointer
SUB  SP,80                 ;Make room for the buffer
MOV  DX,SP                 ;DX ==> buffer area
MOV  AX,Offset A$          ;Offset of string
CALL P$ZeroFile            ;Create ASCIIZ string
...                        ;use the ASCIIZ string
MOV  SP,BP                 ;Discard the buffer
```

Pause \SOURCE\PAUSE.ASM

P.D.Q. Equivalent: Pause

Use

Pauses the computer for the specified number of timer ticks.

Calling Convention

```
PUSH Offset of number of timer ticks
CALL Pause
```

No return value.

Notes

This routine is accurate to within +0 and -.056 seconds.

Example

Pause for 1 second (18 timer ticks):

```
Extrn Pause:Proc

.DATA
Ticks  dw  18              ;Storage for delay count

.CODE
MOV  AX,Offset Ticks       ;Address of Ticks
PUSH AX
CALL Pause
```

PDQCompare \SOURCE\PDQCOMP.ASM

Use

Compares two regions of memory of equal length.

Calling Convention

```
PUSH Segment of region 1
PUSH Offset of region 1
PUSH Segment of region 2
PUSH Offset of region 2
PUSH Number of bytes to compare
```

Returns -1 in AX if the bytes match in both regions, or 0 if they are different.

Notes

The return value in AX from this routine is either True or False to show whether the byte strings are the same or not. However, the flags are undisturbed from the Repe Cmpsb instruction, so you can use them to determine which byte string is greater.

Example

Compare two 20-byte regions of near memory:

```
Extrn PDQCompare:Proc

.DATA?
Region1  db  20 dup (?)   ;Two memory areas to compare
Region2  db  20 dup (?)

.CODE
PUSH DS                   ;Segment of region 1
MOV  AX,Offset Region1    ;Get offset of first region
PUSH AX
PUSH DS                   ;Segment of region 2
MOV  AX,Offset Region2    ;Get offset of second region
PUSH AX
MOV  AX,20                ;Get length to compare
PUSH AX
CALL PDQCompare
OR   AX,AX                ;Were they the same?
JNZ  TheSame              ;Yes -- Go
```

PDQCPrint \SOURCE\PDQCPRNT.ASM

P.D.Q. Equivalent: PDQCPrint

Use

Prints a string at a given row and column location quickly by bypassing DOS and the BIOS.

Calling Convention

```
PUSH Offset of string descriptor of text to print
PUSH Offset of video row for first character
PUSH Offset of video column for first character
CALL PDQCPrint
```

No return value.

Notes

This routine uses the current color value for the characters that it prints. The color value is stored in the external byte P$Color.

Both the row and column values are 1-based.

This routine does not check the validity of the row and column number you provide. That is your responsibility.

You can direct the text to any segment (not just the video display) by setting the segment value in the external word P$MonSeg. You must either set that segment manually, or call P$MonSetup once before calling this routine to make sure the video segment value is correctly set.

Example

Print the string Work$ at Row 5, Column 1:

```
Extrn PDQCprint:Proc

.DATA?
EVEN
Work$  dd  1 dup (?)      ;Space for a string descriptor
Row    dw  1 dup (?)      ;Space for the row
Column dw  1 dup (?)      ;Space for the column

.CODE
MOV  [Row],5              ;Set the row value
MOV  [Column],1           ;  and the column value
MOV  AX,Offset Work$      ;Get address of descriptor
PUSH AX
MOV  AX,Offset Row        ;Get video row for output
```

```
PUSH AX
MOV  AX,Offset Column     ;Get video column
PUSH AX
CALL PDQCPrint
```

PDQExist \SOURCE\PDQEXIST.ASM

P.D.Q. Equivalent: PDQExist

Use

Determines whether a specified file exists.

Calling Convention

```
PUSH Offset of string descriptor for the filespec
CALL PDQExist
```

Returns AX = -1 if file exists or 0 if it doesn't exist.

Notes

This routine resets the DTA to its own space on the stack.

This routine always clears the error word P$PDQErr.

If this routine finds a file, it may have any or all of the following attributes set: read-only, hidden, system, and archive. This routine will not find subdirectory names or volume names.

Example

Determine whether the file exists whose name is stored in FileName$:

```
Extrn PDQExist:Proc

.DATA?
EVEN
FileName$  dd  1 dup (?)  ;Room for the string descriptor

.CODE
MOV  AX,Offset FileName$  ;Get offset of string
PUSH AX
CALL PDQExist             ;Does the file exist?
OR   AX,AX                ;Test the result
JNZ  FileExists           ;Go if it does exist.
```

PDQInkey \SOURCE\PDQINKEY.ASM

P.D.Q. Equivalent: PDQInkey

Use

This routine returns the value of the next keystroke, or 0 if no keystroke is pending.

Calling Convention

```
CALL PDQInkey
```

AX contains 0 if no key was waiting, a positive value for a regular ASCII character, or a negative value for an extended key.

Notes

This routine uses DOS Service 6 to see if a keystroke is pending, so it supports redirection of input.

Example

Loop until a keystroke is ready and then test whether it is an ASCII key or extended key.

```
      Extrn PDQInkey:Proc

      .CODE
@@:
      CALL PDQInkey          ;Get the next key
      OR   AX,AX             ;Was one waiting?
      JZ   @B                ;Loop until one is ready
      JS   ExtKey            ;Go if it was an extended key
      ;Stay here for an ASCII key
```

PDQInput \SOURCE\PDQINPUT.ASM

P.D.Q. Equivalent: PDQInput

Use

Get a line of input using the DOS Line Input routine.

Calling Convention

```
PUSH Offset of result string descriptor
CALL PDQInput
```

No return value (result returned in passed string).

Notes

Maximum input line length is 127 characters.

This routine uses DOS service 0Ah, so it supports redirection. However, you should never call this routine from a TSR program (use BIOSInput or BIOSInput2 instead).

Example

Get a line of input from user and store it in Work$:

```
Extrn PDQInput:Proc

.DATA?
EVEN
Work$  dd  1 dup(?)       ;Space for string descriptor

.CODE
MOV  AX,Offset Work$      ;Get descriptor address
PUSH AX
CALL PDQInput
```

PDQMessage \SOURCE\PDQMSG.ASM

P.D.Q. Equivalent: PDQMessage

Use

Copy a BASIC error message into a string.

Calling Convention

```
PUSH Offset address of error number
CALL PDQMessage
```

Returns offset of descriptor of string holding the message.

Notes

The returned string and descriptor are in PDQMessage's own data area. The messages are actually stored in the code segment to avoid impacting DGROUP. Only enough DGROUP memory (25 bytes) is reserved to hold a copy of the longest string. Copy the string to your own string space if you need to do further work with it. The previous message will be overwritten with new calls to PDQMessage.

You can add error messages to the list by editing and recompiling the PDQMSG.ASM source code file.

Example

Display error 14, "File not found":

```
.DATA?
MyErrNum  dw  1 dup (?)

.CODE
MOV  [MyErrNum],14        ;Store error number
MOV  AX,Offset MyErrNum   ;Get address of error number
PUSH AX
CALL PDQMessage           ;Get descriptor address in AX
PUSH AX                   ;Pass descriptor address
CALL P$PESD               ;And print it out
```

PDQMonitor \SOURCE\MONITOR.ASM

P.D.Q. Equivalent: PDQMonitor

Use

Determines and reports the type of monitor that is installed. If two adapters and monitors are being used, PDQMonitor reports the one that was currently active the first time it was called.

Calling Convention

```
CALL PDQMonitor
```

Returns monitor type in AX.

Notes

Currently, PDQMonitor can report 11 different types of monitors. See the table describing the return value under PDQMonitor in the reference portion of this manual.

Example

Determine whether a color monitor is being used:

```
        Extrn PDQMonitor:Proc

        .CODE
        CALL  PDQMonitor     ;Get monitor type
        CMP   AX,2           ;Monochrome or herc?
        JLE   NoColor        ;Yep -- it's not color
        CMP   AX,10          ;EGA w/ CGA or 8514/A?
        JAE   Color          ;Yep -- we've got color
        TEST  AX,1           ;Else check color bit
        JZ    NoColor        ;It wasn't color at all
Color:
        ; If we get here, they're using a color monitor
```

PDQMonSetup \SOURCE\MONSETUP.ASM

Use

Sets P$MonSeg and P$CGAPort, two internal variables that are used by other routines to specify the video segment and the CGA snow flag.

Calling Convention

```
CALL PDQMonSetup
```

No return value.

Notes

Once P$MonSeg has a non-zero value, this routine exits without doing anything. If you want print output to go to a different segment, you can change P$MonSeg directly. If you want to force PDQMonSetup to reset P$MonSeg (for example, after switching between a color and monochrome adapter), you can set P$MonSeg to 0 and then call this routine.

Example

Get the video segment for the current video adapter in AX:

```
Extrn PDQMonSetup:Proc

.DATA
Extrn P$MonSeg

.CODE
CALL PDQMonSetup          ;Make sure P$MonSeg is current
MOV  AX,[P$MonSeg]        ;Get value in AX
```

PDQParse \SOURCE\PDQPARSE.ASM

Use

Divide a string into substrings by searching for a specified string delimiter character.

Calling Convention

```
PUSH Offset of descriptor of string to parse
CALL PDQParse
```

Returns AX = Offset of descriptor for next substring.

Notes

See the description of PDQParse in the reference portion of this manual.

The returned string descriptor is in this routine's data space and will be overwritten by successive calls to PDQParse. Copy the string to your own string descriptor with B$SASS if you want to process it further.

Set the external word P$NextChar to zero to perform the function of PDQRestore. Set the external byte P$DelimitChar to the delimiter that you want PDQParse to use.

Example

Print the first entry of the DOS Path statement (also ensure that the default delimiter of ";" is in place):

```
Extrn PDQParse:Proc
Extrn B$FEVS:Proc          ;Gets an environment string

.DATA
EVEN
DefStr Path$, "PATH"
Extrn P$NextChar:Word      ;Set to 0 to start new parse
Extrn P$DelimitChar:Byte   ;Set to the delimiter

.CODE
MOV  P$NextChar,0          ;Restore the parser for new string
MOV  P$DelimitChar,';'     ;Set the delimiting character
MOV  AX,Offset Path$       ;Get address of string descriptor
PUSH AX
CALL B$FEVS                ;Get the path string
MOV  SI,AX                 ;Copy offset
CMP  WordPtr [SI],0        ;Did we find the path?
JE   NoPathFound           ;Go on error
PUSH AX                    ;Pass the string descriptor
CALL PDQParse              ;Get the first directory
PUSH AX                    ;Push descriptor for printing
CALL P$PESD                ;And print it out
```


PDQPrint \SOURCE\PDQPRINT.ASM

P.D.Q. Equivalent: PDQPrint

Use

Prints a string directly to the screen using the specified color attributes.

Calling Convention

```
PUSH Offset of string descriptor
PUSH Offset of row value
PUSH Offset of column value
PUSH Offset of color value
```

No return value.

Notes

This is faster than a standard PRINT operation (using, for example, P$PESD) because it writes directly to screen memory instead of using DOS or the BIOS. On the other hand, the output from PDQPrint cannot be redirected.

This routine calls P$MonSetUp to set the video segment if it hasn't been set previously. It also requires an explicit color setting for the string that it prints.

Example

Print Work$ at Row 5, Column 1, in bright white on blue:

```
Extrn PDQPrint:Proc

.DATA?
EVEN
Work$   dd  1 dup (?)     ;Space for string descriptor
Row     dw  1 dup (?)     ;Space for other parameters
Column  dw  1 dup (?)
Color   dw  1 dup (?)

.CODE
MOV  [Row],5              ;Set row value
MOV  [Column],1           ;  and column
MOV  [Color],1Fh          ;Set color value
MOV  AX,Offset Work$      ;Get offset of string
PUSH AX
MOV  AX,Offset Row        ;Get video row offset
PUSH AX
MOV  AX,Offset Column     ;and video column
PUSH AX
MOV  AX,Offset Color      ;Get color last
PUSH AX
CALL PDQPrint
```

PDQRand \SOURCE\PDQRAND.ASM

Use

Returns a random integer between 0 and a specified limit.

Calling Convention

```
PUSH Offset of limit value
CALL PDQRand
```

Returns AX = random integer.

Notes

The limit and the return value in AX are signed integers with values between 0 and 32,767. See PDQRandomize to set the random number seed if you want to repeat a sequence of random integers.

Example

Get a random number between 1 and 6:

```
Extrn PDQRand:Proc

DATA?
Limit  dw  1 dup (?)      ;Space for the limit

.CODE
MOV  [Limit],5            ;Store the limit value
MOV  AX,Offset Limit      ;Get address of Limit
PUSH AX
CALL PDQRand
INC  AX                   ;1 <= AX <= 6
```

PDQRandomize \SOURCE\PDQRAND.ASM

Use

Set the random number seed used by PDQRand.

Calling Convention

```
PUSH Offset of seed value
CALL PDQRandomize
```

No return value.

Notes

The seed value should be a positive integer in the range of 1 to 32,767. PDQRand uses a seed value of 7,397 by default.

Example

Set the seed value to 100:

```
Extrn PDQRandomize:Proc

.DATA?
Seed  dw  1 dup (?)       ;Space for the seed value

.CODE
MOV  [Seed],100           ;Set new seed value
MOV  AX,Offset Seed       ;Get its address
PUSH AX
CALL PDQRandomize
```

PDQSound \SOURCE\PDQSOUND.ASM

P.D.Q. Equivalent: PDQSound

Use

Play a note of a specified duration through the computer's speaker.

Calling Convention

```
PUSH Offset of frequency in Hz.
PUSH Offset of duration in timer ticks (1/18 second)
CALL PDQSound
```

No return value.

Notes

See the description for PDQSound in the reference portion of this manual for the use of negative duration values. If you use a frequency of 37 Hz. or less, this routine will return immediately and not produce any sound (nor turn off the speaker).

Example

Produce a tone of 440 Hz (concert A) for one second:

```
Extrn PDQSound:Proc

.DATA?
Frequency  dw  1 dup (?)  ;Data space for parameters
Duration   dw  1 dup (?)

.CODE
MOV  [Frequency],440      ;Set frequency value
MOV  [Duration],18        ;A full second
MOV  AX,Offset Frequency  ;Get frequency value
PUSH AX
MOV  AX,Offset Duration   ;and duration value
PUSH AX
CALL PDQSound
```

PDQValL

\SOURCE\PDQVAL.ASM
\SOURCE_PDQVAL.ASM

Synonym: PDQValI

P.D.Q. Equivalent: PDQValI and PDQValL

Use

Return the value of a number stored in a string. This routine can be used both for integers and long integers.

Calling Convention

```
PUSH Offset of descriptor for string containing number
CALL PDQVALL
```

Value returned in DX:AX.

Notes

P.D.Q. uses the same routine for PDQValI and PDQValL. It is up to you to be sure that the string contains a value that can be represented as an integer or a long integer; it does not check for overflow.

If you are certain that the returned value will be less than 65,536 you can ignore DX and use AX only.

See the description of PDQValI and PDQValL in the reference section of this manual.

The versions of these routines in the stub file, _PDQVAL.ASM, do not accept values in &H hexadecimal format nor will they accept positive values with a leading plus sign. Otherwise, they are identical to the regular routines.

Example

Find the long integer value of Work$:

```
Extrn PDQVALL:Proc

.DATA?
EVEN
Work$  dd  1 dup (?)      ;Space for a string descriptor

.CODE
MOV  AX,Offset Work$      ;Get address of string
PUSH AX
CALL PDQVALL              ;Get value in DX:AX
```

PoolOkay \SOURCE\POOLOKAY.ASM

P.D.Q. Equivalent: PoolOkay

Use

Check the integrity of the string pool.

Calling Convention

```
CALL PoolOkay
```

Returns AX = -1 if the string pool is okay, or 0 if it has been corrupted.

Notes

This routine works by walking through the active portion of the string pool. It makes sure that each string's back pointer points to a valid string descriptor that in turn points back to the string data. You may find this routine most valuable during debugging.

Example

Check the integrity of the string pool:

```
Extrn PoolOkay:Proc

.CODE
CALL PoolOkay           ;Check the string pool
OR   AX,AX              ;Is it okay?
JZ   StringError        ;No -- go take care of it
```

PopDeinstall \SOURCE\POPDEINS.ASM

P.D.Q. Equivalent: PopDeinstall

Use

Removes a TSR from memory (after a PopDown if TSR is popped up).

Calling Convention

```
PUSH Offset of DGROUP variable
PUSH Offset of ID$ string descriptor
CALL PopDeinstall
```

Returns AX = 0 if deinstallation failed, or AX = -1 if deinstallation succeeded.

Notes

The "DGROUP" variable is the value returned from a previous call to TSRInstalled. The ID$ string must be the same string as was used for identification when the TSR was installed in memory.

Example

Remove the currently-running TSR from memory:

```
Extrn PopDeinstall:Proc

.DATA
DGRP  dw  ?               ;Space for DGROUP variable
DefStr ID$,"My ID string: version 1.0"

.CODE
MOV  AX,Offset DRGP       ;Get DGROUP variable
PUSH AX
MOV  AX,Offset ID$        ;And ID string
PUSH AX
CALL PopDeinstall         ;Try to remove from memory
OR   AX,AX                ;Test the return
JNZ  Okay                 ;Okay -- go ahead
; We get here if deinstall failed
```

PopDown \SOURCE\POPUP.ASM

P.D.Q. Equivalent: PopDown

Use

Return to underlying application (or DOS prompt) from a TSR.

Calling Convention

```
CALL PopDown
```

No return.

Notes

This routine does not return to your code. Call this routine to end a TSR session that was begun by P.D.Q. code.

Example

End a popup session in a TSR:

```
Extrn PopDown:Proc

.CODE
```

```
CALL PopDown              ;Give back control
; We'll never get here
```

PopRequest \SOURCE\POPREQST.ASM

P.D.Q. Equivalent: PopRequest

Use

Requests that the current TSR pop up.

Calling Convention

```
PUSH Offset of Flag word
PUSH Offset of Timerticks word
CALL PopRequest
```

Returns AX = 0 if request failed (if another request was pending), or AX = -1 if the request has been registered.

Notes

The Flag is a word that is set to 0 (False) by PopRequest. When (if) the TSR eventually pops up, the Flag will be set to a non-zero value to show that this request caused the popup.

TimerTicks is an unsigned word which designates how long this request should stay in effect. Each tick is approximately 1/18th of second. If you specify the largest possible value (65,535) the request will stay in effect for about an hour.

PopRequest is normally called from an interrupt handler that has detected a situation that necessitates that the program pop up. A keyboard popup request is the only thing that takes precedence over a request registered with this procedure.

Example

Ask that the program pop up in the next 5 seconds:

```
Extrn PopRequest:Proc

.DATA
PopupFlag   dw  ?         ;Value is unimportant
TimerTicks  dw  18 * 5    ;Time to wait

.CODE
MOV  AX,Offset PopupFlag  ;Get address of flag
PUSH AX
MOV  AX,Offset TimerTicks ;And address of tick count
```

```
PUSH AX
CALL PopRequest
OR   AX                  ;Was request acknowledged?
JZ   PopReqFailed        ;No--go
;  Popup request was accepted
```

PopUpHere \SOURCE\POPUP.ASM

P.D.Q. Equivalent: PopUpHere

Use

Sets the popup address and does a bunch of housekeeping for a P.D.Q. simplified TSR program.

Calling Convention

```
PUSH Offset of HotKey and Shiftmask word
PUSH Offset of ID$ string descriptor
CALL PopUpHere
JMP  somewhere in this code segment (see below)
```

No return value.

Notes

Store the hot key and shift mask as **ShiftMask * 256 + scan code**. See the information about PopUpHere in the reference portion of this manual for more information.

The Jmp instruction should be followed by a full word address. That is, it must be a NEAR (not a SHORT or FAR) jump to the rest of your initialization code. PopUpHere depends on the length of the jump to set the correct popup address. See SCRNCAP.ASM for an example of using a Nop to guarantee that at least three bytes are reserved.

Example

Set up a simplified TSR (also see the example programs on the distribution disk) to pop up on Alt-S:

```
Extrn PopUpHere:Proc

.DATA?
HotKey  dw  1 dup (?)     ;Place for Hotkey definition

.DATA
DefStr ID$, "My ID string: version 1.0"

.CODE
MOV  [HotKey],81Fh        ;Shiftmask = 8: Alt key
```

```
MOV  AX,Offset HotKey     ;Scan code = 1Fh: "S"
PUSH AX
MOV  AX,Offset ID$        ;Get address of ID$
PUSH AX
CALL PopUpHere            ;Set Popup address
JMP  Continue_Install
;  this begins the popup code
```

Power \SOURCE\POWER.ASM

Use

Raise an integer (the mantissa) to an integer power (the exponent).

Calling Convention

```
PUSH Offset of mantissa
PUSH Offset of exponent
CALL Power
```

Returns result in DX:AX.

Notes

This routine returns 0:0 in DX:AX in case of an overflow error.

Example

Calculate 12 ^ 5:

```
Extrn Power:Proc

.DATA?
Mantissa  dw  1 dup (?)   ;Room for the arguments
Exponent  dw  1 dup (?)

.CODE
MOV  [Mantissa],12        ;Set values in place
MOV  [Exponent],5
MOV  AX,Offset Mantissa   ;Get the first argument
PUSH AX
MOV  AX,Offset Exponent   ;and the second
PUSH AX
CALL Power                ;Result (248832) in DX:AX
```

Power2 \SOURCE\POWER2.ASM

Use

Calculates 2 ^ n for n values between 0 and 31.

Calling Convention

```
PUSH Offset of n
CALL Power2
```

Returns result in DX:AX.

Notes

2 ^ 31 will return with the high bit of DX set. Technically, this is a negative number (if DX:AX is taken as a signed long integer). But if you interpret it as an unsigned integer, the result is correct.

Example

Calculate 2 ^ 12:

```
Extrn Power2:Proc

.DATA?
n  dw  1 dup (?)          ;Room for the argument

.CODE
MOV  [n],12               ;Set the argument value
MOV  AX,Offset n          ;Get argument address
PUSH AX
CALL Power2               ;The result (4096) is in DX:AX
```

Sort

\SOURCE\SORT.ASM
\SOURCE_SORT.ASM

P.D.Q. Equivalent: Sort

Use

Sorts a string array in ascending or descending order.

Calling Convention

```
PUSH Offset of first array element descriptor to include in sort
PUSH Offset of number of elements to sort
PUSH Offset of sort direction word
CALL Sort
```

No return value.

Notes

Use zero as the sort direction for an ascending sort, or any other value for a descending sort.

Note that the number of elements and the sort direction are passed by reference, not by value.

In a string array, the descriptors are in an array table. The text of each string is in the string pool. If you are sorting the entire array, pass the address of the first item. You can find the address of the first item at offset 0Ah in the array descriptor.

The version in _SORT.ASM is identical except that it is slower in exchange for less code. It uses B$SCMP to compare strings, and B$SCMP has the overhead of checking every string to see if it is temporary and deleting it if so. Of course, no string descriptors in your array will be temporary, so the check is simply wasted time during the sort.

Example

Sort the first 10 elements of the single-dimension string array Array$ into ascending order:

```
Extrn Sort:Proc

.DATA?
Array$     db  16 dup (?)          ;Room for array descriptor

.DATA
Size       dw  10                  ;Elements to sort
Direction  dw  0                   ;0 = ascending

.CODE
MOV  AX,Word ptr [Array$ + 10]     ;Get offset of first
                                   ;  string descriptor
PUSH AX
MOV  AX,Offset Size                ;Point to size to sort
PUSH AX
MOV  AX,Offset Direction           ;Point to sort direction
PUSH AX
CALL Sort                          ;Sort it all
```

StuffBuf \SOURCE\STUFFBUF.ASM

P.D.Q. Equivalent: StuffBuf

Use

Places up to 15 key strokes in the computer's type-ahead buffer as if they had been typed from the keyboard.

Calling Convention

```
PUSH Offset of string descriptor
CALL StuffBuf
```

No return value.

Notes

The string may contain no more than 15 key strokes. Extended keys are represented by a CHR$(0) followed by the key's scan code.

Example

Place "Testing", an up arrow key, and "123" into the buffer:

```
.DATA
DefStr StufChar$, <"Testing", 0, 72, "123">

.CODE
MOV  AX,Offset StufChar$  ;Pointer to descriptor
PUSH AX
CALL StuffBuf             ;Characters in keyboard buffer
```

Swap2Disk \SOURCE\SWAP2DSK.ASM

P.D.Q. Equivalent: Swap2Disk

Use

When invoked before calling EndTSR, tells the popup handler to swap the program's code and data to a disk file while inactive.

Calling Convention

```
PUSH Offset of string descriptor for swap file name
PUSH Offset of integer program ID
CALL Swap2Disk
```

Returns success or failure (-1 or 0) in AX. Also sets P$PDQErr to a BASIC error number, or clears it to zero if no error occurred.

Notes

The only reason Swap2Disk will return 0 is if it is unable to locate sufficient hard disk space to hold the program, or if the file name is invalid.

Example

```
.DATA
EVEN
ProgramID  DW  0
DefStr SwapFile, <"PROGRAM.SWP">
DefStr ErrorMsg, <"Unable to swap to disk.">

.CODE
MOV  AX,Offset SwapFile          ;pass the name of the swap file
PUSH AX
MOV  AX,Offset ProgramID         ;and the address of the ID
PUSH AX
```

```
CALL Swap2Disk                  ;try to enable disk swapping
OR   AX,AX                      ;did it work?
JNZ  Success                    ;yes, continue
MOV  AX,Offset ErrorMsg         ;no, print the error message
PUSH AX
CALL B$PESD
```

Swap2EMS \SOURCE\SWAP2EMS.ASM

P.D.Q. Equivalent: Swap2EMS

Use

When invoked before calling EndTSR, tells the popup handler to swap the program's code and data to expanded memory while inactive.

Calling Convention

```
PUSH Offset of integer program ID
CALL Swap2EMS
```

Returns success or failure (-1 or 0) in AX.

Notes

Swap2EMS will return 0 if there is insufficient expanded memory available, or no expanded memory at all.

Example

```
.DATA
EVEN
ProgramID  DW  0
DefStr ErrorMsg, <"Unable to swap to EMS.">

.CODE
MOV  AX,Offset ProgramID        ;pass along the program ID
PUSH AX
CALL Swap2EMS                   ;try to enable swapping to EMS
OR   AX,AX                      ;did it work?
JNZ  Success                    ;yes, continue
MOV  AX,Offset ErrorMsg         ;no, print the error message
PUSH AX
CALL B$PESD
```

SwapCode \SOURCE\SWAPCODE.ASM

P.D.Q. Equivalent: SwapCode

Use

Retrieves the optional code passed in BX if CALL INTERRUPT was used to invoke the swapping TSR program.

Calling Convention

```
CALL SwapCode
```

AX holds the code number.

Notes

See the section *TSR Programs That Swap To Disk Or EMS* for information on passing additional information to a swapping TSR.

Example

Retrieve the code passed in BX by a calling program:

```
Extrn SwapCode:  Proc

.Code
CALL SwapCode
OR   AX, AX              ;Did the caller pass a code?
JZ   NoCode              ;No
```

TestHotKey \SOURCE\TESTKEY.ASM

P.D.Q. Equivalent: TestHotKey

Use

Tests whether the last key pressed matches a given scan code and shift mask.

Calling Convention

```
PUSH Offset of Argument
CALL TestHotKey
```

If AX = -1, then the key matches; if AX = 0, then no match.

Notes

The Argument value is **Shiftmask * 256 + Scancode**. Notice that the address of Argument is passed to TestHotKey, not the value itself.

Example

Test whether Alt-S was the last key pressed:

```
Extrn TestHotKey:Proc

.DATA
Arg  dw  8 * 256 + 1Fh    ;Alt mask is 8, S has scan code
                          ;  of 1Fh
.CODE
MOV  AX,Offset Arg        ;Get address of argument
PUSH AX
CALL TestHotKey           ;Was it pressed?
OR   AX,AX                ;Test the result
JNZ  AltSPressed          ;Yes it was pressed -- go
```

TSRFileOff \SOURCE\TSRFILE.ASM

P.D.Q. Equivalent: TSRFileOff

Use

Restores the foreground application's PSP and DTA after a call to TSRFileOn.

Calling Convention

```
CALL TSRFileOff
```

No return value.

Notes

TSRFileOff and TSRFileOn are not needed in a P.D.Q. simplified TSR program.

This routine assumes that it is safe to use DOS services 2Fh, 50h, and 51h—that is, it assumes it is safe to call DOS.

Example

Return to the original PSP and DTA after using file services in a TSR program:

```
Extrn TSRFileOff:Proc

.CODE
CALL TSRFileOff           ;Make the switch
```

TSRFileOn \SOURCE\TSRFILE.ASM

P.D.Q. Equivalent: TSRFileOn

Use

This routine is used in a non-simplified TSR to switch to the local PSP and DTA before performing DOS file services. This guarantees that the TSR can open and access files without interfering with the foreground application.

Calling Convention

```
CALL TSRFileOn
```

No return value.

Notes

Use TSRFileOff to restore the original PSP and DTA before returning to the foreground application. If you don't, you may destroy files or even a directory structure.

This routine assumes that it is safe to use DOS services 2Fh, 50h, and 51h—that is, it assumes it can safely call DOS.

Example

Prepare to use file or device I/O in a TSR:

```
Extrn TSRFileOn:Proc

.CODE
CALL TSRFileOn          ;Switch to local file structures
```


TSRInstalled \SOURCE\TSRINST.ASM

P.D.Q. Equivalent: TSRInstalled

Use

Find an installed (resident) copy of a TSR program in memory.

Calling Convention

```
PUSH Offset of ID$ descriptor
CALL TSRInstalled
```

Returns AX = DGROUP (data segment of installed copy of the program), or AX = 0 if the program has not been previously installed.

Notes

The DGROUP value is needed to call DeinstallTSR and PopDeinstall later to remove a TSR program from memory.

You should call TSRInstalled before going resident (before calling EndTSR) if you don't want more than one copy of the TSR to be installed in memory at one time. If the return value is not 0, you can avoid calling EndTSR and display a warning message instead.

See the SCRNCAP.ASM demonstration program for a complete example of detecting prior installation.

Example

Check whether the current TSR program is already resident in memory:

```
Extrn TSRInstalled:Proc

.DATA
DefStr ID$, "MyTSR Version 9.75"

.CODE
MOV  AX,Offset ID$        ;Point to string descriptor
CALL TSRInstalled         ;Is another copy resident?
OR   AX,AX                ;0 means no
JNZ  AlreadyInstalled     ;Oops -- go
;  If we get here, another copy of this program is not
;  already in memory.
```

UnhookInt0 \SOURCE\HOOKINT0.ASM

Use

Releases the Interrupt 0 trap established by a call to HookInt0, and lets DOS handle any subsequent division by zero errors.

Calling Convention

```
CALL UnhookInt0
```

No return value.

Notes

If you happen to call UnhookInt0 when the P.D.Q. Interrupt 0 trap is not active, this routine simply returns without doing anything.

Example

```
Extrn UnhookInt0:Proc

.CODE
CALL UnhookInt0          ;That was easy, wasn't it?
```

_FLUSH

\SOURCE\FLUSH.ASM
\SOURCE_FLUSH.ASM

P.D.Q. Equivalent: Flush

Use

Flushes all open BASIC files; that is, it makes sure that all pending changes are written to disk and the directory entries are updated without having to close and then reopen the files.

Calling Convention

```
PUSH Each of the file numbers to flush
CALL _FLUSH
ADD  SP,(number of files passed) * 2
```

No return value. May report an error by calling P$DoError.

Notes

To flush all open files do not pass any arguments, and also do not include an **Add SP,n** instruction after the call.

_FLUSH is written as a C-style routine to accept a variable number of arguments. It determines the number of arguments that it receives by looking for the instruction **Add SP,n** at the return address. If that instruction is present, _FLUSH divides the *n* value by 2 and assumes that it has received that number of arguments. If that instruction is not present, _FLUSH flushes all open file buffers. _FLUSH will work correctly only if you include or omit the **Add SP,n** instruction to tell it how many arguments it has received.

_FLUSH works by calling two DOS services. It uses Int 21h, Service 45h to create a duplicate file handle for an open file. Then it calls Int 21h, Service 3Eh to close the duplicate handle. For this method to work, there must be at least one file handle available.

The version of this routine in the file _FLUSH.ASM flushes all open files. To reduce code size it does not accept any arguments.

Example

```
FLUSH BASIC file #3:

Extrn _FLUSH:Proc

.CODE
MOV  AX,3              ;Get file number
PUSH AX                ;  and send it on
CALL _FLUSH            ;Flush that file
ADD  SP,2              ;Remove file number from stack
```

Undocumented Procedures

The following procedures from the P.D.Q. library are not fully documented in this chapter. We're not trying to hide anything from you; rather, we simply believe that these routines are less important for assembly language programmers than the preceding group which we have documented fully.

Many of these procedures perform actions which your programs can do more easily and quickly without the help of a library routine. There is little reason, for example, to call a special routine to place a 2-byte value into memory somewhere.

If you do want to call one of these procedures, or if you just want to study our code, read the associated source code files. Most of the source files are fully commented and you should have little trouble understanding how to call these procedures or how to emulate their activities yourself.

A few procedures which are entirely internal to P.D.Q., and which probably have no usefulness in an assembly-language program, are not listed here at all. Routines that handle the BASIC 7 Currency data type are also omitted.

PROCEDURE NAME	P.D.Q./QB EQUIVALENT	SOURCE FILE
Absolute	**Call Absolute**	**\SOURCE\ABSOLUTE.ASM**

Assembly language routines can call each other directly. There is no reason to use this mechanism.

Allocmem	**Allocmem**	**\SOURCE\ALLOCMEM.ASM**

P.D.Q. uses this procedure to request a block of far memory from DOS. You can make the call to DOS service 48H of Int 21H yourself more quickly.

PROCEDURE NAME	P.D.Q./QB EQUIVALENT	SOURCE FILE
B$BEEP	**Beep**	**\SOURCE\BEEP.ASM**

This procedure sends a beep to the speaker by calling the video BIOS (Int 10h, service 0Eh, with 7 in the AL register).

B$CENC	**END() (BASIC7)**	**\SOURCE\ENDBC7.ASM**
B$STP1	**STOP() (BASIC7)**	**\SOURCE\ENDBC7.ASM**
EndLevel	**EndLevel (P.D.Q.)**	**\SOURCE\ENDLEVEL.ASM**

These three routines store the errorlevel value they receive into the data byte called P$TermCode and then call B$CEND to clean up and end the program.

B$CHDR	**CHDRIVE (BASIC 7)**	**\SOURCE\CHDRIVE.ASM**

This procedure implements BASIC 7's CHDRIVE statement to select a new default drive. You will probably want to call Int 21h, service 0Eh directly instead of waiting for this procedure to extract the drive number from a letter in a BASIC string and then make the same call.

B$CHOU	**PRINT #n,**	**\SOURCE\PRNHANDL.ASM**

This routine puts the DOS handle number of the device to receive output from PRINT into the external word P$PrintHandle.

B$CPI4 (386 version) **\SOURCE\COMPAR43.ASM**

This is a long integer comparison routine for 386 and 486 CPUs. If you are writing your program with 386 instructions enabled, you can perform this task directly without the call. If not, you can't use this routine. However, see the normal B$CPI4 in the reference section for an 8088 and 80286 long integer comparison.

B$DIV4 (386 version) **\SOURCE\DIVLONG3.ASM**

This divides a 4-byte long integer by another long integer using the extended registers of the 386 processor. If you are programming on and for a 386, you can do the operation yourself much more quickly than setting up for and calling this routine.

B$DSEG	**DEF SEG**	**\SOURCE\DEFSEG.ASM**

All DEF SEG does is set a segment address in an external word-sized variable called B$SEG. You will save time by setting the variable yourself. If you want to return to the default data segment—equivalent to DEF SEG without an argument—simply use **Mov B$Seg,DS**.

PROCEDURE NAME	P.D.Q./QB EQUIVALENT	SOURCE FILE
B$DWID	WIDTH (device)	\SOURCE\WIDTH2.ASM
B$FWID	WIDTH (file number)	

Neither of these routines do anything except return because the P.D.Q. print routines do not automatically add a carriage return/line feed every 80 characters like BASIC does.

B$ENFA **\SOURCE\DEFFN.ASM**
B$EXFA

These routines are called on entry to and exit from a DEF FN function. Their purpose in BASIC is to check for sufficient stack space for the function. B$ENFA also calls B$LINA for debugging purposes.

B$ENRA **\SOURCE\SUBRECUR.ASM**

This routine is called upon entry to a recursive SUB or FUNCTION to create an appropriate stack frame.

B$ENSA **\SOURCE\SUBSTAT.ASM**

This routine is called upon entry to a STATIC SUB or FUNCTION to create the necessary stack frame.

B$EXSA **EXIT SUB/FUNCTION** **\SOURCE\EXITSUB.ASM**

This routine does the cleanup necessary to exit from a subprogram or function early. It should never be necessary in an assembly-language program.

B$EXTS **EXIT Function** **\SOURCE\EXITFUNC.ASM**

This routine performs a RET and nothing else. It is used in P.D.Q. to exit a BASIC 7 function when ON ERROR is used.

B$FCVD **\FPSOURCE\CVS.ASM**
B$FCVS

These two routines convert the field buffer form of a single-precision or double-precision number into numeric form. They return a pointer to the number in AX. What they really do is read the address from a string descriptor into AX and return that address as the pointer to the number.

B$FIL2** **\FPSOURCE\B$FILD.ASM

Converts a 2-byte integer in AX into a 4-byte integer in DX:AX with a single CWD instruction and then falls into B$FILD.

B$FILD** **\FPSOURCE\B$FILD.ASM

Pushes a 4-byte integer in DX:AX onto the floating point stack.

PROCEDURE NAME	P.D.Q./QB EQUIVALENT	SOURCE FILE
B$FIS2		**\FPSOURCE\B$FIST.ASM**

Converts the floating point value in ST(0) into a 2-byte integer, and returns the integer in AX.

B$FIST		**\FPSOURCE\B$FIST.ASM**

Converts the floating point value in ST(0) into a 4-byte integer, and returns the integer in DX:AX.

B$FIX4	**FIX()**	**\FPSOURCE\B$FIXINT.ASM**
B$FIX8	**FIX()**	
B$INT4	**INT()**	
B$INT8	**INT()**	

Truncates (B$FIX*n*) or rounds (B$INT*n*) the floating point value in ST(0). The value is left in ST(0) for further use.

B$FLEN	**LEN**	**\SOURCE\LEN.ASM**

The first word of a string descriptor holds the length of the string. Just read the length directly.

B$FPOS	**POS(0)**	**\SOURCE\POS0.ASM**

This routine uses BIOS routines to find the current cursor position.

B$LINA		**\SOURCE\DEBUG.ASM**

This routine supports the BC compiler's /d switch to make a program perform additional runtime error checking for debugging purposes.

B$LPRT	**LPRINT**	**\SOURCE\LPRINT.ASM**

To send text to the printer, BASIC calls B$LPRT to set up printer output and then one of the PRINT routines to actually do the output. The output is directed by an external data word called P$PrintHandle. You can store a value of 4 in that word and then call a PRINT routine yourself instead of calling B$LPRT.

B$OEGA	**ON ERROR GOTO**	**\SOURCE\ONERROR.ASM**
B$RESA	**RESUME address**	**\SOURCE\RESUME.ASM**

You can use ON ERROR without any problems in an assembly language program. However, RESUME to an address assumes that BASIC's system of SUBs and FUNCTIONS is being used, and won't work properly in a normal assembly language program.

PROCEDURE NAME	P.D.Q./QB EQUIVALENT	SOURCE FILE
B$RSTA	**RESTORE**	**\SOURCE\RESTORE.ASM**
B$RSTB	**RESTORE**	**\SOURCE\RESTORE.ASM**

Moves the READ/DATA pointer to the beginning of the data list or to a new location in the data list. See the comment about READ and DATA below under R$READDATA.

B$SADD STRINGADDRESS	**SADD**	**\SOURCE\SADD.ASM**

Gets the address of a string by looking in the second word of its string descriptor and returning DS:address in DX:AX. You can do the same thing without the overhead of a call.

B$SETM	**SETMEM**	**\SOURCE\SETMEM.ASM**

Returns without doing anything. This is a dummy routine that is included in P.D.Q. for compatibility purposes.

B$SSEG	**SSEG (BASIC 7)**	**\SOURCE\SSEG.ASM**

This routine simply copies DS into AX and then returns.

BlockCopy	**\SOURCE\BLOCKOPY.ASM**

This procedure copies a block of memory from one location to another. You can save time and code by setting up DS:SI, ES:DI, and CX yourself and using MOVS.

B_OnExit	**\SOURCE\B_ONEXIT.ASM**

P.D.Q. uses this routine to register up to 10 routines for clean-up processing as a program is ending. Since you have complete control over program termination in an assembly language program (and since you may not be calling B$CEND to terminate your program), you probably won't find this routine useful. Full calling instructions are in the source code file if you need them.

CallOldInt	**CallOldInt**	**\SOURCE\CALLINT.ASM**

This procedure is used to give BASIC programs access to interrupts. It requires setting up a user TYPE with the necessary register values, and reading those values back out of the TYPE when the call returns. It is much easier to make direct calls to the computer's interrupts yourself from assembly language.

PROCEDURE NAME	P.D.Q./QB EQUIVALENT	SOURCE FILE
ColorSave	**COLORSAVE**	**\SOURCE\COLORSR.ASM**
ColorRest	**COLORREST**	

These two routines retrieve and store the current color value used by CLS and PDQCPrint. The color value is stored in an external data byte called P$Color. You can more quickly read or set P$Color yourself (see the B$COLR procedure for information about P$Color).

DOSVer	**DOSVer**	**\SOURCE\DOSVER.ASM**

P.D.Q. reads the DOS version number during program initialization and stores it in an external data word called P$DOSVer. You can read that value directly.

Get1Byte	**Get1Byte (P.D.Q.)**	**\SOURCE\GET1BYTE.ASM**

This routine reads a single byte from a specific memory location, given the segment and offset + 1 (the element number).

Get1Word	**Get1Word (P.D.Q.)**	**\SOURCE\GET1WORD.ASM**

This routine reads a word from a specific memory location into AX, given the segment and (offset / 2) + 1 (the element number).

GetSeg	**GetSeg (P.D.Q.)**	**\SOURCE\GETSEG.ASM**

This routine returns the current segment set by DEF SEG. That value is stored in an external data word called B$Seg, which you can read (and write) directly.

Interrupt	**Interrupt (P.D.Q.)**	**\SOURCE\INTRPT.ASM**
InterruptX	**InterruptX (P.D.Q.)**	**\SOURCE\INTRPTX.ASM**

It is much easier in an assembly language program (and much faster) to call interrupts directly instead of through these routines.

P$DoError **\SOURCE\DOERROR.ASM**

This routine is called when an error occurs in another library routine to save the error number and jump to the correct location if ON ERROR is in effect. You would need to call this routine only if you were adding library routines that could report an error.

P$TempStr **\SOURCE\TEMPSTR.ASM**

This routine finds free space in the string pool to match the requested space for a new string. If necessary, it calls the pool compaction routine. It does not create a back pointer or a string descriptor.

PDQPeek2	**PDQPeek2**	**\SOURCE\PDQPEEK2.ASM**

Returns a 2-byte value from the DEF SEG segment.

PROCEDURE NAME	P.D.Q./QB EQUIVALENT	SOURCE FILE
PDQPoke2	**PDQPoke2**	**\SOURCE\PDQPOKE2.ASM**

Pokes a 2-byte value into the DEF SEG segment. You can do the same more easily by addressing memory directly.

PDQRestore	**PDQRestore**	**\SOURCE\PDQREST.ASM**

See the description of PDQParse to set the restore word directly.

PDQSetMonSeg	**PDQSetMonSeg**	**\SOURCE\PDQSETMS.ASM**

Sets the value you pass to it in the external word P$MonSeg.

PDQSetWidth	**\SOURCE\PDQSETWD.ASM**

Sets a value equal to twice the screen width in the external word P$PrintWidth for use by PDQPrint and PDQCPrint.

PDQShl	**\SOURCE\PDQSHL.ASM**

Shifts a word value left by a specified number of bytes.

PDQShr	**\SOURCE\PDQSHR.ASM**

Shifts a word value right by a specified number of bytes.

PDQTimer	**PDQTimer (P.D.Q.)**	**\SOURCE\PDQTIMER.ASM**

Reads the number of timer ticks stored in the BIOS data area in the double word at 0:46Ch.

PointIntHere	**PointIntHere**	**\SOURCE\POINTINT.ASM**
HookInt	**UnhookInt**	**\SOURCE\UNHOOK.ASM**

These routines are necessary in a P.D.Q. BASIC TSR program to get and set the address of an interrupt handler in your program. It is much easier to set and restore the addresses yourself in assembly language, because you can find the segment and offset of a routine directly.

R$READDATA	**READ**	**\SOURCE\READ.ASM**

When you use DATA and READ statements in a BASIC program, each value is stored twice: once in ASCII form in the DATA statement, and once in a normal variable. Assembly language DB, DW, etc. statements let you avoid this duplication and wasted program space.

ReleaseMem	**ReleaseMem (P.D.Q.)**	**\SOURCE\RELMEM.ASM**

Releases memory allocated with ALLOCMEM. Just call DOS Int 21h, service 49h yourself to save some time.

ResetKeyboard	**\SOURCE\RESETKBD.ASM**

Resets the keyboard and 8259 PIC chip.

PROCEDURE NAME	P.D.Q./QB EQUIVALENT	SOURCE FILE
SeekLoc	SeekLoc (P.D.Q.)	\SOURCE\SEEKLOC.ASM

SeekLoc simply calculates ((RecNumber - 1) * RecLength) + 1.

Set1Byte	Set1Byte (P.D.Q.)	\SOURCE\SET1BYTE.ASM
Set1Long	Set1Long (P.D.Q.)	\SOURCE\SET1LONG.ASM
Set1Type	Set1Type (P.D.Q.)	\SOURCE\SET1TYPE.ASM
Set1Word	Set1Word (P.D.Q.)	\SOURCE\SET1WORD.ASM

All four routines simply set bytes in memory, given the appropriate segment, offset, and value. You already have this capability in assembly language, and will only slow your programs down if you call these routines.

SetDelimitChar	SetDelimitChar	\SOURCE\SETDELIM.ASM

Sets the character you specify in the byte variable called P$DelimitChar.

StringShort	StringShort (P.D.Q.)	\SOURCE\STRSHORT.ASM

This routine simply returns the contents of the word variable P$BytesShort.

StringUsed	StringUsed (P.D.Q.)	\SOURCE\STRUSED.ASM

This routine simply returns the contents of the variable P$BytesUsed.

Appendices

Appendices

Appendix A: How We Did It

You may be wondering how we are able to achieve such impressive size and performance improvements. Or perhaps even more important, why Microsoft has not seen fit to optimize their BASIC this way. In order to appreciate these improvements and how they are possible—especially within the context of an add-on library—we must first understand how a compiler operates. The discussion that follows requires only an understanding of BASIC programming concepts such as simple math operations, GOTO, and CALL.

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. Even programmers that write in assembly language are shielded to some extent from the low-level details of CPU instruction bytes and memory addresses. Like BASIC, an assembly language program can refer to memory variables using names the programmer makes up, and the assembler itself will keep track of which memory locations they are assigned to.

High level languages such as QuickBASIC add an extra layer of insulation between the programmer and the microprocessor. With BASIC, the BC.EXE compiler reads your BASIC source file, and translates the instructions into the equivalent assembly language statements. For simple operations such as **X = X + 1**, the compiler can create a direct translation—in this case **INC WORD PTR [X]**. The INC instruction tells the PC's processor to increment (add 1 to) the word-sized memory location named "X". Other simple operations such as subtraction, multiplication, division, and integer assignments are likewise translated directly.

Statements such as PRINT, INPUT, and STRING$ that perform more complicated actions are instead converted into calls to the BASIC runtime language library. Even though these could be translated directly to assembly language statements, that would be very inefficient. Suppose, for the sake of argument, that the code to print an integer variable comprises 300 bytes. Generating that code in-line would add the same 300 bytes repeatedly to your program every time you printed a number. Clearly, using subroutines is a better method. And this is where P.D.Q. comes in, because P.D.Q. is a rewrite of the BASIC libraries. When you use BASIC commands that call library routines, your program is really calling the P.D.Q. versions instead of Microsoft's.

It is important to point out the use of subroutines—even in assembly language programs—in this discussion. Many people mistakenly believe that BASIC programs (or those written in any high-level language for that matter) are large and slow simply because they are compiled, and then linked to a library of subroutines. Nothing could be further from the truth. Indeed, Microsoft BASIC is an outstanding compiler, and in many cases the code it generates is as good as a human hand-coding in assembly language. The real problem with BASIC is the enormous amount of code that is added to the start of each program, and additional code to protect you from runtime errors. With that in mind, let's consider some of the problems facing users of traditional high-level languages, and how P.D.Q. can overcome them.

Traditional Programming Languages

The overwhelming problem with most high-level languages is the sheer size of the resultant code. For example, a QuickBASIC 4.5 program that consists solely of an END statement produces a final .EXE file size of nearly 10K. Once you begin adding statements, the program size rises considerably.

There are several contributors to excessive program size with regular BASIC. One is the mandatory code that is called at the beginning of every program, to query the installed hardware and current display mode. Additional code is provided to support linking with C language subroutines. Another factor is the manner in which the language library is implemented. Many different assembly language subroutines are required to support BASIC's language statements. But rather than place each routine into its own object file, Microsoft has chosen to group like routines within the same file. These routines are thus forever joined together.

This joining is called *granularity*, and it causes routines that you may not need to be added to your program. A QuickBASIC 4 program that uses CLS will also receive the code for COLOR, CSRLIN, POS(0), LOCATE, and the function form of SCREEN. Further, many of these routines add more capability than most programs need. For example, the display commands can work in both text and graphics mode.

Yet another problem is the inordinate amount of hand-holding that is present in Microsoft BASIC. Where a simple statement to locate the cursor could be translated into only a few machine instructions, additional code is added to check the monitor type and current video mode, to ensure that the LOCATE parameters are within a legal range.

The P.D.Q. library solves the granularity problem elegantly, by placing each language statement subroutine into its own source file. Further, there is almost no error trapping in any of the P.D.Q. subroutines. That is, if you try to locate the cursor to, say, row 300 and column -75, the P.D.Q. LOCATE routine will happily oblige and pass the values on to the BIOS. Fortunately, the BIOS simply ignores your request and leaves the cursor where it is.

Appendices

Appendix B: Graphics Programming With P.D.Q.

Although we have decided not to add graphics capabilities to P.D.Q. by supporting LINE, CIRCLE, and so forth, we have included some simple routines that work in the EGA and VGA screen modes only. P.D.Q. does support the SCREEN statement to switch video modes, so you can use that to enter graphics and then return to text mode later. Of course, for serious graphics work we recommend our Graphics Workshop library. This product includes many assembly language routines for incorporating high-performance graphics into BASIC programs including those linked with P.D.Q.

All of the positioning parameters for these routines are expressed as pixels, and the maximum range depends on the video mode you are using. For SCREEN 9 the valid pixel values range from 0 to 639 horizontally, and 0 to 349 vertically. VGA SCREEN 12 allows more pixels vertically, and the legal range is between 0 and 479. The color parameter may be any value from 0 through 15 inclusive.

Each routine is provided as a separate BASIC source file having the same name as the subprogram it contains. The files are kept separate so you can add only those capabilities that your program actually needs. These routines are described briefly below, and you should also look at DEMOEGA.BAS for an example of using them.

EGABox draws a single-line or double-line box, and is called as follows:

```
CALL EGABox(ULRow%, ULCol%, LRRow%, LRCol%, Colr%, Style%)
```

Where ULRow%, ULCol%, LRRow%, and LRCol% define the box borders in pixels, Colr% is the box color, and Style% is either 1 for a single-line box, or 2 for a double-line box.

EGADot plots a single point, and is called like this:

```
CALL EGADot(X%, Y%, Colr%)
```

Where X% and Y% indicate the plot position in pixels, and Colr% is the dot color ranging from 0 to 15.

EGAEllipse draws ellipses and circles, using the following syntax:

```
CALL EGAEllipse(X%, Y%, RadiusWide%, RadiusHigh%, Colr%)
```

Where X% and Y% describe the center of the ellipse in pixels, RadiusWide% and RadiusHigh% are the width and height in pixels, and Colr% is the color ranging from 0 to 15.

EGALine draws a single line, as follows:

```
CALL EGALine(X0%, Y0%, X1%, Y1%, Colr%)
```

Where X0% and Y0% indicate the line's starting point, X1% and Y1% are the line's ending point, and Colr% is the line color.

EGAPrint is a graphics-mode print routine, and it begins printing at the current cursor location (use LOCATE to position the text). The syntax is:

```
CALL EGAPrint(Work$, Colr%)
```

Where Work$ is the text to print, and Colr% is the text color.

Appendix C: Debugging P.D.Q. Programs

Because P.D.Q. performs little or no runtime error checking, it is possible to create a program that locks up or reboots your PC without any indication as to what went wrong. Other times a program may not crash, but it also won't work as expected. Even if a program works perfectly when linked with the regular BASIC libraries, it may not work at first when linked with PDQ.LIB.

As a first step, carefully review the section *Differences Between P.D.Q. And Microsoft BASIC*. There are a number of subtle differences that you must be aware of, and it is difficult to remember them all! At the time you are having a problem, go back and reread that section to see if any of these differences might be affecting your program. Also, be sure that your program does in fact run correctly when linked with the regular BASIC libraries, before assuming the problem is related to P.D.Q. Finally, if you are using BASIC 7 PDS you *must* link with the BASIC7.LIB file as described in the section *Compiling and Linking*.

Using /D

The single most effective way to track down errant behavior in a P.D.Q. program is to compile it using the /d (Debug) compiler switch. This tells BASIC to add a call to a special error detection routine just before each BASIC statement. That is, if you have code like this:

```
Work$ = "Testing 1, 2, 3"
PRINT Work$
PRINT LEN(Work$)
```

BASIC creates code similar to this:

```
CALL CheckForError
Work$ = "Testing 1, 2, 3"
CALL CheckForError
PRINT Work$
CALL CheckForError
PRINT LEN(Work$)
```

Using /d tells BASIC to test all array accesses, to ensure that the specified element numbers are legal. Without /d, reading from or writing to an element that doesn't exist is not trapped, and memory is accessed that shouldn't be. When reading an invalid array element, you simply get whatever random nonsense happens to be in memory. But when writing past the end of an array you will likely overwrite data incorrectly, or worse, destroy code that will crash when later executed.

Another condition using /d will assist is running out of stack space. The stack in a P.D.Q. program is located after the string pool, at the top of DGROUP. As new items are added to the stack they are placed at ever lower addresses. Thus, overflowing the stack overwrites the string pool causing a "String space corrupt" error. To save memory, the default stack size in a P.D.Q. program is smaller than that of regular BASIC. This is one reason a program that runs correctly in the QuickBASIC editor may fail when linked with P.D.Q.

Although a BASIC programmer cannot access the stack directly, stack memory is claimed when procedures are called. Also, non-static procedures require more stack memory than static procedures, and recursive procedures require even more. If a program crashes for no apparent reason, try compiling using the /d switch. If an "Out of stack space" error is reported, you can then link with the /stack option.

Please see the section *Other Link Options* in Chapter 2, *Compiling And Linking*, for information on controlling the stack size in a P.D.Q. program. Also see the section *The Stack* in Appendix H, *Miscellaneous Considerations* for advice on selecting a suitable stack size.

Debugging TSR Programs

Unfortunately, /d may *not* be used in a TSR program. Therefore, a different strategy must be used. In truth, there is no good way to debug a P.D.Q. TSR program short of using a hardware-assisted debugger such as Periscope, or a protected-mode software debugger like Soft-ICE. Without heavy-duty debugging products like these your only recourse is to add calls to PDQPrint to display variable values, or show which part of a program is currently executing.

Simplified pop-up TSR programs can be tested in the QuickBASIC editor, as long as they do not also intercept system interrupts. To do this you will first load PDQSUBS.BAS as a module, and then comment-out the GOTO that follows the call to PopUpHere. Then when the program is run, it will execute the pop-up handling code once.

Appendix D: String Memory Considerations

This section describes how strings and other data items are stored in a P.D.Q. program. It is not necessary to understand the material presented here to use P.D.Q. successfully. However, by understanding how near memory is organized you can avoid "Out of string space" errors, and also control the amount of memory taken by P.D.Q. TSR programs.

The P.D.Q. String Pool

Every P.D.Q. program contains a block of memory we call the string pool. By default, this memory holds up to 32K of data; however, you can change this by linking with an alternate string pool in the form of a stub file. Three different types of data are stored in the P.D.Q. string pool:

1. The current contents of each string variable. The amount of memory taken is equal to the length of the string.

2. A table of string descriptors for each conventional (not fixed-length) dynamic string array. Four bytes per element are needed for each descriptor, so the number of bytes taken is determined by the total number of elements in all arrays.

3. A back pointer for each active string. A back pointer occupies two bytes of storage, and is present only when a string contains data. Memory is not taken for a back pointer before a string has been assigned, or after it is cleared using a statement such as **Work$ = ""**.

The string pool lies within the same 64K near data segment known as DGROUP. Several other items also compete for memory in DGROUP; these include your program's static data (variables and static numeric and TYPE arrays), variables used by P.D.Q. internally as it works (currently open file numbers, the current COLOR setting, and so forth), and the P.D.Q. stack. Non-array strings and static string arrays also use a string descriptor and that descriptor is in DGROUP. However, those string descriptors are not located in the P.D.Q. string pool.

Figure 1 shows a memory map of the various code and data segments that comprise a P.D.Q. program.

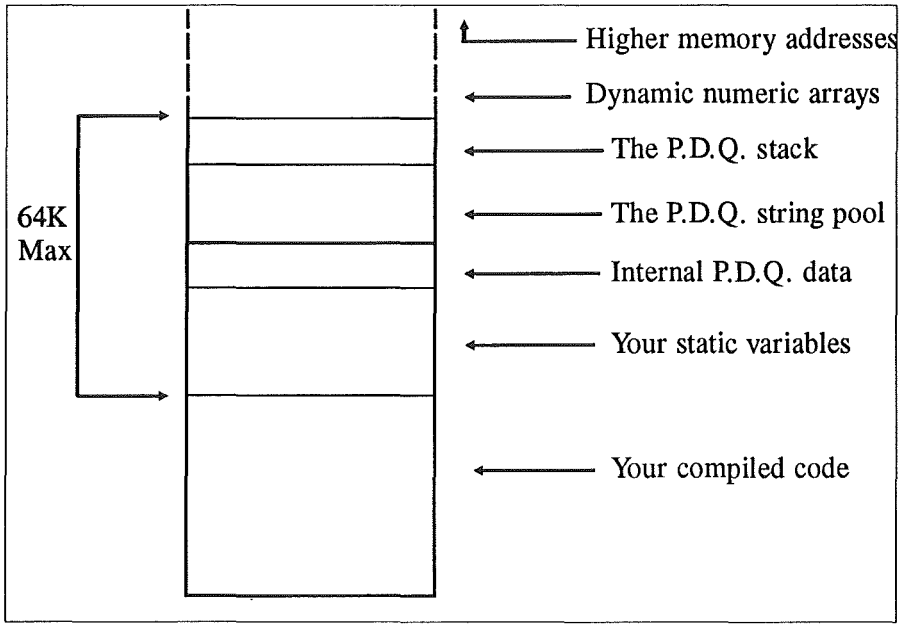


Figure 1: Memory Map Of A P.D.Q. Program

Although a program's code can grow beyond 64K in size by linking one or more compiled modules together, all near data—even when multiple modules are used—is combined into a single segment that cannot exceed 64K. Please understand that this 64K limitation is due to the design of the Intel 80*x*86 series of microprocessors, and has nothing to do with BASIC or P.D.Q.

Determining A Suitable String Pool Size

In many programs the size of the string pool does not matter. As long as the 32K default is large enough, there is no harm in having more memory than is actually needed. But there are two exceptions:

1. When a program operates as a TSR, all of the program's code and data remain in memory. Thus, less remains for other programs that are subsequently run.

2. When using a task switching program like Windows, DESQview, or Software Carousel, you can control how much memory is provided to each partition to preserve system resources. That is, you can tell Windows, *"The program I plan to run in this partition requires only* xxx*K to run, so set aside only that much and leave the rest for my other programs."*

If you have only a few variables that occupy, say, 3K of DGROUP memory, then you could conceivably have a string pool as large as 64K - 3K = 61K. Conversely, if you have a 10,000-element static long integer array (40K taken), then the string pool cannot be larger than 64K - 40K = 24K. Understand that this merely defines how big the string pool *can* be, and not what is ideal.

Regardless of how small an .EXE file size is, the amount of memory taken at run time will always be somewhat larger. For example, a short program with the single statement **X$ = "Hi mom"** may compile and link to only 1K or so. But at run time the entire string pool is expanded by DOS, and occupies that much memory in the PC. Thus, it is useful to link with a smaller string pool stub file when memory is at a premium.

The important point, therefore, is how to determine the amount of string pool memory your program actually requires. This may be determined using FRE(""), and also with the P.D.Q. StringUsed and StringShort functions. If you add the statement **PRINT FRE("")** after all of a program's strings have been assigned, you will know how much is available.

Likewise, StringUsed reports how many bytes were actually used—that is, **32K - FRE("")**. StringShort tells if your program ever asked for more than was available, and if so how much more. StringUsed is the most direct way to determine a program's string memory requirements. But if StringUsed reports that you used all 32K, then you should also query StringShort to see if you actually needed more than that at some point.

The MAKESTR Utility

P.D.Q. includes several alternate string pool stub files which you can link with a program to obtain a variety of fixed pool sizes. You can also control exactly the amount of string memory using the MAKESTR utility program. When compiled and run, MAKESTR.BAS creates a custom string pool object file holding any size between 10 and 63,000 bytes.

Other Memory Considerations

Because the stack is stored in DGROUP, increasing its size with the LINK /stack: option switch may require also reducing the size of the string pool. The /stack: option is described in the section *Other Link Options* in Section I, Chapter 2, *Compiling And Linking*.

When a program is designed as a TSR, all dynamic far memory allocations *must* be made before the call to EndTSR. Dynamic far memory is allocated either manually with the AllocMem routine, or automatically using REDIM with a numeric, TYPE, or fixed-length string array. DIM also allocates memory dynamically when a '$DYNAMIC metacommand is in effect, or when a variable is used to specify the array size. This issue is described further in the section entitled *Memory Allocation And Dynamic Arrays*.

Appendix E: Using CALL Interrupt

Of all the language features that have been added to BASIC in recent years, one of the most powerful is CALL Interrupt. By being able to tap into system interrupts directly, BASIC programs can access virtually all of the available DOS and BIOS services. Even though BASIC has more commands than any other high-level language, it is unable to communicate directly with the operating system. Such an intimate interaction is obviously quite useful—for example, to determine the current drive or directory, or to count the number of files whose names match a given specification. Therefore, modern versions of BASIC include a callable subroutine which provides access to all of the PC's system interrupts.

This section presents a general overview showing how to invoke interrupts from within a P.D.Q. BASIC program. Of course, we can't cover all of the DOS and BIOS services your programs may need to call in this brief tutorial. However, several important DOS and BIOS functions will be discussed. Also, many of the demonstration programs that are included with P.D.Q. use CALL Interrupt, and these provide further examples.

Before we get to the specific services, it is important to point out that the Interrupt statement is really an external called routine. With Microsoft QuickBASIC 4.0 and later it is provided in the QB.LIB and QB.QLB library files. BASIC 7 PDS uses the library names QBX.QLB and QBX.LIB. A similar version is included with P.D.Q. in the PDQ.LIB library file.

IMPORTANT:

Note that P.D.Q. differs slightly in its implementation of the Interrupt routine. The version that comes with Microsoft BASIC expects three parameters: an interrupt number, and two TYPE variables. The P.D.Q. Interrupt routine uses only one TYPE variable to hold both the incoming and outgoing registers.

What Is An Interrupt?

The IBM PC/XT/AT and compatibles support 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 PC's built-in BIOS routines.

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 and the PC's microprocessor, the keyboard is 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 specialized type 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. Even though assembly language programs may use the Call statement to invoke a subroutine, the Int (Interrupt) instruction is needed to access the operating system. Let's see why.

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 when it combines the various portions of your program into a single .EXE 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 still 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 will be located on the Tandy computer.

Interrupts, therefore, rely upon a table of addresses stored in a known place in low memory called the *interrupt vector table*. The very first 1,024 bytes in every PC contains a table of addresses for all 256 possible interrupts. Each table entry contains two words (four bytes)—one word holds the routine's segment, and the other holds its address within that segment. Some of the interrupt vector entries are assigned by the BIOS when you turn on your PC; others are assigned by DOS during bootup.

Whenever an assembly language program issues an Int instruction, the PC's microprocessor automatically fetches the correct segment and address stored in this table and then calls that address. Thus, any program may access any interrupt, without having to know where in memory the interrupt routine actually resides.

Registers

Every microprocessor contains a set of built-in integer variables called *registers*. Each register holds a single word (two bytes), which corresponds nicely to the size of a BASIC integer variable. Because these registers are contained within the microprocessor itself, they are accessed very quickly—much faster than variables that are stored in memory. The 80*x*86 family of microprocessors contains fourteen different registers. Some of these are intended for a specific use, while others may serve 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 assembler FOR/NEXT loop. We're not going to pursue a lengthy discussion of microprocessor theory here, however, because it's not really necessary if you simply want to access a few system interrupts.

Most DOS and BIOS services are specified by an interrupt number, as well as a service number. Nearly all of the DOS services are accessed through Interrupt 21h (Hex, or &H21), with the desired service specified in the AH register. In many cases, information is also returned in these registers. For example, the DOS service that obtains the currently selected disk drive is specified by placing the value 19h in the AH register. When the interrupt has finished, the current drive number is returned in the AL register.

The DOS services that accept or return a string (such as a file or directory name) also expect an address in another register to tell where that string is located. The DOS service that changes the current directory is called with AH set to 3Bh, and DX holding the address of the string that contains the name of the directory to change to. Likewise, to obtain the current directory you load AH with the value 47h, and SI with the address of a string that is to 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.

When a string is sent as a parameter to a DOS routine, it must be terminated with a CHR$(0) zero byte so DOS will know where it ends. Likewise, when DOS returns a string to your program (such as the current directory), it indicates the end with a CHR$(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 byte.

It is impossible to describe every DOS and BIOS service you may want to access here. Indeed, a complete discussion would fill several books. Two excellent books are *Peter Norton's Programmer's Guide to the IBM PC* and *Advanced MS-DOS* by Ray Duncan. Both of these are published by Microsoft Press, and are available in most book stores. An excellent book that specifically addresses calling interrupts from BASIC programs is (ahem) *PC Magazine's BASIC Techniques And Utilities* by Ethan Winer (Ziff-Davis Press).

To get you started, this section presents a few short examples. The first obtains the current drive, the second returns the current directory, and the third clears a rectangular portion of the display screen using one the PC's video BIOS services.

Accessing DOS

The short program example below returns the current drive, and it is designed as a DEF FN-style function. This was done both to create a smaller function, and because using a function is a sensible and natural way to design a routine that returns information. It is important that the statement **'$INCLUDE: 'PDQDECL.BAS'** be present at the beginning of the program source file. PDQDECL.BAS contains declarations for all of the external P.D.Q. routines including Interrupt, as well as the TYPE definition needed to access the processor's registers. But it is up to your program to DIM the Registers TYPE variable before using it.

```
'$INCLUDE: 'PDQDECL.BAS'
DIM Registers AS RegType            'create the TYPE variable

DEF FNGetDrive%                     'define the function
  Registers.AX = &H1900             'put &H19 into AH
  CALL Interrupt(&H21, Registers)   'call Interrupt &H21
  'return just the low (AL) part of AX and convert to ASCII
  FNGetDrive% = (Registers.AX AND &HFF) + 65
END DEF

PRINT "The current drive is " CHR$(FNGetDrive%)
```

As you can see, this function is very simple to implement; now let's see how it works. First the AX register is loaded with the value &H1900, which specifies the DOS "get current disk" service. Internally, the AX register is comprised of two separate "half-registers" called AL and AH. The "L" in AL stands for Low, and the "H" in AH means High. Therefore, AX is comprised of both a low- and high-byte portion, as shown in the following BASIC formula:

```
AX = AL + (256 * AH)
```

In an assembly language program it is simple to access each register half separately. However, BASIC does not offer a byte-sized numeric variable type to use within a TYPE declaration. Therefore, a bit of math is required to get at each half separately. Using Hexadecimal notation simplifies this somewhat, as the function example above shows. That is, assigning AX to &H1900 is the same as placing &H19 into AH, and zero into AL. In this case we don't care what is assigned to AL so using a zero is acceptable.

Next, the Interrupt routine is called specifying Interrupt &H21 and the Registers TYPE variable is passed to it. This TYPE variable is used to load the processor's registers with the appropriate values before the interrupt is called, as well as to examine those values when the interrupt has finished. Here, &H19 is passed to Interrupt &H21, and the value that is returned in AL indicates the current drive. For this service DOS uses 0 to mean drive A, 1 for drive B, and so forth. Therefore, you must use AND with the value &HFF to extract just the low portion in AX, and then add 65 to adjust that to the equivalent ASCII character value.

Obtaining the current directory is only a little more difficult than getting the current drive, as you can see in the example below.

```
DEFINT A-Z
'$INCLUDE: 'PDQDECL.BAS'
DIM Registers AS RegType              'create the TYPE variable

DEF FNGetDirectory$(Drive)
  STATIC Dir$, Zero                   'these are local variables
  Dir$ = SPACE$(65)                   'the longest possible name
  Registers.AX = &H4700               'put 47h into AH, 0 into AL
  Registers.DX = Drive                'specify the drive number in DL
  Registers.SI = SADD(Dir$)           'the string address goes in SI
  CALL Interrupt(&H21, Registers)     'call Interrupt &H21
  Zero = INSTR(Dir$, CHR$(0)) - 1     'find terminating zero byte
  FNGetDirectory$ = LEFT$(Dir$, Zero)'return directory name
END DEF
                              'display the name for drive C
  PRINT "The current directory is C:\";
  PRINT FNGetDirectory$(3)     '1 = A, 2 = B, 3 = C, etc.
```

Regardless of whether you are using regular BASIC or P.D.Q. to create this function, the string that receives the directory name must be initialized to a length of at least 65 characters. This is the longest path name that DOS can return.

Only three registers are needed to tell DOS to return the current directory. The first is AH, which specifies DOS service 47h. The second is DL, and it tells DOS the drive to examine. For this service drive A is indicated with a 1, drive B with a 2, and so forth. You may also use a value of 0 in DL to mean the current default drive. The last register is SI which tells

DOS where to place the directory name. Therefore you will use the SADD (String Address) function when assigning SI.

The next step is to call the Interrupt routine specifying Interrupt &H21, while passing the Registers variable to it for the remaining information. INSTR then locates the zero byte that DOS uses to mark the end of the string. Finally, the function output is assigned from the correct number of leading characters using LEFT$.

Although this simple example doesn't attempt to detect DOS errors, that would be simple to add. For example, if you ask for the current directory in a drive that doesn't exist, you'll probably want to know that. For most of its services, DOS uses the processor's Carry flag to indicate the success or failure of an operation. This flag is the first of several bits that are located in the Flags register. Therefore, you could add this statement to the FNGetDirectory$ function, immediately after calling the Interrupt routine:

```
IF Registers.Flags AND 1 THEN ... an error occurred
```

Accessing The BIOS

This last example calls upon a BIOS video service to clear just a portion of the display screen. Similar to the way DOS interrupts are invoked, the BIOS video routines are accessed through Interrupt &H10, with a service number specified in the AH register.

```
DEFINT A-Z
'$INCLUDE: 'PDQDECL.BAS'
DIM Registers as RegType

CLS
PRINT STRING$(1920, "A");          'fill the first 24 lines
CALL ClearScreen(5, 10, 20, 70)    'call the routine

SUB ClearScreen(ULRow, ULCol, LRRow, LRCol)
  Registers.AX = &H600             'service 6, scroll screen
  Registers.BX = &H700             'clear to white on black
  ' the corner parameters go into CX and DX below
  Registers.CX = (ULCol - 1) + (256 * (ULRow - 1))
  Registers.DX = (LRCol - 1) + 256 * (LRRow - 1)
  CALL Interrupt(&H10, Registers) 'call the BIOS to do it
END SUB
```

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 service recognizes zero as a special flag, which tells the BIOS to clear the screen rather than scroll it. Service 6 also expects the color to clear to in the BH register, and in

this case 7 is used for normal white on black. Of course, you may use any color you like, or even pass the color as an additional parameter to the subprogram.

The next two instructions take the upper left and lower right corner arguments, and place them into the appropriate registers. 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 the Registers TYPE variable.

Finally, the Interrupt routine is called specifying BIOS video Interrupt &H10. There are two important benefits to having the BIOS do the work. One is of course the reduced amount of code that is needed, compared to manually looping through video memory using POKE to clear each character position. The second is that the BIOS is responsible for determining which type of monitor is installed, and selecting the correct video segment.

Summing Up

As you have seen, accessing DOS and BIOS services is not at all difficult. Armed with a good reference book that describes the various services, you can quickly and easily write programs in BASIC that tap the full power of the operating system. As implemented in the P.D.Q. library, CALL Interrupt provides performance and code size that is competitive with pure assembly language.

Appendices

Appendix F: Accessing The Environment

Every program has an environment, which is an area of memory maintained by DOS that holds a collection of string variables. One important use for the environment is to allow programs to pass information to other programs that they SHELL to. DOS also uses the environment for remembering your current PATH and PROMPT settings. Further, LINK and BC can also use the environment. Those programs let you specify where to look for libraries or Include files that aren't in the current directory.

Environment variables are assigned using the DOS SET command, or, in BASIC, with the ENVIRON statement. All of the active environment variables may be viewed from the DOS command line by entering SET with no arguments. Environment variables may be retrieved individually in a BASIC program with the ENVIRON$ function.

The environment is often overlooked by BASIC and other programmers due to several serious limitations in the way that it has been implemented. The first is that a program can alter only its own environment. Therefore, it is impossible using normal means for a program to pass return information to a program that shelled to it. When DOS executes a program, it provides a copy of the current environment to the program being executed, which is then abandoned when that program terminates. Further, the copy which is created is only as large as necessary to hold the current set of variables. It is usually not possible for a program to add new variables, or extend appreciably the length of existing variables. Finally, DOS and BASIC always capitalize variables before they are added to the environment, which further limits its usefulness.

P.D.Q. overcomes these limitations by providing several important enhancements to BASIC's ENVIRON and ENVIRON$ statements. These enhancements are accessed with the EnvOption routine, prior to using ENVIRON and ENVIRON$. EnvOption lets you access the current program's environment, the environment of the parent (usually, but not always DOS), or the environment of the underlying application from a TSR program. Another option lets you tell ENVIRON and ENVIRON$ to honor or ignore capitalization. EnvOption is described fully in the reference portion of this manual, so we won't belabor those options here. However, it is important to point out a few minor differences between the way regular BASIC and P.D.Q. implement the environment commands.

Please understand that these differences were established to be more fully compatible with the way DOS and COMMAND.COM work. For ex-

ample, regular BASIC removes leading and trailing blanks from environment variables. This prevents your programs from accessing what are otherwise perfectly legal environment variable names. Regular BASIC also accepts illegal assignments, contrary to the way DOS works internally.

Where Microsoft BASIC generates an "Illegal function call" error if you attempt to assign a variable with leading or trailing blanks, P.D.Q. is perfectly happy to assign such variables into the environment. Further, regular BASIC always capitalizes variable names and their contents, which prevents lower-case characters from being used. Finally, regular BASIC accepts an assignment string that is missing the equals sign (=), where P.D.Q. does not. The examples in Table F-1 illustrate some of the differences between regular BASIC and P.D.Q.

TABLE F-1
Differences Between BASIC and P.D.Q.

REGULAR BASIC ALLOWS	P.D.Q. REQUIRES	ACTION
ENVIRON("PATH=;")	ENVIRON("PATH=")	Clear the current PATH
ENVIRON("PATH \WP")	ENVIRON("PATH=\WP")	Set PATH=\WP
ILLEGAL IN MS BASIC	**P.D.Q. ACCEPTS**	**ACTION**
ENVIRON(" X=Y")	ENVIRON(" X=Y")	Set " X=Y"
ENVIRON("hi=there")	ENVIRON("hi=there")	Set "hi=there"

Five error codes are used by the P.D.Q. environment routines to indicate the success or failure of an operation. All of the possible error codes are listed in Table F-2 with some typical causes, and these errors may be retrieved by examining BASIC's ERR function.

TABLE F-2
P.D.Q. Environment-Related Errors

ERROR NUMBER	POSSIBLE CAUSE
101 COMSPEC not found	Someone erased the COMSPEC= variable from the environment being accessed.
102 Environment not found	The program being accessed has released its own environment.
103 ENVIRON string invalid	The equal sign (=) was omitted when assigning a variable with ENVIRON.
104 Out of string memory	There was insufficient string pool memory for ENVIRON or ENVIRON$ to do their work.
105 Out of environment space	There was insufficient environment memory to add the specified variable.

Appendices

Appendix G: Performance Optimizations

This section presents a variety of hints and suggestions for improving the performance of your P.D.Q. programs. Notice that many of these tips are relevant for regular BASIC as well. Please understand that these few techniques are by no means the last word on the subject. The only way to really determine which program statements are the most efficient is to compile a program, and then examine the resultant machine code using CodeView. This is the method we used when researching the information presented here.

String Versus Integer Operations

In general, string operations and comparisons are slower and require more code to implement than integer operations. Therefore the first example below is slightly more efficient than the second:

```
WHILE LEN(INKEY$) = 0: WEND  'this creates less code
WHILE INKEY$ = "": WEND      'this creates more code
```

Likewise, when a character is being compared with IF/ELSEIF or SELECT CASE, you should obtain its ASCII value once if possible, and use that for subsequent comparisons. The first example that follows generates more code and is much slower than the second.

```
SELECT CASE X$          'this works, but generates more code
  CASE "Y"
   ...
  CASE "N"
   ...
  CASE "Q"
   .     ..
END SELECT


SELECT CASE ASC(X$)     'obtain the ASCII value once
  CASE 89               'these integer comparisons create
   ...                  '  very little code
  CASE 78
   ...
  CASE 81
   ...
END SELECT
```

When integers are compared, BASIC generates in-line assembler code to directly compare the two values. This is extremely efficient, as shown following.

```
IF X = 65 THEN GOTO 100     'here's the BASIC statement
  Cmp  Word Ptr [X],41      'and the code that BC generates
  Jne  Label1               '(41 is the Hex equivalent of 65)
```

```
    Jmp  _100
Label1:

  IF X$ = "A" THEN GOTO 100  'here's a similar BASIC statement
    Mov  AX,X$               '  which creates much more code
    Push AX
    Mov  AX,Offset "A"
    Push AX
    Call B$SCMP              'B$SCMP is BASIC's string compare
    Jnz  Label2              '  routine
    Jmp  _100
Label2:

  100 END
_100:
```

Constants Versus Variables

Contrary to Microsoft's BASIC documentation, using constants (literal numbers) often makes your programs larger than when variables are used. This is true regardless of whether you are using actual numbers or named constants. For example, in the program fragment below many calls are made to PDQPrint to display a box. The first example uses numbers for the color and column parameters, while the second uses variables.

```
1.  CALL PDQPrint("                ", 1, 1, 7)
    CALL PDQPrint("                ", 2, 1, 7)
    CALL PDQPrint("                ", 3, 1, 7)
    CALL PDQPrint("                ", 4, 1, 7)
    CALL PDQPrint("                ", 5, 1, 7)

2.  Colr = 7: Column = 1
    CALL PDQPrint("                ", 1, Column, Colr)
    CALL PDQPrint("                ", 2, Column, Colr)
    CALL PDQPrint("                ", 3, Column, Colr)
    CALL PDQPrint("                ", 4, Column, Colr)
    CALL PDQPrint("                ", 5, Column, Colr)
```

Each time the numbers 7 and 1 are encountered BASIC generates code to assign those values to new locations in memory, and then passes the addresses of those locations to PDQPrint. If you instead assign variables for the 7 and the 1 once ahead of time as is done in the second example, the resultant code is much smaller.

In this case, the actual saving is only 12 bytes of code per call to PDQPrint. But you can see how this could quickly add up in a large program. Further, because each instance of the same constant is always placed into a new area of memory, this seriously impinges on available string space. Of

course, a variable is stored only once. So in addition to the 12 bytes of code, another 4 bytes of string space is also saved per call. The first example therefore comprises 305 bytes of code and uses 30 bytes of string memory, while the second requires only 257 bytes of code, and 14 bytes of string space. (The two variable assignments occupy 6 bytes each.)

One important exception, though, is when comparing and assigning variables. Using a constant as in **IF X >3 THEN** is always preferable to using a variable such as **IF X >Three THEN**, because the extra step of loading the variable Three is avoided. That is, the 8088 can directly compare a variable with a constant, where it cannot directly compare two variables. Assignments are usually better from constants as well, with one notable exception. If the same value is assigned more than twice in a single block of code, using a variable is preferred. That is, the first example below generates more compiled code than the second.

```
A = 1   'each of these statements is 6 bytes
B = 1
C = 1
D = 1
E = 1

A = 1   'this is 6 bytes
B = A   'and so is this
C = A   'but the rest are only 3 bytes each
D = A
E = A
```

Internally, BC remembers what values are in which registers as it compiles your program. Of course, the best way to know for sure is to try it both ways and see which results in a smaller .EXE program. Or you could use CodeView to view the generated code.

Finally, each use of CALL (especially with parameters) creates a fair amount of code. Therefore, when many of the parameters do not change between calls it can be advantageous to use GOSUB to branch to a central subroutine that in turn performs the call:

```
  X = 12: GOSUB CallIt
  X = 13: GOSUB CallIt
  X = 14: GOSUB CallIt
   ...
   ...
   ...
CallIt:
  CALL Something(X, Y, Z, A, B, C)
```

Short Circuit Techniques

Another useful and important optimization is called *short circuit expression evaluation.* When multiple conditions are tested using AND and OR, QuickBASIC evaluates all of them before acting on the result. The example below tests if a string is not null and if the row and column coordinates are within a legal range, before attempting to locate the cursor and print the string:

```
IF Work$ <> "" AND Row <= 25 AND Column <= 80 THEN
    LOCATE Row, Column
    PRINT Work$
END IF
```

If the most likely case is that Work$ is null a better approach is to first test Work$, and then use a separate test for Row and Column:

```
IF LEN(Work$) THEN
    IF Row <= 25 AND Column <= 80 THEN
      LOCATE Row, Column
      PRINT Work$
    END IF
END IF
```

This way, if Work$ is empty your program can quickly skip the remaining tests, and go on to the rest of the program. While this technique does not reduce code size (code to implement each comparison is still generated), it greatly increases the program's speed.

Note that BASIC 7 PDS performs this optimization automatically, and creating separate IF tests is not necessary or desirable with that compiler.

String Concatenation

Yet another important program optimization is to avoid unnecessary string concatenation. Whenever you join two or more strings with the plus sign (+) BASIC generates code to find each string in memory, then combine them, and finally copy the resultant characters to a new location. But when strings are merely being printed as in the example below using a semicolon as a separator avoids this concatenation, thus reducing the size of your .EXE program. Not only does this let the compiler generate less code, but you also avoid adding the concatenation library routine to your program.

```
PRINT X$ + Y$ + Z$          'not recommended
PRINT X$; Y$; Z$            'preferred
```

Even more important than the reduction in code size is the tremendous improvement in speed realized by avoiding unnecessary concatenation.

Although P.D.Q. is much faster at combining strings than regular BASIC, why add unnecessary code when it is not needed?

When a string is assigned from a concatenated list, the overhead is even worse. The short examples below assign the codes to enable enhanced printing on Epson/IBM type printers. The first generates a whopping 55 bytes of code, while the second does the same thing in only 13. The QuickBASIC editor lets you enter control characters directly into a quoted string by first pressing Ctrl-P, and then typing the ASCII code on the numeric key pad while pressing Alt. Note, however, that CHR$(27) must be entered by pressing Ctrl-P and then Ctrl-[. The Escape key may not be used for this purpose in the QuickBASIC editor.

```
PrintCode$ = CHR$(27) + "E" + CHR$(27) + "G"
PrintCode$ = "←E←G"
```

Speeding Up File Processing

One of the most important areas where program optimization is helpful is file processing. In many cases, a program's speed will be limited by the amount of time it takes DOS to physically read or write information on the disk. But again, by applying some simple tricks file access speed can often be dramatically improved.

Perhaps the biggest I/O bottleneck is created when numeric variables are read and written. Whenever a statement such as **PRINT #1, X%** is encountered in a BASIC program, the two-byte integer value must be converted to an equivalent string of ASCII digits. Besides adding the code overhead and time required to perform the conversion, additional disk space is taken. Although an integer variable is stored in memory using only two bytes, as many as eight are required for the equivalent digits in the file (including a possible minus sign and terminating carriage return/line feed). Of course, the added length further increases the amount of time needed to read or write each variable.

Similarly, when you use **INPUT #1, X%** in a BASIC program, each character must be read from disk, and examined to see if it is either a comma or carriage return that marks the end of the value, or a CHR$(26) that marks the end of the file. Further, once the digits have been read into memory, they must then be evaluated and converted into two bytes before being placed into the integer variable. Obviously, this takes much longer than simply reading two bytes without regard to their meaning.

Therefore, a better method for storing numeric values is to use the BINARY file commands PUT and GET:

```
OPEN FileName$ FOR BINARY AS #1   'open the file
PUT #1, , X%                      'write three integers
PUT #1, , Y%
PUT #1, , Z%
CLOSE #1                          'close the file
```

Likewise, to read those values again later you would use:

```
OPEN FileName$ FOR BINARY AS #1   'open the file
GET #1, , X%                      'read three integers
GET #1, , Y%
GET #1, , Z%
CLOSE #1                          'close the file
```

Like their sequential access counterparts, PUT and GET step through the file, incrementing the current file SEEK position as they work. However, they are much faster because the numbers do not need to be converted to ASCII digits, and because a fixed number of bytes is always used regardless of the value being read or written. In a test that wrote and then read 1000 integer variables using both methods, the binary access took 2.74 seconds and created a file 2000 bytes in length. Contrast that with the 4.56 seconds needed to write and read a sequential file which occupied 6893 bytes of disk space.

Perhaps the ultimate speedup trick is to read or write more than one variable at a time, when an entire array is being processed. Each time a call is made to the DOS routines that handle file operations, some amount of time is required by DOS just to handle the request. Even though normal BASIC commands do not allow this, you may easily call BASIC's internal routines directly to process any number of bytes (up to 65,535). This is a very important speedup technique which is demonstrated in the BIG-PUT.BAS example program.

One final point to be aware of is that EOF in P.D.Q. is fairly slow, and should be avoided whenever possible. Each time EOF is queried it calls DOS to attempt to read a single character from the specified file. If no character was read, then EOF knows it's at the end of the file. Otherwise, it has to call DOS again to seek one byte backward, to compensate for the byte just read. But precisely because of the way errors are handled by P.D.Q. you can instead use the ERR function to trap for an end-of-file condition. This is shown below, though it will not work if you are using ON ERROR.

```
OPEN "XYZ" FOR INPUT AS #1        'open the file
DO
    LINE INPUT #1, Work$          'get a line of text
    PRINT Work$                   'print it just for fun
```

```
LOOP UNTIL ERR                  'continue until an error occurs
CLOSE #1                        'all done
```

Compiling With /S

The /s switch has been around since the very first BASCOM 1 compiler, and it is probably the least understood of the BC.EXE options. Since Microsoft has failed to clearly describe it in their manuals, we'll spend a few moments here and explain it for them.

The /s switch does two things: First, it tells the BC compiler not to combine identical strings that are longer than four characters. By default, BC looks for repeated occurrences of string constants, and assigns them only once in the resulting .OBJ file. For example, if you use

```
PRINT "Press any key to continue"
```

five times in a program, the actual quoted text is stored only once in the .EXE file. Each time PRINT is used with that message the same string is passed to it, which of course helps preserve string space. But for very large programs that have many quoted strings, BC must remember all of them until it is ready to create the object file. At some point BC will run out of memory, since it has a lot of other information it needs to keep track of as well.

To avoid this problem in large programs you should use the /s switch. In that case BC simply writes each string to the object file as it is encountered, rather than saving it until the end in its work space. Although it might appear that not combining like strings will make programs that use /s larger, often that is not the case. Now let's see why.

The second thing /s does is tell BC to add two short assembler subroutines (eight bytes each) at the beginning of your program (or at the beginning of each module in a multi-module program). Two of the most common string operations in a program are assignments and concatenations. Normally, a call to the string assignment or concatenation routines generates thirteen bytes of code, including the statements needed to pass the source and destination parameters. However, a call to these subroutine "wrappers" takes between three and nine bytes each. BC therefore uses these subroutines to offset the increased size that results from multiple copies of the string constants. This approach is much like the "GOSUB to a CALL" shown earlier in this section.

In many cases—especially if there are few or no duplicated strings—using /s reduces the size of your programs. Indeed, it would be wonderful if the subroutine option of /s were available independently, perhaps by adding another compiler switch.

Appendix H: Miscellaneous Considerations

This section serves as a "catch all" of miscellaneous information relating to P.D.Q. Several differences between P.D.Q. and regular BASIC programs are described, and additional information is also provided for programmers who are familiar with assembly language.

Functions In P.D.Q.

BASIC's SUB and FUNCTION constructs are fully supported in P.D.Q. It is important to point out, however, that employing GOSUB routines or DEF FN functions when practical results in smaller and faster programs. Using GOSUB or invoking a DEF FN function (with no parameters) generates only three bytes of code. Compare that with five bytes for a SUB or FUNCTION, plus an additional four for each parameter being passed. In addition, two library routines are also added to programs that use conventional SUB or FUNCTION definitions further increasing a program's size.

True/False Functions

When a numeric function returns only a true or false result, it is most sensible to use the values -1 and 0 respectively. This has two distinct advantages, which are shown using the PDQExist function as an example. First, the caller can use the simplified form of IF, to test for a true condition:

```
IF PDQExist%(FileName$) THEN                'the file exists
```

In this case any non-zero value is considered true, and an explicit comparison to a value of zero is not required. The second, and more important, advantage is that BASIC's NOT operator may be used to test if the result was not true, yielding a more readable program:

```
IF NOT PDQExist%(FileName$) THEN            'the file was not there
```

The NOT operator simply reverses all of the bits in an integer variable or function result, which effectively alternates between the values 0 and -1. That is, the value 0 is represented internally like this:

```
0000 0000 0000 0000
```

And the value -1 looks like this when viewed in Binary:

```
1111 1111 1111 1111
```

Thus, NOT reverses the bits toggling the value between -1 and 0 each time it is used.

Fixed-Length And TYPE Variables

Although P.D.Q. fully supports BASIC's fixed-length strings, we recommend that you avoid using them for general string handling except in special situations. Each time a fixed-length string is accessed or printed, BASIC generates a substantial amount of code to copy it into a conventional string. Even worse, whenever a fixed-length string is used as an argument to a called subroutine, additional code is generated to copy the string back again in case the subroutine changed it.

You can avoid having BASIC create a copy of the fixed-length string, but only by using VARPTR to pass its address. Of course, the routine being called must be designed to expect this address, rather than the address of a string descriptor.

Fixed-length and TYPE variables may be read and written to disk using GET and PUT, and their numeric components may be freely assigned and compared using normal BASIC statements. Using these variables this way does not impose any penalties on code size or speed. As with fixed-length strings, the string portion of a TYPE variable is converted to a conventional string whenever it is accessed.

Integer Values Greater Than 32K

Some of the P.D.Q. extensions such as AllocMem use an integer parameter to specify a value between 0 and 65,535. Since integers in a BASIC program range from -32,768 to 32,767, you can use an equivalent negative number to specify values between 32,768 and 65,535. This is shown in the following short code example.

```
IF NumBytes& > 32767 THEN
  NumBytes& = NumBytes& - 65536
END IF
PassedArg% = NumBytes&
Segment% = AllocMem%(PassedArg%)
```

Similarly, you can convert a negative number to an equivalent positive value like this:

```
IF Value% < 0 THEN
  Answer& = Value% + 65536
ELSE
  Answer& = Value%
END IF
```

Initialized Versus Uninitialized Data

Quoted string constants are referred to as *initialized data*, because the actual data is placed into the program's object file. Even if the quoted string is merely a series of blank spaces, those spaces must end up somewhere within the final program. However, P.D.Q. programs also support what is called *uninitialized data*, whereby header information in the .EXE file contains instructions to DOS to allocate DGROUP memory when the program is loaded and run.

The advantage of using uninitialized data is that the necessary bytes are not present in the .EXE program file. One disadvantage is that the data area contains whatever random bytes happen to be in memory at the time the program loads. However, all of the P.D.Q. routines that access uninitialized data assign or clear that memory before using it.

Using P.D.Q. With QuickPak Professional

If you have our QuickPak Professional product, do not use the alternate BASIC string functions such as QPMid$, QPLTrim$, and so forth. Although they will work, BASIC adds a lot of extra and unnecessary code to process external string functions. The MID$, LTRIM$, and other string functions in the P.D.Q. library use the same tight, error-free techniques as QPMid$, but they are meant to be integrated directly with BASIC.

This limitation also extends to the QuickPak Professional Trim$ function, which is less efficient and generates more code than simply using **LTRIM$(RTRIM$(Work$))**. Again, it is not the string functions themselves that are inefficient. Rather it is the extra setup that BASIC performs to invoke them, and then copy the function's output to another temporary string. External numeric functions also suffer from this added overhead, but to a much lesser degree.

String Arrays

When string arrays are passed to an external assembler routine such as the Sort routine provided with P.D.Q., an extra step is required to obtain the correct address of the first element. This is done using a combination of BYVAL and VARPTR as follows:

```
CALL SomeRoutine(BYVAL VARPTR(Array$(FirstElement)))
```

Please understand that the generated code is no larger than when the element is specified directly. That is, BYVAL VARPTR only looks like it requires more code to implement.

Assembly Language Considerations

Unlike virtually every other company in the BASIC support business, we are unique in our attitude and willingness to share all of our source code. A wealth of useful information is contained in the assembly source files that accompany P.D.Q., and you are invited to study them and learn from them if you are so inclined. Indeed, beyond just learning how P.D.Q. works, you can also use our source code as a starting point for routines and language extensions of your own. Please note, though, that all of the P.D.Q. library routines are meant to be assembled using Microsoft MASM 5.1 or later. If you have MASM 6.0 do not use the ML.EXE assembler program. Instead use the MASM.EXE interface provided with that product.

Another reason for investigating the source code is to create your own custom stub files (see the section entitled *Linking With Stub Files*). The routines that are listed following contain useful header comments that describe the internal workings of P.D.Q. in detail.

ASSIGN$.ASM describes string descriptors and back pointers, and how they are implemented in P.D.Q. It also discusses string heap compaction (sometimes called *garbage collection*), as well as temporary strings and string assignments in general.

B_ONEXIT.ASM shows how to tie your own assembly language routines into BASIC's exit procedure, to cause a particular routine to be executed when the program ends. This is useful to ensure that hooked interrupt vectors are released.

COLOR.ASM describes the unusual method BC.EXE uses to pass a varying number of parameters to some of its internal routines—in this case COLOR.

COLORDAT.ASM shows how the foreground and background colors are combined into a single byte in display memory.

COMPACT.ASM contains additional comments about string compaction and temporary strings, and discusses some of the techniques we used in P.D.Q. to make these as fast as possible.

DIM.ASM presents a thorough discussion of dynamic arrays, and how they are dimensioned and erased. These routines are quite complex, especially when dynamic string arrays are involved, and the comments contained therein go a long way toward explaining what is really going on.

DOERROR.ASM shows how ON ERROR is handled in a P.D.Q. program.

ERRDATA.ASM describes the method P.D.Q. uses to handle the various error codes that the ERR function reports.

FHANDLES.ASM describes the interaction between BASIC file numbers and DOS file handles, and discusses how record lengths and the current TAB position are maintained by P.D.Q.

FLOATS.ASM discusses floating point issues related to using P.D.Q. as a toolbox with assembly language.

FLUSH.ASM provides an example of a routine that accepts a varying number of parameters, and then determines how many parameters were passed. When an external routine is declared using CDECL, BASIC adds the instruction **Add SP,n** after the call. Thus, the routine can read the value for "n" from the caller's code segment, to know how many parameters there are.

GETTEMP.ASM describes how temporary string descriptors are allocated by P.D.Q.

HOOKINT0.ASM explains the particularly messy details of writing an Interrupt 0 handler.

PDQ.ASM contains the main startup code for P.D.Q. Although it does not contain descriptive comments, it is a fairly simple procedure to follow. In particular, it shows BASIC's segment ordering, and provides some insight into how a high-level language is designed.

PDQDATA.ASM contains many useful data items that can be accessed by routines you design. For example, the current version of DOS and the program's PSP segment are stored there, and you can access these directly.

TEMPSTR.ASM is used by many of the P.D.Q. internal string routines, and the short discussion in the header comments shows how it may be called by any routine that needs to allocate string memory. Also see **MAKETEMP.ASM** which provides a similar service but at a higher level.

TIMER.ASM shows how to add floating point math to a BASIC program, and tie into BASIC's 8087 emulator. The beauty of this method is that either the 8087 or emulator is used automatically, based on whether an 8087 is detected at startup. Note that this technique works with regular BASIC as well as P.D.Q.

UBOUND.ASM contains information on array descriptors, and also provides a table detailing each entry in the descriptor.

Appendix I: Link Errors

Fixup Overflow Errors

Most error messages you will receive from LINK are "Unresolved external", which means that you used a BASIC statement or function that is not supported by P.D.Q. We'll discuss those in a moment, but there is another possible error condition you should be aware of.

Each time you use a variable in a BASIC program, the BC.EXE compiler sets asides space in the .EXE file to hold it. For each integer variable two bytes are reserved, and for strings and long integers four bytes are used. The actual string data is created at run time when those variables are assigned, so it is not present in the file.

If the combination of your program's variables and the 32K string pool memory exceed the CPU's 64K limit, LINK displays either a "Fixup overflow" or "Stack plus data exceeds 64K" error message. When this happens, you must explicitly link with one of the smaller STR#####.OBJ files to reduce the size of the string pool. This is described in the sections *Linking With Stub Files* and *String Memory Considerations*.

Unresolved External Errors

If you attempt to use one of the BASIC commands that P.D.Q. doesn't support, in most cases you will receive an "Unresolved external" error message from LINK. This happens because BASIC has made a call to one of its library routines, but LINK was not able to find that routine in the PDQ.LIB library file.

Unfortunately, LINK reports only the name of the routine that it couldn't find, which in some cases bears little resemblance to the name of its associated BASIC command. Fortunately, many internal routine names are in fact similar to their corresponding statement names. For example, using the unsupported CLEAR statement tells LINK to access a routine named B$SCLR. The table that follows indicates some of the unsupported commands, and lists the external names with their corresponding BASIC statements.

Almost all of the BASIC language routines begin with the characters "B$", and the following character is often an "S" (for Statement) or "F" (for Function). However, there are exceptions to this naming convention.

You may want to create a listing file of all of the routines and data items in the BCOM library that comes with BASIC. This would help you to identify other routines that are not listed here. To do this, enter the following command, specifying the library name appropriate for your version of QuickBASIC:

```
lib bcomxx.lib , filename.ext ;
```

If you are using BASIC 7 PDS specify the BCL71ENR library instead:

```
lib bcl71enr.lib , filename.ext ;
```

This tells LIB.EXE (supplied with BASIC) to generate a fully cross-referenced listing of each item in the library and name it FILENAME.EXT. Two lists are created in this file—the first shows each routine name and data item, followed by the name of the object module that holds it. The second list shows each object module, and all of the routines and data items contained therein. Thus, a routine that does not have a self-explanatory name can often be identified by the name of the object module it is in, or by the names of other routines in the same object module.

If you elect to create a list file, we recommend using a file browsing utility that has a search facility such as Vernon Buerg's excellent LIST program (shareware, available on CompuServe). The first occurrence of the routine name shows the object module it is in, and the second search shows all of the other related routines in that module.

Table I-1 is by no means complete, but it does list many of the common unsupported routines that P.D.Q. programmers might attempt to use.

TABLE I-1
P.D.Q. Unresolved Externals Often Reported

ROUTINE NAME	POSSIBLE CAUSE
B$?EVT	using KEY ON, ON KEY, ON TIMER, or ON PLAY
B$ERDS	using the ERDEV$ function
B$ERDV	using the ERDEV function
B$ETK0	using KEY (n) ON
B$EVCK	using ON KEY, ON TIMER, ON PLAY, (or ON DASHER, ON PRANCER ...)
B$FERL	using the ERL function
B$KFUN	using KEY ON
B$LPOS	using the LPOS function
B$ONKA	using ON KEY (n) GOSUB
B$ONLA	using ON PLAY (n) GOSUB
B$ONTA	using ON TIMER (n) GOSUB
B$POW4	raising a single precision variable to a power (such as X! ^ 2)
B$RDMP	using BASIC PDS's REDIM PRESERVE
B$RUNL	using RUN to restart the program
B$SCHN	using the CHAIN statement
B$SCLR	using the CLEAR statement
B$SCPF	omitting STATIC from a string function definition
B$SRUN	using RUN "filename"
B$USNG	using PRINT USING
B$VWPT	using VIEW PRINT
B$WRIT	using WRITE

Appendices

INDEX

Index

A

B

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

11 BAILEY AVENUE, RIDGEFIELD, CONNECTICUT 06877
PHONE: (203) 438-5300 - SALES: 1-800-35-BASIC
CRESCENT SOFTWARE, INC.