Professional Programming Tools
for BASIC Compilers

Crescent Software's

QuickPak Professional

Owner's Manual

CRESCENT
SOFTWARE, INC.

QuickPak
PROFESSIONAL

Advanced Programming Library for
BASIC Compilers
Version 4.0

CRESCENT SOFTWARE, INC.
11 Bailey Avenue
Ridgefield, CT 06877
PHONE: (203) 438-5300 ■ FAX: (203) 431-4626

Table of Contents

Chapter 1
Introduction

Thank you for purchasing QuickPak Professional! We have put every effort into making this the finest and most complete collection of BASIC utilities available. We sincerely hope that you find it both useful and informative. If you have a comment, a complaint, or perhaps a suggestion for another product you'd like to see, please let us know. We want to be your *favorite* software company.

Before we begin discussing the contents of the QuickPak Professional utilities, 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 upgrades 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!

Also, please mark the product serial number on your disk labels. License agreements and registration forms have an irritating way of becoming lost, and doing this will insure that the number is always handy if you need to contact us.

You may also want to note the version number in a convenient location, since it is stored directly on the distribution disk in the volume label. If you ever have occasion to call us for assistance, we will probably need to know the version you are using. To determine the version number for any Crescent product, simply display a directory of the original disk. The first thing that appears will be something like:

```
Volume in drive A is QPPro V4.00
```

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

WELCOME TO QUICKPAK PROFESSIONAL

QuickPak Professional is a comprehensive collection of subroutines and functions designed to complement Microsoft QuickBASIC and BASIC 7 PDS. There are three key components to QuickPak Professional:

1. Assembly language routines that provide a dramatic improvement over what would be possible using BASIC alone. Some of the routines improve on BASIC's speed and code size, while others allow access to DOS and BIOS services not possible any other way. Assembler routines can also eliminate the need to use ON ERROR.
2. BASIC subprograms and functions to perform a variety of chores that would be tedious or difficult to write yourself. In some cases, routines that are provided in assembly language are also present in a BASIC equivalent so they may be easily customized.
3. The Assembly Tutor which will get you started writing your own assembly language routines, and this manual.

All of the programs include heavily commented source code, not only to show how they are used, but to explain how they work as well.

OVERVIEW

QuickPak Professional contains many different and varied utilities which are intended to be added to programs you write in BASIC. Literally hundreds of routines are provided to search and sort arrays, do windowing and quick printing, perform date and time arithmetic, and to access the keyboard in ways not possible using BASIC alone.

In addition to the DOS, video, and other services common to most "toolbox" products, we have also provided a number of major subprograms. For example, QEdit is intended to be used whenever you want to add a text editor to your programs. However, it could be turned into a full-blown word processor with very little additional programming. Likewise, the SPREAD.BAS subprogram provides a nearly complete spreadsheet, lacking only user-defined formulas and macro interpretation.

Other major subprograms include a sophisticated pull-down menu complete with mouse support, a vertical menu with scrolling, a dialog box data entry system, and several pop-up utilities. In addition, QuickPak Professional comes with routines to search, browse, and encrypt files, and sort an array or file on any number of keys.

Finally, an assembler program is provided that will transfer the contents of a graphics screen *in any screen mode* to nearly any printer. Supported printers include those that follow either the Epson/IBM dot-matrix standard, or accept the codes used by the Hewlett-Packard LaserJet. Besides honoring all of the BASIC-supported graphics modes, this routine will also provide intelligent pattern-substitution based on the screen colors.

There are many, many other programs included, and only by examining this manual in detail will you become acquainted with all of them. Most of the routines are amply illustrated by an accompanying demonstration program. All of the demo programs for the BASIC programs begin with the letters DEMO, to make them easy to identify. BASIC programs with the same names as their assembler counterparts are used to illustrate those routines in context.

QUICKPAK PROFESSIONAL IS EASY TO USE

In designing QuickPak Professional, major effort has gone into making the routines as easy to use as possible, without sacrificing any of their power, flexibility, or speed.

For example, the number of passed parameters has been kept to the absolute minimum which helps to reduce the size of your programs. Further, all of the DOS services that require a drive letter will accept either upper or lower case, or a null string to indicate the default drive.

In addition, the routines that read a list of file and directory names from a disk are designed to read all of them in one operation. Where other methods require you to loop repeatedly getting them one by one until an error occurs, the QuickPak Professional routines process the entire directory from a single call.

All of the routines that expect a file or directory name are given exactly as they would be in BASIC. Where other commercial toolbox routines require you to append a CHR$(0) to the end of a DOS file name, we do this for you automatically.

Finally, QuickPak Professional employs functions where appropriate. In many cases this will greatly simplify their use. For example, to obtain the current default drive is as easy as:

```
CurDrive$ = CHR$(GetDrive%)
```

Contrast that with the usual method:

```
CALL GetDrive(Drive%)
CurDrive$ = CHR$(Drive%)
```

Clearly, the QuickPak Professional method is easier, and contributes less code as well.

QUICKPAK PROFESSIONAL VERSIONS

QuickPak Professional is available in two versions with one intended for use with QuickBASIC 4.0 or later, and another for QuickBASIC 2. The correct version of QuickPak Professional is required for use with each of these compilers.

This version of QuickPak Professional is for use with QuickBASIC 4.0, 4.5, BASIC 6.0, and BASIC 7. *x* PDS.

QUICK START

This manual covers the many important topics you will need to know to use QuickPak Professional effectively. Besides providing a list of each routine and its calling syntax, many other details are described in depth. In addition, a number of tutorials on various aspects of BASIC programming are also contained in the Tutorial.

If you are familiar with BASIC programming and want to begin right away, simply start QuickBASIC 4 by loading the PRO.QLB Quick Library like this:

```
QB /L PRO
```

If you are using BASIC PDS, then you would specify the PRO7.LIB file as follows:

```
QBX /L PRO7
```

Once BASIC has been started, you may run any of the demonstration programs to quickly see what the various routines do, and how they are called. Many of the demonstration programs start with the letters DEMO, to make them easy to identify from the QuickBASIC editor Files menu.

The following sections are intended as an introduction to using libraries and multiple program modules for those not familiar with these concepts.

Introduction to QuickPak Professional

We have designed QuickPak Professional to be as easy to use as possible. Quick Libraries are provided and are readily available to be loaded with QuickBASIC. Many useful and informative demonstration programs are also included, and our technical support staff is eager to assist you when help is needed.

This tutorial is intended for programmers who are not familiar with using external subroutines and libraries. It also includes useful information about advanced concepts employed by some of the QuickPak Professional routines.

USING LIBRARIES

When you compile a BASIC program, the Microsoft BC compiler converts your code into equivalent machine language instructions and places the result in an object file with an .OBJ extension. Programs written in other Microsoft languages such as assembler, C, or Pascal are also compiled and converted to the same object file format. Subroutines written in these languages can be added to your compiled BASIC object files when the final program is created with LINK.EXE.

Subroutines contained in object files can also be placed into a library, simplifying the linking process. A library is simply a collection of object files, and its advantage is that you do not need to explicitly specify which object files are to be added to the program.

There are two types of libraries that you need to know about when using QuickPak Professional: Quick Libraries with a .QLB extension, and regular libraries with an .LIB extension.

The Quick Library is a special type of library made from any combination of object files and regular .LIB libraries. Because the QuickBASIC editor is, in fact, an interpreter and not a true compiler, Quick Libraries are needed in order to access external compiler assembled subroutines. It is important to understand that a Quick Library is used only while you are in the QuickBASIC editor.

You load a Quick Library when starting QuickBASIC, by using the /l option switch as follows:

```
qb /l pro
```

The /l switch tells QuickBASIC to load a Quick Library — in this case PRO.QLB. Note that the .QLB file extension is implied and not required. If the Quick Library is in a directory or drive other than the current default, a complete DOS path to the library must be given after the /l switch as follows:

```
qb /l d:\quickpak\pro
```

QuickBASIC allows only one Quick Library to be loaded at a time. Therefore, all of the external routines that your program will be using in the QB environment must be contained in that Quick Library. This is not a problem if you are using only QuickPak Professional routines, since both the .LIB and .QLB libraries contain the same routines. However, if you need to access routines from more than one library while in the QB editor, it will be necessary for you to make a single quick library from the various libraries required by your program. This will be discussed in the following section that describes how to build Quick Libraries.

The second type of library that you will need to know about is the .LIB library. This library is used in the linking process and is accessed by LINK.EXE whenever you use external library routines in your programs. In fact, LINK also includes routines contained in the .LIB libraries that come with BASIC.

When you are in the QB environment and select "Make .EXE file..." from the pulldown menu, QB checks to see if there is a Quick Library loaded. If there is, it searches for a corresponding .LIB library and adds the required routines from that library to your program. If you are linking from the DOS command line, you must specify this .LIB library at the end of your link line, thus:

```
link /options myprog , , nul, pro ;
```

In this case LINK assumes an .LIB extension, so it is not entirely necessary to include it. Understand that LINK is very smart, and includes only those routines that are actually called by your program. You may also specify that multiple object files and libraries be included:

```
link /options object1 object2 , , nul , lib1 lib2 ;
```

BUILDING LIBRARIES USING LIB.EXE

Libraries are built from object files. Once a file has been compiled to an .OBJ file format, it may be added to a library by using the LIB.EXE program included with QuickBASIC. LIB.EXE is a utility program that allows you to manipulate .LIB libraries. For example, to create a new library, you would type the following command at the DOS prompt:

```
lib libname +my.obj +his.obj +her.obj ;
```

Here, libname is the name of the library that you are creating, and the plus sign tells LIB.EXE that the various files with an .OBJ extension are to be added to this library. You may also include one or more complete libraries:

```
LIB libname + object1 +lib1.lib +lib2.lib
```

LIB.EXE assumes an .OBJ extension when adding modules to a library. Therefore, you must include the .LIB extension when adding an existing library file.

LIB.EXE may also be used to delete, replace, or extract a copy of selected .OBJ files from an existing library. The syntax for these operations is as follows:

```
lib pro +my.obj +your.obj
lib pro -my.obj -her.obj
lib pro *my.obj *her.obj
```

The first example adds the routines in MY.OBJ and YOUR.OBJ to the PRO.LIB library. The second deletes the routines contained in MY.OBJ and HER.OBJ from the library. The last example extracts a copy of the routines, without affecting the library.

In case you are wondering why you would ever want to modify a library, consider this: When you are in the QuickBASIC environment, the Quick Library that is loaded occupies memory. As your program grows in size, it too will require more memory. At some point, the combined size of both your code and the loaded Quick Library may exceed the amount of available memory. This will cause an "Out of string space" or "Out of memory" error message. Since it is very unlikely that you are using every routine in QuickPak Professional, you can preserve a substantial amount of memory by using a subset of PRO.QLB containing only the routines that your program actually needs.

There are two ways you can build a subset library. One is to extract the object files that your program needs by using the asterisk (*) command of LIB.EXE, and then build a new Quick Library from those files. Another way is to delete the routines that you do not need from a copy of the .LIB library by using the "-" command. You would then create a new Quick Library by utilizing this modified .LIB library. Please understand that there is no need to reduce the size of an .LIB library. Rather, it is only a Quick Library that is loaded into memory. However, creating a new .LIB library can in some cases make it easier to create a new Quick Library that contains the same routines. This will be discussed in further detail in the section entitled *Building Quick Libraries and Using MakeQLB.*

When many commands are needed to manipulate an .LIB library, you may optionally place those commands into a file, instead of having to enter them all at once from the command line. Such a file is called a *response file*, because it holds a series of responses to the LIB.EXE prompt messages.

A LIB.EXE response file is simply an ASCII text file that lists each command on a separate line. A typical LIB.EXE response file is shown below.

```
+ my.obj    &
+ your.obj  &
+ his.obj   &
+ \qb\their.obj ;
```

The ampersand (&) at the end of each line tells LIB.EXE that more commands will follow. To indicate the end of the response file, a semicolon rather than an ampersand is used following the last .OBJ file. By convention you should save this file with an .RSP extension, and then run LIB.EXE as follows:

```
lib libname @my.rsp
```

The at sign (@) tells LIB.EXE that MY.RSP is a response file, and not an object file with that name.

Another useful feature of LIB.EXE is its ability to create a file that lists all of the routines in a specified .LIB library. This is called a list file, and can be created by invoking LIB.EXE manually as follows:

you enter:

```
lib pro
```

LIB.EXE responds:

```
Operations: <Enter>
List File: pro.lst <Enter>
```

When LIB.EXE asks what operations you want to perform, just press Enter. When it asks for the name of a list file, enter PRO.LST and press Enter.

Once LIB.EXE has created this file, it may be viewed in any ASCII text editor. A list file is organized in two different ways. The first portion alphabetically lists all of the routines found in your library along with the object module that contains it. The second portion of the file alphabetically lists each object module in the library, showing the routines it contains below and to the right. Understand that the object module is the name you will provide when adding or extracting object files; the routine names are what you use when actually calling the routine.

In many cases these names are the same. For example, the QuickPak Professional QPrint routine is contained in an object file named QPRINT.OBJ. However, some routine names are longer than eight characters, and the object file name must be slightly altered.

The information contained in a list file will help you to determine what module name to extract, based on the call name listed in the QuickPak manual. It can also be used to verify whether or not a particular routine is in your library.

Now that you know how to create and modify .LIB libraries, let's take a look at how to create a Quick Library.

BUILDING QUICK LIBRARIES AND USING MAKEQLB

As mentioned before, Quick Libraries are used only while you are in the QuickBASIC environment. There are several ways to create a Quick Library, but the easiest is to use the MakeQLB utility program that we have included with QuickPak Professional.

To use MakeQLB you must first compile and link MAKEQLB.BAS as shown in the program's header comments. Once you have created the MAKEQLB.EXE executable program, you are ready to put it to use.

If your program already exists and you want to create a smaller subset Quick Library, run MakeQLB as follows from the DOS command line:

```
makeqlb
```

MakeQLB will then prompt you for the information it needs, using an interface similar to LIB and LINK:

```
Main Module Name [*.BAS]:
Output Library Name [*.QLB]:
List File Name [*.LST]:
INPUT libraries [PRO.LIB]:
BQLB## Library Name [BQLB45.LIB]:
```

At the first prompt, simply type in the name of your main module with or without an extension (.BAS is implied), and press Enter.

The second prompt asks for the name of the Quick Library that is being created. The .QLB extension is implied, and does not need to be entered. If you simply press Enter, MakeQLB will use the same name as your main module; however, it employs a .QLB extension.

The third prompt is the name of the list file that MakeQLB will create, and this file will hold the names of all of the routines that are added to the Quick Library. If you press Enter, MakeQLB will use the same name as the main module.

The fourth prompt is for the names of the .LIB libraries that contain the routines used by your program. The default name is PRO.LIB, and you may type in any additional library names that your program requires. Multiple libraries may be specified as well, with each separated by a space. Of course, each file name may optionally be preceded by a full DOS path name if necessary.

The final prompt is for the appropriate BASIC support library, and this name will vary depending on which version of BASIC you are using. BQLB45.LIB is the default, which is meant for QuickBASIC version 4.5. For QuickBASIC version 4.0, the support library is BQLB40.LIB, and for version 4.00b it is BQLB41.LIB. If you are using BASIC 7.*x you will instead enter QBXQLB.LIB.*

After MakeQLB has created the Quick Library, it writes a list file. This is simply an ASCII text file containing the names of the routines that were just added to the Quick library. This list file can be very useful for a number of reasons. First, it lets you verify the routines that are actually in your Quick library. But more important, you may modify this file to either add or delete routines from your Quick library. You may then run MAKEQLB.EXE again, but this time by specifying the list file instead of the BASIC source program. You may also manually create a list file from scratch, and avoid the time required for MakeQLB to process your source program.

The format for a MakeQLB list file is very simple. You merely use any ASCII text editor, and type the name of the routine as it appears in your manual on a single line. The next routine goes on the next line, and so forth:

```
APrint0
BLPrint
OneColor
AMenuT
...
...
```

The most common problem people encounter when running MakeQLB is that they fail to specify all of required .LIB libraries. This results in the error message "Unresolved external in CHRISMAY.OBJ". If you instead receive an error message that indicates an Unresolved External and the subroutine is written in BASIC, then it is most likely that these modules were not loaded when you saved your main program. Finally, if you get the error message "Subscript out of range", then you failed to use the /ah switch when compiling MAKEQLB.BAS.

It is important to understand that MAKEQLB.EXE creates only a Quick Library. Since you may use as many .LIB libraries as necessary when linking, there is no need to combine them into a single library. Further, .LIB libraries are not loaded into memory, so there is no need to develop a subset library containing only the routines that you are using.

If you prefer to manually create a Quick Library, then you will have to use the version of LINK.EXE that came with your compiler. Manually creating a Quick Library from the DOS command line is described in detail elsewhere in this manual.

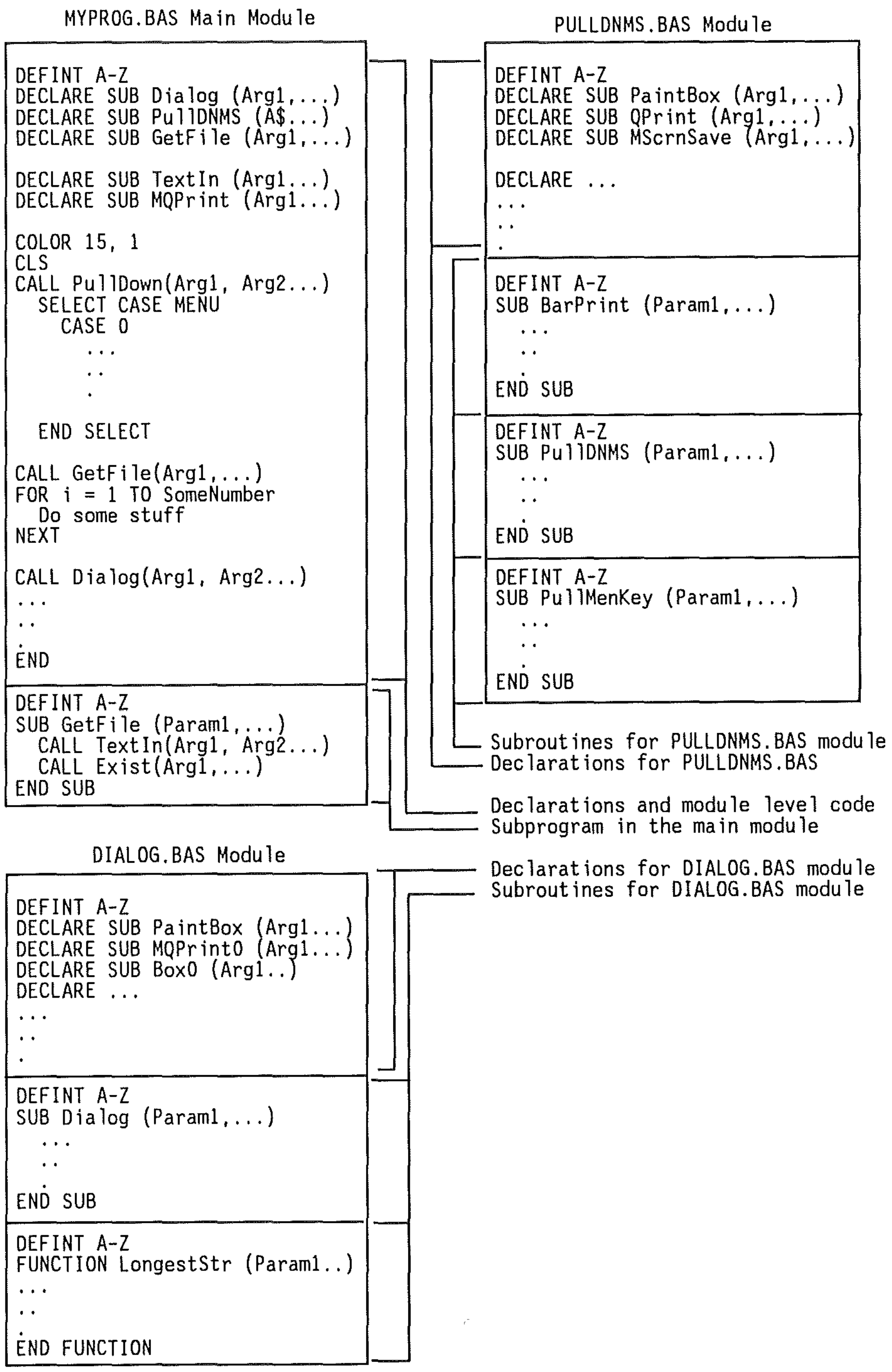
MODULES AND SUBPROGRAMS

Beginning programmers using QuickBASIC are often confused about the differences between modules, subroutines, and what Microsoft calls module-level code. A module is simply a file that holds BASIC source code. Every program has a main module, and some programs also have additional modules that contain ancillary subprograms and functions.

External modules can be thought of as a "folder" that holds the subroutines. In such a module, the only executable code is in the actual subroutines, and the main portion may contain only compiler directives such as DECLARE, CONST, and TYPE definition statements. When you press F2 while in the QuickBASIC editor, a dialog box appears listing all of the BASIC subs and functions in your program. The text that appears shows each module in capital letters, and the subprograms and functions it contains are indented and in mixed case.

One of the biggest advantages that modules offer is easy expansion to your programs. You may add routines that you (or we) have already written, tested, and debugged, and simply make calls to them from anywhere in your program. This also lets you reuse the same routines in more than one program; therefore, it reduces the amount of programming effort needed.

Except for a few isolated cases, module-level code must be located in the main module. The only exceptions to this are DEF FN-style functions, and the target of an ON ERROR label. The first executable statement in your main module is the first statement that will be executed by your program. The example that follows shows this in context.



MYPROG.BAS Main Module

```
DEFINT A-Z
DECLARE SUB Dialog (Arg1,...)
DECLARE SUB PullDNMS (A$...)
DECLARE SUB GetFile (Arg1,...)

DECLARE SUB TextIn (Arg1...)
DECLARE SUB MQPrint (Arg1...)

COLOR 15, 1
CLS
CALL PullDown(Arg1, Arg2...)
  SELECT CASE MENU
    CASE 0
      ...
      ..
      .

  END SELECT

CALL GetFile(Arg1,...)
FOR i = 1 TO SomeNumber
  Do some stuff
NEXT

CALL Dialog(Arg1, Arg2...)
...
..
.
END
```

```
DEFINT A-Z
SUB GetFile (Param1,...)
  CALL TextIn(Arg1, Arg2...)
  CALL Exist(Arg1,...)
END SUB
```

DIALOG.BAS Module

```
DEFINT A-Z
DECLARE SUB PaintBox (Arg1...)
DECLARE SUB MQPrint0 (Arg1...)
DECLARE SUB Box0 (Arg1..)
DECLARE ...
...
..
.
```

```
DEFINT A-Z
SUB Dialog (Param1,...)
  ...
  ..
  .
END SUB
```

```
DEFINT A-Z
FUNCTION LongestStr (Param1..)
...
..
.
END FUNCTION
```

PULLDNMS.BAS Module

```
DEFINT A-Z
DECLARE SUB PaintBox (Arg1,...)
DECLARE SUB QPrint (Arg1,...)
DECLARE SUB MScrnSave (Arg1,...)

DECLARE ...
...
..
.
```

```
DEFINT A-Z
SUB BarPrint (Param1,...)
  ...
  ..
  .
END SUB
```

```
DEFINT A-Z
SUB PullDNMS (Param1,...)
  ...
  ..
  .
END SUB
```

```
DEFINT A-Z
SUB PullMenKey (Param1,...)
  ...
  ..
  .
END SUB
```

Subroutines for PULLDNMS.BAS module
Declarations for PULLDNMS.BAS
Declarations and module level code
Subprogram in the main module
Declarations for DIALOG.BAS module
Subroutines for DIALOG.BAS module

This example graphically illustrates how a three-module program is organized by BASIC. The dummy code inside the three main boxes is what might be found in the individual module text files. These boxes are then subdivided to show what portion of code would appear inside the Quick Basic editing window when you select the module or subprogram by that name.

Note that there is no module level code in the two modules PULLDNMS.BAS and DIALOG.BAS. The only executable code is in the individual subprograms within those modules, and the only executable module level code is in the main module. The main module can then make calls to any subprogram or function in any module. In addition, once a procedure has been called, it too can access other procedures in any module.

Finally, notice that the DECLARE statements that are needed for a given module depend only on the procedures that will be accessed from that module. It is not necessary to declare routines in other modules if they are not going to be called directly.

ADDING QUICKPAK PROFESSIONAL ROUTINES TO YOUR PROGRAMS

Nearly all of the routines in QuickPak Professional expect integer arguments. The easiest way to insure that only integers are passed is to use DEFINT A-Z as the first line in any module or subroutine that calls a Quickpak Professional routine. This statement tells the compiler that variables starting with the letters A through Z (that is, all variables) are to be treated as integers unless otherwise specified.

If you do not add the DEFINT A-Z statement, BASIC assumes that all variables are single precision. If you need to have variables of a different type, you may append the appropriate type suffix to those variables to override the integer default:

```
X! = 14.2      'single precision
Y# = 173.9     'double precision
Z& = 1098657   'long integer
```

There are several advantages to using integer variables wherever possible. First, operations on integers are many times faster than the equivalent operations on single or double precision numbers. Second, integers require less memory since they only take 2 bytes of memory. Single and double precision variables require 4 and 8 bytes respectively. In addition, if you call an assembler routine that expects integer arguments with any other type of variable, you are certain to get the wrong results. In some cases you could also lock up the PC requiring a reboot.

Before you add a QuickPak Professional routine to your program you will need to determine whether the routine is a BASIC or assembler subroutine or function. This is clearly documented at the top of the page that describes each routine. If you are adding an assembler routine, you can access the routine with a CALL, as shown in the routine's description found in the manual.

Initially, you must load PRO.QLB as described earlier, if the routine is written in assembly language. If the routine is a BASIC subroutine or function, you will instead add it to your program by using the *Load* option of QuickBASIC's *File* menu. For example, if you wanted to use the QuickPak Professional Parse function contained in the module FNOTHER.BAS, you would load FNOTHER.BAS as a module.

It is also important that you declare the various QuickPak Professional routines correctly. We have provided a module called DECLARE.BAS, if you are not sure of the correct syntax. This file contains the appropriate declarations for every routine contained in QuickPak Professional. We suggest that you load DECLARE.BAS as a document, and then copy the individual DECLARE statements to your program as needed. Although you could load DECLARE.BAS as an Include file, this will reduce substantially the amount of memory that is available for your program.

ORGANIZING YOUR DIRECTORIES

As our installation instructions indicate, we recommend that you create a directory called \PRO, and place all of the QuickPak Professional BASIC modules and library files in that directory. Once this is accomplished, you will probably want to run the various demonstration programs from within the QuickBASIC environment. For the sake of this discussion, we will assume that your version of QuickBASIC is installed in a directory called \QB45 located on drive C.

The easiest way to get started is to change to the \PRO directory, and then type the path to QuickBASIC and start QB as follows:

```
C:\> cd \pro
C:\PRO> \qb45\qb /l pro
```

This starts QB.EXE from the \QB45 directory, although that directory is not the current one. If your system PATH is set to include C:\QB45, then you do not have to type in the full path name.

Since the \PRO directory is still the current directory, you may now select Open or Load from QuickBASIC's File menu. The dialog box will then display all of the files with a .BAS extension in the current directory. To run the demonstration programs you must use Open from the file selection menu, not Load. Open tells QB to also look for a corresponding .MAK file, which indicates additional modules that are to be loaded automatically.

Many of the QuickPak Professional demonstration programs have a .MAK file; in particular, those that show how to use a BASIC subprogram or function. In that case, the demonstration program is the one you will load and run initially, and the demonstrated QuickPak Professional routine is loaded along with it, based on its name being listed in the .MAK file. Load is primarily used when adding a module to an existing program.

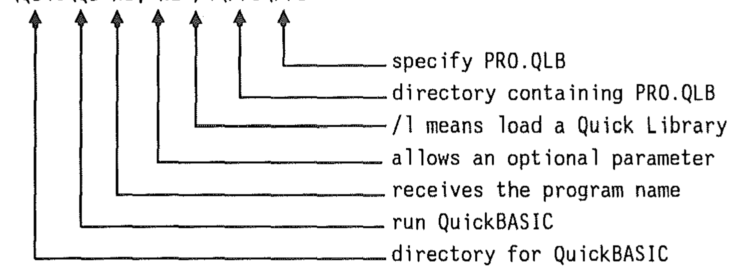
Although this is the method we recommend — at least when getting started — there are other ways to organize your directories. Some programmers place all of their .BAS files and libraries in one enormous directory along with QuickBASIC. However, this defeats the purpose of using directories, since the whole point is to let you keep related files and modules in their own directory. At some point, with such a large directory, it can become very confusing as to which files came with QuickBASIC, which are part of QuickPak Professional, and which are those you have written.

To eliminate this confusion, we recommend that you keep a separate directory for QuickBASIC, a separate directory for QuickPak Professional, and a separate directory for each project that you work on. With this arrangement you can clearly see which files go with which project. When you need a BASIC source file from QuickPak Professional, you can load it from the \PRO directory, and that directory will be remembered in the main program's .MAK file. Then, each time you want to work on a program you will first change to the appropriate directory, and start QuickBASIC like this:

```
\qb45\qb program /l \pro\pro
```

You can also create a short batch file to perform this automatically for you. As long as the batch file is in a directory listed in the system PATH setting, it can be run from any other directory on your hard disk. The sample QB.BAT batch file that follows shows one way to do this:

```
\QB45\QB %1, %2 /l\Pro\Pro
```


specify PRO.QLB
directory containing PRO.QLB
/l means load a Quick Library
allows an optional parameter
receives the program name
run QuickBASIC
directory for QuickBASIC

You can create this file in any ASCII text editor, or by using the DOS COPY CON: command. Then to start QuickBASIC you simply enter:

```
qb program
```

Where *program* is the name of the main program to load, and the %1 and %2 parameters are replaced with the name of the program you enter and one optional parameter such as /ah.

Since you will not be in the \PRO directory while QuickBASIC is running, you will need a way to tell QuickBASIC where the PRO.LIB file is when you create an .EXE program. This is the purpose of the LIB= environment variable. If you add the statement LIB=C:\PRO to your AUTOEXEC.BAT file, LINK will read that variable and know to look there for PRO.LIB.

With Microsoft PDS 7.1 object files may also be placed in the directory set with LIB=, and LINK.EXE will be able to find these files as well. This can be useful if you have an object module that is not in a library, but is used often in your programs. This way you do not have to type in the complete path to it each time you run LINK.EXE.

USING POLLED ROUTINES

When you call most subroutines, execution is passed to that routine until it has completed its task. Usually this is what you want. However, there are times when it is desirable to have the routine return periodically, so your program can monitor what is happening while it is working. For example, when you use BASIC's INPUT statement, you have no control over what happens or for how long, until the operator presses Enter. Also, BASIC does not provide any way for you to recognize the F1 key, when pressed, to display a help screen. To solve this problem, several of the QuickPak Professional routines are designed to be called repeatedly in a loop, allowing you to monitor what is going on as they operate.

One example of a polled routine that you are probably familiar with is BASIC's INKEY$ function. Using INKEY$ in a loop as shown below lets you check for a key press and print it on the screen, while simultaneously printing the current time.

```
DO
   Ky$ = INKEY$
   LOCATE 1, 1: PRINT Ky$
   LOCATE 2, 1: PRINT TIME$
LOOP UNTIL Ky$ = CHR$(27)        'until they press Escape
```

BASIC's INPUT$ function can also be used to return a single key press. But if it were used in place of the INKEY$ function above, the program will halt until a key is pressed. Therefore, INKEY$ is a very simple example of a pollable subroutine. As you can see, polling opens up many possibilities that are not otherwise possible with INPUT; in fact, it is not possible with most subprograms.

A number of subroutines in QuickPak Professional take advantage of the additional power that a polled routine can provide. For example, the PullDown and PullDnMS menu routines may be called in either a polled or non-polled manner. This is accomplished by setting the Action argument in the call to either 1 or 0, respectively.

When called in a polled mode, these menu routines allow the calling program to determine what menu is currently being viewed, which choice is currently highlighted, and which key was last pressed by the operator. This allows your program to display a descriptive message based on the current menu choice while the menu is still active. When PullDown and PullDnMS are called in a non-polled manner, your program cannot do anything until the user selects a choice from the menu or presses Escape.

Another example of a QuickPak Professional polled routine is the QEdit text editor. This subroutine is a complete word processor with word wrap, cut, copy, and paste operations, and full mouse support. Using QEdit in a polled mode lets you monitor what keys are being pressed even while text is being entered or edited. This way you can test for specific keys and act on them, without having to make any changes to the QEDIT.BAS source file.

For example, you may wish to look for a specific function key and if pressed, change the margins, print the file, or perhaps save the text to disk. Without the polling capability of this routine, you would have to wait until Escape was pressed and then display a menu of choices. Therefore, polling lets you extend the capabilities of QEdit, and also regain control between keystrokes.

The following example shows how this is done.

```
Action = 1                         'Action = 1 sets the polled mode

DO
  CALL QEdit(Array$(), Ky$, Action, Ed)
  IF LEN(Ky$) = 2 THEN             'Test for extended key
    Ky = ASC(RIGHT$, 1)
    SELECT CASE Ky
      CASE 60                      'F2
        CALL ChangeMargin          'Change the margins
      CASE 67                      'F9
        CALL PrintFile             'Print the file
      CASE 68                      'F10
        CALL SaveFile              'Save the file
    END SELECT
  END IF
LOOP UNTIL Ky$ = CHR$(27)          'Loop until user presses Escape
Action = 5                         'restore the previous screen
CALL QEdit(Array$(), Ky$, Action, Ed)
```

Here, QEdit is called with an initial action of 1, which starts it in polled mode. When QEdit is called this way, it looks at each keystroke, handles it as required, and then exits the subprogram without delay. Since QEdit does not test for function keys other than F1, the next section of code in the loop checks for the F2, F9, and F10 function keys. This code will execute the appropriate CALL if one of these keys is pressed, and will then loop back to call QEdit. This process repeats continuously until the user eventually presses Escape.

It would appear to the typist that everything is happening within QEdit; however, part of the code that is executing is actually external to the routine. The DEMOEDIT.BAS program shows this type of polling in context.

Note that this type of extensibility is often associated with Object Oriented programming. As you can see, this polling technique provides a similar capability to BASIC subprograms and functions.

USING ACTION PARAMETERS

When you call a pollable QuickPak Professional routine using an Action value of zero, it behaves like a normal, non-polled subroutine. Most of the pollable menu routines begin by saving the underlying screen. Then the menu is displayed, and the user is allowed to navigate the choices by using the various cursor keys. When Enter or Escape is finally pressed, the routine restores the screen to what it had been, erases the menu, and then returns to your program.

When you call the same routine in polled mode, the internal process is slightly different. First, you call the routine with an Action of 1, which establishes the polled mode. But after saving the screen and displaying the menu, Action is set to 3 by the routine. If a key has been pressed it is acted on. However, the important point is that the subroutine returns immediately, regardless of what keys were or were not pressed.

The next time through the loop with Action set to 3, the routine skips over the initialization and display code, and goes directly to the key processing portion. After acting on any keystrokes as needed, it again returns immediately.

This process continues as long as Action is set to 3. If Action is set to 4, you know that the user has pressed a terminating key (Enter or Escape). This is a signal for you to examine the values returned, and to set Action to 5. Action 5 tells the subroutine to restore the original screen and then exit.

The last Action variable is 2. With Action set to 2, you are telling the subroutine to redisplay any data passed to it, but without redrawing the surrounding window. This also resets Action to 3, so you don't have to assign that value manually to continue polling. Using an Action value of 2 is useful when you determine that the current screen information needs to be updated or modified. For example, QEdit lets you establish any arbitrary row to be positioned anywhere on the screen. Once you have set the correct parameters for this, you would then use Action = 2 to force QEdit to update the display screen.

PASSING THE CNF VARIABLE

The Cnf variable is used by a number of QuickPak Professional routines. It is a TYPE variable that is used to pass color values and certain system information to those routines that require them. The advantage of using a TYPE to relay this information is that only one parameter is required, although many different values are being passed.

For example, the PullDown menu subprogram requires several different colors — text foreground color, background color, highlight color, inactive text color, and so on. Passing a single parameter makes the call much simpler. Reducing the number of parameters also helps to reduce the size of your programs. But more important, when Cnf is used correctly it will automatically sense the type of display (color or monochrome) that is being used, and assign a set of colors to their appropriate values.

An additional benefit of using Cnf is that if you call several different routines that use Cnf, you will be able to define a consistent look for your programs, since all the routines will be using the same color scheme. Cnf also tells the various menu routines if a mouse is installed in the host PC.

One of the most common problems that new users experience is failing to understand how to correctly set up and pass the Cnf TYPE variable. The process is fairly straight-forward, though perhaps a bit confusing at first. There are two Include files supplied with QuickPak Professional that make this process as easy as possible. These files are DEFCNF.BI and SETCNF.BI, and they are used in the following manner:

In the main module of your program just after any DEFtype and DECLARE statements, you will include the DEFCNF.BI file. This file simply defines the Config TYPE structure, which is used when dimensioning the Cnf variable. Immediately following that you will include the SETCNF.BI file. (.BI is a Microsoft convention, and stands for BASIC Include.) SETCNF.BI also uses the QuickPak Professional Monitor function to determine the type of display adapter being used. Additional code then sets the colors fields in Cnf to the appropriate values. Since SETCNF.BI requires the Monitor function, you must also declare Monitor in your program. The code in your main module would appear as follows:

```
DEFINT A-Z
DECLARE SUB AnySub (A, B)
DECLARE FUNCTION AnyFunction ()
DECLARE FUNCTION Monitor% ()

'$INCLUDE: 'DefCnf.Bi'
'$INCLUDE: 'SetCnf.Bi'
```

We suggest that you look at the code in these include files, just so you can see what they are doing. You can do this either by loading the files into the QuickBASIC editor, or by selecting "View included lines" from the QuickBASIC pulldown menus.

In most cases you can merely call the various QuickPak Professional routines that use the Cnf variable, as long as DEFCNF.BI and SETCNF.BI have been included in your program as shown. However, a complication arises when you are calling one subprogram that, in turn, calls one of the menu programs that requires Cnf. In this case, you must first pass Cnf to your subprogram, and then pass it on to the QuickPak Professional routine that expects it.

Say, for example, that you have written a subroutine called GetInput, and this routine may occasionally need to display an error message. The QuickPak Professional routine MsgBox is ideal for this situation, but it does require the Cnf variable. To make this work, you would pass Cnf to the GetInput subroutine as a parameter, and then call MsgBox. Since Cnf has been passed to the GetInput subroutine, MsgBox will also be able to access it. This is shown following.

```
'MAIN.BAS
DEFINT A-Z
DECLARE FUNCTION Monitor% ()
DECLARE SUB MySub(X, Y, Z, Cnf AS ANY)
'$INCLUDE: 'DEFCNF.BI'
'$INCLUDE: 'SETCNF.BI'
...
...
CALL MySub(X, Y, Z, Cnf)
...
...
SUB MySub(X, Y, Z, Cnf AS Config)
  ...
  ...
  CALL MsgBox(Message$, Wide, Cnf)
  ...
END SUB
```

If Cnf had not been passed into MySub, the Cnf referenced within MySub would appear to BASIC as a regular numeric variable. This would cause BASIC to report a "Type mismatch" error, since it knows that MsgBox is expecting a variable of type Config.

The only additional complication arises if you are calling an intermediate subprogram or function in another module; you must then add the TYPE declaration file DEFCNF.BI to that module. Without including DEFCNF.BI, Cnf would again appear to the second module as a simple integer variable, and again cause a "Type mismatch" error message.

Note that in the module-level code, the declaration for the subroutine that calls MsgBox defines Cnf AS ANY. This is needed because DEFCNF.BI has not yet been included, yet we do not want BASIC to treat Cnf as an integer.

As supplied, Cnf has already been assigned attractive colors for both color and monochrome monitors. However, you should feel free to change these colors to whatever you prefer, by simply modifying the SETCNF.BI file.

TO CALL OR NOT TO CALL

Depending on your programming style, you may either prefer to use the CALL statement to invoke a subroutine, or not use it. If you do not use the CALL statement, you must declare every routine that you plan to access. If you do use the CALL statement, then you do not have to declare subroutines, although you may opt to. Regardless of whether or not you use a CALL statement, you must always declare BASIC or assembly language functions.

The following two statements are equivalent in QuickBASIC:

```
CALL NameFile(OldName$, NewName$)
```

or

```
DECLARE SUB NameFile (A$, B$)
 ...
 ...
NameFile OldName$, NewName$
```

Although the arguments in the declaration statement do not use the same names as those in the actual CALL statement, note that they must be of the same data type.

The reason why you must always declare functions should be clear when you consider the following BASIC statement:

```
Print MonthName$(Month)
```

When the compiler sees this statement, it would assume that you want to print the string found in the MonthName$ array, element Month. However, MonthName$ is actually a QuickPak Professional function that returns a month's name in string form, given the month's number 1-12. Without an explicit declaration for this function, QuickBASIC has no way to know that you want to call a function named MonthName$, as opposed to accessing an array with that name.

USING THE QUICKPAK PROFESSIONAL FILE ACCESS ROUTINES

One of BASIC's few quirks is the way that disk and other errors are handled. Unlike other languages that let you test the success or failure of the most recent operation, QuickBASIC instead requires ON ERROR. It is up to you to first establish a block of code that will receive control when an error occurs, and then figure out what happened and where you were before the error occurred. Besides being unnecessarily convoluted, ON ERROR makes your programs larger and also makes them run more slowly.

QuickPak Professional lets you avoid ON ERROR by including replacement file access routines that return an error code. After any operation that may result in an error, you can simply query the QuickPak Professional DOSError and WhichError functions. The short program fragment shows this in context:

```
CALL CDir("\QB45")
IF DosError% THEN
 PRINT "Unable to change to the \QB45 directory"
END IF
```

All of the QuickPak Professional routines operate this way, although some routines return additional error information in their passed parameters. The DOSError function merely tells if an error occurred, and WhichError lets you determine which one. WhichError returns the same codes that BASIC does for the same situations. Both of these routines are designed as functions, and must therefore be declared.

OPENING FILES

QuickPak Professional includes a complete set of routines that can optionally replace BASIC's file handling statements. Which you use depends on your needs, and there are advantages and disadvantages to both. If you are writing a small program that will only occasionally be used, using ON ERROR makes a lot of sense. But for programs where size and performance are critical, using the QuickPak Professional replacements will probably be worth the added effort.

Before a file can be accessed by using either method, it must first be opened. Unlike BASIC's OPEN statement that accepts an argument to specify the access mode (INPUT, OUTPUT, APPEND, BINARY, or RANDOM), the QuickPak Professional routines operate in binary mode only. We do include routines for inputting lines of sequential text and accessing fixed-length records. However, from the perspective of DOS, all file access is purely binary. When you use BASIC's OPEN, it is BASIC that remembers the mode, and it is BASIC that prevents you from using LINE INPUT with a random access file. The QuickPak Professional file routines make no such distinction, and this added flexibility is yet another reason to use them.

Unlike BASIC's OPEN, the QuickPak Professional FOpen family of routines will not create a file if it does not exist. If you plan to read from an existing file only, then FOpen can be used in the same way as BASIC's OPEN FOR INPUT. Likewise, if you plan to write to a file that already exists — if it currently has a length of zero — then FOpen will also work as expected. However, if you need to create a file, you must first use FCreate. You can use the Exist function to determine if a file already exists.

Like BASIC's OPEN FOR OUTPUT, FCreate will create a file if it does not exist, or truncate it to a length of zero if it does. There are two additional OPEN replacement routines: FOpenS and FOpenAll. FOpenS (the S stands for Shared) is intended for use in a network application, and lets you read and write data to disk while allowing others read access only. FOpenAll is similar, except it lets you specify all of the possible sharing options.

The three FOpen routines are similar in that they expect the name of the file and a Handle parameter. With BASIC's OPEN you make up a file number, then use the number whenever you need to access the file. Contrast that with the method used by QuickPak Professional, where the file number is returned to you. Please understand that this is the same method that DOS uses, and indeed, all other high-level languages. It is important to understand that the Handle is returned to you by these routines, and you should never assign it yourself.

After you have called one of the FOpen routines, you will check to see if an error has occurred. For example, an error is possible if the diskette drive door is open, or if the file to be opened was not found. You will use the DOSError and WhichError functions to detect errors, as shown below.

```
IF NOT Exist%(FileName$) THEN       'no such file
 CALL FCreate(FileName$)            'so create it
END IF

CALL FOpen(FileName$, Handle%)      'then open it
IF DOSError% THEN                   'check for an error
  BEEP                              'sound an alarm
  LOCATE MsgRow, 1                  'prepare to display
  PRINT ErrorMsg$(WhichError%)      'print an error message
  EXIT SUB                          'bail out
END IF
```

DOSError would detect an error if it had occurred during the opening process. Then, the WhichError and ErrorMsg$ functions can be used to display an appropriate message. WhichError returns an error number, and ErrorMsg$ returns an equivalent string. For example, if you attempt to open a file that does not exist WhichError will return 53, and ErrorMsg$ will return "File not found". If no error occurred, you would continue the program and either read or write data to the file.

SEQUENTIAL FILES

Normally when you save a sequential file in BASIC, you use either the WRITE# or PRINT# statements. These statements write a string to disk, followed by carriage return and line feed characters. These characters later tell INPUT or LINE INPUT that they have encountered the end of a line. With QuickPak Professional you instead use the FPut family of subroutines to write sequential data to disk. FPut was designed to write as many bytes to disk as are in the string being written.

This is ideal for binary files, and with only slightly more work, FPut can also be used for sequential files as well. Since FPut does not automatically add a carriage return and line feed to the end of the string that you are saving, you must add these manually:

```
Source$ = "Crescent Software"
CRLF$ = CHR$(13) + CHR$(10)
CALL FPut(Handle%, Source$ + CRLF$)
...                                 'write other data here
...
CALL FClose(Handle%)                'then close the file
```

To retrieve sequential data from a file with BASIC you would normally use the LINE INPUT# command. With QuickPak Professional you will instead use FLInput. FLInput is similar to BASIC's LINE INPUT statement, except that it too will set an error flag that may be detected with DOSError should an error occur while you are reading. You must also create a temporary buffer string to hold the data as it is being read from disk.

To read data from a file you might first check to see if the file name is valid, using the Exist function as before. Then you would open the file and read the data.

```
IF NOT Exist%(FileName$) THEN
  LOCATE MsgRow, 1
  Print "File not found"
  EXIT SUB
END IF

CALL FOpen(FileName$, Handle%)          'open the file
Buffer$ = SPACE$(82)                    'create a file buffer
Work$ = FLInput$(Handle%, Buffer$)      'Work$ receives data
...
...
```

FLInput will read data into Buffer$ until it finds a CHR$(13) line terminator, a CHR$(26) end of file mark, or until Buffer$ is filled. Please see the FLINPUT.BAS program for a working demonstration of the FLInput routine.

RANDOM FILES

BASIC uses the GET and PUT commands for accessing random data files with fixed-length records. GET and PUT can work in different ways, depending upon the mode in which the file was opened, and which optional arguments were used with PUT and GET. QuickPak Professional offers several replacement routines, with each designed for a specific use.

QuickPak Professional replaces GET and PUT with FPut, FPutR, FPutRT, FPutRTA, and FPutT, and their counterparts FGet, FGetR, FGetRT, FGetRTA, and FGetT. The routines containing an "R" (random) as part of the suffix will access data in a file at a designated record number. If the suffix contains a "T" (TYPE) then the routine is intended for use with TYPE and fixed-length string variables. If the suffix also includes an "A", the routine will read or write all or part of a TYPE or fixed-length string array.

Code for saving and loading a simple random file using a TYPE variable could be done as follows:

```
DEFINT A-Z                               'All integers please
DECLARE FUNCTION Exist (FileName$)       'Declare all functions

TYPE DBase                               'Define the TYPE
  FirstName AS STRING * 12
  LastName AS STRING * 20
END TYPE

DIM DRecord AS DBase                     'Create the TYPE variable
RecSize = LEN(DRecord)

DRecord.FirstName = "Crescent"           'Make something to save
DRecord.LastName = "Software"
RecNumber& = 1                           'Just 1 record req'd for
                                         ' this example

FileName$ = "C:\Cust.Dat"                'Assign the file name
IF NOT Exist%(FileName$) THEN            'See if it already exists
  CALL FCreate(FileName$)                'No, so create it
END IF

CALL FOpen(FileName$, Handle)            'Open the file
CALL FPutRT(Handle, DRecord, RecNumber&, RecSize) 'Read it
CALL FClose(Handle)                      'Close the file

DRecord.FirstName = ""                   'Clear the string
DRecord.LastName = ""

CALL FOpen(FileName$, Handle)            'Re-open the file
CALL FGetRT(Handle, DRecord, RecNumber&, RecSize) 'Get the record
CALL FClose(Handle)                      'Close the file
                                         'Print it to show it worked
PRINT DRecord.FirstName, DRecord.LastName
```

Finally, it is worth mentioning that with random files in BASIC, the first record number is 1, and the first byte in the file is also considered to be byte 1. With QuickPak Professional, while the first record in a random file is still 1, the first byte in a binary file is byte 0. This is important to understand when specifying a file offset with FSeek.

ADDING TO QUICKBASIC

Before we present a complete description of each program, let's first discuss some of the ways external routines may be added to QuickBASIC.

QuickBASIC allows calling external routines by name, which is a tremendous improvement over the older BASIC interpreter, where exact memory addresses must be known. Further, in addition to assembly language subroutines, QuickBASIC also allows creating subprograms in BASIC.

External routines and subprograms are accessed with the CALL command, and as many variables as required may be exchanged between them and the main program.

A new feature added to QuickBASIC beginning with version 4 allows invoking a BASIC or assembler subprogram without requiring the CALL key word, as long as the routine has been previously declared. This is the approach we have used in several of the demonstration programs that come with this version of QuickPak Professional.

All of the QuickPak assembly language routines are contained in the library files PRO.LIB and PRO.QLB for use with QuickBASIC 4 and BASCOM 6. The PRO7.QLB and PRO7.LIB files are intended for use with BASIC PDS. Note that if you are using BASIC PDS and compiling with near strings from the DOS command line, using PRO.LIB will result in slightly smaller programs.

The QuickPak Professional BASIC routines are provided in source format only, and are intended to be added by you to your programs. The BASIC programs will be added by loading them as modules using the Load option of BASIC's file menu.

Besides the library files, each individual assembler routine is available in source code format to aid your understanding of how they work. It is *not* necessary to know assembly language to use QuickPak Professional, and we provide the source code solely for the benefit of those who are interested.

If you are new to QuickBASIC and this all seems a bit confusing, don't worry—the instructions for each program contain all of the details, and many complete examples are included on the QuickPak Professional disks.

USING THE ASSEMBLER ROUTINES

The easiest way to begin is to start up QuickBASIC, and load the QuickPak Professional Quick Library into the editor. From the DOS prompt enter:

```
QB /L PRO
```

If you are using BASIC PDS instead enter:

```
QBX /L PRO7
```

This starts QuickBASIC, and tells it to load the QuickPak Professional Quick Library into memory. Once you are in QuickBASIC's editor, you may begin using all of the routines. Numerous example programs are provided both to show how the various BASIC and assembler routines are used, and to provide ideas for your own programs. For example, QD.BAS is a complete DOS file utility, and READDIRS.BAS shows many of the other QuickPak DOS services in a useful context.

VERY IMPORTANT!

Most of the QuickPak Professional routines and functions that use numeric variables expect integers. Using single or double precision values is guaranteed to cause a crash. The only exceptions are those routines that are specifically intended for use with floating point or long integer variables.

Also, many array manipulation routines are provided, and it is up to you to insure that the correct type and number of elements are specified.

As a safety precaution, we have provided a BASIC file named DECLARE.BAS that shows the correct declarations and calling syntax for each routine.

USING THE BASIC ROUTINES

External routines written in BASIC will be added to your programs by loading them as modules. Unlike earlier versions of QuickBASIC (or indeed, most application programs) the QuickBASIC editor allows you to load and manipulate more than one program file at a time.

Once you have started QuickBASIC, files may be brought into the editor with either the Open or Load commands. Using Open causes QuickBASIC to first clear out any programs that may already be in memory. Then, it loads the selected file as the "main" program. If you use Load, the named file is read into memory, but without disturbing other program files already loaded. Thus, you will use Load to add the QuickPak BASIC routines to your own programs, and Open to run their demos.

Each loaded module is entirely separate. However, subprograms and functions in one module may be freely called by any other module. When the main file is saved, a "make" file (with a .MAK extension) is created automatically by QuickBASIC. Then each time you Open the main program again, the make file is examined, and all of the modules that had previously been loaded will be read in automatically.

One final note about the BASIC subprograms and functions is that most of them have a demonstration program to show how they are used. Some provide a fairly substantial demo, for example DEMOSS.BAS shows all of the steps needed to set up and call the spreadsheet program SPREAD.BAS.

FUNCTIONS

Another interesting capability of QuickBASIC 4 and later not present in earlier versions of the BASIC compiler is that a user-defined function can actually *do* something, as opposed to merely calculating a result. For example, the ScanFile function first opens a file, then searches for a specified string, and finally returns where in the file the text was found.

Beginning with QuickBASIC 4, functions may also be written in assembly language. In the past, the only way an assembler routine could return information was to pass it a variable, and then examine the variable when the routine finished.

Now, however, assembly routines may return a value directly. This feature is used extensively in QuickPak Professional in those cases where returning a value is appropriate. It is important to understand that assembler functions *must* be declared before they can be used. For example, consider the GetDrive function which returns the currently active drive:

```
PRINT "The current drive is "; CHR$(GetDrive%)
```

If the function hasn't first been declared, BASIC would have no way to know that GetDrive% isn't simply an integer variable. Even though QuickBASIC will place the appropriate DECLARE statements for BASIC subprograms and functions in your program automatically, it will not do this for functions or subroutines contained in a Quick Library.

QuickPak Professional assembler functions are declared as:

```
DECLARE FUNCTION GetDrive%()
```

Here, the empty parentheses indicate that the function does not require or expect any incoming parameters. Functions such as FileSize that do require variables will have the parameters given in the argument list:

```
DECLARE FUNCTION FileSize&(FileName$)
```

Functions are also valuable because they may be used directly within BASIC test, assignment, or PRINT statements:

```
IF ReadTest%("A") THEN PRINT "Drive A is ready"
```

or

```
CALL QPrint(X$, OneColor%(FG%, BG%), 0)
```

or

```
PRINT MaxInt%(X%, Y%)
```

Finally, all of the QuickPak Professional functions that return a true or false value use -1 for true and 0 for false to allow the optional use of the BASIC NOT operator:

```
IF AltKey THEN . . .
```

or

```
IF NOT ReadTest%("B") THEN . . .
```

PASSING VALUES

All of the calling conventions for each of the QuickPak routines are shown beginning in the next section, though some interesting features of QuickBASIC are worth mentioning here.

Arguments that are passed to external programs do not always need to be variable names. As an example, to clear a window on the screen from 5,10 to 10,70 the numbers themselves may be specified:

```
CALL ClearScr(5, 10, 10, 70, 7, -1)
```

Numeric and string expressions are also valid:

```
CALL ClearScr(X% + 3, 10, X% + 8, 70, 7, -1)
```

or

```
CALL BPrint("Testing" + Array$(15) + LEFT$(Y$, 5))
```

And when using SetDrive, the drive letter may be given directly:

```
CALL SetDrive("A")
```

When you do this, QuickBASIC first creates a temporary variable set to the value or expression being passed, and then passes the temporary variable to the routine. For quoted or concatenated strings, a temporary variable comprising the various pieces is created and passed.

Notice that when BASIC finishes, the temporary variable is not discarded. Of course, when an assembler routine will be returning information to the calling BASIC program, these shortcuts will not work and a variable must be used.

Notice that a bug in QuickBASIC versions through 3.0 does not correctly handle a concatenated string being passed to an assembler routine. If you are using a version of QuickBASIC prior to 4, a temporary string should be created first, and then passed as a single variable:

```
Temp$ = "Testing" + Array$(15) + LEFT$(Y$, 5))
CALL QPrint0(Temp$, Colr%)
```

Also notice that when values are passed to an external routine as variables, any variable name may be used. Even if a BASIC subprogram uses one set of names to define the incoming variables, you are free to substitute other names when you call it.

Non-integer values may also be passed to subroutines and functions by using the correct type identifier suffix:

```
CALL Routine(1!, 2&, 3#)
```

Here, the one is passed as a single precision, the 2 is a long integer, and the 3 is forced to a double precision.

Notice that variables of one type may also be converted to any other type on the fly with BASIC's built-in conversion functions. For example, if you already have a value in a single precision variable and want to pass it to a routine that requires an integer, you would use CINT (Convert to Int) like this:

```
CALL QPrint(X$, CINT(SColor!), Page%)
```

As long as the variable SColor! is within the legal range for an integer, BASIC will first copy it into a temporary location, and then send the copy to QPrint. Likewise, you could pass a single precision value to one of the QuickPak Professional financial functions that expect a double precision variable with the help of CDBL:

```
X! = QPFVP#(CDBL(Fv!), CDBL(Intr!), Term%)
```

or

```
PRINT QPROUND$(CDBL(Amount!), Places%)
```

PASSING ARRAYS

Passing an array to an assembler routine is quite different from passing a simple variable. Although BASIC allows sending an entire array to a BASIC subprogram or function by following the array name with empty parentheses, this method does not work with the QuickPak assembler routines.

In QuickBASIC version 4 and later, all of the QuickPak routines that operate on a conventional (not fixed-length) string array require an extra step when they are called. In previous versions of QuickBASIC it was sufficient to simply use the starting array element as part of the call. Once the routine knew where the first element was located, it could count on subsequent elements being in adjacent locations.

With current versions of BASIC; however, a new calling method is needed. Though it would appear to require more effort, Microsoft has now decided to instead make a *copy* of the array element, and send the address of the copy to the assembler routine. Of course, the address of a copy of an array element has no relevance to the actual location of the array.

The best way around this problem is to use a combination of VARPTR and the new BYVAL key word, as shown in the string sort example below:

```
CALL SortStr(BYVAL VARPTR(Array$(1)), Size%, Dir%)
```

Using VARPTR forces QuickBASIC to obtain the true address of the array element, and BYVAL then sends the *value* of the address to SortStr. Usually, addresses are passed to assembly routines rather than values, however the address of an *address* would not be appropriate.

Numeric and TYPE arrays must be handled yet another way. As with string arrays, QuickBASIC will normally make a copy of the named element in a numeric array, rather than pass the original address directly. Further, because these arrays may be located anywhere in memory, the assembler routines that operate on them need to know both the address and the segment. The solution this time is to use the SEG command:

```
CALL SortI(SEG Array%(1), Size%, Dir%)
```

When QuickBASIC sees that SEG is used as part of a call, it understands that the actual segment and address of the element are needed, and passes both of them unchanged to the named routine.

If a subprogram or function has been declared with BYVAL or SEG, it is not necessary to use it again each time later:

```
DECLARE SUB Delete(BYVAL Address%, NumEls%)
CALL Delete(VARPTR(Array$(1)), NumEls%)
```

or

```
DECLARE SUB AddInt(SEG Element%, Value%, NumEls%)
CALL AddInt(Array%(Start), Value%, NumEls%)
```

Notice in the first example that the Address% variable is an integer. This is necessary because VARPTR returns an integer value that corresponds to the variable's address.

PASSING FIXED-LENGTH STRING AND TYPE ARRAYS

Passing a fixed-length string array or a TYPE array to a BASIC subprogram is also possible, even though all of the examples in the QuickBASIC manuals skirt the issue by showing the various arrays as being SHARED. The short program below illustrates the steps required to do this.

Notice that to pass an array of fixed-length strings it must be declared as a TYPE array, even if the TYPE is comprised solely of a single string member.

```
TYPE FLen                               'define the TYPE
    Stuff AS STRING * 11                'before you refer
END TYPE                                'to it later

DECLARE SUB FLSub (Param() AS FLen)     'declare the sub
DIM A(100) AS FLen                      'DIM the array

A(89).Stuff = "Testing #89"             'assign element 89
CALL FLSub(A())                         'call the subprogram

SUB FLSub (Array() AS FLen)             'refer to the TYPE
     PRINT Array(89).Stuff              'print the element
END SUB
```

LINKING WITH QUICKPAK PROFESSIONAL

The PRO.QLB and PRO7.QLB Quick Libraries are intended to be used only while you are in the QuickBASIC editing environment. Once you have completed developing and testing a program, you will need to create a final stand-alone .EXE file before it can be run from DOS. If you are compiling and linking manually from DOS, then you would specify PRO.LIB like this:

```
LINK basicprogram ,, NUL, PRO.LIB
```

If you are using BASIC PDS with far strings, then you must link with PRO7.LIB as follows:

```
LINK basicprogram , , NUL, PRO7.LIB;
```

If you prefer, you can start LINK without any options, and wait for it to prompt you for the information it needs.

One important point you should be aware of is that only those routines that are called will be added to your program. Many people mistakenly believe that if they link with a library such as PRO.LIB, all of the routines contained in the library will be added to their program. However LINK is very smart, and knows to add only the routines that are actually needed. (Also see the section entitled "Compiling and Linking from DOS" for an important warning about how QuickBASIC 4.0 invokes LINK from within the editor.) You should also understand that more than one library can be specified when linking. For example, if you need assembler routines from both QuickPak Professional and the Graphics Workshop, you may tell LINK to use both of them:

```
LINK basicprogram ,, NUL, PRO GW
```

Further, single object modules may also be added in at link time, even if they are not in a library at all:

```
LINK basicprogram object ,, NUL, PRO GW
```

If you do wish to combine several libraries into a single file, that is quite easy too. Though the LIB library manager is usually employed to add or remove object modules, you may also add one or more complete libraries like this:

```
LIB libname +libname2.lib +libname3.lib
```

One useful LINK option you should be aware of is the /EX command line switch. When LINK is invoked with /EX, it creates an .EXE file in a special "packed" format. Not unlike the various archive programs, your program's code and data are compressed to take up less disk space. When the program is run, the first code that actually executes is an unpacking routine that puts everything back together again.

For programs compiled with QuickBASIC 4.0, /EX is intended to be used *only* with stand alone programs that do not require the QuickBASIC BRUN support module. Further, though a packed program will take up less disk space, it of course requires the same amount of memory when it is run.

BUILDING QUICK LIBRARIES

Quick Libraries enable you to access programs and functions written in languages other than BASIC, while you are within the QuickBASIC editing environment. Besides the assembly language programs included with QuickPak Professional, this feature also lets you call routines written in C, Pascal, and any of the other supported Microsoft languages.

All of the QuickPak Professional assembler routines are already contained in the PRO.QLB library. However, there may be occasions when you need to build a Quick Library. For example, you may wish to incorporate routines of your own as well as those provided with QuickPak Professional.

Quick Libraries are created using the version of LINK that comes with your version of BASIC. Microsoft has documented many of the capabilities of LINK in the QuickBASIC manuals, though there are additional useful features you should be aware of.

Normally, LINK expects a list of object file names to be specified for inclusion in the Quick Library being built. But what Microsoft failed to mention anywhere is that you can also combine one or more entire .LIB libraries as well. (A .LIB library simply contains one or more .OBJ modules in a single file.) This feature is particularly useful if you need to combine several Crescent .LIB library files into a single Quick Library.

To create a new Quick Library comprised of one or more .LIB and .OBJ files, invoke LINK like this:

```
LINK /Q L1.LIB L2.LIB OBJ1 OBJ2, LIBNAME, NUL, BQLB45
```

The /Q option tells LINK that it is to create a Quick Library rather than an .EXE file. L1.LIB and L2.LIB are the library files being incorporated into the new Quick Library, and OBJ1 and OBJ2 are object modules. Of course, you can mix and match library and object files in any order you would like.

The .LIB extension is very important, because without it LINK would be looking for files with an .OBJ extension. Notice that an extension is not needed for the object files.

The LIBNAME parameter is the name you want to use for the new Quick Library being built. Without the LIBNAME parameter, LINK will default to the same name as the first .LIB or .OBJ file in the list. In the example above, the resultant file would have been named L1.QLB if LIBNAME had been omitted. In that case, though, the comma "place holder" is still required:

```
LINK /Q L1.LIB L2.LIB OBJ1 OBJ2, , NUL, BQLB45
```

The NUL parameter prevents LINK from generating a useless list file, and BQLB45 is a special library supplied with QuickBASIC. BQLB45.LIB has nothing to do with QuickPak, and is required for making any Quick Library that will be used with BASIC. Note that BQLB45 may have a slightly different name, depending on the version of QuickBASIC. For example, if you are using BASIC 7 PDS the library is named QBXQLB.LIB.

One final important note about making Quick Libraries is the /SEG option. When many separate object modules are being combined, the number of code and data segments may exceed LINK's default maximum of 128. When we create PRO.QLB from PRO.LIB, we always tell LINK to allow for up to 500 segments like this:

```
LINK /Q /SEG:500 PRO.LIB ,, NUL, BQLB45
```

BUILDING LIBRARIES FOR QUICKBASIC 2 AND 3

Versions of QuickBASIC prior to 4 use a special program called BUILDLIB.EXE to create libraries for use in the editing environment. As with LINK, BUILDLIB expects a list of object file names to know what it is to include in the library. And again, combining one or more .LIB files is both undocumented yet simple. In this case, however, the /L option is required:

```
BUILDLIB /L OBJ1 OBJ2, LIBNAME, NUL, L1.LIB L2.LIB
```

Note that Microsoft supplies a special program called PREFIX.OBJ on the QuickBASIC 2.0 and 3.0 program disks, and recommends that you include it in all of the libraries that you build. Also on the QuickBASIC 2 and 3 disks are several object files that contain other routines you may need.

If you are using BUILDLIB (or LINK for that matter) to create a library comprised of many separate object modules, you may receive a “Too many segments” error message. If this happens you will need to use the Segment option like this:

```
BUILDLIB /L /SEG:300 OBJ1 OBJ2 . . .
```

In fact, there are a number of LINK and BUILDLIB options that for some unknown reason Microsoft has failed to document in the QB 2 and 3 manuals. To see a list of the options that are available with any given version, start BUILDLIB (or LINK) with the /HELP command:

```
BUILDLIB /HELP
```

ADDING BASIC PROGRAMS TO A QUICK LIBRARY

With earlier versions of QuickBASIC it was often advantageous to add pre-compiled BASIC programs into a library. Since those versions always compiled an entire program each time you made any changes, pre-compiling could save an enormous amount of time. Those routines that have been completed and tested could be accessed, without having to wait for them to be compiled over and over.

With QuickBASIC 4 we recommend that you *not* do this for several reasons. First, because the QB editor is really an interpreter, there will be no significant decrease in the time needed before your program starts to run. Second, any routines that are in a Quick Library may not be traced or debugged. The final reason requires a bit of explanation.

The BC compiler can produce two very different types of programs. The BCOM type is a stand-alone program that can run without requiring the BRUN run-time module. Many programmers prefer this method because it creates the simplest type of program. Also, a BCOM program usually requires the least amount of memory when it runs.

BRUN programs require the presence of a run-time module each time they are invoked from the DOS command line. In situations involving many program modules that are chained together, compiling a program for BRUN makes a lot of sense. Each program file is substantially smaller because the BASIC language routines are all contained in the run-time module, and don't have to be added to each program.

What complicates the issue is when a BASIC program is being compiled for inclusion in a Quick Library, some versions require it to be compiled as a BRUN program without the /O option. Once you have completed development and are ready to create a stand-alone .EXE program, you must re-compile each of the modules, this time using the /O BCOM option.

If you forget to do this and combine a BCOM main program with one or more BRUN modules, your final program may not work, and you'll have no indication of what went wrong. Neither BC nor LINK will report the error.

COMPILING AND LINKING FROM DOS

For many programs, you can simply let QuickBASIC do all the work, and create an .EXE program directly from within the editing environment. But there are several disadvantages to doing this, and we recommended that you compile and link from the DOS command line manually. This is especially true if you are using QuickBASIC 4.0.

One very important reason is that you can specify exactly the options that you want to add to your program. For example, even if you are not using any of the BASIC communications features, that code will otherwise be added to your program. Further, if you are using any external programs in a Quick Library, you must be sure to have a parallel .LIB library file that has the same name and contains all of the same routines.

Worse still, if you link with a library such as PRO.LIB, QuickBASIC 4.0 tells LINK to add *all* of the routines from the library. (We can only assume that this is a bug—they *couldn't* have done that on purpose, could they?) The whole point of using a linker is that it should add only those procedures that your programs call for. Finally, because of the way the QuickBASIC 4.0 editor invokes LINK, programs comprised of many separate modules simply cannot be linked.

When QuickBASIC 4.0 runs LINK to combine all of your various program modules, it places their names on a single command line. But since DOS limits the length of a line to 127 characters, attempting to link more than ten or so modules will fail. Of course, even if you compile and link manually as we recommend, you will still face this limit. The best solution is to use a LINK response file.

Fortunately, Microsoft has fixed these problems beginning with QuickBASIC 4.5.

RESPONSE FILES

When you need to link a large number of modules, the most efficient way to tell LINK their names is with a response file. A response file merely contains the answers for all of the various LINK prompts, and it can greatly simplify your job by eliminating a lot of repetitive typing. Further, since the answers are contained in a file, the file can be edited as needed.

There are two types of response files that you should be familiar with—the type used by LINK, and the type used by the LIB.EXE library manager. Each serves a similar purpose, but they use a slightly different syntax.

A LINK response file contains a list of file names to be linked together. When more names must be given than can be entered onto a single DOS command line, a plus sign (+) is used as a line-continuation character. A typical LINK response file looks like this:

```
Object1 Object2 Object3 Object4 +
Object5 Object6 Object7 Object8 +
Object9 Object10
```

Rather than entering all of the object names when you start LINK, instead give it the name of the response file preceded by an "at" sign (@) as shown below.

```
LINK [/options] @filename.ext
```

A LIB response file uses a slightly different format:

```
+Object1 +Object2 +Object3 +Object4 &
+Object5 +Object6 +Object7 +Object8 &
+Object9 +Object10
```

The plus signs tell LIB that the named object module is to be added to the library. Notice that a LIB response file uses an ampersand (&) as the continuation character.

EXTRACTING OBJECT MODULES FROM PRO.LIB

All of the QuickPak Professional assembler routines are provided both in library files and in assembler source code. There are very few situations in which you would need access to the individual object modules. However, extracting one or more object modules from the .LIB library is actually quite simple.

As we have already seen, LIB uses the plus sign followed by a file name to indicate that the file is to be added to the current library. To extract an object file you would instead use an asterisk. The example below shows how to extract object files for the QuickPak QPrint and APrint routines contained in PRO.LIB.

```
LIB PRO *QPrint *APrint
```

After entering this command, two new files will have been created—QPRINT.OBJ and APRINT.OBJ.

These are only some of the many useful options you can use with LIB and LINK, however discussing all of them is beyond the scope of this documentation. For more information you should refer to the manuals supplied by Microsoft.

Note that it is not usually necessary to extract object files from the QuickPak Professional libraries. If you merely need to create a subset Quick Library to preserve memory in the QB editor, the MakeQLB utility we provide will do this for you automatically. MakeQLB is described in the Miscellaneous chapter of this manual.

USING INTEGERS

You will notice that nearly all of the demonstration programs we have provided begin with DEFINT A-Z as the very first statement. Though this is of course entirely up to you, we strongly recommend that you use this practice in all of your programs. Besides guaranteeing that the variables passed to the QuickPak assembler routines are integers where appropriate, this will also make your programs both faster and smaller.

Integers require the least amount of memory for storage, and math operations on integers are much faster than for any other type of variable. Of course, when an integer is not adequate for a given situation, you should use a variable type that is. If at all possible, however, use long integers if you simply need to accommodate larger numbers.

Even though a long integer occupies the same amount of memory as a single precision variable, most math operations on long integers will be considerably faster. Long integers are also more accurate in situations involving repeated calculations, to avoid rounding errors. For example, if many dollar amounts are to be added together, a common technique is to treat all of the values as pennies. Then, when the final result is printed, a single floating point divide will return the actual dollar value:

```
Total& = 0
FOR X = 1 TO 3000
    Total& = Total& + Value&(X)
NEXT
PRINT USING "The total is $######,.##"; Total& / 100
```

ELIMINATING "ON ERROR"

Many of the assembly language routines provided with QuickPak Professional may at first appear to duplicate functions built into the BASIC language. For example, we provide routines to create and remove subdirectories, delete and rename files, plus many other services that BASIC can already do directly.

Unfortunately, there are several problems with the way that DOS errors are handled by BASIC. With the introduction of QuickBASIC 4.0, error handling has become even more complicated and convoluted. It seems to us that if you attempt to, say, open a file for input that isn't there, your program should be able to know that right away. Having to first specify where to GOTO if an error occurs is both non-intuitive and inconvenient. Further, using ON ERROR creates programs that are both larger and slower.

Because of these problems with ON ERROR, we have provided a number of DOS and BIOS routines to bypass BASIC altogether, and manipulate printers and disk files directly. In all cases, these routines are called with arguments similar to those used by BASIC. However, rather than use ON ERROR to direct your program to an error handler, you may simply query the new reserved "variables" DOSError% and WhichError%.

Of course, an external toolbox package can't really add new reserved words to QuickBASIC. Rather, these are integer functions that indicate whether or not an error occurred on the last DOS operation, and if so which one. Because these are designed as functions for use with QuickBASIC 4, they must be declared before they can be used.

This method allows you to quickly determine if an error has occurred without having to constantly specify new locations for error handling in each portion of a program. For example, deleting a file using the QuickPak KillFile routine is as simple as:

```
CALL KillFile(FileName$)
IF DOSError% THEN PRINT FileName$; " wasn't there"
```

or

```
CALL KillFile("XYZ.DAT")
IF NOT DOSError% THEN PRINT "XYZ.DAT was deleted"
```

Many other services are available to test a disk drive for reading and writing, lock and unlock network files, send text to a printer, and so forth.

The QuickPak Professional file services are also handy for use with the original BASCOM 1 compiler. Many programmers are still using that version because it creates programs that are much smaller than those produced by QuickBASIC. But BASCOM 1 does not support path or directory names, and these services nicely fill that gap. Note, however, that BASCOM 1 does not support using functions such as DOSError%.

When using any of the QuickPak Professional file routines, it will be up to you to keep track of the file handle numbers that DOS assigns. When a file is opened using the BASIC OPEN command, the file number that will be used for subsequent references to the file is determined by you. BASIC then maintains an internal table of file numbers, based on the ones you have chosen and those that were assigned by DOS.

Internally, it is actually DOS that assigns the file numbers, and BASIC simply translates the numbers for you. But when an assembly language routine such as FOpen opens a file, it is up to your program to remember the number that DOS assigned.

Notice that both the QuickPak Professional file routines and those provided by QuickBASIC may be used together in the same program. QuickBASIC provides the FILEATTR command to determine the DOS file handle for any file it has opened, based on the file number you originally specified. The example below shows how to use FILEATTR to obtain the DOS handle:

```
Handle% = FILEATTR(FileNumber, 2)
```

If you had originally opened the file as #1, then the FileNumber variable above would be 1 and Handle will receive the equivalent DOS handle associated with that file.

MULTI-TASKING MENUS

One of the most exciting capabilities we have provided in QuickPak Professional is a system of menus that can simulate multi-tasking. Where most menu programs simply sit in an empty loop waiting for a key press, both the pull-down and BASIC vertical menu subprograms let you continue your program if you want.

To accomplish this, a new "Action" parameter (Action%) has been added to the calling sequence. Depending on the setting of this variable, the menus can be instructed to operate in a number of different ways. Besides the Action variable about to be described, these menu programs also make use of two special $INCLUDE files—DEFCNF and SETCNF. These are described separately in the section entitled "SETCNF and DEFCNF" elsewhere in this manual, and we will not belabor them here.

The Action variable has six different possible settings, to tell PullDown and VertMenu how they are to behave. Each of the possible Action values is described in detail below. We should also mention that while mouse support is built in to the pulldown and vertical menus, you may easily remove that code if you don't need mouse support. The portion of the programs that process the mouse are clearly marked showing what you should REM out or delete. Search for "Rodent" from within the QuickBASIC editor.

If Action is set to zero, then the menus will operate the way you would expect a "normal" menu to work. That is, the underlying screen is first saved, then the menu is displayed, and finally an INKEY$ loop repeatedly waits for the user to press a key or a mouse button. Once a key or mouse button has been pressed, the original screen is restored, and control is returned to the calling program. The Choice variable(s) may then be examined to see what selection the user chose.

When Action is set to 1, both PullDown and VertMenu simply save the screen and display themselves. Control is then returned to the calling program immediately, however Action is also set to 3 for subsequent calls. Since Action 3 is how you will be polling the menu subsequently, this saves you an extra step.

Setting Action to 2 lets you re-display the menu, in those cases where it may have been overwritten by another, possibly overlapping, menu. Action 2 also resets itself to 3 for subsequent calls. If the menus are called with Action equal to 3, the keyboard and mouse are merely polled to see if a key or button has been pressed.

If Action is still set to 3 when the menu returns, it means that no keys or mouse buttons were pressed.

If Action is returned set to 4, the user either made a selection or pressed Escape. In this case, the Choice, Menu, and Ky$ variables should be examined.

The last Action value is 5, and this simply tells VertMenu or PullDown to remove itself and restore the original screen.

If you intend to create stacked menus, you should be aware of one important point. Because each menu saves its own underlying screen, the screen that was saved first will be destroyed when the menu is called again. This means that it is up to you to save each screen in succession manually, except for the last one. This may be done either manually using ScrnSave and ScrnRest, or automatically with the WINDOMGR.BAS subprogram.

DEMOMENU.BAS provides a complete demonstration of using PullDown and VertMenu in a multi-tasking context. DEMOPULL.BAS and DEMOVERT.BAS are also provided to illustrate the minimum steps needed to call these routines.

BIT ARRAYS

Before the introduction of QuickPak Professional, the smallest type of variable you could use in BASIC was an integer. In cases where a program needs the range of values offered by integers, using them makes a lot of sense. But when the values are very small and there are a lot of them, an integer can waste a considerable amount of memory.

One effective solution is to create a "byte" array by defining a string variable with a length equal to the number of elements that are needed:

```
X$ = STRING$(Size, 0)
```

Each "element" can hold a value between 0 and 255, and with only a little extra work, you can coerce the range to be from -128 to 127. But often, all that is really needed is a simple True or False type of variable, and even using the individual characters in a string will be wasteful.

Three routines are provided to manipulate what we call Bit arrays. Like byte arrays, a string variable is used for the actual storage, however each character in the string really represents 8 separate elements. Thus, creating a Bit array with a size of 1000 elements will occupy only 125 bytes, as opposed to the 2000 bytes that an integer array would require. At sixteen to one, this represents a substantial savings indeed!

The two Bit routines that assign and retrieve array elements are written in assembly language, and a BASIC subprogram is provided to "dimension" the array. Because an assembler routine cannot easily create strings or change their length, this part must be done in BASIC. Of course, creating the string is very simple, and you could just as easily use the formula shown below.

```
Array$ = STRING$(NumEls \ 8 + 1, 0)
```

SetBit and GetBit may then be used to set and get the various elements as they are needed.

THE QUICKPAK PROFESSIONAL EDITOR AND SPREADSHEET

Two of the programs included with QuickPak Professional are a complete spreadsheet, and a full-screen text editor. Each of these is intended to be added to your programs as a module, and then called when needed. This section describes both subprograms, and provides additional details on how they will be accessed by your programs.

QEdit - The QuickPak Professional Editor

The QEdit editing window may be positioned anywhere on the screen, and sized to nearly any number of rows and columns. It features on-line help, full mouse support, word-wrap, horizontal and vertical scrolling, as well as block operations for insert, delete, and copy.

Notice that the QEdit help text and paste buffer are stored outside of BASIC's normal string space, to leave your program as much string memory as possible. The technique used to accomplish this is described in detail in the Appendix of this manual.

All of the standard editing keys are supported. For example, Home and End move to the beginning and end of the line, the PgUp and PgDn keys scroll the screen by pages, and Ctrl-PgUp and Ctrl-PgDn move to the first and last lines respectively. The cursor may also be moved to the top or bottom of the edit window with the Ctrl-Home and Ctrl-End keys.

Similar to the QuickBASIC editor, QEdit uses the Ctrl-Left and Ctrl-Right arrow keys to move the cursor by words, and blocks are marked with any of the Shift-Cursor keys. Notice that blocks may also be marked using a combination of the Shift *and* Ctrl arrows to jump by whole words. Beyond the usual block operations, however, QEdit also supports block operations on *columns*.

To mark a block in sentence mode, place the cursor at the beginning of the block and press the Shift-Right arrow key. The marked block will be highlighted as the cursor travels over the text. Alternately, entire lines may be marked by placing the cursor at the desired starting point and pressing the Shift-Down arrow key.

To mark a column, place the cursor at the upper left corner of the block and press the Shift-Right arrow key until the highlight extends to the desired right edge. Then, press Shift-Down until the bottom of the block has been reached.

Regardless of which marking mode is used, the block will be captured as soon as any non-marking key has been pressed. If either Del or Shift-Del is pressed, the block will first be captured and then deleted. To paste the block from the buffer into the text at the current cursor location, simply press Shift-Ins.

For users that have a mouse, the text may be scrolled, new margins set, and the editing window may be moved or resized dynamically while editing. Using a mouse for scrolling the text is done almost exactly like the QuickBASIC editor. That is, you may click the left button on the scroll bar icons at the bottom or right edge and move them, or press and hold the left button on the arrow icons.

One improvement you will notice over the way QuickBASIC works is that QEdit actually scrolls the text while you move the scroll bar. Where the various Microsoft editors make you release the mouse button before you can see how far you actually scrolled, QEdit provides immediate feedback.

Marking a block of text with a mouse is also improved over the QuickBASIC method in that marking may be on columns. As when using the keyboard to mark a block, simply press and hold the left button while sliding the mouse to the right, and then move down to define a column. Moving downward first will instead mark in sentence mode.

To resize the window, press the left mouse button at either the upper left or lower right corner of the editing window, and simply drag the corner to its new location. Setting a new right margin is just as easy. Click on the right margin indicator at the top of the screen and move it. Please understand that if the document being edited is very large, there will be a small delay because the entire text must be rewrapped to the new margin setting.

All of the code to process block operations is located in a single section of the QEdit subprogram. This way you can easily remove or REMark out those lines, to reduce the size of your program if that feature is not needed. All of the mouse code has also been placed into a single section of the program for the same reason. Search the QEDIT.BAS source file for the text "Mouse Handling" for further instructions.

The call for QEdit is fairly simple to set up, as illustrated in the DEMOEDIT.BAS demonstration program. Your program will need to dimension a conventional (not fixed-length) string array to hold the lines of text. The size to which the string array is dimensioned dictates the maximum number of lines that may be entered.

If you intend to present a blank screen to your user, then no additional steps are needed to prepare the array. If you already have text that is to be edited, it may be placed in the array before QEdit is called.

The text may also be sent to QEdit as a single long line in the lowest array element. In that case, it will be wrapped automatically before being presented for editing. If you intend to read files prepared by a word processor that places each paragraph on its own line (such as XyWrite), you will probably want to read each line into *every other* element in the string array. This will preserve the spacing between paragraphs, and can be accomplished as shown below:

```
OPEN X$ FOR INPUT AS #1          'open the file
CurLine = 1                      'set the current line counter
WHILE NOT EOF(1)                 'read until the end
   LINE INPUT Array$(CurLine)    'get a line
   CurLine = CurLine + 2         'skip over the next line
WEND
CLOSE #1                         'close the file
```

If you do pre-load the array with individual lines of text, an extra blank space should be present at the end of each line. When QEdit wraps words to the next or previous line, the space is needed to prevent the end of one word from running into the beginning of another. Comments in the QEdit source code show how to insure that every line has at least one trailing space.

Like most of the other QuickPak Professional routines, the current cursor location indicates where to position the upper left corner of the editing window. Parameters passed to QEdit are then used to indicate the width and height of the window, the margins, colors, and so forth. Let's take a close look at each of these in turn, beginning with the QEdit calling syntax.

```
CALL QEdit (Text$(), Ky$, Action%, Ed)
```

The Text$() array holds the text to be edited, as described above.

Ky$ returns holding the last key that had been pressed. For example, it would hold CHR$(27) if the user pressed Escape to exit QEdit.

The Action% parameter sets the operating mode for QEdit as follows:

Action = 0—Use the editor in a non-polled mode. QEdit will take control, and return only when the user presses the Escape key. The underlying screen will be saved upon entry, and restored when QEdit is exited.

If you do not intend to add features to QEdit or take advantage of its multi-tasking capability, you may set Action% to 0 and simply ignore the remaining Action parameters described below.

Action = 1—Initialize the editor for polled mode. The underlying screen is saved, the edit window will be drawn, and the text is displayed. Control will be returned to the caller immediately without QEdit checking the keyboard. The Action flag is also set to 3 automatically (see below).

Action = 2—Redisplay the edit window and text, but without resaving the underlying screen. Control is then returned to the caller immediately without QEdit checking the keyboard. As above, the Action parameter will be set to 3 automatically.

Calling QEdit with an Action of 2 would be useful when changing the window size or location, to force QEdit to redisplay the text at the new location.

Note that if word wrap is on, Actions 0, 1, and 2 will cause the text to be re-wrapped to the value of Ed.Wrap
(see below).

Action = 3—This is the idle state of the editor. Each time the editor is called with this value, it will check the keyboard and perform tasks dictated by a key press. Control will then be returned to the caller.

While the editor is being polled, the caller may examine the Ky$ parameter to determine which, if any, keys were pressed. The members of the "Ed" structure can also be examined and changed. Note that if the caller does change these, the editor should always be called again with an Action of 2 to redisplay the edit window.

Action = 5—Restores the screen that was saved when QEdit was called with Action% set to 1.

The Ed parameter is a TYPE structure defined as EditInfo in the file QEDITYPE.BI. All of the additional parameters for QEdit are contained in this structure. Therefore, you must include QEDITYPE.BI in your calling program, and assign the elements needed to establish the window size, colors, and so forth. This is fully detailed in the DEMOEDIT.BAS demonstration program.

Note that passing a pointer to a TYPE variable this way is much faster and more concise than passing all of these parameters as part of the call. The following is a list of the elements in the EditInfo structure.

Ed.Rows sets the number of rows to be displayed in the window. It can range up to 25 lines with no border on a CGA or monochrome display, or 23 if you do specify a border. If an EGA or VGA adapter is present and WIDTH is used to set more screen lines before QEdit is called, then the window may occupy up to 43 or 50 lines respectively.

Ed.Wide sets the number of columns to be displayed in the window. The width can range up to 80 if no border is specified, or 78 if a border is used.

Ed.Wrap sets the right margin for word wrapping. This is independent of the rightmost visible column, and may be set to nearly any value (up to 255). If the right margin extends beyond the right edge of the window, QEdit will scroll the text to accommodate it. Word wrap may also be disabled entirely by setting Ed.Wrap to 0.

Ed.HTab sets the number of columns to move when the Tab or Shift-Tab key is pressed. This parameter will default to 8 if a value of zero is given.

Ed.AColor sets the color of the edit window. The color number follows the same convention as the QuickPak video routines in that both the foreground and background colors are packed into a single value.

Ed.Frame is used to specify whether or not a frame is drawn around the edit window. If it is set to any non-zero value, the ruler line, scroll bars and cursor position numbers will be drawn around the window.

The remainder of the parameters are intended to be read by your program, and do not have to be set before QEdit is called. For example, you could determine if the user has resized the screen, or if a block of text has been marked. Also, if you are adding some sort of search capability for the text, you could force QEdit to display the screen at a particular row and column.

Ed.LSCol holds the current left screen column of the editable window (not including the frame).

Ed.LC holds the leftmost column of text being displayed, which will be greater than 1 if text is scrolled to the right.

Ed.CurCol holds the current text column number of the cursor, which is not necessarily the current screen column.

Ed.TSRow holds the top screen row of the editable window, not including the frame.

Ed.TL holds the topmost row of the displayed text, which will be greater than 1 if text has been scrolled down.

Ed.CurLine holds the current text line number at the cursor, which is not necessarily the current screen row.

Ed.UlCRow holds the upper left row number of the most recently marked text block.

Ed.UlCCol holds the upper left column number of the most recently marked text block.

Ed.BrCRow holds the bottom right row number of the most recently marked text block.

Ed.BrCCol holds the bottom right column number of the most recently marked text block.

Ed.CBlock indicates the type of marking for the most recently marked block. Ed.CBlock will be 0 if marking was in sentence mode, or -1 if the block was marked using the QEdit column marking feature.

Ed.Presses indicates whether a mouse button has been pressed, but not handled by the editor. This information is for your program to use if you intend to handle mouse presses that occurred outside of QEdit. Since Ed.Presses is non-zero only in that situation, you would then examine the Ed.MRow and Ed.MCol parameters (see below) to know where the mouse cursor was when the button was last pressed.

Ed.MRow holds the row where the mouse cursor was at the time the button was last pressed, or if it is currently being pressed.

Ed.MCol holds the column where the mouse cursor was at the time the button was last pressed, or if it is currently being pressed.

Ed.Instat is used to determine the current insert state mode. Ed.Instat will be 1 if QEdit is currently in the overtype mode, or -1 if inserting is active.

Ed.Changed can be used to see if the text has been changed. This parameter will be set to -1 if any changes or additions have been made to the text, otherwise it will be 0. This lets you know whether the file needs to be saved or not, however you must clear this variable once the text has been saved.

Ed.LCount holds the number of active lines in the text string array.

Ed.MErr is an error flag used to signal errors that occurred within the editor. Ed.MErr will be 1 if there is insufficient memory. This could be caused by running out of string space with a large document, or not having enough far memory to capture a large block the user has marked for pasting. Ed.MErr will be set to 2 if the user attempted to enter (or paste) more text lines than the string array has been dimensioned to.

Spread - The QuickPak Professional Spreadsheet

The QuickPak Professional spreadsheet is a full-featured application lacking only user-defined formulas and macro capability. Not unlike the various spreadsheet compilers on the market, the formulas and cell formatting information are hard-coded into the program.

Where a traditional spreadsheet must interpret the contents of each cell each time it recalculates, this approach allows the program to operate very quickly. However, it also means that the spreadsheet must be modified for each application.

We have provided many of the features needed by a spreadsheet program such as cell formatting and GOTO, however it is up to you to establish the table of cell formats and formulas. We have also provided an example showing how to implement an "intelligent" recalculation, where only those cells that are affected by an edit will be processed.

Because the QuickPak Professional spreadsheet offers much more than simply a means to input data in rows and columns, there are a number of steps that must be performed before it is called. These steps are outlined below, followed by a brief description of some of the programs internal workings.

To truly understand how this spreadsheet works, however, will require a close examination of the BASIC source code. We have commented the subprogram as thoroughly as possible, to explain the purpose of each statement.

The first step you must perform is to dimension three arrays that tell the spreadsheet how many rows and columns to use, how the numeric cells are to be formatted, and how wide each column is to be. In the description that follows, we will be referring to the demo program DEMOSS.BAS, and we will use the same variable and array names.

The main spreadsheet is contained in a two-dimensional string array, which must be a conventional (not fixed-length) array. The size to which Wks$() is dimensioned determines the number of rows and columns that can be accessed. The example below establishes the size of the work area to be 150 rows from top to bottom, and 30 columns wide:

```
DIM Wks$(150, 30)
```

A parallel string array must also be dimensioned, to hold the formatting information that describes how numeric fields will be displayed. The format strings are intended to be set up in the same manner that BASIC's PRINT USING uses. If numeric formatting is not required, then the array must still be dimensioned, though it is not necessary to assign any of the elements.

The final array that is needed indicates the width of each column. Unlike the two-dimensional worksheet and format string arrays, the columns are kept in a one-dimensional integer array. The Columns array must be dimensioned to the number of columns in the main spreadsheet array, and each element establishes the width of its corresponding column. That is, ColWdths%(1) sets the width for the "A" column, ColWdths%(2) holds the "B" column, and so forth.

Besides the arrays, three additional parameters are required by the spreadsheet. The first one (WindWdth in the demo) indicates how wide the active window is to be. This may range up to a value of 78, because the border requires two columns. The second parameter is the number of rows to display, and the allowable upper range depends on the number of screen lines that a given monitor can display.

The final parameter is an action flag, which allows you to call the spreadsheet more than once, without having to re-display the screen. Action is used here as it is with QEdit, in that a value of zero means to enter the program and wait there until the slash key is pressed. At that time, it will restore the original screen and return to the caller.

When Action is set to 1, the screen is saved as with Action 0, but pressing the slash key does *not* restore the original screen. Thus, your main program could display a menu to change the column widths or load a new worksheet, and it would appear to the user that they are still within the spreadsheet. You would then re-enter the spreadsheet with an Action of 2, which simply re-displays the screen. Notice that the column widths may be freely changed, and the spreadsheet will be re-displayed correctly in the new size.

Finally, calling QEdit with an Action of 5 tells it to restore the original screen contents and return.

Because the QuickPak Professional spreadsheet is supplied as BASIC source code, there is virtually no limit as to how it may be modified. The incoming keystrokes are read by the program immediately below the remark "Main key processing loop". Thus, you could put additional key traps there, and branch to other parts of the subprogram that you create.

Three function keys are recognized in the program—F2 to edit a cell, F5 to prompt for a GOTO address, and F9 which causes a complete recalculation. Whenever a cell is edited, the program branches to the label QPCellEdit. Approximately 25 lines below that label is a GOSUB to the QPCalc1 routine, which recalculates only that portion of the work sheet that depends on the edited cell. Of course, it is up to you to modify the calculations, and determine which cells must be revised.

QuickPak Professional includes all of the scientific and financial functions used by commercial spreadsheets, and these would be a natural addition to applications you create with the QuickPak Professional spreadsheet.

Spreadsheet Program Description

The spreadsheet subprogram begins by determining the type of monitor that is installed, and then sets appropriate window, highlight, and border colors. These are clearly marked to make it easy for you to change them if you prefer.

Next, the Action variable is examined to see if the program is being initialized or not. If so, the current cursor location and underlying screen are saved, and the spreadsheet is displayed. Otherwise, the original screen is restored, and the array that held it is erased to free up the memory.

The next few steps calculate the total width of the display window, and create the header title line. Finally, four GOSUB's are used to print the header and cell contents, calculate the entire spreadsheet, and highlight the current cell at A1.

We have used GOSUB rather than CALL for several reasons. First, a GOSUB can branch and return considerably faster than can a CALL. Second, GOSUB eliminates having to share many variables, or worse, pass them all which would take even more time. (The next time some clown tells you how much faster C is compared to BASIC, point out that GOSUB is always faster than any C function, because it avoids time-consuming pushes to pass parameters.)

All keystrokes are read in the main key processing loop, which is clearly marked. Any custom keys that you would like to process may be examined here. For example, if you intend to add macros you could trap Alt keys and perhaps function keys as well. Extended keys may also be examined in the SELECT CASE block below, however normal non-extended keys must be trapped here.

Notice the string variable Status$, which holds a message such as "READY" or "WAIT" to be printed in the upper right corner. You may print custom messages in the status box, however the maximum length is limited to five characters.

It is important to use LSET to assign Status$ as we have done, to clear any remnants of a previous longer message. Using LSET also maintains the original string length, which guarantees that the entire message area is filled.

One technique you may find interesting is how we trap the first key that is entered, to know whether the status box is to display "LABEL" or "VALUE". If the key is either a slash or one of the extended keys, the program branches to the section meant to handle it. However, any other key will be examined to see if it matches against a table of value keys, and Status$ is modified and displayed.

Of course, we can't simply discard that key, so it is then stuffed back into the keyboard buffer, prior to calling the Editor routine. Thus, Editor also gets a chance to process the key.

The next long block of code examines and processes all of the extended keys that are recognized by the program. Each key is clearly labeled within the CASE statement, again to make it easier for you to add additional processing if desired.

The final portion of the subprogram contains the routines that do most of the real work. Separate subroutines are used to highlight the current cell, create the "A1" cell address marker, and print the column header, a cell's contents, or the entire worksheet.

The next two subroutines create a line of data for display, and process a cell after it has been edited. Even though a two-dimensional string array is used to hold the worksheet, each line is created separately which allows it to be quickly displayed.

The last two subroutines are used to calculate the worksheet, and these must of course be customized by you. The first subroutine is entered whenever the F9 Calc key is pressed, and it must contain statements to recalculate the entire worksheet. The second is invoked whenever a cell has been edited, and may be set up to recalculate only those cells that depend on the one that was edited.

QUICKPAK PROFESSIONAL FUNCTIONS

We have provided a number of useful scientific and financial functions that are intended to be added to your programs. Nearly all of the scientific and financial functions that are present in the popular spreadsheet programs are included, along with many general purpose functions.

All of the functions that are equivalent to those found in most spreadsheet programs have names that begin with the letters QP, and are contained in a file named FNSPREAD.BAS. This is done both to identify them, and to allow you to use meaningful names for your variables. For example, the QuickPak Professional Future Value function is called QPFV#, which allows you to also have a variable named FV#.

All of the scientific and financial functions are set up to accept and return double precision values. If you prefer, they may easily be changed to use single precision variables. Simply modify the function header and parameter list as shown in the example below. Of course, you must also change all references to those variables in the function definition.

```
FUNCTION QPATAN2#(X#, Y#)
```

becomes

```
FUNCTION QPATAN2!(X!, Y!)
```

Because these functions are small, they have all been placed into a single file, along with a short demonstration. Therefore, we recommend that you load FNSPREAD.BAS as a module, and then move the ones you need to your main program. This eliminates having to deal with many individual files.

To move a subprogram or function with the QuickBASIC 4 editor, first press F2 which brings up a list of all of the modules that are currently loaded. Simply move the cursor highlight bar to the function or subprogram to be moved, and then press Alt-M. You will be prompted for the destination module that is to receive the subprogram or function.

VERY IMPORTANT

If a function has been moved from one module to another, it is no longer present in the source module in memory. When you subsequently unload the original function file, QuickBASIC will warn that the file has been modified, and ask if you want to save it. Be sure to answer "No", otherwise the file will be saved, but without the function that had been moved.

In only a few cases, one function will require the presence of another. For example, the QPAVG# (average) function also calls upon QPSUM# and QPCOUNT#. Those functions that require another function are clearly marked at the beginning of their source code as to what additional routines are needed.

All of the functions are identified in a header portion of FNSPREAD.BAS, with the meaning of each expected variable clearly identified. Functions that operate on arrays are set up to access the entire array. If you need to process only a portion of an array, the source code may be easily modified as follows.

The first few lines in each of the array functions obtain the lower and upper bounds of the array like this:

```
FUNCTION QPSUM#(Array#())
    Lo% = LBOUND(Array#, 1)
    Hi% = UBOUND(Array#, 1)
    .
    .
    FOR X% = Lo% TO Hi%
```

Simply change the incoming parameter list to also include the Lo% and Hi% parameters:

```
FUNCTION QPSUM#(Array#(), Lo%, Hi%)
```

Next, remove the lines that assign Lo% and Hi%. Then when the function is used, also pass the desired range of elements to be included.

The remaining functions are contained in the file FNOTHER.BAS and are also intended to be moved to your programs. Among the many routines provided are functions for parsing strings, replacing and inserting Tab characters, and converting between signed and unsigned numbers. Each is fully described in the Functions portion of this manual.

Adding these functions to QuickBASIC 2, or 3 is also fairly easy. With QuickBASIC 2 or 3, first load the appropriate function file, and mark the functions of interest with the Shift-arrow keys. Then press F2 to capture the text into the editor's paste buffer. Finally, load your main program, put the cursor at the point where the function is to be inserted, and press the Insert key. This will paste the text into your program.

It is important to understand that functions in QuickBASIC 2 and 3 *must* be defined before they are used. That is, they must be physically positioned in the BASIC source code before any lines that refer to them.

VERY LONG INTEGERS

The recent addition of long integers to BASIC is welcome indeed. Where many repeated calculations with conventional floating point variables often produce small cumulative errors, long integers are always accurate to the penny. Further, long integer operations are considerably faster than the same operations on double precision variables.

Unfortunately long integers don't always provide a sufficient range of values. Considered as pennies, a long integer can hold numbers ranging between plus and minus 21 million dollars or so. That's better than the range of conventional integers, but clearly inadequate for serious financial work.

Therefore, we have provided a set of six subroutines to manipulate what we call Very Long integers. A Very Long integer is comprised of eight bytes (64 bits), and it can hold any number between -9,223,372,036,854,775,808 and +9,223,372,036,854,775,807. Now, *that's* a range of numbers—just like a mainframe!

All of the standard four-function math operations are included to add, subtract, multiply and divide Very Long integers. Of course, you'll also need some way to print and assign them. Therefore, a pair of routines to convert between the Very Long format and a conventional string are also provided.

Because Very Long integers occupy eight bytes, a natural parallel exists between them and conventional double precision variables. Therefore, you will use those to hold the values manipulated by the Very Long integer routines.

A Very Long integer may be included in a field statement for use with disk files, however it is important to understand that you must always pack and unpack the variable whenever it is to be assigned or printed. The example below shows a Very Long being assigned from a string, and written to a disk file:

```
CALL VLPack("1234567890123", VL#, ErrFlag%)
OPEN "TEST.DAT" FOR RANDOM AS #1 LEN = 65
    FIELD #1, 8 AS VL$, 57 AS Stuff$
    LSET VL$ = MKD$(VL#)
    LSET Stuff$ = "This is a silly test message"
    PUT #1, 1
CLOSE #1
```

The next example shows how all of the records in a data file may be read, and a cumulative total obtained for each Very Long field:

```
OPEN "TEST.DAT" FOR RANDOM AS #1 LEN = 65
    FIELD #1, 8 AS VL$, 57 AS Stuff$
    Records% = LOF(1) / 65              'find number of records
                                        'and zero out the total
                                        'accumulator
    CALL VLPack#("0", Total#, ErrFlag%)
    FOR X% = 1 TO Records%              'process entire file
          GET #1, X%                    'read a record
          CALL VLAdd(Total#, CVD(VL$),Total#,ErrFlag%)
                                        'accumulate total
    NEXT
CLOSE #1

Tot$ = SPACE$(20)                       'we MUST make room for the
                                        'returned string
CALL VLUnpack(Total#, Tot$, ErrFlag%)
Tot$ = LTRIM$(Tot$)                     'optional to strip leading
PRINT "The total is "; Tot$             'spaces in the string
```

or

```
PRINT USING "############,.##"; VAL(Tot$)
```

The example above accumulates a running total, but to save an extra scratch variable Total# is used twice in the calling argument list. That is, Total# is one of the operands being added, as well as the variable that receives the result.

Normally passing the same variable twice like this would be considered a bad programming practice. However, all of the Very Long subroutines have been designed to work correctly when variables are passed in this manner.

The ErrFlag% variable is handled somewhat differently by the various Very Long routines. If a string to be packed contains letters, punctuation, or any other extraneous characters, VLPack will set ErrFlag% to -1 to indicate the error. Otherwise, ErrFlag% will return holding zero.

However, the only error that could occur with VLUnpack is failing to give it a string long enough to hold the returned information. As with VLPack, ErrFlag% receives a zero if no error occurred, or a -1 if the string length was insufficient.

The four VL math routines set ErrFlag% only if the result of any adding, multiplying, and so forth result in a value that is either too high or too low.

SORTS VS. INDEXED SORTS

Many different types of sorts are provided in the QuickPak Professional package, however all of the sorts fall into one of two categories. The "plain" sorts are intended to order all of the elements in an array to be either ascending or descending. However, we have also provided a set of *indexed* sorts that instead sort a table of pointers. Let's take a closer look at what this means.

When one of the normal sorts is called to sort an array (or a portion of an array), the elements are actually exchanged to fall into the desired order. If you print all of the elements in sequence after the array has been sorted, you can easily see how they have been arranged:

```
CALL SortI(SEG Array%(Start), Size%, Direction%)
FOR X = Start TO Size%
   PRINT Array%(X)
NEXT
```

The indexed sorts, on the other hand, sort a parallel integer array rather than the primary array, thus retaining the original order while also providing access in sorted order. The example below shows the same array being sorted as above, except the elements are printed out by referring to the index array subscripts.

```
CALL ISortI(SEG A%(Start), SEG I%(0), Size%, Direction%)
FOR X = Start TO Size%
   PRINT A%(I%(X))
NEXT
```

It is important to understand that you *must* dimension the parallel index array to the correct number of elements. Further, it is also up to you to initialize the index array to increasing values beginning with zero. The indexed sorts require a reference point to be able to access each element in the primary array being considered. This means that when the sorting begins, index element zero must contain a 0, element one will hold a 1, and so forth.

Even though it would be simple for the sorting routine to initialize the index array itself, this would disallow resorting a second time based on another key. Therefore, we have provided the routine InitInt which is specifically intended to initialize an integer array to ascending values very quickly. The example below shows how InitInt would be used in the typical context of sorting an entire array:

```
DIM A%(1000), I%(1000)          'dim the arrays
CALL InitInt(SEG I%(1), 1, 1000) 'initialize the index
CALL ISortI(SEG A%(0), SEG I%(0), 1001, 0)
FOR X = 0 TO 1000
   PRINT A%(I%(X))
NEXT
```

One other point that you should be aware of is reserving sufficient stack space when sorting very large arrays. The Quick Sort algorithm used by the various sort routines uses the PC's stack as it is working. Unfortunately, it is not always clear exactly how much stack memory will be required.

We recommend that you set aside one byte of stack space for each five elements being sorted. In the example above, 1000 elements are being sorted, so at least 200 bytes of stack space will be needed.

Unless you instruct BASIC otherwise, it will always set aside at least 1,000 bytes of memory for a stack. So in this case, no extra effort is needed on your part. But when the number of elements exceeds, say, one or two thousand, you should use the CLEAR command as shown below to establish additional stack memory:

```
CLEAR ,, stacksize
```

If you are using BASIC PDS, the STACK statement provides the same service, but without the side effects of CLEAR.

Remember that besides the sort routines, other resources in a PC may also need the stack at the same time. For safety, we recommend setting aside one byte per five elements *plus* an extra 512 bytes just in case. By now you may be thinking, "Okay, but what will happen if I don't provide enough stack memory?" Good question.

In a BASIC program, the stack resides near the bottom of the 64K string data segment. As new items are added to it, they are placed at ever lower addresses, and an internal counter in the PC's microprocessor tracks where the next available stack location is. If too many items are added to the stack, they will extend into and overwrite your variables and other data, which will cause all sorts of problems.

Even though BASIC provides the FRE(-2) function to determine how much stack space is available at any given time, FRE is *not* able to monitor the stack while the sort routines are working.

BASIC also uses the stack for its own purposes—retaining the address to return to after a GOSUB or CALL, and to store the addresses of variables being passed to a subprogram. For example, every time a GOSUB is performed, two bytes are used, while a CALL to a BASIC subprogram with no parameters requires ten bytes.

Each simple variable passed in a parameter list uses an additional two bytes, and each one-dimensional array adds ten more. On top of that, another two bytes per dimension are taken when using multi-dimensioned arrays.

Using this as a guideline, you can determine fairly easily the amount of stack memory required by your programs, based on the number of parameters being passed, as well as the depth of subprogram nesting.

One final point worth mentioning concerns the QuickPak Professional assembler sorts. We have provided a number of different sort routines, each optimized for a particular type of variable. That is, separate sorts are included for integer arrays, long integers, single precision, double precision, normal strings, and fixed-length or TYPE arrays.

If you intend to sort, say, only long integers, then that routine will provide the highest performance possible. However, if you need to sort several different types of arrays in one program, you might consider using only the TYPE sort routine. (Or the indexed TYPE sort where applicable.)

The TYPE sort can be used to sort all of the other variable types, except conventional string arrays. Even though the speed will be slightly lower than with the sorts that are optimized for a single variable type, your programs will be that much smaller.

DEFCNF AND SETCNF

DEFCNF.BI defines a TYPE variable that contains all of the information about the host PC that most programs would need to know. A complimentary program named SETCNF.BI then queries the system and assigns each component defined in DEFCNF. For example, the type of monitor installed, whether or not a mouse is present, and so forth.

Two separate files are used because the system hardware really needs to be examined only once. However, the TYPE variable that is defined in DEFCNF will also be needed in any other modules that are linked to the main program. This way, SETCNF will assess the system and fill in all of the elements of the Config TYPE variable. Then, the entire variable and all of its components may be passed to any of the other modules in one operation—as long as they also include DEFCNF to define the Config TYPE variable.

DEFCNF and SETCNF will thus be added near the beginning of your main programs like this:

```
'$INCLUDE: 'DEFCNF.BI'
'$INCLUDE: 'SETCNF.BI'
```

Then, any external modules that also need access to the configuration will need to include only DEFCNF.BI:

```
'$INCLUDE: 'DEFCNF.BI'
```

To be consistent with the Microsoft way of doing things, the type of monitor detected may be ignored and forced to mono by placing a /B argument on the command line of any program that uses SETCNF. If you do *not* want this feature, simply REMark out that statement in the SETCNF.BI Include file. Likewise, you could add a "/C force color" command line option using a similar technique.

CREDITS

A number of people have made important contributions to QuickPak Professional, and they all deserve credit for their programming skills and hard work. Each is listed below in alphabetical order, followed by the routines they have written.

John C. Bean:	ReadFileX
David Cleary:	The entire suite of XMS routines.
John Conrad:	FCopy, FEof, FOpenAll, FOpenS, NameDir, FStamp and WhichError
Bill Eppler:	All of the spreadsheet functions except those noted below
Ed Ernst:	QPIRR and QPNPV spreadsheet functions
Robert Hummel:	All of the assembler numeric and TYPE array sorts, and the floating point Compare routines used by the assembly language Max, Min, and Search programs
Robert Karp:	Clock and Keyboard
Chris May:	AMenu, AMenuT, Box, DirFile, Editor, FUsing, MenuVert, PUsing, YesNo. Chris also did all of the research and coding to create the object modules generated by MakeQlb.
Jay Munro:	PSwap, AsciiPick, ColorPick, MouseRange, QPStrI and QPStrL.
Jeff Prosise:	Pause2 and the entire suite of EMS routines.
Thomas Renckly:	DEMOEMS2.BAS
Don Resnick:	Calc
Philip Martin Valley:	DEMO123.BAS
Harald Zoschke:	Calendar

Crescent partner Don Malin contributed many important programs including MsgBox, PullDown, QEdit, Spread, VertMenu, ViewFile, and the QD "QuickDOS" utility. Don also wrote MakeQlb, PickList and VertMenuT.

Paul Passarelli is a member of the Crescent Software staff, and he wrote ASCIIChart, C2F and F2C, Delimit, MouseTrap, Parse, QPrintAny, Rand, and all of the Very Long integer routines. Paul also wrote FileSort, Evaluate, QPValI and QPValL.

Phil Cramer, another member of the Crescent Software staff, wrote the Dialog subprogram and demonstrations. Phil also wrote the SCROLLIN. BAS subprogram.

Brian Giedt who wrote our GraphPak and GraphPak Professional packages created the graphics mode ScrnDump subroutine.

Warren Bofinger designed and typeset the original version of this manual, with later additions by Jacki W. Pagliaro.

All of the remaining subroutines, functions, and example programs were created by Ethan Winer. This manual and the Assembly Tutor were written by Ethan Winer.

DIFFERENCES FROM PREVIOUS VERSIONS OF QUICKPAK

Besides the enormous number of new programs in QuickPak Professional, we have also made important improvements to many of the routines that are derived from the original QuickPak.

One of these is the addition of a Page parameter to the various video routines. Where the original QuickPak video routines could only deal with text page zero in standard eighty-column mode, the video services in Professional can accommodate any number of rows and columns, on any legal video page.

Many of the video routines now require a Page parameter to indicate which text page is to be used. For example, the original version of QPrint specified only the string to print and a color:

```
CALL QPrint(X$, Colr%)
```

The new version is instead called like this:

```
CALL QPrint(X$, Colr%, Page%)
```

If Page% is set to zero, then QPrint will write to text page zero, regardless of which page is actually active. Likewise, setting Page% to 2 will print on text page 2, even if page zero is currently being displayed. This capability lets you build screens in the "background", to be displayed at a later time by using BASIC's SCREEN command.

Of course, multiple pages will work only with color display adapters—for mono screens you should always use page zero. Otherwise, the information will be written to an area of memory that is not active. If a new monochrome standard that allows more than one text page should ever emerge, though, the QuickPak Professional video routines will be ready.

As a further enhancement, you may also print on whatever page happens to be active at the time QPrint is called by using -1 as the page parameter. In a similar fashion, the color parameter may also be -1.

In the original QuickPak we provided both QPrint and QPrint2. Where QPrint expects you to tell it the color to use, QPrint2 would always honor whatever colors were already on the screen. In the new versions of QPrint (and APrint), the original colors may be preserved by specifying a color value
of -1.

Because these new capabilities require a fair amount of additional code to accomplish, we have also provided versions of most routines that operate on page zero *only*. All of these alternate video routines have a "0" appended to their name. For example, ScrnSave0 will save portions of the screen on page zero only. However, the "0" versions will operate correctly in either the 40-column or 43-line modes.

Another important difference relates to the QuickPak Professional subprograms written in BASIC. When life was simpler and SHARED always worked as expected, we used shared variables in several of the input and menu programs. Unfortunately, SHARED does not work across modules in a QB 4 program, so we have modified those routines that had in the past used SHARED to now pass all parameters.

If you are converting an existing program that uses QuickPak routines for use with QuickPak Professional, please consult the routine description portion of this manual for each of the QuickPak services you are using. We have made every effort to maintain as much compatibility with the original versions as possible, however in some cases a change to the CALL parameters was unavoidable.

Also, several of the original QuickPak routines have been renamed. This was done mostly to provide some consistency with the many new routines that have been added. However, in a few instances they were changed simply because we didn't like the originals. For example, many people had trouble pronouncing BasDir, so it is now named ReadFile.

Another change made to QuickPak Professional affects those routines that manipulate arrays. In the good old days when most arrays started with element zero, it seemed logical for us to assume element zero in the size calculations. Therefore, routines such as Sort were designed for you to tell it the number of elements to sort *minus one*. This example is for the *original* QuickPak:

```
DIM Array$(10)                   'really 11 elements
FOR X = 0 TO 10                  'read some data
   READ Array$(X)
NEXT
CALL Sort(Array$(0), 10)         'sort 11 elements
```

Even though the Owner's Guide referred to the Size parameter as the number of elements to sort, it actually had to be one less. But now that QuickBASIC and Turbo Basic allow nearly any starting and ending range of subscripts, we have changed the array routines to expect the actual number of elements.

Two other important changes are the addition of assembler *functions* for QuickBASIC 4, and the re-writing of several BASIC routines into assembly language. Where the original QuickPak routine to get the current default drive required passing it an integer variable, the Professional version of GetDrive returns the result directly.

Likewise, GetDir now returns the current directory without having to preassign and pass it a string. This allows GetDir to return only as many characters as needed, and without the extra trailing zero byte. We have also enhanced GetDir to add the leading backslash automatically.

All of the assembler routines that logically should return a value are now set up this way. Besides being more sensible, in all cases one less parameter will need to be used when calling the routines.

In some cases where we have translated a BASIC routine into assembly language, we are still providing the original BASIC version. It is very difficult for most BASIC programmers to modify someone else's assembler routine, and the BASIC versions allow you to customize them if you really need to.

Both assembler and BASIC versions are provided for several of the menu and input routines. Also, it is more difficult to implement the scrolling bars and mouse capabilities in an assembler menu, so if you need those features you must use the BASIC versions.

Further, only the BASIC input routines show the status of the Cap and NumLock keys automatically. Likewise, the BASIC TextIn subprogram recognizes the Ctrl-arrow keys to jump by whole words, where its Editor assembler counterpart does not.

All of the original QuickPak routines that process arrays are now also provided in a second version meant for use with fixed-length string and TYPE arrays. In all cases these routines have the same name as their original counterparts, but with the letter "T" appended as a suffix. For example, APrint for use with fixed-length strings is called APrintT.

The final improvements relate to the QuickPak routines that process strings. Many of these are now provided in two versions—one that honors capitalization and one that ignores it. Rather than build both capabilities into all of the programs, we have opted to supply separate versions. This allows each dedicated routine to be that much smaller and faster, while eliminating yet another parameter to be passed.

All of the string services that are provided in two forms use a 2 appended to the name to indicate case-insensitive. That is, SortStr will sort a string array and honor the difference between "A" and "a", while SortStr2 considers "a" to come before "B". Likewise, InCount2 will find "Hi Mom" within the string "HI MOM" or "hI mOM", while InCount would not.

DIFFERENCES FROM EARLIER VERSIONS OF QUICKPAK PROFESSIONAL

This version of QuickPak Professional improves on the original 1.XX version in two important ways. The first is simply the addition of more routines. The second improvement is in all of the DOS and video routines, as well as in some of the BASIC subprograms. Also, several of Chris May's assembler routines use a new calling convention that lets you know which key was used to exit the routine. Please see DEMOCM.BAS for additional information.

New routines fall into one of two categories—those present in some earlier versions but not mentioned in the manual, and routines that were not present at all. If you are already using any of the routines that have been changed, be sure to see the appropriate pages in this manual for the correct usage.

All of the DOS routines have been rewritten to eliminate the need for an extra variable solely to indicate if an error occurred. The new method uses a pair of assembler functions to tell if an error occurred on a prior DOS access. This is described in detail in the section "Eliminating ON ERROR".

All of the video routines have been rewritten to use the Monitor function to determine the current monitor type. Where the original versions duplicated the same code over and over, the new versions utilize a common routine to do this. Notice that this change is transparent to your programs, and does not require any modifications to the code.

A new method for storing a string array in an integer array (the string manager routines) has been implemented. Where the original version stored the length of each string immediately before the string data, the new method uses a carriage return/line feed pair to delimit the strings. This approach lets you directly write the array to disk quickly, while storing it in a conventional ASCII text file format. The new FGetA and FPutA routines are provided for this purpose.

Finally, several of the BASIC routines have been enhanced. In most cases, no change is needed in the way they are called, however in the QEdit text editor, a new TYPE variable has been added to the parameter list.

TUTORIALS

COMPARING CALL, GOSUB, AND MULTI-LINE FUNCTIONS

With all the features and capabilities in the latest wave of BASIC compilers, we are often asked to explain the difference between a called subprogram, a GOSUBed routine, and a multi-line user-defined function. In many cases it really doesn't matter which you use, however some situations definitely favor one approach over the others. For example, using separately compiled subprograms is the only way to create a final program with more than 64K of code with QuickBASIC.

Essentially, a called subprogram would be used when you want to *do* something, or need to alter a variable. On the other hand, a function would be indicated when a value is to be computed based on the contents of one or more variables. DEF FN type functions have the added characteristic that they cannot alter any of the incoming variables. Finally, GOSUB routines are useful when a large number of variables must be shared with the main program. Let's take a closer look at each of these methods in turn.

SUBPROGRAMS

When a subprogram is called, the addresses of all the variables included in the parameter list are made available to the subprogram. Whenever one of these variables is assigned in the subprogram, the changes will also be reflected in the main program because the variable was actually altered. Further, any of the variables used in a subprogram that were *not* passed in the list are considered to be "private". That is, you can have a variable named Price within the subprogram, and it will not conflict with a variable of the same name outside the subprogram.

Of course, you can also share variables between a main program and its subprograms with the SHARED statement. There are two ways to do this. One approach is to declare the variables as being shared within the subprogram. The only problem with this method is that in QuickBASIC, a bug causes the SHARED statement to be ignored once a program grows beyond a certain size. I have observed this with QB versions through 3.0, so it isn't recommended. Also, variables that are declared as shared within a subprogram will be shared only with the main program, and not with any other subprograms.

The other method is to use DIM SHARED, which will establish one or more variables as being shared throughout an *entire* program. Even though DIM is usually meant for declaring the size of an array, you can also use it to indicate which variables are to be global. Here's one example:

```
DIM SHARED Max, Position, Array$(250)
```

Notice that DIM SHARED is not supported by Turbo Basic as of version 1.1, so you must place separate SHARED statements within each subprogram.

FUNCTIONS

Functions are useful in a variety of programming situations, and differ from subprograms in that they are not invoked by the BASIC CALL command. Rather, you simply refer to a function by its name as part of an expression. If you defined a function to return the square of a number, it might be written like this:

```
DEF FnSquare(X) = X * X
```

Then to square a number, you would simply include the function as part of an assignment, or perhaps use it in a PRINT statement:

```
Value = FnSquare(2.3)
```

or

```
PRINT FnSquare(A)
```

Functions also have the unique ability to return a numeric result based on a string, or vice versa. For example, BASIC's built-in STR$() function will accept a numeric value, and convert it into the equivalent string form.

```
X$ = STR$(Value#)
```

Similarly, the VAL function accepts an incoming string, but returns its numeric value:

```
Number% = VAL("43562.89")
```

Since a function can't actually change any of the variables being passed to it, it would be less appropriate for, say, modifying a variable in some way. Of course, BASIC lets you do just about anything you'd ever want; however, using a function that way would require extra program instructions. For example to convert a string to all upper case with the UCASE$ function available in Turbo Basic and QuickBASIC 4.0 you would have to assign a new variable:

```
CustName$ = UCASE$(CustName$)
```

As opposed to calling a subprogram that capitalized the string directly:

```
CALL Upper(CustName$)

SUB Upper(X$) STATIC
    FOR X = 1 TO LEN(X$)
          Char = ASC(MID$(X$, X))
          IF Char > 96 AND Char < 123 THEN
               MID$(X$, X) = CHR$(Char - 32)
          END IF
    NEXT
END SUB
```

If you wanted to use a capitalized version of a string, though, without actually changing it permanently, then a function is ideal.

```
IF UCASE$(Answer$) = "YES" THEN . . .
```

or

```
PRINT UCASE$(CustName$)
```

This way, the string isn't actually altered, but your program can use it as if it were. Because of this behavior, functions are not often used to modify or print variables, or to perform file access, though of course they could be.

Functions are also ideal for providing a single result based on a number of input variables. Many of BASIC's built-in functions work like this as well, for example INSTR accepts two different string arguments, and returns one numeric result. Another place a function might be useful is when you need to find the maximum or minimum of two or more values:

```
DEF FnMax(X, Y)
    IF X > Y THEN
          FnMax = X
    ELSE
          FnMax = Y
    END IF
END DEF
```

Then to print whichever variable is greater, simply include the function within a PRINT statement.

```
PRINT FnMax(Value1, Value2)
```

Unlike subprograms where the incoming variables may be freely changed, BASIC instead makes a temporary copy of any parameters being passed to a function. Then, the addresses of these copies are made available to the function, rather than the addresses of the original variables. Thus, any variable assignments within a function will be in effect only while the function operates, since only the copies are being manipulated.

In the function below, even though the incoming variable Z is being modified, the assignment will not be reflected in the main program once the function has finished.

```
DEF FnSigned(Z)
    IF Z > 32767 THEN Z = Z - 65536
    FnSigned = Z
END DEF
```

Another important point is that unlike subprograms, variables that are not part of the function's incoming parameter list will be common to the rest of the program. Of course, functions can also utilize local variables, but they must first be declared STATIC.

Which brings up an important point. Even though STATIC is generally used to declare a variable as being private to a function or subprogram, its *real* purpose is to permanently allocate storage space while a program is being compiled. Normally, a programmer uses a high-level language to avoid having to deal with the messy details of memory allocation, DOS functions, and so forth. In this case, however, it is important to understand how variables are stored.

When a variable has been declared as STATIC, an area of memory is set aside to hold it at the time the program is compiled. This is exactly the same way that regular variables are handled in a main program. But with the introduction of QuickBASIC 4.0 and its support for recursive functions and subprograms, you may now stipulate that variable space is to be created when the program is run. In this case, temporary space is set aside on the processor's stack to hold the variable's value.

In a recursive subprogram, any variable that is not declared as static will be created and initialized to zero (or a null string) each time the subprogram is invoked. But in some situations you might want to retain the value of certain variables between successive calls. This is precisely what the STATIC statement is for—it creates a permanent memory location in normal RAM for those variables. Otherwise, the variables exist only while the subprogram is running, and the stack memory that holds them is abandoned when the subprogram ends.

This is an important distinction, because recursive functions must preserve all of their local variables if they are to be repeatedly invoked. If a recursive function or subprogram is midway through a series of calculations and then calls itself, the second invocation must not destroy or overwrite any variables from the level above. This is why new memory areas must be found to hold the non-static variables at the time the function is used, rather than once at the time the program is compiled.

GOSUB

In most cases, a GOSUB routine is not as useful as a called subprogram, since the names of each variable being manipulated must be whatever the subroutine is expecting. In the old days, the only way you could have a subroutine do the same thing to many different variables was to constantly assign and reassign:

```
Temp$ = CustName$ : GOSUB 1200 : CustName$ = Temp$
```

One exception, though, is when a large number of variables must be shared between the main program and the subroutine. When I first began writing a word processor I've been tinkering with for about a year, I started to write the margin-setting routine as a subprogram.

But then I was faced with either passing all of the margins and other assorted default values as parameters, or else declaring the twenty or so variables as being SHARED. It finally hit me that a plain old GOSUB made the most sense—the routine could access all of the variables it needed, and I didn't have to pass or share any of them.

Yet another advantage of GOSUB is due precisely to the fact that it is not kept separate from the rest of the program. One of QuickBASIC's many restrictions in its error handling is that errors that occur in a subprogram must always be handled in the main program. If the routines that perform disk reading and writing are all in the main program to begin with, you'll have less difficulty resuming a program's execution where you want to.

Finally, a GOSUB can be executed much more quickly than a CALL can. Each time a CALL is used in a BASIC program, the address of each passed parameter must be placed on the PC's stack. When an entire array is passed, many bytes will be involved which takes even more time. (See the section about sorting elsewhere in this manual for an additional discussion of how the stack is used by BASIC.) Stack operations are notoriously slow on the 8088 series of processors, and are avoided by assembly language programmers whenever possible.

FUNCTIONS IN QUICKBASIC 4.0

QuickBASIC 4.0 introduces yet another type of function, which Microsoft refers to as a Function Procedure. This type of function is really just a variant of the subprogram, though it does return a value as part of an expression. Like subprograms, variables that are not declared STATIC in a Function Procedure will be local to the procedure. Also like subprograms, any of the incoming variables may be freely altered by the function. Further, this type of function does *not* need to be physically positioned before the lines that invoke it in a program, as required by the original DEF FN functions.

It is also possible to call a Function Procedure (or subprogram) in such a way that the incoming variables may *not* be altered if you prefer. This is done by enclosing the variable's name in parentheses, which QuickBASIC will then treat as an *expression*. Even though this may seem a little odd, it makes perfect sense once you think about it. For example, if you have a subprogram that expects an incoming numeric variable, you could also call it with a numeric expression like this:

```
CALL DoSomething(X * 5)
```

In this case, it would not be appropriate for BASIC to pass the address of the variable X, since what you really want is to send it the result of multiplying X by five. Therefore, QuickBASIC first performs the calculation, then stores the result in a temporary location, and finally sends the address of the temporary location to the subprogram.

In a similar manner, you can trick QuickBASIC into thinking a variable is really an expression by enclosing it in parentheses:

```
CALL DoSomething((X))
```

Microsoft's otherwise excellent BASIC documentation mistakenly describes this technique as "passing by value," claiming that the *value* of the variable is being passed rather than its address. In truth, a temporary copy of the variable is created, and the address of the copy is then passed to the function. Thus, any changes made to the incoming variable will affect only the copy, and the original variable's value will be preserved.

SUMMING UP

We have looked at a variety of different ways that blocks of BASIC code can be organized. Even though functions now have the capability to extend across multiple lines and do operations like printing and file I/O, they are more often used to perform calculations based on variables or values.

And while subprograms can be used to do nearly anything you'd like, they are better suited to manipulating variables, or performing specific actions. Finally, GOSUB routines provide a simple way to share many variables with the main program, as well as simplifying error and event trapping.

DYNAMIC VS. STATIC ARRAYS

When QuickBASIC was first introduced, one of the most important new features offered was the addition of Dynamic arrays. Where earlier versions of the BASCOM compiler supported only Static arrays that impinged on string space, Dynamic arrays finally allowed the programmer to exceed the 64K limit on total data size.

When an array is dimensioned as static using a statement like:

```
DIM Array(1000)
```

the area of memory that will hold the array is permanently set aside when the program is compiled. This area of memory is located within the same 64K memory block that also contains strings, simple (non-array) variables, and any DATA items. Further, QuickBASIC also uses this memory for its own work variables such as the current color settings, cursor location, and the most recent DEF SEG statement.

However, an array may be dimensioned as Dynamic, which causes the memory to be allocated when the program runs, *outside* of the normal 64K data segment. There are several ways to tell QuickBASIC that an array is to be Dynamic. One is to use a variable or expression to specify the number of elements. Since the variable's value isn't known when the program is compiled, the memory must be set aside later when it is run.

Another way to force QuickBASIC to create an array as Dynamic is to use REDIM instead of DIM, which brings up an important point. Since the memory for a Static array is permanently set aside at compile time, it cannot be recovered. Thus, when a Static array is erased all of the elements will be cleared to zero, but the memory that holds the array is not released. Of course, Static arrays may not be redimensioned.

When a Dynamic array is erased, the memory it occupies is made available to the program again. And when an array is redimensioned, the memory is first released, and then claimed again for the new number of elements. This is an important distinction, not only because of where the array is located in memory, but also in the way ERASE operates.

Yet another way to indicate how an array is to be treated is with the $STATIC and $DYNAMIC metacommands. By default, any array dimensioned using a constant for the number of elements will be Static. However, if the $DYNAMIC metacommand precedes the dimension statement, QuickBASIC will consider the array to be Dynamic. If a $STATIC metacommand is then used later on, subsequent arrays will be static.

We do not recommend using $STATIC or $DYNAMIC, because it is confusing to keep track of which metacommand is most recently active. If $DYNAMIC is placed on the first page of a very long listing, it would be easy to miss next month or next year when you go back to the program. Using REDIM makes it very clear that the array is Dynamic—right at the point in the source listing where the array is dimensioned.

Since Dynamic arrays are always created outside of the normal 64K string data area, you might think that Dynamic arrays are always preferable. However, this is not necessarily the case. QuickBASIC always sets aside as much string space as possible (up to the 64K limit), but you may not always need that much.

If a program must be able to run on a PC with only 256K of memory, it is conceivable that your program could end up with lots of string memory sitting there unused, and not enough far memory because of all the Dynamic arrays. The best approach is probably to estimate how much string space a program will require, and then plan on creating Static arrays to fill what remains.

Fortunately, QuickBASIC provides an easy way to determine exactly how much memory is being used for both strings and Dynamic arrays. The FRE("") function returns the amount of string space currently available, and FRE(-1) tells how much far memory remains. You could either sprinkle a few PRINT statements throughout the program, or use the QuickBASIC Watch capability to monitor these as the program runs.

The last point that you should be aware of is how dynamic string arrays are treated. In a conventional string array (not fixed-length), the strings are always in the 64K data segment regardless of whether they are static or dynamic. However, a dynamic string array is completely removed from memory when it is erased, where a static string array is merely cleared to all null strings.

SAVING SCREEN IMAGES TO DISK

One of the things we are often asked to explain is how text and graphic screen images may be saved to disk and loaded again later. This is actually quite easy to do, however it does require knowing which type of display is present, and whether the screen is currently in text or graphics mode.

Text screens are the simplest to save and load, and only requires knowing whether the monitor is monochrome or color. In IBM PC/XT/AT/compatible computers, a different area of RAM is used to hold the screen characters for each of the two display types.

The QuickPak Professional Monitor%() function can tell you the exact monitor type that is currently active, but this information may also be determined by looking in low memory at the BIOS video status byte.

You may use either DEF SEG and PEEK to access this byte, or the QuickPak Professional Peek1 function which eliminates having to use DEF SEG. Both methods are shown below.

```
DEF SEG = 0
IF PEEK(&H463) = &HB4 THEN
    'mono
ELSE
    'color
END IF
```

or

```
IF Peek1%(0, &H463) = &HB4 THEN . . .
    'as above
```

Once the monitor type is known, a screen may be saved with BASIC's BSAVE command, or the QuickPak Professional QBSave which returns an error code. For simplicity, we'll use BASIC's BSAVE here.

Notice that this example assumes the screen is in the normal twenty-five line by eighty-column mode. If it is not (and this can be determined with the GetVMode routine), the number of bytes being saved must be changed. For example, a 40 by 25 screen occupies only 2000 bytes, while an 80 by 43 screen requires 6,880 bytes.

The size of any text screen may be easily determined by multiplying the number of rows by the number of columns, and then multiplying the result by two. The rows and columns must be multiplied by two because for each character on the screen, a second byte is also used to hold its color.

```
IF Peek1%(0, &H463) = &HB4 THEN          'mono
   DEF SEG = &HB000
ELSE                                     'color
   DEF SEG = &HB800
END IF
BSAVE "screen.", 0, 4000
```

Here,"screen." would be whatever file name you want to use, the zero means to save the screen starting at offset zero, and then store all 4000 bytes.

Loading a previously saved screen is just as easy:

```
IF Peek1%(0, &H463) = &HB4 THEN
   DEF SEG = &HB000
ELSE
   DEF SEG = &HB800
END IF
BLOAD "screen.", 0
```

Notice the extra period after the screen name. If this is omitted, BASIC will stupidly append a .BAS extension, which is obviously inappropriate. The period forces BASIC to use no extension at all, though you could of course add one. Also notice that even though a text screen may have been saved from a color display, it may be loaded later to a monochrome display or vice versa.

Saving a graphics screen is slightly more complicated, mostly because of the large number of graphics modes and display adapters. We'll look at how to save a CGA graphics screen first, and then an EGA and VGA.

As with the text screens, it is up to you to know how many bytes must be saved. For this, using GetVMode makes a lot of sense because among other information, it also returns the size of the screen currently in use. The page size may also be found by peeking the two bytes at address 00:44Ch using the QuickPak Peek2 function:

```
CALL GetVMode(Mode%, Page%, PageSize%, Rows%, Cols%)
```

or

```
PageSize% = Peek2%(0, &H44C)
```

To save the graphics screen use:

```
DEF SEG = &HB800
BSAVE "gscreen.", 0, PageSize%
```

To load it back use:

```
DEF SEG = &HB800
BLOAD "gscreen.", 0
```

EGA (and VGA) screens are much more complicated than CGA screens, due to the sheer amount of memory that each screen occupies. When the original IBM PC was first designed, the engineers probably had no idea that such high-resolution screens would ever be connected to it. Because EGA and VGA graphics screens require much more memory than a PC sets aside for video, a technique known as "bank switching" is used.

In a bank switched system, more than one block of memory is accessed by physically switching electrical connections from one range of addresses in the processor to different areas of RAM. This is sometimes called "mapping", and in the case of the EGA, only one of four possible memory banks (or planes) is mapped to the video address space at a time. Separate planes are used for the red, green, blue, and intensity (brightness) information. In BASIC, the current plane may be selected by a series of OUT statements.

Rather than provide a detailed program listing here, we have included the program EGABSave on the QuickPak Professional disk. EGABSave is configured to both save and load an EGA hi-res screen image, though comments in the save subprogram show how to modify it to instead save the VGA screen 12. When you run EGABSave, you can clearly see each portion of the screen as it is reloaded into screen memory.

SAVING ARRAYS TO DISK

One very important application for BSAVE (and the QuickPak Professional QBSave routine) is to reduce the time needed to save a large data file from a numeric array. When an entire array of numeric data must be saved on a disk, the most common approach is to create a sequential file, and use the PRINT # statement to write each value.

Unfortunately, using a sequential file this way is unbearably slow, because each number must be converted from BASIC's internal format into a string of ASCII digits. Worse, each number will take up extra space on disk to store the digits.

That is, the integer value 12345 will occupy six bytes, with one required for each digit, and another for the separating comma that is also needed. Contrast this with BASIC's method for storing an integer in memory, where only two bytes are used, regardless of the variable's value.

Using QBSave (or BASIC's BSAVE) overcomes both of these problems, by quickly transferring the contents of an entire numeric array to a disk file. Because the file is simply a "snapshot" of memory, the operation happens very quickly, without regard for the meaning of the values in the array. The QuickPak Professional FPutA routine is equally useful for saving an area of memory, and it is described in the routines section of this manual.

Though an entire numeric array may be saved in this fashion, you cannot save a conventional string array. The elements in a string array are *not* in contiguous memory locations, rather they are scattered throughout the 64K string space. A table of pointers then holds their actual location. However, a fixed-length string array may be saved, but only if it is no larger than 64K in size. Which brings up an important point.

The DOS service that writes to a file uses a single word to specify how many bytes are to be written. This means that in order to create a file larger than 64K (65536) bytes in length requires multiple passes. Thus, if you intend to save a "huge" array using either BSAVE or QBSave, you will need to do so in pieces.

A related point is how you would specify saving more than 32767 bytes, since that is the largest value that may be held in an integer variable. In BASIC, integer numbers that are larger than 32767 are instead considered to be negative.

Though the designers of BASIC could just as easily have allowed integers to range from 0 through 65535, they decided that a range of -32768 through 32767 offers more flexibility. Therefore, if you need to save, say, 43788 bytes, you would first subtract 65536, and then use the result as an argument to QBSave, as shown below:

```
ActualBytes! = 43788          'or whatever
IF ActualBytes! > 32767 THEN
   NumBytes% = ActualBytes! - 65536
ELSE
   NumBytes% = ActualBytes!
END IF
```

Even more effective would be to use a long integer to specify the number of bytes. Although QBSave and the other QuickPak Professional DOS routines expect an integer variable, they will work just as well with a long integer. The important difference is that a long integer may be assigned values greater than 32767, without having to create an artificially negative number.

Determining the number of bytes a given array occupies is quite easy to do, based on the number of elements it contains and the size of each element. Each element in an integer array comprises two bytes, while long integer or single precision variables require 4. Double precision values need eight bytes each, though a fixed-length string may of course be any size.

Based on this information, the number of bytes to specify for any single-dimension array may be computed as follows:

```
NumBytes = (UBOUND(Array) - LBOUND(Array) + 1) * ElSize
```

Where ElSize is either two, four, eight, or the length of each fixed-length string element. But again, remember that this is for arrays occupying no more than 64K of memory.

It is very important to dimension an array that is to be loaded again later, prior to calling QBLoad or BASIC's BLOAD. Following a similar logic, you would determine the number of elements needed by first using the QuickPak Professional FileSize function to obtain the size of the file. Next, you would subtract the seven bytes taken by the BSAVE header (see below), since that portion of the file isn't actually loaded. Finally, divide the result by the size of each element.

In truth, you don't really need to subtract the seven byte header, since dimensioning to more elements than needed won't cause any harm. However, failing to dimension it large enough is *guaranteed* to cause problems.

One additional topic worth mentioning here is the use of a BSAVE header which is part of the file being saved. When you ask BASIC to create a file using BSAVE, it creates a seven byte header immediately before the saved data. This header contains a special "signature" byte, as well as other information about the file. The QBSave routine creates this header just as BASIC would, which allows the file to be read again later using either method.

The very first byte in a BSAVE file is &HFD, which merely identifies it as having been created by BSAVE. The next four bytes hold the segment and address from which the data has been saved respectively. The final byte holds the length of the data in the file, not including the seven byte header. Notice that the QuickPak Professional QBLoad command ignores this header, and simply loads the entire file into memory at the segment and address you give it.

Finally, you may have noticed in the syntax examples that both QBLoad and QBSave may be called with a different number of arguments. When the SEG option is used in a call parameter list, *two* addresses are actually passed to the routine. One holds the segment of the variable, and the other holds its address. SEG simply lets you pass both of them in a single operation.

However, this means that QBLoad and QBSave must be designed to expect the *value* of those addresses, as opposed to BASIC's usual method of passing an address of a variable. Thus, BYVAL will be needed if you intend to pass a separate segment and address rather than specifying an array.

Because these routines may be called either way, it is important that you declare them using the same method you intend to use when you call them. In the DECLARE.BAS file, they have been declared as SEG, under the assumption that you will be saving and loading arrays. However, the other method is also shown as comments immediately below.

A complete discussion of SEG, BYVAL, and other related topics is given in The Assembly Tutor that accompanies this package.

CALLING WITH SEGMENTS

Many of the QuickPak Professional routines are intended to help you manipulate string and numeric arrays. For example, Insert and Delete will insert or delete elements in a string array much faster than would ever be possible with a FOR/NEXT loop. However, some of these array routines are intended to be used with Dynamic arrays that are not necessarily within BASIC's normal 64K data segment.

For those routines, it is important that they know not only the address for a given range of array elements, but the segment as well. However that segment could be nearly anywhere within a PC's memory. Beginning with QuickBASIC 4 and BASCOM 6, a BASIC program can now inform an assembler routine of both the segment and address.

In QuickBASIC 4 and later, the SEG option is used to indicate that the segment is to also be included when the array address is passed. But the problem is using these routines with earlier versions of QuickBASIC and BASCOM. To solve this, we have provided a set of routines called Pointers with the QuickBASIC 2 and 3 versions of QuickPak Professional.

Whenever a routine is shown in the syntax example as being passed with a SEG statement, you will have to first call one of the Pointers subprograms to get that extra information. The example below shows how to set up the call to Fill2 for QuickBASIC 2 and 3, or BASCOM 1 and 2:

```
CALL PointerI(Array%(), Element%, Segment%, Address%)
CALL Fill2(Segment%, Address%, Value%, Size%)
```

Contrast this with the QuickBASIC 4 calling method:

```
CALL Fill2(SEG Array%(Element%), Value%, Size%)
```

Three different versions of Pointers are provided, with one each intended for use with integer, single precision, and double precision arrays. That is, PointerI will locate an integer array element, while PointerS and PointerD locate a single and double precision element respectively.

QuickBASIC 2 and 3 also come with an assembler routine named PTR86 that serves the same purpose, and you may use that if you prefer.

Turbo Basic passes all arrays (and normal variables as well), with the segment automatically, so a separate SEG command is not needed. Please understand that in each case, the correct version of QuickPak Professional *must* be used. The routines that expect the information as a separate segment and address are not the same internally as those that expect a SEG address.

Related to this is how a fixed-length string array is passed to the QuickPak Professional assembler routines. All of the programs that operate on fixed-length strings are set up to expect a segmented address when they are called. However, due to a "design decision" at Microsoft (they tell us it's on purpose), the SEG call option does not work with fixed-length string arrays.

Rather than pass the true segment and address of the array element as it does with a numeric array, QuickBASIC instead copies the element into a conventional string in near memory, and then passes a segmented address of the copy.

The solution is to create a TYPE array that is comprised solely of a single fixed-length string member. When SEG is then used on the TYPE element, the SEG statement will work as expected. Which brings up an interesting point.

When an array is passed with SEG, *two* parameters are actually sent to the subroutine—a segment and an address. Therefore, a SEG call may be simulated by passing the value of a segment and the value of an address. Even though it appears that a different number of parameters is being sent to the routine, in truth, both methods do exactly the same thing.

Both of the examples below call the ReadFileT routine correctly, except the second requires the fixed-length array to be dimensioned as a TYPE:

```
DIM A(1 TO 100) AS STRING * 12
CALL ReadFileT(Spec$, BYVAL VARSEG(A$(1)), BYVAL VARPTR(A$(1)))
```

or

```
TYPE FLen
   Dummy AS STRING * 12
END TYPE
DIM A(1 TO 100) AS FLen
CALL ReadFileT(Spec$, SEG A(1))         'A() is a TYPE
```

An example of passing a fixed-length string array both ways is also given in the APRINTT.BAS demonstration program.

STORING DATA ITEMS OUTSIDE BASIC'S STRING SPACE

One of the unfortunate limitations with QuickBASIC's handling of strings is the 64K size limit. Even though QuickBASIC offers "huge" string arrays that can be nearly any size, the real problem is frequently DATA statements that hold constant information.

As an example, suppose you have a program that relies on many text strings for various on-line help messages. One typical way to get these into a program might be with a sequence of READ/DATA statements like this:

```
DIM Help$(253)
FOR X = 1 TO 253
   READ Help$(X)
NEXT

DATA "Press any key to continue"
DATA "Insert the PROGRAM disk into Drive A"
.
.
DATA "Press F1 for help"
```

Whenever a quoted string appears within a program listing, QuickBASIC must allocate space somewhere to hold it. As you might imagine, that space is always located within the normal string data segment. However, even *numbers* kept as DATA will steal string memory:

```
FOR X = 1 TO 10
   READ Info%(X)
NEXT
DATA 71, 102, 451, 17, 33, 999, 37, 199, 200, 1034
```

Since BASIC has no way to know whether the DATA items will ultimately be read as strings or numbers, it must preserve the text exactly as it was entered. In the example above, forty five bytes of string space are taken, not counting the additional twenty bytes of variable storage also needed when they are assigned into the Info%() array.

Yet another problem with DATA statements is that they are very slow to read. This problem is made even worse in QuickBASIC 4, which takes approximately four times longer to read data than previous versions. Microsoft has advised us that READ/DATA statements are more flexible than before, which accounts for the loss in speed. For example, DATA statements in a main program may now be read by commands in a Quick Library.

One possible solution is to read the data from a disk file when the program first runs. Using BLOAD to load an entire numeric array is very fast, especially when compared to opening a file for sequential input, and reading each item one by one.

However, this requires not only an extra file, but for string data the space will still be taken away from the available string memory. A much better method is to store large string constants or array data within the *code* segment of a program. As you must know, QuickBASIC allows a program's code to grow to nearly any size, and doing this can conserve a considerable amount of string memory.

The technique about to be described requires an assembler, and is by necessity more complicated than the READ/DATA method. However, if there are many strings or a very large amount of numeric values, the savings will definitely be worth the trouble. In fact, this is the method we used to store the help screens for the QEdit text editor included with QuickPak Professional. Besides the examples about to be described, you may also want to examine the EDITHELP.ASM source file.

Two files are provided on the QuickPak Professional disk to show how this may be accomplished. The first is DATA.ASM, and it shows how to incorporate both string and integer data within an .OBJ file. The other is DATA.BAS which illustrates how to access it later. For the purposes of this discussion, you should have printouts of the programs handy, or have the DATA.ASM file on your screen.

For each group of data items that are to be stored, you will need five sets of statements. The first is a Public clause declaring the assembler functions that return the string address and length. You will also need a function to locate the code segment where the data is being kept—in this case it is called QPGetCS. Once the string has been located, it is a simple matter to copy it into a "normal" BASIC string (or a numeric array where appropriate) using the QuickPak Professional BCopy routine.

Second, a label defining the data's location is needed. This may be any label name you would like to use, though the examples use String1, String2, and IArray. For string data the label will be followed by a DB (define byte) statement, which is then followed by the quoted string or strings.

Integer data must be defined using DW (define word), followed by the words of integer data being stored. Other numeric data types may also be stored, for example DD will set aside space for the four-byte double words used by long integers.

The third item needed is a DW statement to indicate the length of the data being stored. Understand that the assembler does all of the dirty work in locating the data and determining its length. You can freely add or remove items whenever you want, without ever having to calculate anything.

The final two are assembler functions that BASIC can call to find the data, and know how long it is. While all of this may seem like a lot of work, as you can see from the source listing, each individual item is very small. Understand that it is *not* necessary to know how the assembler functions work, though you will of course need an assembler to create the .OBJ modules.

A few items deserve special mention, most notably the use of the "$" operator that indicates to the assembler the *current address*. Rather than requiring you to manually count all of the characters in a string, it makes much more sense to have the assembler do this for you. Consider the first string, which in the listing is called String1.

Once the label has been defined, the assembler assumes that you will want access to the string's address, so it remembers it internally as it works. The next label—Length1—defines a word of storage, however its *value* will be automatically set to the difference in bytes between the String1 address and itself. Again, to the assembler a dollar sign means *here*, and Offset means the address of whatever label is being referred to.

This process is repeated for the second string variable, and again, both the string's address and its length are calculated by the assembler automatically. The integer array data is only slightly different, because the length of each data element is really two bytes, not just one. Thus, we ask the assembler to first calculate the difference in bytes between the labels IArray and LengthI, and then divide the result by two to derive the number of *words*. We want the number of words, so we'll know how large to dimension the array later in the BASIC program.

Each assembler routine contains only two instructions—a command to load either the desired address or length into
the AX register, and a Return instruction. Assembler functions are one of the truly nifty things added to QuickBASIC with version 4, and they provide a very simple way to access information without requiring any passed parameters. As you can see in the DATA.BAS demo, assembler functions *must* be declared in BASIC before they are called.

COMMON PROBLEMS (AND SOLUTIONS)

While we are always pleased to assist you in using any of our products, you will find the answers to some common problems and symptoms described below. If you still need assistance, please call us and we'll be more than happy to help you.

Complete crash or other weird results:

You used the wrong number of variables in a CALL.

You failed to use integer variables with a routine where they are mandatory.

You specified too many elements with a routine that manipulates an array.

A SHARED variable isn't being shared correctly:

Using SHARED within a subprogram shares with the main *only*, not with any other subprograms. To specify that a variable or array is to be shared throughout a program requires DIM SHARED in the main program.

Shared works only within a single program module. To share variables across modules also requires you to use a COMMON declaration.

An assembler function doesn't work:

You forgot to declare it, or

You omitted the type-identifier suffix. For example, PrnReady is an integer function and thus must be declared as

```
DECLARE FUNCTION PrnReady%()
```

QuickBASIC Error Messages

Subprogram not defined

You attempted to access an assembler routine in a Quick Library without using the CALL keyword, but failed to declare the subprogram in your main BASIC program.

You called a subprogram that has been declared, but it is not in the current Quick Library, or you forgot to load the PRO.QLB Quick Library.

You attempted to call one of the QuickPak Professional BASIC routines, but have not loaded it as a module. The BASIC subprograms and functions are *not* in the Quick Library, and must be loaded manually.

You used the QuickBASIC Load option to load one of the demonstration programs instead of using Open. Open is needed for all of the QuickPak Professional BASIC demos to ensure that any necessary BASIC modules are also brought in to the QuickBASIC editor.

String Space Corrupt

This can be caused by a number of different problems. The biggest culprit is simply a bug in early releases of QuickBASIC 4. This has been corrected in version 4.00b and 4.5. The version number is always displayed at the bottom of the screen each time you start the QuickBASIC editor.

Corrupting the string space can also be caused by incorrectly using a QuickPak routine that manipulates string arrays. For example, attempting to sort 100 elements in an array that has not been dimensioned to that size.

Similarly, if you use ReadFile to get a list of file names into a string array, you must first dimension the array to a sufficient size, and also set aside space in each string to hold the file names. The routine FCount is specifically intended as a companion to ReadFile, and it will report the number of files that match a given file specification.

String space can also be corrupted in programs with many levels of subprogram call nesting. The BASIC CLEAR and STACK commands are needed to reserve additional stack space, and this is discussed in the section entitled "Sorts vs. Indexed Sorts."

Chapter 2
Array Routines

AddInt

assembler subroutine contained in PRO.LIB

Purpose:

AddInt adds a constant value to all of the elements in a specified portion of an integer array.

Syntax:

```
CALL AddInt(SEG Array%(Start), Value%, NumEls%)
```

Where:

Array%(Start) is the first element in the array that is to be modified, Value% is the value to be added to each element, and NumEls% is the number of elements to be processed.

Comments:

AddInt is useful in conjunction with the various indexed sort routines. Because the indexed sorts have no way to determine which portion of an array is actually being sorted, the index array is always returned holding values ranging from zero to the number of elements minus one. AddInt allows you to quickly adjust the sorted index values to reflect the correct element numbers in the primary array.

Values may also be subtracted from each element by simply using a negative value.

AddInt is shown at work in the sort demonstration ISORTSTR.BAS.

DeleteStr

assembler subroutine contained in PRO.LIB

Purpose:

DeleteStr removes an element from a normal (not fixed-length) string array.

Syntax:

```
CALL DeleteStr(BYVAL VARPTR(Array$(Element)), NumEls%)
```

Where:

Array$(Element) is the element to be deleted, and NumEls% is the number of elements that follow.

Comments:

DeleteStr is functionally equivalent to a FOR/NEXT loop that copies elements downward, but it operates much faster than would be possible using BASIC alone. An assembler routine cannot actually create or assign strings, therefore the deletion is handled internally by swapping. The algorithm used is:

```
FOR X = Element TO Element + NumEls%
    SWAP Array$(X), Array$(X + 1)
NEXT
```

Because the deletion is performed by swapping, the deleted string ends up in the last element of the array. You can easily erase the last element like this:

```
Array$(Element + NumEls%) = ""
```

A complete example of DeleteStr (and its companion program InsertStr) is contained in the file INSERT.BAS.

DeleteT

assembler subroutine contained in PRO.LIB

Purpose:

DeleteT removes an element from a fixed-length string, numeric, or user-defined TYPE array.

Syntax:

```
CALL DeleteT(SEG Array(Element), ElSize%, NumEls%)
```

For fixed length strings, see "Calling with Segments" in the Tutorial section of this manual.

Where:

Array(Element) is the element to be deleted, ElSize% is the size of each array element (or a special code that is described below), and NumEls% is the number of elements that follow.

Comments:

DeleteT is functionally equivalent to a FOR/NEXT loop that copies elements downward, but it operates much faster than would be possible using BASIC alone. Unlike the Delete routine meant for string arrays, DeleteT performs a direct memory move copying as many bytes as needed to effect the deletion. Because memory is simply copied downward into lower elements, the last element is still present in the array. You can easily clear it if needed like this:

```
Array(Element + NumEls%) = 0      'numeric element
Array$(Element + NumEls%) = ""    'fixed-length string
```

The element size indicates how many bytes each element occupies. For example, you would use 2 for an integer array, and 8 for a double precision. DeleteT will also accept the codes used by the TYPE sorts to indicate the element size:

-1 = integer
-2 = long integer
-3 = single precision
-4 = double precision
-5 = currency (BASIC PDS only)
+n = fixed-length string or TYPE with a size of n bytes

DimBits

BASIC subroutine contained in the file BITS.BAS

Purpose:

DimBits creates a BASIC string that will be used to hold a Bit array.

Syntax:

```
CALL DimBits(Array$, NumEls%)
```

Where:

Array$ is returned set to the length needed to hold NumEls% bit elements, and it is initialized to all zeros.

Comments:

Bit arrays are useful when a large table of True/False information must be maintained in as little memory as possible. Where a conventional integer array requires two bytes to hold even a single bit of information, a Bit array occupies only the amount of memory actually needed.

Once a Bit array has been dimensioned, individual elements are read and assigned using GetBit and SetBit respectively. These routines are described elsewhere in this manual. Bit arrays are also discussed at length in the section of this manual entitled "Bit Arrays".

DimBits is a very simple routine, and may be entered into your program directly. The actual code is:

```
Array$ = STRING$(NumEls \ 8 + 1, 0)
```

A short example of the Bit array routines is contained in the file BITS.BAS.

Fill2, 4, 8

assembler subroutines contained in PRO.LIB

Purpose:

Fill2 will quickly assign all of the elements in a specified portion of an integer array to any value.

Syntax:

```
CALL Fill2(SEG Array%(Start), Value%, NumEls%)
```

Where:

Array%(Start) is the first element to be assigned, Value% is the value the elements are to receive, and NumEls% is the number of elements to be processed.

Comments:

Fill4 and Fill8 are called exactly the same as Fill2, but are set up to process single precision (or long integer) and double precision arrays respectively. All of the passed parameters have the same meaning as with Fill2, though the appropriate Array and Value variables should be substituted.

To initialize *any* array to all zeros (or null strings), the best approach is to simply redimension it:

```
REDIM Array(NumEls)
```

All of the fill routines are demonstrated in the program FILL.BAS.

Find and Find2

assembler subroutines contained in PRO.LIB

Purpose:

Find will search all or part of a conventional (not fixed-length) string array forward looking for the first occurrence of a given string or sub-string.

Syntax:

```
CALL Find(BYVAL VARPTR(Array$(Start)), NumEls%, Search$)
```

Where:

Array$(Start) is the element at which searching is to begin, NumEls% is the number of successive elements to be examined, and Search$ is the string or sub-string to find.

NumEls% is returned holding the number of elements that were searched before a match was found, or -1 if there were no matching elements. If the first element searched matches, NumEls% will return holding 0.

Comments:

Find is set up to honor capitalization, and Find2 will ignore it. That is, Find2 will find "abcde" within the array element "ABCDE", while Find would not. One or more question marks (?) may also be used in the search string as a wild card to match any character. The DOS "*" wild card is implied, and should not be used.

Because Find2 compares strings regardless of capitalization, the first thing it does is to permanently capitalize the search string. In designing Find2 we would have preferred to not tamper with the search string, however that would have either decreased its performance, or required more string memory to be taken from your program to hold a temporary working copy.

If it is essential that Search$ not be modified, simply pass a copy of Search$ as shown below:

```
CALL Find2(BYVAL VARPTR(Array$(Start)), NumEls%,UCASE$(Search$)
```

When UCASE$ (or LCASE$) is used within a calling argument, BASIC will first copy the string into a temporary location, and then capitalize the copy. Thus, no matter what Find2 does to the search string, only the temporary copy will be affected.

Find and Find2 are amply demonstrated in the program FIND.BAS, which also shows how to continue searching an array.

FindB and FindB2

assembler subroutines contained in PRO.LIB

Purpose:

FindB will search all or part of a conventional (not fixed-length) string array for a string or sub-string. Like Find, FindB also accepts wild cards and is provided in both case-sensitive and case-insensitive versions. However, FindB and FindB2 search the string array *backwards*.

Syntax:

```
CALL FindB(BYVAL VARPTR(Array$(CurEl%)), CurEl%, Search$)
```

Where:

Array$(CurEl%) is the current array element at which searching is to begin, CurEl% is its element number, and Search$ is the string or sub-string to find.

CurEl% is returned holding the actual number of the element in which the match was found, or -1 if there were no matching elements. However, this assumes that the array begins at element zero.

Comments:

All of the comments for Find and Find2 apply to FindB and FindB2, except that continuing an array search is handled differently.

A complete demonstration for using all four of the Find string array routines is contained in the FIND.BAS example program.

FindExact

assembler subroutine contained in PRO.LIB

Purpose:

FindExact searches an entire conventional (not fixed-length) string array for an exact match.

Syntax:

```
CALL FindExact(BYVAL VARPTR(Array$(Start)), NumEls%, Search$)
```

Where:

Array$(Start) is the element at which searching is to begin, NumEls% is the total number of elements to examine, and Search$ is the string to search for. NumEls% returns holding the number of elements it searched before finding a match, with 0 meaning it found it on the first one. If no match is found NumEls% will instead hold -1.

Comments

Unlike Find, FindB, and the other "Find" routines, this one searches for an exact match only. Since capitalization is honored and wild cards are not recognized, this routine is much faster than the other versions.

FindT and FindT2

assembler subroutines contained in PRO.LIB

Purpose:

FindT will search all or part of a fixed-length string array for any string or sub-string.

Syntax:

```
CALL FindT(BYVAL VARSEG Array$(Start), BYVAL VARPTR Array$(Start), _
     ElSize%, NumEls%, Search$)
```

Where:

Array(Start) is the first element to begin searching, ElSize% is the length of each element, NumEls% is the number of successive elements to examine, and Search$ is the string or sub-string to find.

NumEls% is returned holding the number of elements that were searched before a match was found, or -1 if there were no matching elements. If the first element searched matches, NumEls% will return holding 0.

Comments:

FindT is set up to honor capitalization, and FindT2 will ignore it. That is, FindT2 will find "abcde" within the array element "ABCDE", while FindT will not. One or more question marks (?) may also be used in the search string as a wild card to match any character. The DOS "*" wild card is implied, and should not be used.

Because FindT2 compares strings regardless of capitalization, the first thing it does is to permanently capitalize the search string. In designing FindT2 we would have preferred to not tamper with the search string, however that would have either decreased its performance, or required more string memory to be taken from your program to hold a temporary working copy.

If it is essential that Search$ not be modified, simply pass a copy of Search$ as shown below:

```
CALL FindT2(SEG Array(Start), ElSize%, NumEls%,UCASE$(Search$)
CALL FindT2(SEG Array(Start), ElSize%, NumEls%,UCASE$(Search$)
```

When UCASE$ (or LCASE$) is used within a calling argument, BASIC will first copy the string into a temporary location, and then capitalize the copy. Thus, no matter what FindT2 does to the search string, only the temporary copy will be affected.

Notice that Search$ may be either a conventional or fixed-length string. More information about how BASIC passes strings to a subroutine may be found in The Assembly Tutor under the section entitled "Fixed-Length Strings".

FindT and FindT2 are demonstrated in the program FINDT.BAS, which also shows how to continue searching an array.

FindTB and FindTB2

assembler subroutines contained in PRO.LIB

Purpose:

FindTB will search all or part of a fixed-length string array for a string or sub-string. Like FindT, FindTB also accepts wild cards and is provided in both case-sensitive and case-insensitive versions. However, FindTB and FindTB2 search the string array *backwards*.

Syntax:

```
CALL FindTB(BYVAL VARSEG Array$(CurEl%), BYVAL VARPTR _
     Array$(CurEl%), ElSize%, CurEl%, Search$)
```

Where:

Array(CurEl%) is the current array element at which to begin searching, CurEl% is its element number, and Search$ is the string or sub-string to find.

CurEl% is returned holding the actual number of the element in which the match was found, or -1 if there were no matching elements. However, this assumes that the array begins at element zero.

Comments:

All of the comments for FindT and FindT2 apply to FindTB and FindTB2, except that continuing an array search is handled differently.

A complete demonstration for using all four of the FindT string array routines is contained in the FINDT.BAS example program.

FindLast

assembler function contained in PRO.LIB

Purpose:

FindLast scans a conventional (not fixed-length) string array backwards looking for the last non-blank element.

Syntax:

```
NumEls% = UBOUND(Array$)
Last = FindLast%(BYVAL VARPTR(Array$(NumEls%)), NumEls%)
```

Where:

NumEls% is the number of elements to which Array$ has been dimensioned, and Last receives the number of the last element that is not null.

Comments:

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

FindLast is useful when writing routines that need to know how many elements in a string array are actually active, even if the array might have been dimensioned to a larger value.

Though UBOUND will report how large the entire array is, in many cases what you really want to know is what portion of the array contains useful information. For example, in the two Lotus-style menu programs provided with QuickPak Professional, it would be inappropriate to display empty menu choices if the array had inadvertently been dimensioned to more elements than were assigned.

Likewise, the QEdit full-screen editor needs to know how many lines of text are currently in the array, even though the array was probably dimensioned to a much larger value.

GetBit

assembler function contained in PRO.LIB

Purpose:

GetBit will report the status (on or off) of an element in a QuickPak Professional Bit array.

Syntax:

```
Bit = GetBit%(Array$, Element%)
```

Where:

Array$ is a string that was previously set up to hold a Bit array, Element% is the desired element number to retrieve, and Bit receives either a 0 if the element is clear, or -1 if it is set.

Comments:

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

GetBit is set up to return -1 if the element is set rather than a normal 1. This allows the use of the BASIC NOT operator, and eliminates the requirement for an explicit comparison against a particular value. For example, to see if a bit is set, you could use the following:

```
IF GetBit%(Array$, 1499) THEN . . .
```

And to determine if the bit is clear, you could instead use:

```
IF NOT GetBit%(Array$, 1499) THEN . . .
```

Using this approach also eliminates having to perform an explicit comparison to obtain the bit status:

```
IF GetBit%(Array$, 20998) = 0 THEN . . .
```

Bit arrays are discussed in depth separately in this manual under the heading “Bit Arrays”.

IMaxD, I, L, S, C

assembler functions contained in PRO.LIB

Purpose:

IMaxD will search through an entire double precision array, and return the element number of the largest value. The remaining functions are designed to operate on integer, long integer, and single precision arrays respectively. IMaxC is for use with the Currency data type offered in BASIC PDS.

Syntax:

```
Element = IMaxD%(SEG Array#(Start), NumEls%)
```

Where:

Array#(Start) is the first element to consider when searching the array, NumEls% is the total number of elements to search, and Element receives the element number that holds the largest value.

Comments:

Because these routines have designed as functions, they must be declared before they may be used. Unlike the original Max? functions that directly return the value of the largest element, these functions return an element number and are thus designed as integer functions. If you need both the element number and the value, using only the IMax function will give you both and avoid adding redundant code to your program.

The element number returned assumes a zero-based array, and further, that the entire array is being searched. That is, if the largest value is found at the first element, the function will return zero. To accommodate varying start elements simply add the first element number to the answer returned:

```
Element = IMaxD%(SEG Array#(10),NumEls%)'start at element 10
Element = Element + 10               'so add 10 here
```

This is shown in context in the IMINMAX.BAS example program.

IMinD, I, L, S, C

assembler functions contained in PRO.LIB

Purpose:

IMinD will search through an entire double precision array, and return the element number of the smallest value. The remaining functions are designed to operate on integer, long integer, and single precision arrays respectively. IMinC is for use with the Currency data type offered in BASIC PDS.

Syntax:

```
Element = IMinD%(SEG Array#(Start), NumEls%)
```

Where:

Array#(Start) is the first element to consider when searching the array, NumEls% is the total number of elements to search, and Element receives the element number that holds the lowest value.

Comments:

Because these routines have designed as functions, they must be declared before they may be used.Unlike the original Min? functions that directly return the value of the smallest element, these functions return an element number and are thus designed as integer functions. If you need both the element number and the value, using only the IMin function will give you both and avoid adding redundant code to your program.

The element number returned assumes a zero-based array, and further, that the entire array is being searched. That is, if the smallest value is found at the first element, the function will return zero. To accommodate varying start elements simply add the first element number to the answer returned:

```
Element = IMinD%(SEG Array#(10), NumEls%)'start at element 10
Element = Element + 10              'therefore add 10 here
```

This is shown in context in the IMINMAX.BAS example program.

InitInt

assembler subroutine contained in PRO.LIB

Purpose:

InitInt will quickly initialize all or a specified portion of an integer array with increasing values.

Syntax:

```
CALL InitInt(SEG Array%(Start), Value%, NumEls%)
```

Where:

Array%(Start) is assigned Value%, Array%(Start + 1) is assigned Value% + 1, and so forth, and NumEls% is the total number of elements to assign.

Comments:

InitInt is required to initialize the integer index array that will be used with any of the indexed sorting routines. Though you could certainly initialize an array with a FOR/NEXT loop, this dedicated routine will do it much more quickly.

For more information regarding InitInt and the QuickPak Professional indexed sort routines, see the section entitled "Sorts vs. Indexed Sorts" elsewhere in this manual.

InsertStr

assembler subroutine contained in PRO.LIB

Purpose:

InsertStr will insert an element at any point into a conventional (not fixed-length) string array.

Syntax:

```
CALL InsertStr(BYVAL VARPTR(Array$(Element)), Ins$, NumEls%)
```

Where:

Ins$ will be inserted into the array at the element specified by Element, and NumEls% is the total number of elements that follow.

Comments:

InsertStr is functionally equivalent to a FOR/NEXT loop that copies elements upward, but it operates much faster than would be possible using BASIC alone. An assembler routine cannot actually create or assign strings, therefore the insertion is handled internally by swapping. The algorithm used is:

```
FOR X = (Element + NumEls%) TO (Element + 1) STEP -1
    SWAP Array$(X), Array$(X - 1)
NEXT
SWAP Array$(Start), Ins$
```

Because the insertion is performed by swapping, the string that had originally been in Array$(Element) ends up in Ins$ when the routine has finished.

It is up to you to insure that the array has been sufficiently dimensioned to accommodate the inserted element. Specifying more elements than actually exist is certain to cause a crash. However, the elements that follow Array$(Element) may be null.

InsertT

assembler subroutine contained in PRO.LIB

Purpose:

InsertT inserts an element into a fixed-length string, numeric, or user-defined TYPE array.

Syntax:

```
CALL InsertT(SEG Array(Element), ElSize%, NumEls%)
```

For fixed length strings, see "Calling with Segments" in the Tutorial section of this manual.

Where:

Array(Element) is the point at which the insertion is to take place, ElSize% is the size of each array element in bytes (or a special code that is described below), and NumEls% is the number of elements that follow.

Comments:

InsertT is functionally equivalent to a FOR/NEXT loop that copies elements upward, but it operates much faster than would be possible using BASIC alone. Unlike the Insert routine meant for string arrays, InsertT performs a direct memory move copying as many bytes as needed to effect the insertion.

Because memory is simply copied upward into higher elements, the current element is still present in the array. Also, unlike the Insert routine for conventional string arrays, you do not specify an element to be inserted. However inserting the element would then be as simple as this:

```
Array(Element) = NewElement
```

The element size indicates how many bytes each element occupies. For example, you would use 2 for an integer array, and 4 for a long integer. InsertT will also accept the same code as the TYPE sort routines to indicate the element size.

ISortD, I, L, S, C

five assembler subroutines contained in PRO.LIB

Purpose:

The five ISort routines are assembler Quick Sorts for ordering all or a portion of a numeric array. Each is intended to sort a parallel INDEX array into either ascending or descending order.

Syntax:

```
CALL ISort?(SEG Array(Start), SEG Ndx%(0), NumEls%, Dir%)
```

Where:

ISort? is the appropriate indexed sort routine.

Array(Start) is the first element to include in the sort, and Ndx%(0) is the starting element in the integer index array. NumEls% is the number of elements to consider, and Dir% indicates the sort direction. If Dir% is set to zero, then the sorting will be forward (ascending). Any other value will cause sorting to be performed backward (descending).

Comments:

Five separate routines are provided to allow you to add only the sort capabilities you need to your program. However, the indexed TYPE sort routine (ISortT) is capable of sorting *all* of the variable types, and you might consider using that if your program will need to sort more than one type of variable. All of these sorts are intended for IEEE numbers only, and will not work the optional non-8087 /fpa math BASIC 7 offers.

It is extremely important that the integer index array be dimensioned to at least the same size as the portion of the primary array being sorted. It must also be initialized to increasing values prior to calling the indexed sort routines. That is, Ndx%(0) must contain a zero, Ndx%(1) will hold a one, and so forth. The routine InitInt is specifically intended for this purpose.

A complete description of these sorts, along with a comparison of indexed vs. normal sorting, is given elsewhere in this manual. Please see the section "Sorts vs. Indexed Sorts".

ISortStr and ISortStr2

assembler subroutines contained in PRO.LIB

Purpose:

ISortStr will sort all or part of a conventional (not fixed-length) string array into either ascending or descending order by manipulating elements in a parallel integer index array. ISortStr2 is nearly identical, but sorting is performed without regard to capitalization.

Syntax:

```
CALL ISortStr(BYVAL VARPTR(Array$(Start)), SEG Ndx%(0), _
    NumEls%, Dir%)
```

Where:

Array$(Start) is the first element to include in the sort, and Ndx%(0) is the starting element in the integer index array. NumEls% is the number of elements to consider, and Dir% indicates the sort direction. If Dir% is set to zero, then the sorting will be forward (ascending). Any other value will cause sorting to be performed backward (descending).

Comments:

It is extremely important to dimension the integer index array to at least the same size as the primary array being sorted. Also, initialize it to increasing values prior to calling the indexed sort routines. That is, Ndx%(0) must contain a zero, Ndx%(1) will hold a one, and so forth. The routine InitInt is specifically intended for this purpose.

A complete working example of ISortStr is shown in context in the ISORTSTR.BAS demonstration program.

Also, a complete description of these sorts, along with a comparison of indexed vs. normal sorting, is given elsewhere in this manual. Please see the section "Sorts vs. Indexed Sorts".

ISortT and ISortT2

assembler subroutines contained in PRO.LIB

Purpose:

ISortT will sort all or part of a fixed-length string or TYPE array into either ascending or descending order by manipulating elements in a parallel integer index array. ISortT2 is nearly identical, but when considering the string component of a TYPE array, sorting is performed without regard to capitalization.

Syntax:

```
CALL ISortT(SEG Array(Start), SEG Ndx%(0), NumEls%, Dir%, _
     ElSize%, MemberOffset%, MemberSize%)
```

For fixed length strings, see "Calling with Segments" in the Tutorial section of this manual.

Where:

Array(Start) is the first element to include in the sort, and Ndx%(0) is the starting element in the integer index array. NumEls% is the number of elements to consider, and Dir% indicates the sort direction. If Dir% is set to zero, then the sorting will be forward (ascending). Any other value will cause sorting to be performed backward (descending).

ElSize% is the length of each TYPE array element, MemberOffset% is the number of bytes into the TYPE where the key member being considered for the sort is located, and MemberSize% is its length. MemberSize% is coded using the convention described in the Comments section below.

If the array being sorted is simply a fixed-length string array, then MemberSize% and ElSize% will be the same, and MemberOffset% will be zero.

Comments:

A special code is used to indicate the type of variable that is being operated upon. For example, sorting based on the string portion of a TYPE simply involves comparing the ASCII values for each character. However, sorting on a double precision component of a TYPE array requires a very different approach.

The type of variable being considered as the key is indicated to ISortT and ISortT2 as follows:

```
 -1 = integer
 -2 = long integer
 -3 = single precision
 -4 = double precision
 -5 = currency (BASIC PDS only)
 +n = fixed-length string with a length of n bytes
```

Comments in the description of the ISort routines for numeric arrays contain important information about preparing the integer index array. Also, a complete working example of ISortT is presented in context in the ISORTT.BAS demonstration program.

KeySort

assembler subroutine contained in PRO.LIB

Purpose:

KeySort is a recursive subprogram that will sort a user-defined TYPE array based on any number of keys. Each key may be sorted independently, ascending or descending.

Syntax:

```
CALL KeySort(SEG Array(Start), ElSize%, NumEls%,_
    SEG Table%(1, 1), NumKeys%)
```

Where:

Array(Start) is the first element to consider in the TYPE array, ElSize% is the length in bytes of each element, and NumEls% is the total number of elements to be included. Table%() is a 2-dimensional array which indicates how the primary array is to be sorted (see below), and NumKeys% is the number of sort keys.

Comments:

In order for KeySort to operate correctly, the Table%() array must be properly prepared. The size of the first subscript should be dimensioned to the total number of sort keys, and the second subscript must be dimensioned to 3. For example, to sort a TYPE array on two keys you would dimension Table%() like this:

```
DIM Table%(1 TO 2, 1 TO 3))
```

In the second subscript, the first element, Table%(n, 1), tells how many bytes into the structure the current key is located. The second element, Table%(n, 2), holds the size in bytes of the data, using the codes shown below. The third element
(n, 3) is either zero to sort ascending, or anything else to sort descending.

It is important that the table array be dimensioned to the exact size needed. If the main array is to be sorted on, say, 4 keys, then Table%() must be dimensioned to only that size:

```
DIM Table%(1 TO 4, 1 TO 3)).
```

Because KeySort calls upon ISortT to do much of the real work, the data type codes are the same as those used by ISortT:

-1 = 2-byte integer
-2 = 4-byte long integer
-3 = 4-byte single precision
-4 = 8-byte double precision
-5 = currency (BASIC PDS only)
+n = n-byte fixed-length string

The code fragment on the following page was taken from the KEYSORT.BAS example program. It shows how the TYPE and table arrays would be set up for sorting on three keys.

```
TYPE Test
   I AS INTEGER
   L AS LONG
   S AS SINGLE
   D AS DOUBLE
   X AS STRING * 20
END TYPE

NumberOfKeys = 3                  'the total number of sort keys
DIM SHARED Table(1 TO NumberOfKeys, 1 TO 3)

'Table is a 2-dimensional table of sort information constructed 'as
follows:

'Element 1,1 is the offset into the element for the primary key
'Element 1,2 is a code for the type of data being considered
'Element 1,3 is 0 or 1 for ascending or descending respectively
'Element 2,1 is the offset into the element for the secondary
'   key, and so forth, through the last sort key

FOR X = 1 TO NumberOfKeys         'read offsets, type codes, and
   FOR Y = 1 TO 3                 'directions
        READ Table(X, Y)
   NEXT
NEXT

DATA 18, 20, 0                    'The primary sort is based on
                                  'the fixed-length string, which
                                  'starts 18 bytes into the
                                  'structure. The string length
                                  'is 20, and we want to do an
                                  'ascending sort.

DATA 0, -1, 1                     'The integer is the second key,
                                  'its length code is -1, sort
                                  'descending.

DATA 10, -4, 0                    'The double precision part is
                                  'the third key, its length code
                                  'is -4, sort ascending.
```

MaxD, I, L, S, C

five assembler functions contained in PRO.LIB

Purpose:

MaxD, MaxI, MaxL, and MaxS will return the largest value in a specified portion of a numeric array. Five separate functions are provided, with one each intended for double precision, integer, long integer, and single precision arrays. MaxC is for use with the Currency data type offered in BASIC PDS.

Syntax:

```
Max = Max?(SEG Array(Start), NumEls%)
```

Where:

Max? is either MaxD, MaxI, MaxL, MaxS, or MaxC.

Array(Start) is the first element in the array to be considered, NumEls% is the number of elements to examine, and Max receives the value of the largest element. Of course, both the Max variable and the numeric array must be the appropriate type, based on which Max routine is being used.

Comments:

Because these have been designed as functions, they must be declared before they may be used.

The Max functions that operate on floating point arrays expect the numbers to be in the IEEE format used by QuickBASIC 4.x or later.

Because these are functions, the correct type identifier must be used when they are declared. For example, MaxL would be declared with an ampersand like this:

```
DECLARE FUNCTION MaxL&(SEG Element&, NumEls%)
```

Also see the companion routines MinD, MinI, MinL, MinS and MinC.

MinD, I, L, S, C

five assembler functions contained in PRO.LIB

Purpose:

MinD, MinI, MinL, and MinS will return the smallest value in a specified portion of a numeric array. Four separate functions are provided, with one each intended for double precision, integer, long integer, and single precision arrays. MinC is for use with the Currency data type offered in BASIC PDS.

Syntax:

```
Min = Min?(SEG Array(Start), NumEls%)
```

Where:

Min? is either MinD, MinI, MinL, MinS, or MinC.

Array(Start) is the first element in the array to be considered, NumEls% is the number of elements to examine, and Min receives the value of the smallest element. Of course, both the Min variable and the numeric array must be the appropriate type, based on which Min routine is being used.

Comments:

Because these have been designed as functions, they must be declared before they may be used.

The Min functions that operate on floating point arrays expect the numbers to be in the IEEE format used by QuickBASIC 4.x or later.

Because these are functions, the correct type identifier must be used when they are declared. For example, MinL would be declared with an ampersand like this:

```
DECLARE FUNCTION MinL&(SEG Element&, NumEls%)
```

Also see the companion routines MaxD, MaxI, MaxL, MaxS and MaxC.

Search

assembler subroutine contained in PRO.LIB

Purpose:

Search will scan all or part of a numeric array to find the first element that matches a given value. Searching may be performed either forward or backward through the array, and the match may be specified to be exact, less or equal, or greater or equal.

Syntax:

```
CALL Search(SEG Array(Start), NumEls%, Match, Found%, _
     Direction%, MatchCode%, VarType%)
```

Where:

Array(Start) is the first array element to include in the search, and may be of any variable type. NumEls% is the total number of elements to search, Match is the value to compare against, and it too may be any type of variable.

Found% then returns a relative displacement (in elements) to the first element that matches. Direction% is set to 0 to search forward, or anything else to search backward.

MatchCode% indicates the type of search as follows:

0 =	find an exact match only
1 =	match if the element is greater or equal
-1 =	match if the element is less or equal

VarType% is coded following the same convention used by the TYPE sort routines as follows:

-1 =	2-byte integer
-2 =	4-byte long integer
-3 =	4-byte single precision
-4 =	8-byte double precision
-5 =	currency (BASIC PDS only)

Comments:

When Search is used with floating point arrays, it expects the numbers to be in the IEEE format used by QuickBASIC 4.x or later.

Although the array and the match variables may be any numeric type, they both must, of course, be of the *same* type.

A complete working example of using Search in context is shown in the SEARCH.BAS demonstration program. Besides showing the correct syntax, SEARCH.BAS also illustrates how to calculate the found element number based on where Search started, and whether it was scanned forward or backward.

SearchT and SearchT2

assembler subroutines contained in PRO.LIB

Purpose:

SearchT and SearchT2 will search an entire TYPE array for a match on any single TYPE member. When used with the string portion of a TYPE, SearchT honors capitalization, while SearchT2 ignores it.

Syntax:

```
CALL SearchT(SEG Array(Start), NumEls%, Match, Found%, Dir%, _
     Code%, StructSize%, MemberOff%, MemberSize%)
```

Where:

Array (Start) is the first TYPE array element to include in the search.

NumEls% is the number of elements to search.

Match is the value to compare against, and it must be of the same data type as the TYPE member being examined.

Found% is the number of elements searched before the match was found.

Dir% is 0 for searching forward, or anything else for backward.

Code% indicates the type of search, using the following code:
0 = exact match
1 = greater or equal
-1 = less or equal

StructSize% is the total size of the TYPE structure in bytes.

MemberOff% is the offset into the structure for the member being sought.

MemberSize% is coded as follows:

-1 =	2-byte integer
-2 =	4-byte long integer
-3 =	4-byte single precision
-4 =	8-byte double precision
-5 =	8-byte currency type (BASIC 7 version only)
+n =	the length of the string portion being searched

Comments:

SearchT and SearchT2 are modeled after the Search routine. Where Search is limited to a conventional numeric array, SearchT lets you examine just a specified portion of a TYPE array.

Note that the Code% parameter is ignored when searching for strings, and defaults to 0 (find an exact match).

SetBit

assembler subroutine contained in PRO.LIB

Purpose:

SetBit will set an element in a QuickPak Professional Bit array to either one or zero (true or false).

Syntax:

```
CALL SetBit%(Array$, Element%, Bit%)
```

Where:

Array$ is a string that was previously set up to hold the Bit array, Element% is the desired element number to set, and Bit% holds either a 0 if the element is to be cleared, or anything else if it is to be set.

Comments:

SetBit is provided as a companion to DimBits and GetBit. Bit arrays are discussed in depth separately in this manual under the heading "Bit Arrays".

SortD, I, L, S, C

five assembler subroutines contained in PRO.LIB

Purpose:

The five Sort routines are assembler Quick Sorts for ordering all or a portion of a numeric array.

Syntax:

```
CALL Sort?(SEG Array(Start), NumEls%, Dir%)
```

Where:

Sort? is the appropriate sort routine.

Array(Start) is the first element to be included in the sort, NumEls% is the number of elements to consider, and Dir% indicates the sort direction. If Dir% is set to zero, then the sorting will be forward (ascending). Any other value will cause sorting to be performed backward (descending).

Comments:

Five separate routines are provided to allow you to add only the sort capabilities you need to your program. However, the TYPE sort routine (SortT) is capable of sorting *all* of the variable types, and you might consider using that if your program will need to sort more than one type of variable.

Theses sort routines assume the use of the IEEE format for floating point variables.

A complete description of these sorts, along with a comparison of indexed vs. normal sorting, is given elsewhere in this manual. Please see the section "Sorts vs. Indexed Sorts".

SortStr and SortStr2

assembler subroutines contained in PRO.LIB

Purpose:

SortStr will sort all or just a portion of a conventional (not fixed-length) string array into either ascending or descending order. SortStr2 is nearly identical, but sorting is performed without regard to capitalization.

Syntax:

```
CALL SortStr(BYVAL VARPTR(Array$(Start)), NumEls%, Dir%)
```

Where:

Array$(Start) is the first element to be included in the sort, NumEls% is the number of elements to consider, and Dir% indicates the sort direction. If Dir% is set to zero, then the sorting will be forward (ascending). Any other value will cause sorting to be performed backward (descending).

Comments:

A complete working example of SortStr is shown in context in the SORTSTR.BAS demonstration program.

Also, a description of these sorts, along with a comparison of indexed vs. normal sorting, is given elsewhere in this manual. Please see the section “Sorts vs. Indexed Sorts”.

SortT and SortT2

assembler subroutines contained in PRO.LIB

Purpose:

SortT will sort all or part of a fixed-length string or TYPE array into either ascending or descending order. SortT2 is nearly identical, but when considering the string component of a TYPE array, sorting is performed without regard to capitalization.

Syntax:

```
CALL SortT(SEG Array(Start), NumEls%, Dir%, ElSize%, _
     MemberOffset%, MemberSize%)
```

For fixed length strings, see "Calling with Segments" in the Tutorial section of this manual.

Where:

Array(Start) is the first element to be included in the sort, NumEls% is the number of elements to consider, and Dir% indicates the sort direction. If Dir% is set to zero, then the sorting will be forward (ascending). Any other value will cause sorting to be performed backward (descending).

ElSize% is the length of each TYPE element, MemberOffset% is the number of bytes into the TYPE where the key member being considered for the sort is located, and MemberSize% is its length. MemberSize% is coded using the convention described in the Comments section below.

If the array being sorted is simply a fixed-length string array, then MemberSize% and ElSize% will be the same, and MemberOffset% will be zero.

Comments:

A special code is used to indicate the type of variable that is being operated upon. For example, sorting based on the string portion of a TYPE simply involves comparing the ASCII values for each character. However, sorting on a double precision component of a TYPE array requires a very different approach.

The type of variable being considered as the key is indicated to SortT and SortT2 as follows:

-1 =	integer
-2 =	long integer
-3 =	single precision
-4 =	double precision
-5 =	currency (BASIC PDS only)
+n =	fixed-length string with a length of n bytes

Also, a complete working example of SortT is presented in context in the SORTT.BAS demonstration program.

Chapter 3
DOS Services

CDir

assembler subroutine contained in PRO.LIB

Purpose:

CDir serves the same purpose as BASIC's CHDIR command, but it returns a code to indicate an error rather than requiring a separate error trapping procedure.

Syntax:

```
CALL CDir(NewDir$)
```

Where:

NewDir$ is a string containing the new directory to change to.

Comments:

As with DOS, a new directory may be requested for either the current drive or any other legal drive in the system. For example, to change the directory on drive B when the current drive is C would be done like this:

```
CALL CDir("B:\NEWDIR")
```

Only two errors are likely when using CDir—either the specified directory does not exist, or an invalid drive was given. Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

ClipFile

assembler subroutine contained in PRO.LIB

Purpose:

ClipFile will establish a new length for the specified file, and the new length may be either shorter or longer than the number of bytes the file presently occupies.

Syntax:

```
CALL ClipFile(FileName$, NewLength&)
```

Where:

FileName$ is the name of the file to process, and NewLength& is a long integer holding the new length the file is to be set to.

Comments:

One important use for ClipFile is to reduce the length of a file to which data has inadvertently been written. As an example, if SEEK (or the QuickPak FSeek) is accidentally used to set the DOS pointer to a location beyond the end of a data file, the length of the file will automatically be extended. Another use for ClipFile is to truncate a database file to purge deleted records, as described below. For ClipFile to work, the file being clipped must be closed.

Many database programs reserve an extra byte in each record just to indicate whether the record has been deleted or not. Unfortunately, the only way the deleted data can then be removed from the database is to move only the active records to a new file one by one. Once the data has been copied the original file would be deleted, and the new file renamed.

However, there are several problems with that approach, the worst being that sufficient free disk space must be present to hold both the original data and the copy. A better method is to create a program to copy the deleted records to the end of the file, and then use ClipFile to set the new length to the end of the last active record. A program to accomplish this would be written much like a traditional bubble sort, except only a single pass would be required.

The only error that is likely is attempting to access a file that does not exist. Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

DCount

assembler function contained in PRO.LIB

Purpose:

DCount reports the number of directories that match a particular specification.

Syntax:

```
Count = DCount%(DirSpec$)
```

Where:

DirSpec$ holds a DOS directory name specification, and Count receives the number of matching directories.

Comments:

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

DCount is intended to provide a count of the directories on a disk, in preparation for using ReadDir to obtain a list of all their names.

Where FCount provides a count of file names that match a given specification, DCount instead searches for directory names. It is important not to confuse these two routines.

Most people think of the DOS wild cards (? and *) as being applicable only to file names, however they are also intended to be used with directory names. For example, to use DCount to determine the number of directories that are under the root directory, you would specify a search criteria such as "*.*" when using DCount.

DiskInfo

assembler subroutine contained in PRO.LIB

Purpose:

DiskInfo calls on DOS to examine a disk, and reports its sector and cluster makeup.

Syntax:

```
CALL DiskInfo(Drive$, Bytes%, Sectors%, FreeClusters&, TotalClusters&)
```

Where:

Drive$ contains the letter of the drive to examine, and Bytes% returns how many bytes each sector holds. Sectors% contains the number of sectors in each disk cluster, FreeClusters& tells how many of them are available, and TotalClusters& indicates the total disk capacity.

Comments:

As with all of the QuickPak Professional DOS services, Drive$ may be either upper or lower case, or a null string to indicate the default drive. Since only the first character of Drive$ is examined, you may also pass a complete file name to DiskInfo if that is more convenient:

```
Drive$ = "C:WHATEVER.XYZ"
```

The number of Bytes per sector will always be 512 under DOS, though the number of sectors per cluster will vary depending on the type of disk being examined. For example, a 360K floppy disk always stores two sectors within each cluster, while a 1.2 MB floppy allocates only one sector per cluster. Thus, a program can quickly and easily determine what type of disk drive it is dealing with.

The amount of free space on a disk may easily be calculated by multiplying the various components that DiskInfo returns.

Also see the DiskRoom and DiskSize functions.

DiskRoom

assembler function contained in PRO.LIB

Purpose:

DiskRoom returns the number of bytes that are currently available on a specified disk drive.

Syntax:

```
Room = DiskRoom&(Drive$)
```

Where:

Drive$ contains either an upper or lower case letter for the disk drive being examined, or is null to indicate the default drive. Room is assigned the number of available bytes.

Comments:

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

Also see the DiskSize function which returns a disk's total storage capacity.

DiskSize

assembler function contained in PRO.LIB

Purpose:

DiskSize returns the total capacity in bytes of a specified disk drive.

Syntax:

```
Size = DiskSize&(Drive$)
```

Where:

Drive$ contains an upper or lower case letter for the disk drive being examined, or is null to indicate the default drive. Size is assigned the total number of bytes the disk can hold.

Comments:

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

Also see the DiskRoom function which returns the amount of free space remaining on a disk.

DOSError

assembler function contained in PRO.LIB

Purpose:

DOSError reports if an error occurred during the last call to a QuickPak Professional DOS routine.

Syntax:

```
IF DOSError% THEN PRINT "A DOS error occurred"
```

Where:

DOSError% returns 0 if there was no error, or -1 if there was.

Comments:

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

All of the QuickPak Professional routines assign a value to the DOSError and WhichError functions to indicate their success or failure. Rather than requiring you to set up a separate error handling procedure and use ON ERROR, you can simply query these functions after performing any QuickPak Professional DOS operation. DOSError is discussed in the section entitled “Eliminating ON ERROR”.

Also see the complementary function WhichError.

DOSVer

assembler function contained in PRO.LIB

Purpose:

DOSVer returns the version of DOS that is presently running on the host PC.

Syntax:

```
Version! = DOSVer% / 100
Major = DOSVer% \ 100
Minor = DOSVer% MOD 100
```

Where:

Version! is assigned the DOS version number. The major and minor version components may also be determined as shown in the example above.

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. For example, if a PC is using DOS 3.10, then the major version would be 3, and the minor version 10.

DOSVer simply combines the two into a single value, and then multiplies the result times 100. This approach is used to eliminate having to deal with floating point arithmetic in assembly language, which is a genuine pain to put it mildly.

There are a number of situations where DOSVer will come in handy. One would be if you are writing a program meant for use on a network, which of course requires DOS version 3.0 or later. Also, the QuickBASIC SHELL command does not always work correctly when a program is running under DOS version 2. Again, by knowing the DOS version you can avoid potential errors.

ErrorMsg

assembler function contained in PRO.LIB

Purpose:

ErrorMsg returns an appropriate message given any of the BASIC error numbers for a DOS service.

Syntax:

```
Message$ = ErrorMsg$(ErrorNumber%)
```

Where:

ErrorNumber% is a valid BASIC error number for a DOS operation.

Comments:

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

Regardless of how you intend to handle DOS and other errors in your programs, at some point you will probably need to print a message to indicate what went wrong. ErrorMsg provides an easy way to add the standard BASIC error messages without requiring ERROR, or the wasted string space that results from storing the messages as text constants or DATA statements.

The text for each message is kept in a table in the code segment, and is organized such that it may be easily modified or expanded. This is shown in the ErrorMsg assembler source code.

We have purposely omitted the "normal" BASIC errors such as Illegal Function Call and Overflow, though these could be added by modifying the assembler source code. However, two BASIC error messages that have been included are "Out of string space" and "Out of memory", which are used in the FastLoad and FastSave routines described elsewhere.

ExeName

assembler function contained in PRO.LIB

Purpose:

ExeName returns the full name of the currently executing program, including the drive, path, and file name. ExeName requires DOS 3.0 or higher, and returns a null string when run under DOS 2.

Syntax:

```
FullName$ = ExeName$
```

Where:

FullName$ receives a string as "C:\QB\MYPROG.EXE".

Comments:

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

There are several situations in which it is useful to know the full name of a program. The most common is when the program manages one or more data files, or maintains a file of configuration information. Also, a self-modifying program that writes directly to its own .EXE file will need to know if it has been renamed.

If the program is run from the current directory, then any necessary support files will be easily accessible. But when the program is in another directory and was found by DOS via the PATH setting, your program would have to parse and search the entire PATH to find them. And if the program was started by specifying an explicit drive or path name, then you're out of luck.

ExeName provides a simple way to determine where the currently running program resides, by searching the program's environment table for the drive, path, and file name. This information is placed there by DOS when a program is first run, thereby reducing the amount of work that ExeName must do.

If the program is located in the current directory or an explicit directory name was given when the program was started, then the drive letter and colon will be included. However, there is one situation in which ExeName will not return the drive as part of the name.

If the program is not in the current directory but was instead found by DOS via the current PATH setting, *and* the path does not include a drive specifier, then DOS will not place the drive into the program's environment.

The examples below show what ExeName will return for a program named MYPROG.EXE that is located in the \UTIL directory, when \UTIL is not the current directory.

PATH=\UTIL	" \UTIL\MYPROG.EXE"
PATH=C:\UTIL	" C:\UTIL\MYPROG.EXE"

Therefore, if your program lets the operator Shell to DOS or otherwise change the current drive, you should use the GetDrive function early in the program to obtain the complete path information. An example of this is shown below.

```
N$ = ExeName$
IF INSTR(N$, ":") = 0 THEN N$ = CHR$(GetDrive%) + ":" + N$
```

Exist

assembler function contained in PRO.LIB

Purpose:

Exist will quickly determine the presence of a file.

Syntax:

```
There = Exist%(FileName$)
```

Where:

FileName$ is the file or file specification whose presence is being determined, and There is assigned either to -1 if the file exists, or 0 if it does not.

The FileName$ parameter may optionally contain a drive letter, a directory path, and either of the DOS wild cards. For example, "B:\STUFF\*.BAS" would tell if any BASIC program files are present on drive B in the \STUFF directory.

Comments:

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

The main purpose of Exist is to prevent the error caused by attempting to open a file for input when it does not exist. Rather than having to set up an ON ERROR trap just prior to each attempt to open a file, Exist will directly tell if the file is present.

In the past, programmers have tried to avoid an error by opening a file for random access, which does not cause an error. Then the BASIC LOF function would be used to see if the file's length is zero, meaning it wasn't there. The problem with that approach—besides being a lot of extra work—is that an empty file will be created in the process.

FastLoad and FastSave

BASIC subprograms contained in FASTFILE.BAS

Purpose:

FastLoad and FastSave allow your programs to load and save an entire text file to/from a conventional (not fixed-length) string array very quickly.

Syntax:

To load a text file:

```
Lines = FastLoadInt%(FileName$)   'load file, get number of lines
REDIM Array$(1 TO Lines)          'dim the array to receive file
CALL FastLoadStr(Array$())        'load the text to the array
```

To save a string array:

```
CALL FastSave(FileName$, Text$())
```

Comments:

Because FastLoadInt is designed as a function, it must be declared before it may be used.

Loading a file requires two steps. The first invokes a function which actually loads the file into a far (dynamic) integer array, and then returns the number of string array elements that will be needed to receive the text. The second step is to call FastLoadStr to copy the individual string elements into the string array. Saving an array is simpler, and requires only a single call to the FastSave subprogram.

In our own informal tests, we measured an improvement in speed of approximately seven times over the equivalent BASIC statements. Also, you may be interested to know that we created a dedicated assembler routine to write all of the elements in a string array to disk in one operation. Unfortunately, the improvement over BASIC was negligible.

For example, to save 500 elements from a string array requires calling the low-level routines in DOS 1000 times. For each string element being written, its length and address must be found, and then DOS must be called to write it to the file. But after each line is written, another DOS call must be performed to write a carriage return and line feed. Because of the overhead DOS imposes each time it is called, it is actually faster to gather up all of the string elements into an array (using the QuickPak Professional string manager routines), and then save them in a single operation.

Because the entire file is loaded, there must be both sufficient string and far memory available to hold it. "Out of memory" errors (or any DOS errors that are encountered) may be detected with the DOSError and WhichError functions.

A complete demonstration of implementing these routines is given in the DEMOFAST.BAS example program.

FClose

assembler subroutine contained in PRO.LIB

Purpose:

FClose will close a file that had been previously opened with the QuickPak Professional FOpen command.

Syntax:

```
CALL FClose(Handle%)
```

Where:

Handle% is the DOS file handle that was assigned when the file was first opened.

Comments:

The only error that is likely to happen when using FClose would be caused by giving it an invalid file handle. Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

DOS handles and file access using the QuickPak Professional routines are discussed elsewhere in this manual in the section entitled "Eliminating ON ERROR".

FCopy

assembler subroutine contained in PRO.LIB

Purpose:

FCopy will copy a file from within a BASIC program without requiring the use of SHELL.

Syntax:

```
CALL FCopy(Source$, Dest$, Buffer$, ErrCode%)
```

Where:

Source$ is the source file name, Dest$ is the target file name, Buffer$ is a temporary work string used internally as a file buffer, and ErrCode% indicates if an error that occurred was on the source or destination file.

Comments:

FCopy is a vastly superior method for copying files compared to using the SHELL command. Besides being much cleaner in general, using FCopy eliminates a number of problems.

One major limitation of SHELL is that it does not work correctly with most versions of QuickBASIC under DOS 2. Further, SHELL internally runs a second copy of COMMAND.COM which requires sufficient available memory, and also means that COMMAND.COM must be available. If a PC has been booted from a floppy disk and that disk is no longer in drive A, the user will get the "Insert disk with COMMAND.COM" message.

It is important that both the source and destination variables be complete file names. That is, Dest$ may not be given simply as "A:" or "C:\STUFF\". Likewise, Source$ may not contain DOS wild cards to indicate multiple files.

Further, it is up to your program to provide a buffer to hold the file's contents as it is being copied. This may be either a conventional or fixed-length string, but it must be at least 65 characters in length. The minimum recommended buffer size is 512 bytes (the size of a DOS sector), but a string length of 4096 bytes would be ideal.

Though we could have set aside a buffer area within the FCopy program, that memory would then be permanently taken from your program. In truth, having to provide a buffer is a small inconvenience anyway, because the space can easily be reclaimed when FCopy has finished. Rather than assign a string prior to calling FCopy, the best approach would be to have BASIC create it on the fly as shown below:

```
CALL FCopy(Source$, Dest$, SPACE$(4096), ErrCode%)
```

When SPACE$() is used as an argument to a subroutine, the memory it occupies is released back to the program as soon as the subroutine finishes.

If the QuickPak Professional DOSError function is zero, then the copying was successful and you can safely ignore ErrCode%. Otherwise, if an error occurred processing the source file, then ErrCode% will return set to 1. A value of 2 indicates a problem with the destination file. In either case, the QuickPak Professional WhichError% function should be used to determine the type of error that occurred.

Also see the related routine FileCopy.

FCount

assembler function contained in PRO.LIB

Purpose:

FCount will report the number of file names that match a particular specification.

Syntax:

```
Count = FCount%(FileSpec$)
```

Where:

FileSpec$ holds a DOS file name or specification, and Count receives the number of matching entries.

Comments:

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

FCount is intended to provide a count of the files on a disk, in preparation for using ReadFile to obtain a list of all their names.

Where DCount provides a count of directory names that match a given specification, FCount instead searches for file names. It is important not to confuse these two routines.

You will generally use the DOS wild cards with FCount, for example "*.BAS" would provide a count of all the BASIC program files in the current directory on the default disk. Of course, FCount will also accept a drive letter or path name:

```
Count = FCount%("D:\LOTUS\*.WK1")
```

DOS

FCreate

assembler subroutine contained in PRO.LIB

Purpose:

FCreate is used to create a file in preparation for writing to it with the QuickPak Professional file handling routines.

Syntax:

```
CALL FCreate(FileName$)
```

Where:

FileName$ is the name of the file to be created.

Comments:

FCreate serves the same purpose as the BASIC OPEN FOR OUTPUT command followed immediately by a CLOSE. That is, if the file does not exist it will be created, and if it is already present it will be truncated to a length of zero bytes.

As with all of the QuickPak Professional file services, FCreate will also accept an optional drive letter and/or directory path. However, a wild card is not permitted.

FCreate will not cause an error if the disk is full, because it does not attempt to write any information to the disk—it merely establishes a directory entry for the file. In fact, if the file already exists and it contains data, FCreate will instead free up the disk space that had been occupied.

There are two probable causes for an error to occur when using FCreate. Either an invalid file name was given (it contains a wild card or refers to an invalid directory), or the disk's directory is full. For example, a 360K floppy disk can accommodate only 112 entries in its root directory, even if the data area is not filled up.

Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

FEof

assembler function contained in PRO.LIB

Purpose:

FEof will report if the current DOS Seek location is at the end of the specified file.

Syntax:

```
IF FEof%(Handle%) THEN . . .
```

Where:

Handle% is the handle that DOS assigned when the file was first opened using FOpen.

Comments:

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

FEof serves the same purpose as BASIC's EOF function, except it is intended for use with files that are being manipulated using the QuickPak Professional file handling routines.

FEof returns -1 if the current DOS Seek location is at the end of the file, or 0 if it is not. This allows you to test for an EOF condition either with or without the BASIC NOT command, as shown below.

```
IF FEof%(Handle%) THEN
   PRINT "It's at the end"
END IF
```

or

```
IF NOT FEof%(Handle%) THEN
   PRINT "Not at the end"
END IF
```

If an invalid handle is given, the DOSError and WhichError functions will be set to indicate the error condition.

FFlush

assembler subroutine contained in PRO.LIB

Purpose:

FFlush will flush a file's data buffers to disk without requiring the file to be closed.

Syntax:

```
CALL FFlush(Handle%)
```

Where:

Handle% is the file handle that DOS assigned when the file was first opened.

Comments:

When data is read from or written to disk, it is always first passed 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 the CONFIG.SYS file. If CONFIG.SYS is not present, then the number of buffers defaults to either 2 for a PC or XT, or 3 for an AT. 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 it allows information that has previously been read or written to be accessed again later 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 the disk at the time the write is performed. Rather, the information sits there in memory until the buffer becomes full, or the file is closed. If your program has just used PRINT or PUT to write data to a disk file and a power outage occurs, the data will never be transferred to the file.

FFlush allows you to force the file buffer's contents to be written to the disk, but without having to close the file and then re-open it again.

Unfortunately, FFlush will work only with files that have been opened and written to using the QuickPak Professional file routines. It will *not* flush the buffers for a file written using QuickBASIC's file commands, because QuickBASIC performs additional buffering before sending the data to DOS.

The only error that is likely with FFlush is giving it an invalid handle number. Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

FGet

assembler subroutine contained in PRO.LIB

Purpose:

FGet reads data from a disk file in a manner similar to BASIC's binary GET command, but it returns an error code rather than requiring the use of ON ERROR.

Syntax:

```
CALL FGet(Handle%, Destination$)
```

Where:

Handle% is the DOS file handle that was assigned when the file was opened, and Destination$ is the string that is to receive the data. The length of Destination$ determines how many bytes are to be read.

Comments:

FGet reads data from the specified file at the location held in the DOS file pointer. The current pointer location is established by the most recent read or write operation, or by using the BASIC SEEK command or the QuickPak Professional FSeek subroutine.

The length of Destination$ is used to tell FGet how many bytes it is to read to insure that sufficient room has been set aside. If FGet had been written to expect a separate variable to specify the number of bytes, it would be possible to corrupt string memory by failing to first assign the string to a sufficient length.

Only two errors are likely when using FGet—either the handle number was invalid, or the destination string was null. Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

Also see the description for the companion routine FGetT, which reads the file data into a TYPE variable.

FGetA

assembler subroutine contained in PRO.LIB

Purpose:

FGetA is similar to the QuickPak Professional FGetT routine, however it accepts a segmented address thus allowing an entire array to be loaded in one operation. FGetA will load up to 64k bytes at a time.

Syntax:

```
CALL FGetA(Handle%, SEG Array(Element), NumBytes%)
```

Where:

Handle% is the DOS file handle that was assigned when the file was opened, and Array(Element) is any array (except a conventional string array) that is to receive the data.

NumBytes% indicates the number of bytes to be read. If the number of bytes exceeds 32767, then you must instead use a long integer variable when calling FGetA:

```
CALL FGetA(Handle%, SEG Array(Element), NumBytes&)
```

Comments:

FGetA reads data from the specified file at the location held in the DOS file pointer. The current pointer location is established by the most recent read or write operation, or by using the BASIC SEEK command or the QuickPak Professional FSeek subroutine.

FGetA is intended to serve a purpose similar to BASIC's BLOAD (or the QuickPak Professional QBLoad). As described in the sections "Saving Arrays to Disk" and "Saving Screen Images to Disk", loading an entire file in one operation can provide a tremendous improvement in speed. However, in order to BLOAD a file, it must have been originally saved in the special format BASIC uses. FGetA instead loads *any* data file, without requiring the special BSAVE header.

Also see the description for the companion routine FPutA, which writes an entire array to disk in a single operation. For reading data into variables that do not require a segmented address, see the description for FGetT.

Only two errors are likely when using FGetA—either the handle number was invalid, or the number of bytes specified was zero. Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

Also see the description for the companion routine FPutA, which writes a file from a segmented address.

FGetAH

assembler subroutine contained in PRO.LIB

Purpose:

FGetAH will retrieve an entire huge array of any size from disk in a single operation.

Syntax:

```
CALL FGetAH(FileName$, SEG Array(Start), ElSize%, NumEls%)
```

Where:

FileName$ is the file to be loaded, and Array(Start) is where in memory to load the file.

ElSize% indicates the length of each array element in bytes (or a special code to indicate the length), and NumEls% is the number of elements to be loaded.

Comments:

Unlike the other FGet routines that expect a handle to a file that has already been opened, FGetAH assumes you want to load the entire file at once. This eliminates the extra steps of having to first open the file, remember the handle, load the data, and finally close the file. FGetAH is used like BASIC's BLOAD (or the QuickPak Professional QBLOAD routine), except it is not limited to loading 64K or less. FGetAH also accepts the same size code that is used by the QuickPak Professional TYPE sort routines:

```
-1 = 2-byte integer
-2 = 4-byte long integer        } these codes are
-3 = 4-byte single precision    } interchangeable
-4 = 8-byte double precision
```

If an error occurs while reading a file, the DOSError and WhichError functions will be set appropriately.

Also see the related routine FPutAH.

FGetR

assembler subroutine contained in PRO.LIB

Purpose:

FGetR reads data from a disk file in a manner similar to BASIC's random GET command, but it returns an error code rather than requiring the use of ON ERROR.

Syntax:

```
CALL FGetR(Handle%, Destination$, RecNumber&)
```

Where:

Handle% is the DOS file handle that was assigned when the file was opened, Destination$ is the string that is to receive the data, and RecNumber& is a long integer that indicates the record to be read.

The length of Destination$ determines how many bytes are to be read, and is also used with RecNumber& to determine how far into the file the record is located.

Comments:

The length of Destination$ tells FGetR how many bytes it is to read to insure that sufficient room has been set aside. If FGetR had been written to expect a separate variable to specify the number of bytes, it would be possible to corrupt string memory by failing to first assign the string to a sufficient length.

Only two errors are likely to occur when using FGetR— either the handle number was invalid, or the destination string was null. Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

Also see the description for the companion routine FGetRT, which reads the data into a TYPE variable.

FGetRT and FGetRTA

assembler subroutines contained in PRO.LIB

Purpose:

FGetRT and FGetRTA read data from a disk file in a manner similar to BASIC's random GET command, but it returns an error code rather than requiring the use of ON ERROR.

Syntax:

```
CALL FGetRT(Handle%, Destination, RecNumber&, RecSize%)
```

or

```
CALL FGetRTA(Handle%, SEG Destination, RecNumber&, RecSize%)
```

Handle% is the DOS file handle that was assigned when the file was opened, and Destination is TYPE variable that is to receive the data. RecNumber& indicates the record to be read, and RecSize% is the length in bytes of each record. RecSize% determines how many bytes are to be read, and is also used internally by FGetRT and FGetRTA to determine how far into the file the record is located.

Comments:

FGetRT and FGetRTA are nearly identical, except that FGetRTA expects a segmented address. This allows you to load one or more elements at once directly into a dynamic array. Because it is up to the program to tell FGetRT how many bytes are to be read, it is very important that sufficient room has been first set aside in the destination variable. If this is not done, string memory will probably be corrupted.

Only two errors are likely to occur when using FGetRT— either the handle number was invalid, or the record size was given as zero. Errors may be detected with the QuickPak Professional DOSError and WhichError functions. Also see FGetR, which reads the data into a conventional string variable.

FGetT

assembler subroutine contained in PRO.LIB

Purpose:

FGetT reads data from a disk file in a manner similar to BASIC's binary GET command, but it returns an error code rather than requiring the use of ON ERROR.

Syntax:

```
CALL FGetT(Handle%, Destination, NumBytes%)
```

Where:

Handle% is the DOS file handle that was assigned when the file was opened, Destination is the variable that is to receive the data and may be any type of data except a variable length string. NumBytes% indicates how many bytes are to be read.

Comments:

FGetT reads data from the specified file at the location held in the DOS file pointer. The current pointer location is established by the most recent read or write operation, or by using the BASIC SEEK command or the QuickPak Professional FSeek subroutine.

Because it is up to the program to tell FGetT how many bytes are to be read, it is very important that sufficient room has been first set aside in the destination variable. If this is not done, string memory will probably be corrupted.

Only two errors are likely when using FGetT—either the handle number was invalid, or the number of bytes to be read was specified as zero. Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

Also see the description for the companion routine FGet, which reads the data into a conventional string variable. For reading data into variables that require a segmented address, see the description for FGetA.

FileComp

BASIC function contained in FILECOMP.BAS

Purpose:

FileComp will report if any two disk files are the same.

Syntax:

```
Same = FileComp%(File1$, File2$, ErrCode%)
```

Where:

File1$ and File2$ are the two files being compared, and Same receives either -1 if they are the same, or 0 if they are not.

If a DOS error occurs (file not found, drive door open, etc.), then ErrCode% will tell which file the error relates to. An ErrCode% of 1 means the problem was with File1$, and 2 means it was with File2$.

Comments:

Besides returning a true or false condition based on a comparison of the two files, FileComp also uses the DOSError and WhichError services to report disk errors. This is illustrated in the DEMOCOMP.BAS example program.

FileCopy

BASIC subprogram contained in FILECOPY.BAS

Purpose:

FileCopy is intended to serve as a "front end" to the FCopy routine, and allows it to accepts wild cards in the source file specification.

Syntax:

```
CALL FileCopy(Source$, Dest$, Copied%, ErrCode%)
```

Where:

Source$ is a file specification such as "*.BAS", and Dest$ is the target drive or directory. Copied% returns holding the number of files actually copied. If an error occurs during the copying process, ErrCode% indicates if the problem was with the source or destination file.

Comments:

FileCopy augments the assembler FCopy routine by allowing a group of files to be copied to a new drive or directory. Where FCopy does not allow wild cards in the source file name or a destination drive or path only, FileCopy permits both.

FileCopy uses the FCount and ReadFile routines to obtain a list of all the matching files, and then copies them one by one to the destination. In fact, because FileCopy processes multiple files, you must not include a specific target file name.

A typical source and destination specification would be:

```
Source$ = "C:\SUBDIR\FILE*.*"
Dest$ = "B:"
```

or

```
Dest$ = "\SUBDIR"
```

or

```
Dest$ = "A:\"
```

Only two errors are likely when using FileCopy—either no files were found that matched the file specification, or the source or destination drive/directory were invalid.

If the copying is successful, ErrCode% will probably hold a value of 2. This is normal. Therefore, you should always use the QuickPak Professional DOSError% and WhichError% functions to see if there really was an error. If so, then ErrCode% reports with which file the error occurred.

DOS

FileCrypt

BASIC subprogram contained in FILECRPT.BAS

Purpose:

FileCrypt will quickly encrypt any file using a password provided by the calling program.

Syntax:

```
CALL FileCrypt(FileName$, PassWord$)
```

Where:

FileName$ is the name of the file being encrypted, and PassWord$ is the password that is to be used.

Comments:

FileCrypt will be used both to encrypt a file and to decrypt it again later. However, you *must* provide the identical password each time FileCrypt is called. Capitalization is important, as well as leading and trailing spaces.

FileCrypt uses the QuickPak Professional Encrypt routine to actually process the file's contents. If the file is too long to fit into memory all at once, it will be processed in pieces.

Rather than simply applying the password you provide when encrypting the file, FileCrypt first encrypts the password against an internal string of nonsense characters. This provides an extra margin of safety in case the file contains a long series of blanks or zero bytes. Blank spaces and CHR$(0) zero bytes could possibly reveal the password if the file is carefully examined.

For the best protection, you should select a password that is at least five characters in length, because the same password is used repeatedly to encode the file's contents. Since the main objective is to prevent the file from containing any recognizable pattern, a longer password will be repeated less often. See the discussion that accompanies the Encrypt routine for more information about passwords.

A complete demonstration of FileCrypt is given in the DEMOFC.BAS example program. Be *very careful* when you try it.

The only error that is likely when using FileCrypt is that the specified file does not exist. Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

FileInfo

assembler subroutine contained in PRO.LIB

Purpose:

FileInfo returns all of the characteristics of a file, including its date, time, size, and attributes.

Syntax:

```
CALL FileInfo(FileName$, SEG TypeVar)
```

Where:

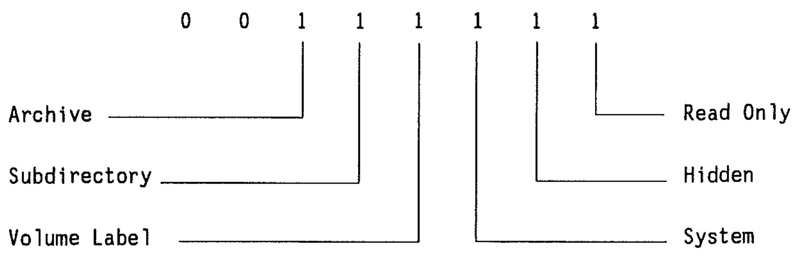
FileName$ is the name of the file to report on, and TypeVar is a TYPE variable that will receive the information.

Comments:

If the file doesn't exist, the Year portion of the TYPE variable will be set to zero. The DOSError and WhichError functions may also be examined to see if an error occurred. If the file name contains wild cards (* or ?), then the first file matching the specification will be reported on. The TYPE variable must be set up like this:

```
TYPE FInfo
    Year   AS INTEGER
    Month  AS INTEGER
    Day    AS INTEGER
    Hour   AS INTEGER
    Minute AS INTEGER
    Second AS INTEGER
    Size   AS LONG
    Attrib AS INTEGER
END TYPE
```

The file's attribute is bit-coded as follows:



Two routines are also included with QuickPak Professional to retrieve only the size of a file. FileSize expects a file name as a string, while FLof accepts a DOS handle for a file that is currently open. Also see the GetAttr function for a discussion of the file attribute byte.

FileInfo is amply demonstrated in the FILEINFO.BAS example program.

FileSize

assembler function contained in PRO.LIB

Purpose:

FileSize will quickly return the length of a named file.

Syntax:

```
Size = FileSize&(FileName$)
```

Where:

FileName$ is the name of the file, and Size is assigned its length in bytes. If the file does not exist Size is instead assigned a value of -1.

Comments:

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

Besides returning a size of -1 if the named file is not present, the DOSError and WhichError functions may also be examined for error information.

Also see the description for FLof which returns a file's size, but it accepts a file handle instead of a name.

FileSort

BASIC subprogram contained in FILESORT.BAS

Purpose:

FileSort will sort a random access disk file on any number of keys, and each key may be ordered independently either ascending or descending.

Syntax:

```
CALL FileSort(DataFile$, IndexFile$, First&, Last&,Table%(), RecLength%)
```

Where:

DataFile$ is the name of the data file being sorted, IndexFile$ is the name of the index file that will be created, and First& and Last& indicate the range of records to be sorted. If both First& and Last& are zero, then the entire file will be considered. The Table%() array tells FileSort which record fields are to be used as the sort keys, using the same method as KeySort. RecLength% is the length of each disk record in bytes. If RecLength% is given as a negative value, sorting is performed without regard to capitalization.

Comments:

When FileSort is called, it does not actually sort the data file. Rather, a separate index file is created which holds a list of record numbers in sorted order. FileSort calls upon KeySort to do the actual sorting. Therefore, the same method of creating the Table array is used here. Please see the KeySort description for more information.

Disk errors may be detected by using the DOSError and WhichError functions. FileSort is demonstrated in the DEMOSORT.BAS example program.

Because FileSort needs to create a huge array as part of its internal operation, you must start BASIC with the /ah option switch.

DOS

FLInput

assembler function contained in PRO.LIB

Purpose:

FLInput will read a line of input from a file that has been opened using the QuickPak Professional FOpen routine.

Syntax:

```
Work$ = FLInput$(Handle%, Buffer$)
```

Where:

Handle% is the file handle that was assigned when the file was first opened, and Buffer$ is a temporary work string needed by FLInput as it reads the file contents.

Comments:

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

Unlike the other QuickPak Professional binary input routines that read a specified number of bytes, words, or elements from a file, FLInput reads until it encounters the CHR$(13) that marks the end of a line, or the CHR$(26) that indicates the end of a file. For files that do not contain an end of file marker, FLInput stops reading at the physical end of the file.

The only errors that are likely when using FLInput are “Input Past End” (number 62) or “Buffer Too Small” (number 83). It is up to the calling program to provide a temporary work buffer for FLInput. If that buffer is too small to read an entire line, you should invoke FLInput again to obtain the remainder. This is illustrated in the FLINPUT.BAS example program.

FLoc

assembler function contained in PRO.LIB

Purpose:

FLoc reports the current DOS file pointer position for files that have been opened using the QuickPak Professional file services. FLoc serves the same purpose as BASIC's built-in LOC() function.

Syntax:

```
Location = FLoc&(Handle%)
```

Where:

Handle% is the DOS file handle that was assigned when the file was opened, and Location is assigned to the current DOS file pointer location. If the handle is invalid Location is instead assigned a value of -1.

Comments:

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

Unlike QuickBASIC's SEEK and LOC, FLoc considers the first byte in a file to be byte 0, not 1. Therefore, FLoc will return a location of 0 if the current file pointer is at the beginning of the file.

Besides returning a location of -1 if the handle given is invalid, the DOSError and WhichError functions may also be examined for the error information.

Also see the companion program FSeek which will set the file pointer location.

FLof

assembler function contained in PRO.LIB

Purpose:

FLof returns the length of a file that has been opened with the QuickPak Professional FOpen subroutine. It is similar to the FileSize&() function, except FLof accepts a file handle rather than a name. FLof serves the same purpose as the BASIC LOF() function.

Syntax:

```
Length = FLof&(Handle%)
```

Where:

Handle% is the DOS file handle that was assigned when the file was opened, and Length is assigned to the file's length in bytes. If the handle is invalid Length is instead assigned a value of -1.

Comments:

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

Besides returning a length of -1 if the handle given is invalid, the DOSError and WhichError functions may also be examined for the error information.

Also see the FileSize function which returns a file's size, but it accepts a file name rather than a handle.

FOpen and FOpenS

assembler subroutines contained in PRO.LIB

Purpose:

FOpen is used to open a disk file in preparation for reading or writing using the QuickPak Professional file access routines. FOpenS is identical, except it opens a file for Shared (network) access.

Syntax:

```
CALL FOpen(FileName$, Handle%)
```

or

```
CALL FOpenS(FileName$, Handle%)
```

Where:

FileName$ is the name of the file to be opened, and Handle% is assigned by DOS for all subsequent references to the file. If the file does not exist or any other error occurs, Handle% will be returned set to -1.

Comments:

FOpen and FOpenS will open any file, and they also accept an optional drive or directory as part of the file name. However, they will not create a file. If you are not sure if a file exists you should first use the Exist function, or call FCreate to create it.

It is up to your program to remember the handle number that DOS assigns, and use that number whenever you access the file again later.

Besides returning a handle of -1 if the named file is not present, the DOSError and WhichError functions may also be examined for this information.

Also see the section entitled "Eliminating ON ERROR" for more information on DOS handles.

FOpenAll

assembler subroutine contained in PRO.LIB

Purpose:

FOpenAll will open a file for any access mode, including all of the variations needed for network operation.

Syntax:

```
CALL FOpenAll(FileName$, AccessMode%, ShareMode%, Handle%)
```

Where:

FileName$ holds the file name to open.

AccessMode%	0	Open file for reading only
	1	Open file for writing only
	2	Open file for reading and writing
ShareMode%	0	Deny sharing access (compatibility mode)
	1	Deny read/write access
	2	Deny write access
	3	Deny read access
	4	Deny none (full share mode)

Handle% returns with the handle number that DOS assigns. If the file cannot be opened, Handle% will be set to -1.

Comments:

Unlike the QuickPak Professional FOpen routine which opens a file for read/write access only, FOpenAll provides you with complete control over all of the possible DOS open parameters. These are shown in the table above.

Also see the description for the DOSError and WhichError functions.

FormatDiskette

assembler function contained in PRO.LIB

Purpose:

FormatDiskette lets you add disk formatting capabilities to your programs.

Syntax:

```
Result = FormatDiskette%(DriveNumber%, Capacity%, SEG BufArray%)
```

Where:

DriveNumber% refers to a physical drive number with drive A represented as zero, drive B as one, and so forth.

The Capacity% argument is given as whole integer values:

```
 360 = 360KB 5.25"
1200 = 1.2MB 5.25"
 720 = 720KB 3.5"
1440 = 1.44MB 3.5"
```

BufArray is a block of memory that FormatDiskette will use as a work area to hold the disk's FAT (File Allocation Table) as it is being built.

Result then receives a code that reports if formatting was successful. See the table below for a list of all possible result codes.

Comments:

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

We recommend that you use an integer array as a buffer because it can be dimensioned before formatting the diskette, and then erased afterwards. We designed FormatDiskette to require a user-supplied buffer to avoid having it take the necessary memory permanently from your program.

The table below shows how big the buffer must be for each of the possible diskette capacities. Of course, the buffer can be larger than necessary and you can use the largest size only, to avoid having to add extra logic to your program.

Table 3-1

Disk Size	**Bytes needed**	**Number of integer elements**
360KB	1060	REDIM BufArray%(1 TO 530)
1.2MB	3644	REDIM BufArray%(1 TO 1822)
720KB	1572	REDIM BufArray%(1 TO 786)
1.44MB	4680	REDIM BufArray%(1 TO 2340)

In the table below, notice that a result code of zero indicates that the diskette was formatted successfully; any other value means there was an error.

The only error that is not fatal is 11, which means that bad sectors were found but were also marked as being bad. This is the same way the DOS FORMAT program works, in that as long as track zero is not defective the disk is still usable. If you do receive error 11 you can use the DiskInfo subroutine to compare the number of total and free clusters. This will tell you how many clusters were marked as being not available.

Table 3-2
FormatDiskette error code return values

0 =	No error
1 =	Invalid disk parameters
2 =	Address mark not found
3 =	Write protect error
4 =	Requested sector not found
5 =	Can't locate drive
6 =	Disk change line is active
7 =	Invalid capacity specified
8 =	DMA Overrun
9 =	DMA boundary error
10 =	Track zero bad
11 =	Bad sectors found and marked (not fatal)
12 =	Media type not found
16 =	CRC read error
32 =	Disk controller failure
64 =	Seek failure
128 =	Drive not ready

FormatDiskette can be used to format a disk at a lower capacity than the drive is capable of. For example, you can specify a capacity of 360KB even if the disk drive can handle 1.2MB disks. Likewise, you can specify that a 1.44MB diskette be formatted to only 720KB.

FormatDiskette is demonstrated in the FORMAT.BAS example program.

FPut

assembler subroutine contained in PRO.LIB

Purpose:

FPut writes data to a disk file in a manner similar to BASIC's binary PUT command, but it returns an error code rather than requiring the use of ON ERROR.

Syntax:

```
CALL FPut(Handle%, Source$)
```

Where:

Handle% is the DOS file handle that was assigned when the file was opened, and Source$ is the string that contains the data to be written.

The length of Source$ determines how many bytes will be written to the file.

Comments:

FPut writes data to the specified file at the location held in the DOS file pointer. The current pointer location is established by the most recent read or write operation, or by using the BASIC SEEK command or the QuickPak Professional FSeek subroutine.

Only two errors are likely when using FPut—either the handle number was invalid, or the source string was null. Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

You can treat the file as sequential by appending a carriage return/line feed to the end of each string as it is written:

```
CRLF$ = CHR$(13) + CHR$(10)
CALL FPut(Handle%, Source$ + CRLF$)
```

Also see the description for FPutT, which writes the file data from information in a string or TYPE variable.

FPutA

assembler subroutine contained in PRO.LIB

Purpose:

FPutA is similar to the QuickPak Professional FPut routine, however it accepts a segmented address thus allowing an entire array to be saved in one operation. FPutA will write up to 64k bytes at a time.

Syntax:

```
CALL FPutA(Handle%, SEG Array(Start), NumBytes%)
```

Where:

Handle% is the DOS file handle that was assigned when the file was opened, and Array(Start) is any array (except a conventional string array) that is to be saved.

NumBytes% indicates the number of bytes to be written. If the number of bytes exceeds 32767, then you must instead use a long integer variable or number.

Comments:

Like its companion program FGetA, FPutA allows your programs to write an entire numeric or TYPE array to disk in a single operation.

See the comments that accompany the description for FGetA. For writing data from variables that do not require a segmented address, see the description for FPutT.

FPutAH

assembler subroutine contained in PRO.LIB

Purpose:

FPutAH will write an entire huge array of any size to disk in a single operation.

Syntax:

```
CALL FPutAH(FileName$, SEG Array(Start),ElSize%, NumEls%)
```

Where:

FileName$ is the file to be saved, and Array(Start) is the first element in the array to be written to the file.

ElSize% indicates the length of each array element in bytes (or a special code to indicate the length), and NumEls% is the number of elements to write.

Comments:

Unlike the other FPut routines that expect a handle to a file that has already been opened, FPutAH assumes you want to save the entire array at once. This eliminates the extra of steps of having to first open the file, remember the handle, write the data, and finally close the file. FPutAH is used like BASIC's BLOAD (or the QuickPak Professional QBLoad routine), except it is not limited to writing 64K or less. FPutAH also accepts the same size code that is used by the QuickPak Professional TYPE sort routines:

```
-1 = 2-byte integer
-2 = 4-byte long integer          }these codes are
-3 = 4-byte single precision      }interchangeable
-4 = 8-byte double precision
```

If an error occurs while writing a file, the DOSError and WhichError functions will be set appropriately.

Also see the related routine FGetAH.

FPutR

assembler subroutine contained in PRO.LIB

Purpose:

FPutR writes data to a disk file in a manner similar to BASIC's random PUT command, but it returns an error code rather than requiring the use of ON ERROR.

Syntax:

```
CALL FPutR(Handle%, Source$, RecNumber&)
```

Where:

Handle% is the DOS file handle that was assigned when the file was opened, Source$ is the string that holds the data to be written, and RecNumber& is a long integer that indicates the record to be written to.

The length of Source$ determines how many bytes are to be written, and is also used with RecNumber& to determine how far into the file the record is located.

Comments:

Only two errors are likely to occur when using FPutR—either the handle number was invalid, or the source string was null. Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

Also see the description for the companion routine FPutRT, which writes the data from a fixed-length string or TYPE variable.

FPutRT and FPutRTA

assembler subroutines contained in PRO.LIB

Purpose:

FPutRT and FPutRTA write data into a disk file in a manner similar to BASIC's random PUT command, but it returns an error code rather than requiring the use of ON ERROR.

Syntax:

```
CALL FPutRT(Handle%, Source, RecNumber&, RecSize%)
```

or

```
CALL FPutRTA(Handle%, SEG Source, RecNumber&, RecSize%)
```

Where:

Handle% is the DOS file handle that was assigned when the file was opened, and Source is a fixed-length string or TYPE variable that holds the data to be written. RecNumber& is a long integer that indicates the record to be written to, and RecSize% is the length in bytes of each record.

RecSize% determines how many bytes are to be written, and is also used internally by FPutRT along with RecNumber& to determine how far into the file the record is located.

Comments:

FPutRT and FPutRTA are nearly identical, except that FPutRTA expects a segmented address. This allows you to save one or more elements at once directly from a dynamic array. Only two errors are likely to occur when using FPutRT or FPutRTA— either the handle number was invalid, or the record size was given as zero. Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

Also see the description for the companion routine FPutR, which writes the data from a conventional string variable.

FPutT

assembler subroutine contained in PRO.LIB

Purpose:

FPutT writes data to a disk file in a manner similar to BASIC's binary PUT command, but it returns an error code rather than requiring the use of ON ERROR.

Syntax:

```
CALL FPutT(Handle%, Source, NumBytes%)
```

Where:

Handle% is the DOS file handle that was assigned when the file was opened, Source is the fixed-length string or TYPE variable that holds the data being written, and NumBytes indicates how many bytes are to be written to the file.

Comments:

FPutT writes data into the specified file at the location held in the DOS file pointer. The current pointer location is established by the most recent read or write operation, or by using the BASIC SEEK command or the QuickPak Professional FSeek subroutine.

Only two errors are likely when using FPutT—either the handle number was invalid, or the number of bytes to be written was specified as zero. Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

Also see the description for the companion routine FPut, which writes the data from a conventional string variable. For writing data from variables that require a segmented address, see the description for FPutA.

FSeek

assembler subroutine contained in PRO.LIB

Purpose:

FSeek will position the DOS file pointer for a file that has been opened using the QuickPak Professional FOpen routine.

Syntax:

```
CALL FSeek(Handle%, Location&)
```

Where:

Handle% is the handle that DOS assigned when the file was first opened, and Location& is the location in the file to seek to.

Comments:

Unlike QuickBASIC's SEEK and LOC, FSeek considers the first byte in a file to be byte 0, not 1. Therefore, to seek to the beginning of a file you would call FSeek with a location value of 0.

The only error that is likely to occur when using FSeek is giving it an invalid handle number. Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

One warning you should be aware of is that seeking to a location beyond the end of a file will cause it to be extended. This is not a fault with FSeek, and in fact will happen with QuickBASIC's SEEK command as well.

To obtain the current seek location for a file, use the QuickPak Professional FLoc function.

FStamp

assembler subroutine contained in PRO.LIB

Purpose:

FStamp creates a new date and time for a specified file.

Syntax:

```
CALL FStamp(FileName$, NewTime$, NewDate$)
```

Where:

FileName$ is the name of the file to process, NewTime$ is a string representing the new time, and NewDate$ is a string holding the new date.

Comments:

FStamp is very capable in that all of the foreign date and time standards are supported, and the format of the new date and time strings is quite flexible.

To apply a new time and leave the current date alone, only the NewTime$ variable should be assigned and NewDate$ will be a null string. Likewise, if only a new date is required, then NewTime$ would be left null. The current system date or time may also be specified by placing an asterisk (*) in either of the strings.

The date and time values must have a valid delimiter between the various components. For example, a valid date would be any of the following:

"05/02/88" or "05-02-1988" or "5.2.88" or "5-2-1988"

Regardless of the delimiter used, FStamp will examine the current country information from DOS, in order to determine which format is being used. The three methods recognized are American MM-DD-YY, European DD.MM.YY, or Japanese YY.MM.DD.

Valid delimiters for the time field are either a colon or a period. If only the hour is given, then the minutes and seconds will be set to zero:

```
NewTime$ = "5"
```

Similarly, if only the hour and minutes are given, then the seconds are assumed to be zero:

```
NewTime$ = "5:05"
```

The QuickPak Professional DOSError% and WhichError% functions should be examined to see if an error occurred during the file stamping operation.

FullName

assembler function contained in PRO.LIB

Purpose:

FullName accepts a partial file name or file specification, and returns a fully qualified name that includes the complete path.

Syntax:

```
Qualified$ = FullName$(PartName$)
```

Where:

PartName$ is a partial path name such as "..\over\name.dat", and Qualified$ receives the full path such as "\root\over\name.dat".

Comments:

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

Please understand that FullName does not check for the validity of either the directory or path names. It merely reports what the full name *would* be under ideal conditions.

FullName uses an undocumented DOS interrupt service to do the actual work, and requires DOS 3.0 or later.

GetAttr

assembler function contained in PRO.LIB

Purpose:

GetAttr examines a disk file and reports the setting of its DOS attribute byte.

Syntax:

```
Attribute = GetAttr%(FileName$)
```

Where:

FileName$ is the file being examined, and Attribute is assigned bit coded with the file's attributes. If the file does not exist or any other error occurs, the attribute variable will be set to -1.

Comments:

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

Every file has an attribute that is assigned at the time it is created. The attribute information is kept in a disk's directory, along with each file's name, date, and time. There are six possible attributes a file can possess, and in many cases it can have more than one. An assembler program can establish any attribute when it asks DOS to create a file, however BASIC offers much less flexibility.

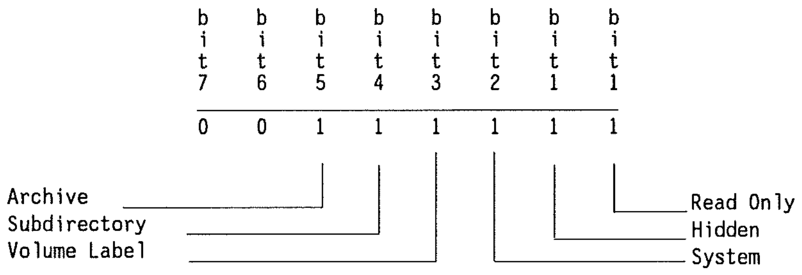
The most common file attribute is Archive, and this simply indicates if the file has been backed up since the last time it was modified. If the archive portion of the attribute byte is set to 1 (true), then the file has been modified but not backed up. DOS sets the archive bit every time a file is written to, and most backup programs clear it as they process each file. Files that are created by BASIC have the archive attribute only.

The next attribute is Subdirectory, and this indicates that a file is a DOS subdirectory, as opposed to a program or data file. There is no real difference between a subdirectory and any other file, other than the setting of this bit in its attribute byte.

The third type of attribute is Volume Label. Of course a volume label isn't actually a file, but rather a name that may be given to a diskette or hard disk. However, a disk's volume label is kept in the root directory along with the other file entries.

The final three attributes are System, Hidden, and Read- Only. The most common system files are IBMBIO.COM and IBMDOS.COM, though any file could be defined as a system file. Hidden files are not displayed by the DIR or FILES commands, and a file that has been marked as read-only may not be altered. Setting a file to read-only is a good way to prevent it from being accidentally overwritten or erased.

The attribute returned by GetAttr is in the form of a single byte, and the placement of the various bits is illustrated in the table below.



Besides returning an attribute of -1 if the named file is not present, the DOSError and WhichError functions may also be examined for error information.

The program GETATTR.BAS provides an example of getting a file's attribute byte, and shows how to isolate and interpret the individual bits.

Also provided with QuickPak Professional is a companion program SetAttr that allows setting a file's attribute.

DOS

GetDir

assembler function contained in PRO.LIB

Purpose:

GetDir will return the current directory for either a specified drive or the default drive.

Syntax:

```
Directory$ = GetDir$(Drive$)
```

Where:

Drive$ is a valid drive letter or a null string to indicate the default drive, and Directory$ is assigned the name of the current directory.

Comments:

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

The Drive$ parameter may be given as either upper or lower case, and only the first character is considered by GetDir.

GetDir returns the complete directory name, minus the drive letter and colon. However, GetDir does not report if the drive was invalid. This may be tested after using GetDir by examining the QuickPak Professional DOSError and WhichError functions.

GetDisketteType

assembler function contained in PRO.LIB

Purpose:

GetDisketteType returns the type of floppy disk drive that is installed.

Syntax:

```
Result% = GetDisketteType%(DriveNumber%)
```

Where:

DriveNumber% refers to the physical drive number as recognized by the BIOS. That is, drive A is specified with a value of zero, drive B with a value of one, and so forth.

The result returned indicates the type of drive as follows:

0 = Drive not present or cannot be identified
1 = 360KB 5.25" 40 track
2 = 1.2MB 5.25" 80 track
3 = 720KB 3.5" 80 track
4 = 1.4MB 3.5" 80 track

Comments:

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

In most cases you will use GetDisketteType before calling FormatDiskette, to ensure that you specify appropriate parameters. Once the drive type is known you can then proceed to format the diskette.

GetDisketteType is demonstrated in the FORMAT.BAS example program.

GetDrive

assembler function contained in PRO.LIB

Purpose:

GetDrive returns the current default disk drive.

Syntax:

```
Drive = GetDrive%                 'get the drive as an integer
```

or

```
Drive$ = CHR$(GetDrive%)          'get the drive as a string
```

Where:

Drive$ is assigned to a value representing an ASCII character that holds the current default drive. GetDrive actually returns an integer result, which can be easily turned into a string as shown above.

Comments:

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

GetDrive returns a value that represents an upper case ASCII drive letter. For example, if the current default drive is A, then GetDrive will return the value 65. Likewise, if the current drive is C, GetDrive would return 67.

GetVol

assembler function contained in PRO.LIB

Purpose:

GetVol obtains the disk volume label for either a specified drive or the current default drive.

Syntax:

```
Volume$ = GetVol$(Drive$)
```

Where:

Drive$ is either an upper or lower case letter of the drive being examined, or a null string to indicate the current default drive. Volume$ is then assigned the disk's volume label.

Comments:

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

Some programmers like to use a disk's volume label as a way to keep track of its contents. One example might be when writing a major accounting program that must also work with a floppy disk system. You could assign labels for each disk using names such as "CUSTOMER" and "SALES". Then if the operator inadvertently put the wrong disk in the drive, your program would know that and prompt for the correct one.

Also see the companion program PutVol, which will create or modify a disk volume label.

GoodDrive

assembler function contained in PRO.LIB

Purpose:

GoodDrive will quickly determine whether a specified drive letter is valid.

Syntax:

```
Okay = GoodDrive%(Drive$)
```

Where:

Drive$ is either an upper or lower case letter that represents the drive to check, and Okay receives either -1 if the drive is valid, or 0 if it is not.

Comments:

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

GoodDrive is provided as a companion for the QuickPak Professional LastDrive function. Where LastDrive will return the last consecutively available drive in a system, GoodDrive checks a single drive letter for validity.

GoodDrive is needed in those cases where the DOS SUBST command has been used to create an alias drive letter. For example, if a system has physical drives A, B, and C, and a RAM disk or SUBST disk drive G, LastDrive would report drive C as the last available drive.

If a user then asks your program to save some data to drive G, you could call GoodDrive to see if G really is valid before displaying a warning message.

Handle2Name

assembler subroutine contained in PRO.LIB

Purpose:

Handle2Name returns the name of an open file, given the DOS handle.

Syntax:

```
CALL Handle2Name(BYVAL Handle%, FilName)
```

Where:

Handle% is the DOS handle for an open file, and FilName is a TYPE variable as described below.

Comments:

The following short complete example shows Handle2Name in context:

```
TYPE NameType
  FileName  AS STRING * 8
  Extension AS STRING * 3
END TYPE
DIM FilName AS NameType

PRINT "Handle FileName Ext"
PRINT "=================="
FOR Handle% = 0 TO 19
  CALL Handle2Name(BYVAL Handle%, FilName)
  PRINT TAB(3); Handle%;
  PRINT TAB(8); FilName.FileName; " "; FilName.Extension
NEXT
```

KillDir

assembler subroutine contained in PRO.LIB

Purpose:

KillDir will remove a specified directory like BASIC's RMDIR command, but it returns an error code rather than requiring the use of ON ERROR.

Syntax:

```
CALL KillDir(DirName$)
```

Where:

DirName$ is the directory to be deleted, and may include an optional drive letter or path.

Comments:

The most common errors that are likely to happen when using KillDir would be caused by asking it to remove a directory that doesn't exist, or specifying a directory that contains files. Two other possibilities would be giving an invalid drive or parent directory, or using a wild card as part of the directory name. Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

The DCount function may be used to determine the presence of any directory. Even though DCount is meant to return the number of directory names that match a given specification, it will also accept a complete directory name. Thus, if the directory exists, DCount will return 1. And if it doesn't, DCount would instead return 0.

Specifying a directory name can be tricky if it isn't located under a disk's root directory. For example, to remove the directory LEVEL2 which is under the directory \LEVEL1 you could use:

```
CALL KillDir("\LEVEL1\LEVEL2")
```

No matter what directory is current, the correct one will be removed. But if you're already in the \LEVEL1 directory, you might instead use:

```
CALL KillDir("LEVEL2")
```

Notice that the leading back slash is *not* specified in this example. If it were, DOS would look for a directory named LEVEL2 under the root directory. Of course, LEVEL2 is really under \LEVEL1, so KillDir would return an error.

Also notice that directory names can have an extension, even though most people don't bother with extensions when creating a directory.

KillFile

assembler subroutine contained in PRO.LIB

Purpose:

KillFile will delete a specified file like BASIC's KILL command, but it returns an error code rather than requiring the use of ON ERROR.

Syntax:

```
CALL KillFile(FileName$)
```

Where:

FileName$ is the file to be deleted, and may include an optional drive or path but not wild cards.

Comments:

The most common error that is likely to happen when using KillFile would be caused by asking it to delete a file that isn't there. Two other possibilities would be specifying an invalid drive or directory, or if the file has a Read-Only attribute. Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

The Exist function may be used to determine the presence of file. Also, a complete discussion of file attributes is given in the section that describes the GetAttr function.

LastDrive

assembler function contained in PRO.LIB

Purpose:

LastDrive will report the last consecutively available drive in a PC.

Syntax:

```
Last = LastDrive%
```

or

```
Last$ = CHR$(LastDrive%)
```

Where:

Last receives the ASCII value of an upper case letter that represents the last consecutively available drive.

Comments:

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

LastDrive returns the last consecutive drive letter that is present, but it does not always take into account "alias" drives that may also be valid.

For example, a system may have physical drives labeled A through C, as well as a RAM disk set up as drive D. In that case, LastDrive will return 68, which is the ASCII for the letter "D". However, if the DOS SUBST command is also being used to assign a phantom drive "H" to a DOS subdirectory, LastDrive will not detect that.

Adding the code to scan through all of the possible drives is not difficult. However, that would mislead your program into believing that drives E, F, and G were also available when they are not.

If you suspect that there may be valid drives beyond the one reported by LastDrive, you should use the GoodDrive function to tell if it really is present. GoodDrive will accept any drive letter, and return -1 if the drive is actually valid.

LineCount

assembler function contained in PRO.LIB

Purpose:

LineCount will quickly count all of the lines of text in a specified file.

Syntax:

```
Count = LineCount%(FileName$, Buffer$)
```

Where:

FileName$ is the name of the file to examine, and may contain an optional drive letter or path name. Buffer$ is a temporary work area needed by LineCount to hold the file, and Count receives the number of lines of text.

If there are more than 32767 lines in the file (very unlikely if you think about it), LineCount will return a negative number. Simply add 65536 to that number to get the actual count.

If the file doesn't exist or some other error occurs, Count will instead receive -1.

Comments:

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

Besides returning a count of -1 if the file is not present, the QuickPak Professional DOSError and WhichError functions may be examined for error information.

The main purpose of LineCount is to report the number of lines in a text file, in preparation for dimensioning an array to hold them. When a program will be reading a text file into an array, knowing this ahead of time can be a big help. The only other possibilities would be to read through the entire file once just to count the lines, or guess at how large to dimension the array to.

BASIC's INPUT command on sequential files is painfully slow, because every character must be examined looking for a CHR$(13) that marks the end of a line, or a CHR$(26) that marks the end of the file. Further, numbers stored in a sequential file must be converted to the internal format used by BASIC which also takes time. This clearly rules out reading the file an extra time just to count the lines.

Guessing at how large to dimension an array is an equally poor practice. If you guess too small, you'll either crash completely, or at best have to re-dimension the array and start all over again. And if you guess too high, you've wasted four bytes of string memory for each extra element that was created.

One important caveat you should be aware of when using LineCount is that it merely counts the number of CHR$(13) carriage returns in the file. (Though it does so *much* faster than BASIC ever could.) Line feeds are not considered at all.

Another important point is that your program must provide a data buffer to hold the file's contents as it is examined. This will be supplied in the form of a conventional or fixed-length string, and it must be at least 256 characters long. The minimum recommended buffer size is 512 bytes (the size of a DOS sector), but a string length of 4096 bytes would be ideal.

Though we could have set aside a buffer area within the LineCount program, that memory would then be permanently taken from your program. In truth, having to provide a buffer is a small inconvenience anyway, because the space can easily be reclaimed when LineCount has finished. Rather than assign a string prior to calling LineCount, the best approach would be to have BASIC create it on the fly as shown below:

```
Count = LineCount%(FileName$, SPACE$(4096))
```

When SPACE$ (or STRING$) is used as an argument to a subroutine, the memory it occupies is released back to the program as soon as the subroutine finishes.

LoadExec

assembler function contained in PRO.LIB

Purpose:

LoadExec is a SHELL replacement that lets you execute another program, and then retrieve its exit code (the DOS error level).

Syntax:

```
ExitCode = LoadExec%(Program$, CmdLine$)
```

Where:

Program$ is the full name of the program to be run, and CmdLine$ is an optional command line parameter that is passed to the program.

Comments:

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

LoadExec can be used to run .EXE and .COM programs only. You cannot run a batch file or other non-executable program directly (but see below).

Note that the DOS service that LoadExec uses does not honor the PATH setting. Therefore, you must state the complete drive and path if they are not the current default. See the SearchPath$ function elsewhere in this manual, which examines the PATH and returns the fully qualified name of an executable program.

Normally, ExitCode will receive a value between 0 and 255, corresponding to the exit code returned by the program that was executed. However, it is also possible that the value may be higher if DOS intervened in the program's termination. Here's how you can determine how the program was ended, and isolate the two separate pieces of information:

```
IF ExitCode <= 255 THEN
  PRINT "Normal termination with an exit code of"; ExitCode
ELSE
  DOSCode = ExitCode \ 256        'isolate DOS's contribution
  ExitCode = ExitCode AND 255     'optionally retain the program's part
  PRINT "DOS intervened"
  SELECT CASE DOSCode
    CASE 1
      PRINT "You pressed Ctrl-C or Ctrl-Break to end the program"
    CASE 2
      PRINT "A critical error occurred and you pressed A (Abort)"
    CASE 3
      PRINT "The program ended as a TSR -- reboot now!"
    CASE ELSE
  END SELECT
END IF
```

Due to a bug in DOS 2.x you should not use LoadExec with that version of DOS. You can prevent this by first querying the DOSVer function, and checking for a value of at least 300. Related to the DOS 2.x issue, some older programs (including CHKDSK and FORMAT) require a DOS 1.x style FCB structure to be present. To avoid adding a lot of code to accommodate what is clearly a waning method, LoadExec does not support those either.

LOADEXEC.BAS shows how to use LoadExec with the SearchPath$ function, and also how to execute batch files. Besides interpreting return codes, it also shows how to detect DOS errors using WhichError.

LockFile

assembler subroutine contained in PRO.LIB

Purpose:

LockFile will lock all or a portion of a network file, but without needing ON ERROR.

Syntax:

```
CALL LockFile(Handle%, Offset&, Length&)
```

Where:

Handle% is the handle assigned by DOS when the file was opened, Offset& is the starting offset into the file where locking is to begin, and Length& is the number of bytes to lock.

Comments:

Only two errors are likely when using LockFile, 73 and 85, and these may be determined with the QuickPak Professional DOSError and WhichError functions.

To use LockFile you must provide a file handle, as well as the range of bytes to lock. These are easily determined based on the record number being locked, and the length of each record. File offsets are zero-based, which means that the first byte in the file is zero, not one. If the file is successfully locked, it will not be necessary to do so again in BASIC. Also, portions of a file must be unlocked in the exact same manner in which they were locked. If you lock, say, bytes 100 through 200, then you must unlock the identical range later.

Understand that when an application locks all or part of a file, the locking applies only to other programs. Although other programs are restricted from both reading and writing to that portion of the file, your application still has complete access to the entire file. See the LOCKFILE.BAS demonstration program for a complete example that shows how to calculate the Offset& and Length& parameters.

MakeDir

assembler subroutine contained in PRO.LIB

Purpose:

MakeDir will create a directory in the same way that BASIC's MKDIR will, but without requiring the need for ON ERROR.

Syntax:

```
CALL MakeDir(DirName$)
```

Where:

DirName$ is the name of the directory to create, and may also include an optional drive letter or path.

Comments:

The only errors that are likely to happen when using MakeDir would be caused by specifying an invalid drive letter, or a parent directory that does not exist. Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

Specifying a directory name can be tricky if it isn't located under a disk's root directory. For example, to create the directory LEVEL2 under the directory \LEVEL1 you could use:

```
CALL MakeDir("\LEVEL1\LEVEL2")
```

No matter what directory is current, the correct one will be created. But if you're already in the \LEVEL1 directory, you might instead use:

```
CALL MakeDir("LEVEL2")
```

Notice that the leading back slash is *not* specified in this example. If it were, DOS would create a directory named LEVEL2 under the root directory, which of course is not what was intended.

NameDir

assembler subroutine contained in PRO.LIB

Purpose:

NameDir will rename a directory.

Syntax:

```
CALL NameDir(OldName$, NewName$)
```

Where:

OldName$ is the original directory name, and NewName$ is the name it is to become.

Comments:

The only errors that are likely to occur when using NameDir would be specifying OldName$ for a directory that doesn't exist, or giving a NewName$ for a directory that does. One other possibility would be using a wild card in either name.

Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

The DCount function may be used to determine the presence of a directory.

It is important to understand that renaming a directory from within a running program can be dangerous. For example, if you allow a user to rename directories, you may not be able to find a data file afterward. Also, if a directory that is one or more levels above the current one is changed, then the current directory's name will be changed as well.

It is important that you provide only the new name as the second parameter, and not a drive letter or path. For example, if OldName$ is "C:\OLD", then NewName$ should be "NEW" and *not* "C:\NEW".

NameFile

assembler subroutine contained in PRO.LIB

Purpose:

NameFile will rename a file in the same way that BASIC's NAME AS command will, but without requiring ON ERROR.

Syntax:

```
CALL NameFile(OldName$, NewName$)
```

Where:

OldName$ is the name the file currently has, and NewName$ is the name it is to become.

Comments:

The only errors that are likely to occur when using NameFile would be specifying OldName$ for a file that doesn't exist, or giving a NewName$ for a file that does. Two other possibilities would be using a wild card in either name, or if the file has a Read-Only attribute. Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

The Exist function may be used to determine the presence of a file. Also, a complete discussion of the Read-Only and other file attributes is given in the section that describes the GetAttr function.

NetDrive

assembler function contained in PRO.LIB

Purpose:

NetDrive reports if a given drive is remote (on a network).

Syntax:

```
Remote = NetDrive%(Drive$)
```

Where:

Drive$ is a letter that specifies the drive to test, or a null string to examine the current drive. Remote then receives -1 if the drive is in fact on a network, or zero if it is not.

Comments:

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

Drive$ may be upper or lower case, and only the first character is considered. If Drive$ is null, the current default drive is tested. NetDrive requires DOS 3.10 or later, and returns 0 (not network) with no other error if the DOS version is less than that.

The following complete example program shows NetDrive in context:

```
DECLARE FUNCTION NetDrive%(Drive$)
IF NetDrive%(Drive$) THEN
  PRINT Drive$; " is a remote network drive"
ELSE
  PRINT Drive$; " is a local non-network drive"
END IF
```

PutVol

assembler subroutine contained in PRO.LIB

Purpose:

PutVol will create a disk volume label for either a specified drive or the current default drive. If a label already exists, PutVol will instead rename it.

Syntax:

```
CALL PutVol(Drive$, Label$)
```

Where:

Drive$ is either an upper or lower case letter for the drive being examined, or a null string to indicate the current default drive. Label$ is then written to the disk's directory as a volume label.

Comments:

Some programmers like to use a disk's volume label as a way to keep track of its contents. One example might be when writing a major accounting program that must also work with a floppy disk system. You could then label the customer file disk, say, "CUSTOMER", and the inventory disk "INVENTORY". If a user inadvertently puts the wrong disk in the drive, your program could know that and prompt for the correct one.

A disk volume label may be up to eleven characters in length.

Also see the companion program GetVol, which will obtain the volume label from a disk.

QBLoad

assembler subroutine contained in PRO.LIB

Purpose:

QBLoad serves the same purpose as BASIC's BLOAD command, but it returns an error code rather than requiring the use of ON ERROR.

Syntax:

```
CALL QBLoad(FileName$, SEG Array%(Element))
```

or

```
CALL QBLoad(FileName$, BYVAL Segment%, BYVAL Address%)
```

Where:

FileName$ is the name of a file that had been previously saved using either QBSave or BASIC's BSAVE.

To load the file into Array(Element) specify the array element as shown, or use the Segment% and Address% method to load it into memory at any location.

Comments:

The only errors that are likely to be encountered with QBLoad would be specifying a file that doesn't exist, or giving an invalid drive letter or path. Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

Two important uses for QBLoad and BASIC's BLOAD are saving and loading graphics or text screen images, or reducing the time needed to load a numeric array. Saving and loading screens is covered in depth in the appendix of this manual, and manipulating arrays this way is discussed in the description of the QBSave routine.

QBSave

assembler subroutine contained in PRO.LIB

Purpose:

QBSave serves the same purpose as BASIC's BSAVE command, but it returns an error code rather than requiring the use of ON ERROR.

Syntax:

```
CALL QBSave(FileName$, SEG Array%(Element), NumBytes%)
```

or

```
CALL QBSave(FileName$, BYVAL Segment%, BYVAL Address%, NumBytes%)
```

Where:

FileName$ is the name of the file, and NumBytes% indicates how many bytes to save. NumBytes may also be a long integer if more than 32,767 bytes will be saved.

To save a file from Array%(Element%) specify the array element as shown, or use the Segment% and Address% method to save it from any memory location.

Comments:

One important use of both QBSave and QBLoad is to overcome a bug in BLOAD and BSAVE with early versions of QuickBASIC 4. However, Microsoft has fixed this problem in version 4.00b.

The only errors that are likely to occur when using QBSave is giving it an illegal file name, specifying an invalid drive letter or path, or not having sufficient room on the disk.

Errors may be detected with the QuickPak Professional DOSError and WhichError functions.

QuickDOS

BASIC program contained in QD.BAS

Purpose:

Unlike most of the QuickPak Professional routines that are intended to be added to your programs, QuickDOS is a complete stand-alone DOS utility program. It allows you to mark a group of files for copying, deleting, or moving to a new disk or directory. The files that are displayed may be sorted by name, extension, date, or size, and an entire directory may be marked or unmarked with a single key press.

Most of QuickDOS' features will be obvious as soon as it is run, however you will have to compile it to a stand-alone program as shown at the beginning of the source listing. Instructions are given for both QuickBASIC 4 and the BASCOM 6 version of the BC.EXE compiler.

QuickDOS is fully "mouse aware", and it lets you control all aspects of the program's operation, except for entering and editing file names and directory specifications. However, there are a few hidden features you should also be aware of.

Pressing Alt-R when an .EXE, .COM, or .BAT file name is highlighted tells QuickDOS to run the program or batch file. Even though QuickBASIC programs can normally run only other .EXE files, the StuffBuf routine is used to place the .COM or batch file's name into the keyboard buffer. Thus when QuickDOS subsequently ends, the program is run as if you had entered the name manually. Notice that this does *not* work if you are running QuickDOS from within the QuickBASIC editor.

Alt-S is similar to Alt-R, except it tells QuickDOS to SHELL and run the currently marked program, and then return.

The last hidden feature is Alt-F, which creates a file that contains a list of the names of all the marked files. This is extremely useful for creating a LINK or LIB response file. Once the list file has been created, simply edit it to add the appropriate response file punctuation. Creating response files is discussed in detail in this manual under the section entitled "Response Files".

QuickDOS will also accommodate any of the text screen modes. That is, if the screen is displaying 43 or 50 lines when QuickDOS is started, the size of the vertical menu and the range of mouse movements will be adjusted automatically.

Finally, QuickDOS recognizes several command line arguments:

/D	sort files by date and show full file information
/E	sort files by extension
/H	use maximum number of lines possible with EGA/VGA
/N	sort files by name
/S	sort files by size
C:\*.*	or any other valid file specification
?	display a list of all command line options
HELP	same as ?

ReadDir

assembler subroutine contained in PRO.LIB

Purpose:

ReadDir obtains a list of directory names from disk, and loads them into a conventional (not fixed-length) string array in one operation.

Syntax:

```
CALL ReadDir(BYVAL VARPTR(Array$(0)))
```

Where:

Array$(0) holds the search specification to indicate which directory names are to be loaded, and subsequent array elements receive each directory name.

Comments:

It is essential that enough elements have been set aside in the string array to hold all of the anticipated directory names. Further, each element must first be assigned to a length of at least twelve spaces to reserve room for each name. All of the steps needed to obtain a list of directory names are outlined below.

Before the array can be dimensioned to hold the directory names, you will need some way to determine how many there are. The easiest way to do this is with the DCount function. DCount accepts the same type of search specification you will give to ReadDir, and it tells how many matching directories there are.

The next step is to dimension the array to the number of elements DCount returned. Finally, space must be set aside in each element to hold the names. Including a possible extension, a directory name may be as long as twelve characters.

Most people think of the DOS wild cards (? and *) as being applicable only to file names, however they are also intended to be used with directory names. For example, to read a list of directory names that begin with the letter "A" and are located under the root directory, you would specify a search specification of "\A*.*" when using DCount and ReadDir.

The program fragment below shows how to obtain a list of all directories under the current directory of drive C.

```
Spec$ = "C:*.*"                    '*.* matches any directory name
Count = DCount%(Spec$)             'see how many there are
DIM Array$(0 TO Count)             'create the string array

FOR X = 1 TO Count                 'fill each element with spaces
   Array$(X) = SPACE$(12)
NEXT

Array$(0) = Spec$                  'put the spec in element 0
CALL ReadDir(BYVAL VARPTR(Array$(0)))   'call ReadDir

FOR X = 1 TO COUNT                 'print them to show it worked
   PRINT Array$(X)
NEXT
```

A complete directory searching program is provided in the file named READDIRS.BAS. READDIRS will search through all of the directories on a disk—no matter how deeply nested—and display all of the files that match a given specification. READDIRS is similar to the various "WHEREIS" programs available in the public domain, and it illustrates how recursion can be used to advantage in such a situation.

Also see the related routine ReadDirT that reads a list of directory names into a fixed-length string or TYPE array.

ReadDirs

BASIC example program contained in READDIRS.BAS

Purpose:

Unlike most of the QuickPak Professional subroutines that are called by your BASIC programs, ReadDirs is intended to serve as an example for calling the various DOS services. It is also a very handy utility program.

ReadDirs first asks for a file specification such as "C:*.*" or "*.BAS", and then searches all of the directory levels on the disk for files that match. When running ReadDirs, do not give it a directory name for the files, because *all* of the disk's directories will be searched automatically.

ReadDirs brings together a number of important DOS routines that are provided with this package, and provides examples for using them in context. It also illustrates how recursion can be used to advantage when navigating tree structured data such as a disk's directory.

ReadDirT

assembler subroutine contained in PRO.LIB

Purpose:

ReadDirT obtains a list of directory names from disk, and loads them into a fixed-length string array in one operation.

Syntax:

```
CALL ReadDirT(Spec$, BYVAL VARSEG(Array$(1)), BYVAL VARPTR(Array$(1)))
```

or

```
CALL ReadDirT(Spec$, SEG Array(1))
```

Where:

Array$() has been dimensioned as a fixed-length string array, and Spec$ holds the search specification to tell which directory names are to be loaded. Subsequent array elements then receive each directory name.

Comments:

It is essential that enough elements have been set aside in the string array to hold all of the anticipated directory names. Further, the size of each element must be exactly twelve, to reserve room for each name.

The steps needed to read directory names into a fixed-length string array are similar to those for a conventional string array, as outlined in the example for ReadDir. However, there are some differences, and a modification of the code shown in the ReadDir example is given below.

```
Spec$ = "C:*.*"
Count = DCount%(Spec$)
DIM Array(1 TO Count) AS STRING * 12

CALL ReadDirT(Spec$, BYVAL VARSEG(Array$(1)), BYVAL VARPTR(Array$(1)))

FOR X = 1 TO Count
   PRINT Array$(X)
NEXT
```

You should be aware that it is also possible to call ReadDirT without having to deal with the BYVAL/VARSEG/VARPTR nonsense, and simply give it the starting array element. The SEG call option allows an entire array to be passed to an assembler routine. However, a "design decision" at Microsoft prevents this from working with fixed-length string arrays. The key is to create a TYPE consisting solely of a single string member, and then pass the TYPE element with SEG.

To use this method you will need to declare ReadDirT with the "SEG varname AS ANY" option as shown below.

```
DECLARE SUB ReadDirT(Spec$, SEG Element AS ANY)

TYPE FLen
   S AS STRING * 12
END TYPE

Spec$ = "C:*.*"
Count = DCount%(Spec$)
DIM Array(1 TO Count) AS FLen

CALL ReadDirT(Spec$, Array(1))

FOR X = 1 TO Count
   PRINT Array(X).S
NEXT
```

See the section entitled "Calling With Segments" in Chapter 1, for a complete discussion of passing fixed-length string and TYPE arrays to assembler routines.

ReadFile

assembler subroutine contained in PRO.LIB

Purpose:

ReadFile obtains a list of file names from disk, and loads them into a conventional (not fixed-length) string array in one operation.

Syntax:

```
CALL ReadFile(BYVAL VARPTR(Array$(0)))
```

Where:

Array$(0) holds the search specification to indicate which file names are to be loaded, and subsequent array elements receive each name.

Comments:

It is essential that enough elements have been set aside in the string array to hold all of the anticipated file names. Further, each element must first be assigned to a length of at least twelve spaces to reserve room for each name. All of the steps needed to obtain a list of file names are outlined below.

Before the array can be dimensioned to hold the names, you will need some way to determine how many there are. The easiest way to do this is with the FCount function. FCount accepts the same type of search specification you will give to ReadFile, and it tells how many matching names there are.

The next step is to dimension the array to the number of elements FCount returned. Finally, space must be set aside in each element to hold the names. Including a possible extension, a file name may be as long as twelve characters.

The DOS wild cards will most likely be used in the search specification, and an optional drive letter or path may also be included. For example, to read a list of file names that begin with the letter "A" and are located under the root directory, you would specify a search specification of "\A*.*" when using FCount and ReadFile. To obtain a list of all BASIC program files in the current directory you would instead use "*.BAS".

The example program below shows how to obtain a list of all the file names under the current directory of drive C.

```
Spec$ = "C:*.*"                    '*.* matches any file name
Count = FCount%(Spec$)             'see how many there are
DIM Array$(0 TO Count)             'create the string array

FOR X = 1 TO Count                 'fill each element with spaces
   Array$(X) = SPACE$(12)
NEXT

Array$(0) = Spec$                  'put the spec in element 0
CALL ReadFile(BYVAL VARPTR(Array$(0)))  'call ReadFile

FOR X = 1 TO Count                 'print them to show it worked
   PRINT Array$(X)
NEXT
```

A complete directory searching program is provided in the file named READDIRS.BAS. READDIRS will search through all of the directories on a disk—no matter how deeply nested—and display all of the files that match a given specification. READDIRS is similar to the various "WHEREIS" programs available in the public domain, and it illustrates how recursion can be used to advantage in such a situation.

Also see ReadFileT which loads a group of file names into a fixed-length string or TYPE array.

ReadFileI

assembler subroutine contained in PRO.LIB

Purpose:

ReadFileI obtains a list of file names, sizes, dates, and times from disk, and then formats and loads them into a conventional (not fixed-length) string array in one operation.

Syntax:

```
CALL ReadFileI(BYVAL VARPTR(Array$(0)))
```

Where:

Array$(0) holds the search specification to indicate which file names are to be retrieved, and subsequent array elements receive each file's name and related information.

Comments:

ReadFileI (Read File names with full Information) is similar to the QuickPak Professional ReadFile routine, except it also obtains the size, date, and time for each file. Before ReadFileI, the only way to do this was to call the ReadFile routine, and then call FileInfo repeatedly for each file. Besides greatly simplifying the amount of work you must do to obtain this information, ReadFileI is also considerably faster.

Because an assembly language routine cannot create BASIC strings, it is up to you to assign each string to a length of 37 characters before calling ReadFileI. The example below shows all of the steps needed to set up and call ReadFileI.

```
Spec$ = "*.*"                   'or whatever
Count = FCount%(Spec$)          'see how many files there are
DIM Array$(Count)               'make an array to hold the files

FOR X = 1 TO Count              'fill each with blanks
    Array$(X) = SPACE$(37)      '37 are needed for the information
NEXT

Array$(0) = Spec$               'assign search spec to lowest element
CALL ReadFileI(BYVAL VARPTR(Array$(0))) 'call ReadFileI

FOR X = 1 TO Count              'print the files to prove it worked
    PRINT Array$(X)
NEXT
```

The information returned in each element of the array is organized as shown in the table below. The name can be up to twelve characters, the size could hold as many as eight digits, the date is always eight digits, and the time is always six. A single blank space is placed between each field area in the string.

```
1234567890123456789012345678901234567
namename.ext sizesize mm-dd-yy hh:mm* <---- either "a" or "p"
```

Also see the comments that accompany the ReadFile routine for more information about the search specification and the FCount function.

ReadFileI is demonstrated in the READFILI.BAS example program.

ReadFileT

assembler subroutine contained in PRO.LIB

Purpose:

ReadFileT obtains a list of file names from disk, and loads them into a fixed-length string array in one operation.

Syntax:

```
CALL ReadFileT(Spec$, BYVAL VARSEG(Array(1)), BYVAL _
    VARPTR(Array(1)))
```

or

```
CALL ReadFileT(Spec$, SEG Array(1))
```

Where:

Array() has been dimensioned as a fixed-length string array, and Spec$ holds the search specification to tell which file names are to be loaded. Subsequent array elements then receive each name.

Comments:

It is essential that enough elements have been set aside in the string array to hold all of the anticipated file names. Further, the size of each element must be exactly twelve, to reserve room for each name.

The steps needed to read file names into a fixed-length string array are similar to those for a conventional string array, as outlined in the example for ReadFile. However, there are some differences, and a modification of the code shown in the ReadFile example is given on the following page.

```
Spec$ = "C:*.*"
Count = FCount%(Spec$)
DIM Array(1 TO Count) AS STRING * 12

CALL ReadFileT(Spec$, BYVAL VARSEG(Array$(1)), BYVAL _
     VARPTR(Array$(1)))

FOR X = 1 TO Count
   PRINT Array$(X)
NEXT
```

You should be aware that it is also possible to call ReadFileT without having to deal with the BYVAL/VARSEG/VARPTR nonsense, and simply give it the starting array element. The SEG call option allows an entire array to be passed to an assembler routine. However, a "design decision" at Microsoft prevents this from working with fixed-length string arrays. The key is to create a TYPE consisting solely of a single string member, and then pass the TYPE element with SEG.

To use this method you will need to declare ReadFileT with the "SEG varname AS ANY" option, and then call it as shown below.

```
DECLARE SUB ReadFileT(Spec$, SEG Element AS ANY)
TYPE FLen
   S AS STRING * 12
END TYPE

Spec$ = "C:*.*"
Count = FCount%(Spec$)
DIM Array(1 TO Count) AS FLen

CALL ReadFileT(Spec$, Array(1))

FOR X = 1 TO Count
   PRINT Array(X).S
NEXT
```

As you can see, the SEG method of calling ReadDirT is simpler than using BYVAL VARSEG() and BYVAL VARPTR(), however it works only if the fixed-length string array has been dimensioned as a TYPE array. A complete discussion of passing fixed-length string and TYPE arrays is given in Chapter 1, under the section entitled "Calling With Segments".

ReadFileX

assembler subroutine contained in PRO.LIB

Purpose:

Like ReadFileI, ReadFileX (Read File names Extended) obtains a list of file names, sizes, dates, and times from disk, except it places these into separate components of a TYPE array.

Syntax:

```
CALL ReadFileX(Spec$, DirSize&, SEG Array(1))
```

Where:

Spec$ holds the search specification to indicate which file names are to be retrieved, DirSize& receives the total of all the file sizes, and elements in the array receive each file's name and related information.

Comments:

ReadFileX expects you to pass to it a user-defined TYPE array that has been dimensioned sufficiently for the number of files that will be read. This array must be structured and dimensioned as follows:

```
TYPE FullInfo                     'this is the TYPE definition
    BaseName AS STRING * 8
    ExtName  AS STRING * 3
    FileSize AS LONG
    FileDate AS STRING * 8
    FileTime AS STRING * 6
    Attrib   AS STRING * 1
END TYPE
Count = FCount%(Spec$)            'use FCount to count the files
DIM Array(1 TO Count) AS FullInfo 'dimension the array
```

Please see the comments that accompany the description for ReadFileI elsewhere in the manual.

ReadFileX is demonstrated in the READFILX.BAS example program.

ReadSect

assembler subroutine contained in PRO.LIB

Purpose:

ReadSect will read the contents of any disk sector into a conventional (not fixed-length) string.

Syntax:

```
CALL ReadSect(Drive$, Info$, Sector%)
```

Where:

Drive$ is an upper or lower case letter specifying the disk drive to read from. Unlike most of the other QuickPak Professional DOS routines, a null string may *not* be used to indicate the default drive.

Info$ is a string already containing at least 512 characters or spaces that receives the sector contents, and Sector% indicates the sector number to read.

Comments:

To accommodate very large hard disks, the sector number is given as an "unsigned" value. For sectors between 0 and 32767, simply assign the number to Sector% and call ReadSect. But for sectors beyond the normal range of an integer variable, you must instead use a long integer. In fact, a long integer variable may be used in either case. To specify a long integer *constant*, simply append a trailing ampersand (&) to the value, as shown below:

```
CALL ReadSect(Drive$, Info$, 43800&)
```

Errors may be detected with the QuickPak Professional DOSError and WhichError functions. Notice that ReadSect is *not* intended for use with network drives.

A complete sector reading utility is provided in the file READSECT.BAS, and the program DEMOSECT.BAS, shows how it may be called.

ReadTest

assembler function contained in PRO.LIB

Purpose:

ReadTest will report whether a specified disk drive is ready for reading.

Syntax:

```
Okay = ReadTest%(Drive$)
```

Where:

Drive$ is either an upper or lower case letter that represents the disk drive to check, or a null string to indicate the current default drive. Okay is then assigned -1 if the drive is ready, or 0 if it is not.

Comments:

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

ReadTest does not check to see if a valid drive letter has been specified. It only tests to see that a disk is in the drive and the door is closed. If you also need to determine whether the drive letter is valid, you should first use the GoodDrive function.

A complete example showing ReadTest and its companion routine WriteTest is given in the RWTEST.BAS demonstration program.

Removable

assembler function contained in PRO.LIB

Purpose:

Removable reports if a given drive's media is removable (a floppy drive).

Syntax:

```
Floppy = Removable%(Drive$)
```

Where:

Drive$ is a letter that specifies the drive to test, or a null string to examine the current drive. Floppy then receives -1 if the drive is in fact meant to hold a floppy disk, or zero if it is not.

Comments:

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

Drive$ may be upper or lower case, and only the first character is considered. If Drive$ is null, the current default drive is tested.

Removable requires DOS 3.00 or later, and returns unpredictable values with no other error if the DOS version is less than that. Therefore, you should use DOSVer manually if there is a chance the host PC is not using DOS version 3.00 or later.

The following complete example program shows Removable in context:

```
DECLARE FUNCTION Removable%(Drive$)
IF Removable%(Drive$) THEN
  PRINT Drive$; " is a removable floppy drive"
ELSE
  PRINT Drive$; " is a hard disk or RAM disk or network drive"
END IF
```

ScanFile

BASIC function contained in SCANFILE.BAS

Purpose:

ScanFile will quickly scan through a specified file looking for a particular text string.

Syntax:

```
Found = ScanFile&(FileName$, Text$, Start&)
```

Where:

FileName$ is the file to be examined, Text$ is the text to find, Start& is the starting offset at which to begin searching, and Found receives the offset at which the text was located. Text$ may contain any number of "?" wild cards, and searching is case insensitive.

If the text is not located, then Found will receive 0. If a disk error occurs, or an invalid path, drive, or file name is given, Found instead receives -1.

Comments:

ScanFile is written in BASIC, however the actual reading of the file is performed by the QuickPak Professional binary file access routines.

Even though ScanFile returns a -1 if an error occurs, the QuickPak Professional DOSError and WhichError functions may also be examined for errors.

ScanFile searches without regard to capitalization, but comments in the source code to SCANFILE.BAS show how this may be changed. Search for "QInstr2" to find where it is used, and replace that with QInstr.

The position in the file that is returned by ScanFile is based at one. That is, if a match is found at the very first byte in the file, ScanFile will return 1, and not 0.

The starting offset parameter is used to let you continue a search, or examine only a portion of a file. For example, the first time ScanFile is used, Start& should be set to 1. If a match is found but it is not the correct one, you would invoke ScanFile a second time with the starting offset set to the offset it first returned.

Calling ScanFile successively this way is amply illustrated by the DEMOSCAN.BAS example program.

SearchPath

BASIC function contained in SRCHPATH.BAS

Purpose:

SearchPath$ accepts the name of any file and returns its fully qualified name by searching the DOS PATH.

Syntax:

```
PathName$ = SearchPath$(FileName$)
```

Where:

FileName$ is a file name such as "FORMAT.COM" and PathName$ receives the program's full path name, for example "\DOS\FORMAT.COM". If a drive letter is included in the user's PATH setting that will be returned as well.

Comments:

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

SearchPath begins by looking in the current directory for the named file, and if not found it then examines all of the directories listed in the DOS PATH. If no file extension is given, then as SearchPath searches each directory it also looks for .COM, .EXE, and .BAT in that order. This is the same search order that DOS uses when running executable programs.

See LOADEXEC.BAS for a demonstration using SearchPath in context.

SetAttr

assembler subroutine contained in PRO.LIB

Purpose:

SetAttr sets the attribute byte for a specified file.

Syntax:

```
CALL SetAttr(FileName$, Attribute%)
```

Where:

FileName$ is the file being modified, and Attribute% is bit coded with the attributes to set it to.

Comments:

Every file has an attribute that is assigned at the time it is created. The attribute information is kept in a disk's directory, along with each file's name, date, and time.

The table below shows some common values for the Attribute% variable:

1	Read-Only
2	Hidden
32	Archive
0	turns off all attributes

A complete description of file attributes is given in the section that describes the GetAttr function. Also, a working example of setting a file's attributes is contained in the SETATTR.BAS demonstration program.

SetCmd

assembler subroutine contained in PRO.LIB

Purpose:

SetCmd lets you establish a new COMMAND$ argument for a program that is subsequently run or chained to.

Syntax:

```
CALL SetCmd(NewCommand$)
```

Where:

NewCommand$ is the string the next program will receive as COMMAND$.

Comments:

SetCmd requires DOS 3.0 or later, and you should use the DOSVer function to check the DOS version because SetCmd does not return an error.

The length of the new COMMAND$ passed to SetCmd must be 125 characters or less; however, leading blanks will be trimmed and not included in that count.

SetDrive

assembler subroutine contained in PRO.LIB

Purpose:

SetDrive allows changing the current default drive.

Syntax:

```
CALL SetDrive(Drive$)
```

Where:

Drive$ is either an upper or lower case letter of the drive to be made the current default.

Comments:

SetDrive is extremely simple to set up and call. Besides using a string variable to specify the new drive letter, you may also give it a string literal:

```
CALL SetDrive("A")
```

or

```
CALL SetDrive("b")
```

You may use the QuickPak Professional DOSError and WhichError functions to determine if the new drive was valid.

SetError

assembler subroutine contained in PRO.LIB

Purpose:

SetError allows a BASIC program to set or clear the DOSError and WhichError functions.

Syntax:

```
CALL SetError(ErrCode%)
```

Where:

ErrCode% is either zero to clear DOSError and WhichError, or an error value to place into WhichError. If ErrCode% is not zero, then DOSError will be set to -1.

Comments:

You probably will not need to set the DOSError and WhichError function values in your programs. However, this capability can be important when you are creating your own DOS services in BASIC. For example, the QuickPak Professional ScanFile function uses SetError to indicate whether the file was read successfully.

SetLevel

assembler subroutine contained in PRO.LIB

Purpose:

SetLevel allows a QuickBASIC program to set the DOS error level.

Syntax:

```
CALL SetLevel(ErrValue%): END
```

Where:

ErrValue% is a value between 1 and 255.

Comments:

Most utility programs use the DOS error level to indicate to a batch file if they terminated successfully. This may then be tested in the batch file with "IF ERRORLEVEL x action", where x is the error level, and action is what to do.

SetLevel has no meaningful effect within the QuickBASIC editing environment — you must compile your program to an .EXE file before it will work. Also, because SetLevel installs itself temporarily as a TSR routine that intercepts each DOS interrupt, we suggest that you not call it until you are ready to end your program.

ShareThere

assembler function contained in PRO.LIB

Purpose:

ShareThere reports if SHARE is installed in the host PC.

Syntax:

```
ShareIsActive = ShareThere%
```

Where:

ShareIsActive receives -1 if SHARE is currently installed, or zero if it is not.

Comments:

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

ShareThere is useful if you are writing a network program, because BASIC will report an error if SHARE (or its equivalent in the network software) is not available when a file is opened for shared access.

A typical usage of ShareThere is as follows:

```
DECLARE FUNCTION ShareThere% ()
IF ShareThere% THEN
  OPEN "ACCOUNTS.DAT" FOR RANDOM SHARED AS #1 LEN = RecLength%
ELSE
  PRINT "Please quit and run the SHARE program"
  END
END IF
```

As with many of the other QuickPak Professional functions, ShareThere uses -1 to indicate True and 0 for False, so you can use BASIC's NOT:

```
IF NOT ShareThere% THEN ...
```

SplitName

assembler subroutine contained in PRO.LIB

Purpose:

SplitName parses out the components in a file name, and returns the drive letter, path name, file name, and extension as separate items.

Syntax:

```
CALL SplitName(WorkName$, Drive$, Path$, FileName$, Extension$)
```

Where:

WorkName$ is a complete file name such as "C:\MYPATH\YOURPATH\MYFILE.EXT". SplitName then returns Drive$ as "C:", Path$ as "\MYPATH\YOURPATH\", FileName$ as "MYFILE", and Extension$ as ".EXT".

Comments:

If there is no drive, the current default is used. And if no path is specified, the current directory is returned. This lets you know everything about the file in one operation.

For assembly language buffs, notice that the same exact assembler source code is used for both the near and far string versions. This is achieved by calling BASIC internal routines exclusively.

SPLITNAM.BAS provides a demonstration program that shows how SplitName works.

Unique

BASIC function contained in UNIQUE.BAS

Purpose:

Unique will return a file name that does not already exist on the default drive in the current directory.

Syntax:

```
FileName$ = Unique$(Path$)
```

Where:

FileName$ receives a unique file name. Path$ is the directory path where a unique name is to be found. If Path$ is null, the current directory is used.

Comments:

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

There are several situations where a program will need to create a temporary file with a unique name. For example, you might want to pass information to a subsequently run program, or perhaps save the intermediate passes of a multiple pass file sort.

Beginning with DOS 3, a similar system service was introduced to create a unique file name. Unique creates a file name based on the system time kept in low memory, and works with all versions of DOS 2.x and later.

The Path$ argument lets you ensure that the file name returned does not already exist in the specified directory.

UnLockFile

assembler subroutine contained in PRO.LIB

Purpose:

UnLockFile will unlock all or a portion of a network file in the same way that BASIC's UNLOCK will, but without needing ON ERROR.

Syntax:

```
CALL UnLockFile(Handle%, Offset&, Length&)
```

Where:

Handle% is the handle that DOS assigned when the file was first opened, Offset& is the starting offset into the file where the unlocking is to begin, and Length& is the number of bytes to unlock.

Comments:

Only two errors are likely when using UnLockFile, and these may be determined by examining the DOSError and WhichError functions.

To use UnLockFile, you must provide a file handle, as well as the range of bytes in the file. These are easily determined based on the record number to be unlocked, and the length of each record. Long integer values are used because you may also need to unlock a large range of records, or even the entire file. Of course, the values to unlock a large file cannot be represented by a conventional integer variable. Notice that if the file is successfully unlocked, it will not be necessary to do it again in BASIC.

Please see the comments and warnings that accompany the description for LockFile.

See the LOCKFILE.BAS demonstration program for an example of how to calculate the Offset& and Length& parameters.

Valid

assembler function contained in PRO.LIB

Purpose:

Valid examines a string to see if it could be a valid file name.

Syntax:

```
Okay = Valid%(FileName$)
```

Where:

FileName$ contains the name of a file, and may optionally include a drive letter and colon, as well as a path name. If the name is legal according to the rules of DOS, Okay will receive -1. Otherwise Okay will be set to zero.

Comments:

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

Valid provides a very quick way to determine if a DOS file name is syntactically correct. However, it does *not* check to see if the named file, drive, and path actually exist. Rather, it simply reports if the name *could* be valid.

Notice that Valid considers blank spaces to be illegal, so you should use LTRIM$ and RTRIM$ if there is any possibility that the name may contain leading or trailing blanks:

```
Okay = Valid%(LTRIM$(RTRIM$(FileName$)))
```

Also notice that Valid follows the rules of DOS exactly. That is, a filename may be longer than eight characters, and an extension may be longer than three. In those cases, DOS will truncate the excess.

WhichError

assembler function contained in PRO.LIB

Purpose:

WhichError reports which error if any occurred during the last call to a QuickPak Professional DOS routine.

Syntax:

```
IF DOSError% THEN PRINT WhichError% "occurred"
```

Where:

WhichError% returns 0 if there was no error, or an error code if there was.

Comments:

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

All of the QuickPak Professional routines assign a value to the DOSError and WhichError functions to indicate their success or failure. Rather than requiring you to set up a separate error handling procedure and use ON ERROR, you can simply query these functions after performing any QuickPak Professional DOS operation. WhichError is discussed in the section entitled "Eliminating ON ERROR".

The table on the following page shows all of the possible DOS errors and the corresponding numbers that WhichError returns.

Also see the complimentary function DOSError.

Table 3-3
QuickPak Professional Error Codes

Err	Description	Typical Situation
7	Out of memory	FastLoadInt
14	Out of string space	FastLoadInt/FastSave
25	Device fault	Disk write error
27	Out of paper	Printer error
52	Invalid file handle	File access
53	File not found	File access
57	Device I/O error	Disk not formatted
58	File/Path already exists	NameFile or NameDir
61	Disk is full	File writes, opens, etc.
62	Input past end of file	File reads
64	Bad file name	Filename null or too long
67	Directory is full	FCreate, MakeDir
68	Device unavailable	Unknown drive
70	Permission denied	Disk write-protected
71	Disk not ready	Drive door open
72	Disk media error	Bad disk sector
73	Advanced feature not available	LockFile with DOS 2.x
74	Rename across disks	NameFile
75	Access denied	Read-only files
76	Path not found	Invalid drive or path
77	Invalid drive spec	DiskInfo, SetDrive
78	Too many handles	FOpen
79	Bad FAT image	WriteSect to FAT
80	Invalid time data	FStamp
81	Invalid date data	FStamp
82	Invalid parameter	File handles, etc.
83	Buffer too small	LineCount
84	Current directory has been renamed	NameDir
85	Lock conflict	LockFile
86	Sharing conflict	FOpenAll
87	Read-only conflict	FCreate
100	Insufficient number of elements	String Restore
127	Undefined error	

WriteSect

assembler subroutine contained in PRO.LIB

Purpose:

WriteSect will write new contents to any disk sector from either a conventional or fixed-length string.

Syntax:

```
CALL WriteSect(Drive$, Info$, Sector%)
```

Where:

Drive$ is an upper or lower case letter specifying the disk drive to write to. Unlike most of the other QuickPak Professional DOS routines, a null string may *not* be used to indicate the default drive.

Info$ is a string that contains the new sector contents to be written (512 bytes), and Sector% indicates the sector number.

Comments:

WARNING:

Like ReadSect, this is a very powerful routine—in fact, perhaps a bit too powerful! It will write directly to any disk sector you specify, filling it with the contents of the stated string.

Be warned, WriteSect has the potential to totally devastate a disk. *Please be careful!* If you aren't sure how to use WriteSect, then you have no business fooling around with it. We have thoroughly tested this routine, however you are on your own. If you trash your hard disk, forget where you got this program.

To accommodate very large hard disks, the sector number is given as an "unsigned" value. For sectors between 0 and 32767, simply assign the number to Sector% and call WriteSect. But for sectors beyond the normal range of an integer variable (up to 65,535), you must instead use a long integer. In fact, a long integer variable may be used in either case.

To specify a long integer *constant*, simply append a trailing ampersand (&) to the value, as shown below:

```
CALL WritSect(Drive$, Info$, 43800&)
```

Errors may be detected with the QuickPak Professional DOSError and WhichError functions. Notice that WriteSect is *not* intended for use with network drives.

WriteSect2

assembler subroutine contained in PRO.LIB

Purpose:

WriteSect2 will write new contents to one or more disk sectors from either a conventional or fixed-length string.

Syntax:

```
Call WriteSect2(Drive$, Info$, Sector%)
```

Where:

Drive$ is an upper or lower case letter specifying the disk drive to write to. Unlike most of the QuickPak Professional DOS routines, a null string may *not* be used to specify the default drive.

Info$ is a string that contains the new sector contents, and Sector% indicates the starting sector number. A long integer may be used to specify sectors numbered higher than 32767.

Comments:

WriteSect2 is similar to WriteSect, except it uses the length of Info$ to determine how many sectors to write at one time. For example, if Info$ is 512 characters long, one sector will be written. Likewise, filling Info$ with 2048 bytes tells WriteSect to write four sectors. We created this special version of WriteSect for our own use in a fast disk copying program, and thought you might find it useful for a similar purpose.

Please see the warnings and comments that accompany the original WriteSect routine.

WriteTest

assembler function contained in PRO.LIB

Purpose:

WriteTest will report whether a specified disk drive is ready for writing.

Syntax:

```
Okay = WriteTest%(Drive$)
```

Where:

Drive$ is either an upper or lower case letter that represents the disk drive to check, or a null string to indicate the current default drive. Okay is then assigned -1 if the drive is ready, or 0 if it is not.

Comments:

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

WriteTest does not check to see if a valid drive letter has been specified. It only tests if a disk is in the drive, is not write-protected, and the drive door is closed. If you also need to determine whether the drive letter is valid, you should first use the GoodDrive function.

WriteTest will also not detect if a disk is merely full. If you need to insure against insufficient disk space, you should use the DiskRoom function after WriteTest.

Notice that WriteTest creates a temporary file named "LIKE-WOW.MAN" on the disk being tested. In the unlikely event that a file with the same name is already present, it will be overwritten in the process.

A complete example showing WriteTest and its companion routine ReadTest in context is given in the RWTEST.BAS demonstration program.

Chapter 4
Functions

Bin2Num

assembler function contained in PRO.LIB

Purpose:

Bin2Num accepts a binary number in the form of a string, and returns an equivalent value.

Syntax:

```
Number = Bin2Num%(Binary$)
```

Where:

Binary$ is a string containing only the characters "1" and "0", and Number receives its value.

Comments:

Because Bin2Num is designed as a function, it must be declared before it may be used.

Bin2Num is designed as an integer function, therefore the values it returns will be considered to be "signed". That is, BASIC always considers integer values larger than 32767 as negative numbers.

To convert a signed number to its equivalent unsigned value, simply add 65536 as shown below:

```
Value& = Bin2Num%("1101111011000101")
IF Value& <0 THEN Value& = Value& + 65536
```

Also see the related functions Num2Bin and Num2Bin2.

C2F

BASIC function contained in FNOTHER.BAS

Purpose:

C2F will convert a Celsius temperature to its Fahrenheit equivalent.

Syntax:

```
FTemp! = C2F!(CTemp!)
```

Where:

CTemp! is a valid Celsius temperature, and FTemp! receives an equivalent in degrees Fahrenheit.

Comments:

Also see the companion function F2C! that converts from Fahrenheit to Celsius.

Delimit

BASIC function contained in FNOTHER.BAS

Purpose:

Delimit counts the number of delimiters in a string, by matching against a second string that contains a table of valid delimiters.

Syntax:

```
Count = Delimit%(Work$, Table$)
```

Where:

Work$ is a string such as the current DOS PATH or COMMAND$, Table$ contains a list of acceptable delimiters, and Count receives the total number of matching delimiters found in Work$.

Comments:

Delimit is intended to be used in conjunction with the Parse function, which extracts individual items from a list and places them into an array. Delimit merely counts the number of delimiters in a string, based on a table you provide.

For example, to isolate the various components of a DOS PATH (obtained with BASIC's ENVIRON$ command), you would use the semi-colon (;) as a delimiter. However, Delimit will also accept a table of possible delimiters, and report how many are contained in the string being examined.

If you need to isolate all of the possible COMMAND$ switches a user gave when starting your program, you would probably include several delimiting characters, as shown below.

```
Count = Delimit%(COMMAND$, "/,-+ ")
```

An example of obtaining each item in the DOS PATH is given in the demonstration portion of FNOTHER.BAS, along with comments showing how to parse COMMAND$.

Eval

BASIC function contained in FNOTHER.BAS

Purpose:

Eval will return the value of a string like BASIC's VAL function, but without regard to dollar signs, commas, or any other punctuation.

Syntax:

```
Value = Eval#(Number$)
```

Where:

Number$ contains a string of digits and possibly punctuation such as commas and dollar signs, and Value receives its actual value.

Comments:

Eval simply steps through all of the characters in a string discarding those that are not valid digits. However, it does employ some intelligence in that only one minus sign or decimal point will be recognized.

ExpandTab

BASIC function contained in FNOTHER.BAS

Purpose:

ExpandTab accepts an incoming text string that contains embedded CHR$(9) tab characters, and replaces them with an appropriate number of CHR$(32) spaces.

Syntax:

```
Expanded$ = ExpandTab$(Original$, NumSpaces%)
```

Where:

Original$ is the original string that contains Tabs, NumSpaces% indicates where the tab stops are located, and Expanded$ receives the expanded result.

Comments:

Rather than simply replace each tab character with a specified number of spaces, ExpandTab takes into account the physical position of the tabs within the string.

The true purpose of a tab is to position the cursor (or a print head) to the next tab position. Therefore, the number of blanks that are being substituted will vary depending on the location of the tab within the string.

ExpandTab takes this into account as it expands the string. Further, you may specify at what intervals the tab stops are to be located. However, tab stops are usually placed at every eighth position.

Also see the companion function ShrinkTab.

F2C

BASIC function contained in FNOTHER.BAS

Purpose:

F2C will convert a Fahrenheit temperature to its Centigrade equivalent.

Syntax:

```
CTemp! = F2C!(FTemp!)
```

Where:

FTemp! is a valid Fahrenheit temperature, and CTemp! receives an equivalent in degrees Centigrade.

Comments:

Also see the companion function C2F! that converts from Centigrade to Fahrenheit.

LastFirst

BASIC function contained in FNOTHER.BAS

Purpose:

LastFirst will reverse the position of a first name and last name in a string such that the last name comes before the first.

Syntax:

```
NewName$ = LastFirst$(OldName$)
```

Where:

OldName$ is in the form “John Smith”, and NewName$ is converted to “Smith, John”.

Comments:

Only one possible problem you may encounter when using LastFirst is when the name contains a suffix such as “Jr.”. We could have trapped for that case, but that still wouldn’t prevent a “Sr.” or “Esq.” from slipping through. Further, testing many cases would slow down the function.

Consider this one as a foundation, and simply add any code you feel is necessary to handle the types of names you anticipate. However, LastFirst will correctly handle three word names such as “John A. Smith” or “John Anthony Smith”.

Also see the companion function LastLast that places a last name after the first.

LastLast

BASIC function contained in FNOTHER.BAS

Purpose:

LastLast will reverse the position of a first name and last name in a string such that the last name comes after the first.

Syntax:

```
NewName$ = LastLast$(OldName$)
```

Where:

OldName$ is in the form “Smith, John”, and NewName$ is then assigned “John Smith”.

Comments:

Also see the companion function LastFirst that places a last name before the first.

Num2Bin and Num2Bin2

assembler functions contained in PRO.LIB

Purpose:

Num2Bin will convert a number into an equivalent binary string with a fixed length of 16 digits. Num2Bin2 performs the same function, but returns only as many digits as required to represent the number.

Syntax:

```
Binary$ = Num2Bin$(Number%)
```

Where:

Number% is an integer value, and Binary$ receives an equivalent string containing only ones and zeros.

Comments:

Because Num2Bin and Num2Bin2 are designed as functions, they must be declared before they may be used.

Num2Bin and Num2Bin2 are designed as integer functions, therefore the values they accept will be considered to be "signed". That is, BASIC always considers integer values larger than 32767 as negative numbers.

Also see the complementary function Bin2Num.

Pad

BASIC function contained in FNOTHER.BAS

Purpose:

Pad will add leading zeros to a number, padding it to a specified number of digits.

Syntax:

```
Padded$ = Pad$(Number!, Digits%)
```

Where:

Number! is any single precision value, Digits% specifies how many total digits are to be used, and Padded$ receives the result. If the number of digits is too few to accommodate the number, Pad$ will append a leading percent sign.

Comments:

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

Pad$ is designed to expect a single precision incoming value, but it may be easily changed to either long integer, single, or double precision if you prefer.

Parse

BASIC subprogram contained in FNOTHER.BAS

Purpose:

Parse will extract individual components from a single string, and place each into a separate element of a string array.

Syntax:

```
CALL Parse(Work$, Delim$, Array$())
```

Where:

Work$ contains the string to be parsed, Delim$ holds a list of the delimiters that separate each component, and Array$() receives each item.

Comments:

Parse is intended primarily to be used for isolating the individual components of COMMAND$. It may also be used to determine the various directories in a PATH, or anywhere that a single string contains multiple items. Even though Parse is actually a called subprogram, it seems most appropriate to place it within the file of related functions.

It is up to you to first dimension the string array that will receive the items. This would be done with the complementary function Delimit.

Though Parse could have been designed to dimension the array, that would require the array to be Shared rather than passed as an argument. BASIC subprograms cannot re-dimension an array that has been passed as an incoming parameter, and this is a better approach.

The example below shows how to combine Parse and Delimit to isolate command line switches such as "/T", "-U", ",3", and so forth.

```
Work$ = COMMAND$
Delim$ = "-,;_ "                     'some common delimiters
                                     '(last one is a blank)
X = Delimit%(Work$, Delim$) + 1      'see how many matching
                                     'delimiters there are
REDIM Array$(X)                      '+1 is needed to account
                                     'for the last item
Parse Work$, Delim$, Array$()        'Parse fills the array
FOR X = 1 TO X                       'print each item
    PRINT Array$(X)
NEXT
```

Because Delimit returns the number of delimiters rather than the actual number of items, the string array must be dimensioned to one more than the value returned by Delimit.

ParseStr

BASIC function contained in FNOTHER.BAS

Purpose:

ParseStr accepts an incoming string that contains numbers separated by commas, and returns a new string consisting of the equivalent ASCII characters.

Syntax:

```
Code$ = ParseStr$(List$)
```

Where:

List$ contains a list of numbers in the form:

```
"27, 69, 27, 72"
```

or

```
"27,69,27,72"
```

and Code$ receives the equivalent ASCII characters:

```
CHR$(27) + CHR$(69) + CHR$(27) + CHR$(72)
```

Comments:

ParseStr$ is primarily intended to allow you to accept a string of printer turn-on or turn-off codes that were entered by a user, and quickly convert them to the appropriate string for sending to a printer.

Also see the companion routine UnParseStr.

QPHex

assembler function contained in PRO.LIB

Purpose:

QPHex is a fast replacement for BASIC's HEX$ function, and it also returns a string padded to a specified number of digits.

Syntax:

```
HexNumber$ = QPHex$(Value, NumDigits%)
```

Where:

Value is either an integer or long integer value, and NumDigits% is the number of digits QPHex is to return.

Comments:

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

When declaring QPHex, you should use the AS ANY clause, which allows you to pass either integer or long integer numbers when you use it:

```
DECLARE FUNCTION QPHex$(Value AS ANY, NumDigits%)
```

If the number of digits is between 1 and 4, then QPHex assumes the incoming value is a two-byte integer. Specifying NumDigits% as 5 through 8 instead tells QPHex that the number is a four-byte long integer. The output for two sample uses is shown below:

```
PRINT QPHex$(12, 2)             'this prints "0C"
PRINT QPHex$(123456&, 8)        'this prints "0001E240"
```

Rand

BASIC function contained in FNOTHER.BAS

Purpose:

Rand returns a random number between the specified lower and upper bounds.

Syntax:

```
R = Rand!(Lower!, Upper!)
```

Where:

Lower! is the lowest acceptable number that should be returned, Upper! is the highest, and R receives the random result.

Comments:

One of the problems with using random numbers in a BASIC program is that the numbers aren't really random! Every time the RND() function is invoked, the exact same sequence of numbers is generated.

To overcome this behavior, BASIC allows a program to "seed" the random number generator with a new starting value. One good way to insure that a new sequence is created each time a program runs is to use TIMER as a seed. This is the approach we took in writing Rand.

ShrinkTab

BASIC function contained in FNOTHER.BAS

Purpose:

ShrinkTab reduces the length of a string by replacing groups of blank spaces with CHR$(9) tab characters.

Syntax:

```
Small$ = ShrinkTab$(Original$, NumSpaces%)
```

Where:

Original$ is the original string that contains groups of blank spaces, NumSpaces% indicates where the tab characters are to be placed, and Small$ receives the new string that has been reduced in length.

Comments:

Rather than simply replace each group of spaces with a tab character, ShrinkTab takes into account the physical position at which the tabs should be placed within the string.

The true purpose of a tab is to position the cursor (or a print head) at the next tab location. Therefore, the placement of each tab that is being assigned will vary, depending on the location of the original blanks.

ShrinkTab takes this into account as it reduces the string. Further, you may specify at what intervals the tab stops are to be located. However, tab stops are usually placed at every eighth position.

Also see the companion function ExpandTab.

Signed

BASIC function contained in FNOTHER.BAS

Purpose:

Signed takes an incoming unsigned long integer value, and returns it in a signed form.

Syntax:

```
S = Signed%(US&)
```

Where:

US& is a number within the range 0 to 65535, and S receives a signed equivalent.

Comments:

The difference between a signed integer value and an unsigned one is really more a matter of semantics than anything else. In either case, there are 65536 values being considered.

When considered on an unsigned basis, integer numbers will range between zero and 65535. But when the same numbers are treated as signed, they instead range from -32768 to 32767. For example, -1 is the same as 65535, -2 equals 65534, and so forth. By definition, a signed number is negative if—when considered in binary—its most significant bit is set.

Also see the companion function UnSigned.

UnParseStr

BASIC function contained in FNOTHER.BAS

Purpose:

UnParseStr accepts an incoming string that contains ASCII characters, and returns the equivalent numeric values separated by commas.

Syntax:

```
List$ = UnParseStr$(Code$)
```

Where:

Code$ contains a string of ASCII characters in the form:

```
CHR$(27) + CHR$(69) + CHR$(27) + CHR$(72)
```

and List$ receives a list of equivalent numbers:

```
"27,69,27,72"
```

Comments:

UnParseStr$ allows editing a string of printer turn-on or turn-off codes that had previously been converted using the companion ParseStr function.

Comments in the source code show how to have UnParseStr place an extra space between the characters:

```
"27, 69, 27, 72"
```

Also see the companion routine ParseStr

UnSigned

BASIC function contained in FNOTHER.BAS

Purpose:

UnSigned takes an incoming signed integer value, and returns it in an unsigned form.

Syntax:

```
US = UnSigned&(S%)
```

Where:

S% is a number within the range -32768 to 32767, and US receives an unsigned equivalent.

Comments:

The difference between signed and unsigned numbers is discussed in the description of the Signed function.

Mathematical functions

BASIC functions contained in FNSPREAD.BAS

QPACOS

Purpose:

Arc cosine of x (rad.) -1 < = x < = +1

Formula:

for ABS(x#) < > 1

```
QPACOS#(x#) = pi / 2 - ATN(x# / SQR(1 - x# * x#))
```

otherwise, if x = 1

```
QPACOS#(x#) = 0
```

else:

```
QPACOS#(x#) = pi
```

Equivalent spreadsheet function:

@ACOS(x)

QPASIN

Purpose:

Arc sine of x (rad.) -1 <= x <= +1

Formula:

for ABS(x#) <> 1

```
QPASIN#(x#) = ATN(x# / SQR(1 - x# * x#))
```

otherwise

```
QPASIN#(x#) = SGN(x#) * pi / 2
```

Equivalent spreadsheet function:

@ASIN(x)

QPATAN2

Purpose:

4-quadrant arc tangent of y/x (rad.) 0 < x < 1

Formula:

for x = 0

```
QPTAN2# = SGN(y#) * pi / 2
```

for x > 0

```
QPATAN2#(x#, y#) = ATN(y# / x#)
```

for x < 0 and y => 0

```
QPATAN2#(x#, y#) = pi + ATN(y# / x#)
```

for x < 0 and y <= 0

```
QPATAN2#(x#, y#) = -pi + ATN(y# / x#)
```

Equivalent spreadsheet function:

@ATAN2(x, y)

QPLOG10

Purpose:

Log of x base 10

Formula:

```
QPLOG10#(x#) = LOG(x#) / LOG(10)
```

Equivalent spreadsheet formula:

@LOG(x)

QPROUND

Purpose:

x rounded to n decimal places

Syntax:

```
Number$ = QPROUND$(Number#, Places%)
```

Equivalent spreadsheet function:

@ROUND(x, n)

Financial functions

BASIC functions contained in FNSPREAD.BAS

Many financial functions involve annuities. An annuity is simply a series of equal payments made at regular intervals of time. It is a compound-interest situation with regular payments. When the payments are made at the beginning of the payment period, the annuity is called an "annuity due". When payment is due at the end of the payment period it is an "ordinary annuity".

Where:

fv# =	future value
pv# =	present value
pmt# =	payment per period
I! =	interest rate *per period* in percent
intr# =	I!/100 (I)
term% =	number of periods
prin# =	principal (same as pv#)
bal# =	balloon payment (may be 0)

Several of the functions we have provided are not available even in Lotus 123. These are identified with an "*".

Sinking fund annuities:

A sinking fund annuity is most easily described as a savings fund designed to accumulate a predetermined amount of money by a specified date.

QPFV - Future value of ordinary annuity (sinking fund)

Formula:

```
QPFV#(pmt#, intr#, term%) = pmt# * ((1 + intr#)^ term% - 1) / intr#
```

Equivalent spreadsheet function:

@FV(pmt#, intr#, term%)

* QPFVN - Term (number of payments) of a sinking fund

Formula:

```
QPFVN#(fv#, pmt#, intr#) = LOG(fv# * intr# / pmt# + 1) / _
     LOG(1 + intr#)
```

* QPFVP - Payment amount of a sinking fund

Formula:

```
QPFVP#(fv#, intr#, term%) = fv# * intr# / ((1 + intr#)^ term% - 1)
```

Annuity due:

An example annuity due (future value) is the future value of a savings account with equal deposits made at the beginning of each period.

* QPFVD - Future value of annuity due

Formula:

```
QPFVD#(pmt#, intr#, term%) = pmt# * (1 + intr#) * _
     ((1 + intr#)^ term% - 1) / intr#
```

* QPFVND - Term (number of payments) of an annuity due/FV

Formula:

```
QPFVND#(fv#, pmt#, intr#) = LOG(fv# * intr# / pmt# + 1 + intr#) / _
     LOG(1 + intr#) - 1
```

* QPFVPD - Payment amount of an annuity due/FV

Formula:

```
QPFVPD#(fv#, intr#, term#) = fv# / ((1 + intr#) / intr# * _
     (1 + intr#)^ term% -1)
```

Ordinary annuity:

When a sum of money is to be repaid with interest in fixed payments for a specified number of periods (such as with a home mortgage), it is called an ordinary annuity. A balloon payment may be associated with this type of annuity.

QPPMT - Loan payment (ordinary annuity)

Formula:

```
QPPMT#(pv#, intr#, term%, bal#) = (pv# - bal# * _
     (1+intr#)^ -term%) / ((1 -(1+intr#)^ -term%) / intr#)
```

Equivalent spreadsheet function:

@PMT(pv#, intr#, term%)

QPPV - Present value of an ordinary annuity

Formula:

```
QPPV#(pmt#, intr#, term%, bal#) = pmt# * _
   (1 - (1 + intr#)^ -term%) / intr# + bal# * (1 + intr#)^ -term%
```

Equivalent spreadsheet function:

@PV(pmt#, intr#, term%)

* QPPVN - Term (number of payments) of an ordinary annuity

Formula:

```
QPPVN#(pmt#, intr#, pv#, bal#) LOG((pmt# - intr# * bal#) / _
     (pmt# - intr# * pv#)) / LOG(1 + intr#)
```

Annuity due relationships:

In order to find the present value of a lease which will involve fixed payments at the beginning of each payment period, use these annuity due relationships.

* QPPMTD - Lease payment (annuity due)

Formula:

```
QPPMTD#(pv#, intr#, term%, bal#) = _
     (pv# - bal# * (1 + intr#)^ -term%) / (1 + intr#) / _
     ((1 - (1 + intr#)^ -term%) / intr#)
```

* QPPVD - Present value of annuity due

Formula:

```
QPPVD#(pmt#, intr#, term%, bal#) = _
      pmt# * (1+intr#) * (1-(1+intr#)^ -term%) / _
     intr# + bal# * (1+intr#)^ -term%
```

* QPPVND - Term (number of payments) of an annuity due

Formula:

```
QPPVND#(pmt#, intr#, pv#, bal#) = _
     LOG((pmt# * (1 + intr#) / intr# - bal#) / (pmt# * _
     (1 + intr#) / intr# - pv#)) / LOG(1 + intr#)
```

Other compound interest relationships

QPCINT— Compounded interest

Purpose:

To find the future value of a savings account drawing compound interest.

Formula:

```
QPCINT#(pv#, intr#, term%) = pv# * (1 + intr#)^ term%
```

QPCTERM—Compounded term of investment

Purpose:

To determine the number of compounding periods it will take an investment to grow to a pre-determined value.

Formula:

```
QPCTERM#(pv#, fv#, intr#) = LOG(fv# / pv#) / LOG(1 + intr#)
```

Equivalent spreadsheet function:

@CTERM(intr#, fv#, pv#)

Note: Interest Rate (intr#) must be in decimal form. For example, the monthly rate of 10% per year is (10 / 100) / 12 or .008333.

QPIRR - Internal rate of return

Usage:

```
QPIRR#(intr#, Array#())
```

This function is seeded by giving it an initial IRR rate (guess). The algorithm first brackets the correct IRR and then converges on the final IRR by a halving method. Convergence ends when a given degree of accuracy is reached.

Equivalent spreadsheet function:

@IRR(guess#, list)

QPNPV - Net present value of future cash flows

Usage:

```
QPNPV#(intr#, Array#())
```

Note: If the initial flow is *not* an outflow, enter zero as the first value in Array#(). This will give the PV of the dollar flow.

Equivalent spreadsheet function:

@NPV(intr#, list)

QPRATE—Rate of investment

Purpose:

To obtain the periodic interest rate required for an investment to grow to a pre-determined value in a specified time.

Formula:

```
QPRATE#(pv#, fv#, term%) = (fv# / pv#)^ (1 / term%) - 1
```

Equivalent spreadsheet function:

@RATE(fv#, pv#, term%)

Depreciation:

BASIC functions for depreciation calculation

The IRS allows depreciation of various assets using some of the methods listed below (depending on the type and life of the asset).

Note: In all cases, year must be greater than or equal to 1 and less than or equal to the depreciable life of the asset.

QPDDB - Double declining balance depreciation

Formula:

```
QPDDB#(cost#, sal#, life%, per%, m!) = _
    m! * cost# / life%^ per% * (life% - 2)^ (per% - 1)
```

Where:

"m" is the depreciation multiplier, i.e.

m = 2—Double Declining Balance
m = 1.5—150% Declining Balance
m = 1—Simple Declining Balance

Note: Adjustments are made to the formula within the function to insure that the total depreciation does not exceed total cost less salvage value.

Equivalent spreadsheet function:

@DDB(cost#, salvage#, life%, period%)

QPSLN - Straight-line depreciation

Formula:

```
QPSLN#(cost#, sal#, life%) = (cost# - sal#) / life%
```

Equivalent spreadsheet function:

@SLN(cost#, salvage#, life%)

QPSYD - Sum-of-years'-digits depreciation

Formula:

```
QPSYD#(cost#, sal#, life%, per%) = (cost# - sal#) * _
          (life% - per% + 1) / (life% * (life% + 1) / 2)
```

Equivalent spreadsheet function:

@SYD(cost#, salvage#, life%, period%)

Statistical functions

BASIC functions contained in FNSPREAD.BAS

QPAVG - returns the average of the values in an array

Usage:

```
Average = QPAVG#(Array#())
```

Formula:

```
QPAVG#(Array#()) = QPSUM#(Array#()) / QPCOUNT%(Array#())
```

Equivalent spreadsheet function:

@AVG(list)

QPCOUNT - returns the number of entries in an array

Usage:

```
Count = QPCOUNT%(Array#())
```

Formula:

```
QPCOUNT%(Array#()) = UBOUND(Array#, 1) - LBOUND(Array#, 1) + 1
```

Equivalent spreadsheet function:

@COUNT(list)

QPMAX - returns the highest value in a list

Usage:

```
MaxValue = QPMAX#(Array#())
```

Note: This function calls an assembly language subroutine that quickly scans the array for the highest value. Assembler subroutines are also provided in QuickPak Professional for integer, long integer, and single precision arrays.

Equivalent spreadsheet function:

@MAX(list)

QPMIN - returns the lowest value in a list

Usage:

```
MinValue = QPMIN#(Array#())
```

Note: This function calls an assembly language subroutine that quickly scans the array for the lowest value. Assembler subroutines are also provided in QuickPak Professional for integer, long integer, and single precision arrays.

Equivalent spreadsheet function:

@MIN(list)

QPSTD - Population standard deviation of items in list

Usage:

```
STD = QPSTD#(Array#())
```

Equivalent spreadsheet function:

@STD(list)

QPSUM - returns the sum of all values in an array

Usage:

```
SUM = QPSUM#(Array#())
```

Equivalent spreadsheet function:

@SUM(list)

QPVAR - Population variance of values in list

Usage:

```
VAR = QPVAR#(Array#())
```

Equivalent spreadsheet function:

@VAR(list)

Chapter 5
Menu/Input Routines

AMenu

assembler subroutine contained in PRO.LIB

Purpose:

AMenu is a sophisticated multi-column menu program that accepts a list of choices in a conventional (not fixed-length) string array. A selection is made by moving the cursor bar to the desired choice, and then pressing Enter.

Syntax:

```
CALL AMenu(BYVAL VARPTR(Array$(Start%)), StartItem%, Count%, _
     ScanCode%, NormColor%, HiLiteColor%, NumRows%, _
     NumCols%, Gap%, ULRow%, ULCol%)
```

Where:

Array$(Start%) is the first element to display in the menu. StartItem% is the item (not the element number) to be initially highlighted. That is , if you want the fifth item in the menu to be highlighted, StartItem% would be set to 5, regardless of the value of Arry$(Start%).

On entry, Count% holds the total number of elements that can appear in the menu. On exit, Count% returns the number of the item (not the element number) that was selected.

ScanCode% indicates which key was used to exit AMenu. You may also tell AMenu not to redisplay the choices by setting ScanCode% to 3 (see below).

NormColor% and HiLiteColor% are the colors to use for the menu items and the currently highlighted item respectively.

NumRows% and NumCols% tell AMenu how large the menu is to be, and Gap% is the number of spaces to place between each menu column. Notice that the length of the first item tells AMenu how long the subsequent items are. Therefore, all of the items should have the same length.

ULRow% and ULCol% indicate where the upper left corner of the menu is to be located on the screen.

Comments:

AMenu is extremely flexible in that the menu choices are displayed in multiple columns anywhere on the screen, and the BASIC program can control all aspects of the menu appearance. AMenu displays itself on the currently active video page.

For maximum flexibility, a box is not drawn around the menu columns. The Box routine would be ideal for this purpose, and it may be used prior to calling AMenu if you'd like.

Besides allowing the number of rows and columns to be varied, this menu has also been designed to be re-entrant. That is, you may call AMenu subsequently with ScanCode% set to 3, and it will maintain the current choice. This allows it to be used for toggling selections on and off, based on the exit code that is returned.

ScanCode% is returned holding either the ASCII value of the terminating key, or a negative value to indicate an extended key. This allows you to trap any keys except the four arrow keys that navigate through the menu. For example, if ScanCode% is set to 13, the Enter key was used to exit AMenu. If ScanCode% is set to, say, -59, then the F1 key was pressed. Notice that the space bar is recognized in the examples as a valid way to exit AMenu, because that key is commonly used for toggling a menu item on and off.

The following example of using AMenu to allow multiple items to be selected is adapted from the DEMOCM.BAS program:

```
DO
   AMenu VARPTR(Array$(1)), 1, Count%, ScanCode%, _
      112, 23, 15, 3, 4, 7, 15
   SELECT CASE ScanCode%
      CASE 13, 32                             'Enter or space
         IF LEFT$(Array$(Count%), 1) = " " THEN
            MID$(Array$(Count%), 1, 1) = CHR$(251) 'check mark
         ELSE
            MID$(Array$(Count%), 1, 1) = " "   'clear the mark
         END IF
      CASE ELSE
   END SELECT
   IF ScanCode% <> 27 THEN ScanCode% = 3     'tell AMenu not to
                                             'reinitialize
LOOP UNTIL ScanCode% = 27
```

To simulate staying in the menu while items are being marked, ScanCode% is set to 3 before subsequently calling AMenu. This tells AMenu to redisplay the current item, rather than evaluate all of the incoming parameters again. If this is not done, the entire menu will be redrawn, and the cursor bar will be placed on the starting element.

As with all of the QuickPak Professional routines that directly access video memory, both the foreground and background colors are combined in a single byte. The OneColor function may be used to calculate the correct color value. The COLORS.BAS program displays a table of color combinations, and can be used as a guide.

Also see the AMenuT routine which is intended for use with fixed-length string arrays.

AMenu and AMenuT are demonstrated in AMENU.BAS.

AMenuT

assembler subroutine contained in PRO.LIB

Purpose:

AMenuT is nearly identical to AMenu, except it is intended for use with fixed-length string arrays.

Syntax:

```
CALL AMenuT(BYVAL VARSEG(Array(Start%)), BYVAL VARPTR(Array(Start%)), _
     StartItem%, Length%,Count%, ScanCode%, NormColor%, HiLiteColor%, _
     NumRows%, NumCols%, Gap%, ULRow%, ULCol%)
```

Where:

Array(Start%) is the first element to display in the menu. StartItem% is the item (not the element number) to be initially highlighted. That is, if you want the fifth item in the menu to be highlighted, StartItem% would be set to 5, regardless of the value of Array(Start%). Length% is the length of each string element.

On entry, Count% holds the total number of elements that can appear in the menu. On exit, Count% returns the number of the item (not the element number) that was selected.

ScanCode% indicates which key was used to exit AMenu. You may also tell AMenu not to redisplay the choices by setting ScanCode% to 3 (see below).

NormColor% and HiLiteColor% are the colors to use for the menu items and the currently highlighted item respectively.

NumRows% and NumCols% tell AMenu how large the menu is to be, and Gap% is the number of spaces to place between each menu column.

ULRow% and ULCol% indicate where the upper left corner of the menu is to be located on the screen.

Comments:

Please see the comments that accompany the description of the AMenu routine.

ASCIIPick and MASCIIPick

assembler subroutines contained in PRO.LIB

Purpose:

ASCIIPick is a menu program that presents the table of ASCII characters, and waits until one is selected. MASCIIPick is similar, but it also supports the mouse for making a selection.

Syntax:

```
CALL ASCIIPick(Char%, Color1%, Color2%, ExitCode%)
```

or

```
CALL MASCIIPick(Char%, Color1%, Color2%, ExitCode%)
```

Where:

Char% is returned holding the ASCII value of the selected character, Color1% is the color to use for the menu, and Color2% is the highlight color for the currently selected character.

ExitCode% will be 0 if Enter (or a mouse button press) was used to make the selection, or 2 if Escape was pressed. If Escape is used, the original value of Char% will be preserved.

The upper left corner of the menu is placed at the current cursor position.

Comments:

A selection is made by using the arrow keys or mouse to move the cursor to the desired character field, and then pressing Enter or the left mouse button. The PgUp and PgDn keys may also be used to move the cursor diagonally, and Home or End will move the cursor to the first or last character respectively. Pressing a character key will place the cursor on that character.

Char% may be pre-loaded to any legal ASCII value, to indicate which character on the menu is to be highlighted initially.

If ExitCode% is set to -1 before calling these routines, the ASCII chart will simply be displayed, and the routines will not wait for a key or mouse button press.

MASCIIPick is demonstrated in the ASCIIPIK.BAS example program.

Also see the related routines ColorPick and MColorPick.

CapNum

BASIC subprogram contained in CAPNUM.BAS

Purpose:

CapNum displays the current setting of the Cap and NumLock keys.

Syntax:

```
CALL CapNum
```

Comments:

CapNum is not really intended to be called separately by your programs, however it could be. Rather, it is called by the various BASIC input routines while they are waiting for a key press.

One aspect of CapNum that you should be aware of is that it always displays the "CAP" and "NUM" messages in black on white (inverse), and clears them to normal white on black.

If you have a colored background that is to be preserved, you must modify the CAPNUM.BAS source code. If the QPrint color variable is changed to -1, the current screen colors will be honored. However, the messages will not stand out as much as when the inverse colors are used.

CapNum is shown in context in the DEMOIN.BAS example program.

ColorPick and MColorPick

assembler subroutines contained in PRO.LIB

Purpose:

ColorPick is a menu program that presents a table of colors and their corresponding values, and waits until one is selected. MColorPick is similar, but it also supports the mouse for making a selection.

Syntax:

```
CALL ColorPick(Colr%, BoxColor%, ExitCode%)
```

or

```
CALL MColorPick(Colr%, BoxColor%, ExitCode%)
```

Where:

Colr% is returned holding the color that was selected, and BoxColor% is the color to use for the box that surrounds the menu.

ExitCode% will be 0 if Enter (or a mouse button press) was used to make the selection, or 2 if Escape was pressed. If Escape is used, the original value of Colr% will be preserved.

The upper left corner of the menu is placed at the current cursor position.

Comments:

A selection is made by using the arrow keys or mouse to move the cursor to the desired color field, and then pressing Enter or the left mouse button. The PgUp and PgDn keys may also be used to go to the top or bottom of the menu, and Home or End will move the cursor to the first or last color respectively.

Colr% may be pre-loaded to any legal color value, to indicate which color on the menu is to be highlighted initially.

If ExitCode% is set to -1 before calling these routines, the color chart will simply be displayed, and the routines will not wait for a key or mouse button press.

MColorPick is demonstrated in the COLORPIK.BAS example program.

Also see the related routines ASCIIPick and MASCIIPick.

DateIn

BASIC subprogram contained in DATEIN.BAS

Purpose:

DateIn provides the ability to enter or edit date fields in a BASIC program. The cursor automatically skips over the separating slashes, and Alt-C will clear the field.

Syntax:

```
CALL DateIn(D$, ExitCode%, Colr%)
```

Where:

D$ is either an existing string in "MMDDYY" format to be edited, or a null string to indicate that a new date is being entered.

ExitCode% tells how editing was terminated, as shown below. Colr% is the field color to be used, and it is packed into a single byte containing both the foreground and background colors.

Comments:

Because DateIn is intended for entering and editing only dates, it does not expect a length parameter. However, it will automatically create the separating slashes during editing, and remove them from the string before returning.

ExitCode% lets your program know whether the user pressed Enter to accept the field, the up arrow to go to the previous field, or Escape.

ExitCode% = 0	Enter, Tab, or the down arrow key was pressed, or the right arrow moved the cursor beyond the end of the field, or the field was filled.
ExitCode% = 1	Shift-Tab or the up arrow key was pressed, or the left arrow put the cursor before the start of the field.
ExitCode% = 2	Escape was pressed.

One important point to be aware of is that DateIn calls the CapNum subprogram to display the current setting of the Cap and NumLock keys. Therefore, any program that uses DateIn must also load or include CapNum.

Because DateIn is written in BASIC, it is simple to add additional exit codes or other features. DateIn provides some degree of date validation in that the month must be between 1 and 12, and the day between 1 and 31. See the description of Date2Num for an example of determining whether a given date is valid.

DateIn is shown in context in the DEMOIN.BAS example program.

Dialog

BASIC subprogram contained in DIALOG.BAS

The Dialog subprogram allows you to add useful dialog boxes that are easily set up and called by your programs, without having to add the large amount of code that usually accompanies such routines. When compiled, DIALOG.OBJ is approximately 22k.

The dialog boxes look and function very much like the ones in QB or QBX, however they are *not* identical. Locations of the various components of the dialog box are handled by the Dialog subroutine, and only their order of appearance may be changed. That is, menus are always located on the right side of the dialog box, check boxes and option buttons are always located on the left side of the dialog box, and text entry fields are always centered unless there is a menu. In addition, the cursor always remains at the upper left corner of the menu field when a list box is active.

Please understand that these differences were made to minimize code size, and not to reduce functionality.

All of the Dialog edit keys function the same as in the QuickBASIC and QBX editing environments, as shown below:

Tab:	Advances through each input field in sequence, from top to bottom. The order in which items are picked when you use the Tab key is determined by their order in the Text$ array. Text$ element 1 is accessed first, element 2 is second, and so forth.
Shift-Tab:	Advances backwards through each selection.
Up/Down/Right/ Left Arrows:	Navigates through each Option Button or menu choice.
Enter:	Selects the entries as displayed and exits. If <Cancel> is highlighted when Enter is pressed, Dialog responds as if Escape had been pressed.
Space Bar:	Is equivalent to hitting the Enter key, or if the cursor is in a Check Box field it toggles the check mark on and off.

Escape:	Exits the dialog box and returns the original data, regardless of any edits that were made.
Left or Right Mouse Buttons:	Selects the Command Button, or places the cursor in the field pointed to. Double clicking the left mouse button on a menu item selects that item, and exits the dialog box as if Enter had been pressed.

While in a text entry field all cursor keys are supported, including the Left and Right arrow keys, Home, End, and Insert. In addition, Alt-C clears the field, and Alt-R restores its previous contents.

Some additional sacrifices were made to keep code size down, as follows:

1. Hot keys are not supported (Alt + first letter of a choice)
2. Only one menu is allowed per dialog box
3. Only one grouping of Option Buttons is supported
4. Mouse scroll bars for menus are not available

Calling Syntax:

```
CALL Dialog(Choice%, Text$(), Resp$(), Menu$(), Row%, Style%,_
    Colr%, Action%)
```

Where:

On exit, Choice% is returned holding the number of the active Command Button. If the Escape key was pressed or Cancel was selected, Choice% instead returns -1.

Text$() is an array of data indicating the type of data to be displayed, and its related text (see "Setting up a Dialog Box" below).

Resp$() is a parallel array that on entry passes the text to be displayed within the various entry fields, and on exit returns the data entered. For Option Buttons, the number of the box checked is returned in all of the Option Button elements so you need only check any one of these elements to find which option was selected.

Menu$() is an array of data to be displayed in the menu/list box. The lengths of the strings in the array *must* all be equal.

Row% if set to 0 will automatically center the dialog box vertically on the screen. Setting Row% to -1 will allow you to locate the dialog box at any screen row. Simply use LOCATE to position the cursor on the desired upper row before calling Dialog. The dialog box must of course fit in the space that you have allowed.

Style% is a number between 1 and 4 that determines the line type to use when drawing the box and dividing line in the Dialog Box. Style% uses the same number code as the QuickPak Professional Box routine. Adding 10 to the line type number causes a drop shadow to be displayed.

Colr% is the desired dialog box color, coded in the format used by the various QuickPak Professional video routines. See COLORS.BAS for a chart of all the color combinations.

Action% lets you specify whether or not the Dialog box is to operate in a re-entrant manner. The table below summarizes the possible values for Action.

Action = 0	Dialog is not re-entrant. On entry, the underlying screen is saved, and the dialog box is displayed. Dialog maintains control until Enter, Escape, or the space bar is pressed, or the mouse is clicked on a Command Button. When that happens, Dialog restores the original screen and returns.
Action = 1	Initializes Dialog in re-entrant mode. The underlying screen is saved, the dialog box is displayed, Action is set to 3, and through a DO LOOP control is passed alternately between the dialog box and the calling program after each key press. (Menu or text field data is returned only after exiting those fields.) This is shown in the example program DEMODIA2.BAS and DEMODIAP.BAS.
Action = 2	Without re-displaying the dialog box itself, Action = 2 re-displays any data in the Response$ array, as well as any plain text strings from the Text$ array.
Action = 3	This does not need to be set by you.

Action = 4	Indicates that a terminating key has been pressed and that the Choice and Response$() variables should be examined.
Action = 5	Closes the dialog box and re-displays the underlying screen.

Setting up a dialog box:

The strings passed in the Text$ array to DIALOG.BAS determine the appearance of the dialog box. Depending upon the first few characters in the string, Dialog interprets them as either Command Buttons, Option Buttons, Check Boxes, Text entry fields, plain strings, or List/Menu fields.

Command Buttons are designated by surrounding the text with "greater than" and "less than" characters, and a single blank space must be used as shown here:

"< OK >"

The " < " symbol defines this string as a Command Button. The text inside " < > " must have a space between it and the " < > " symbols. Command buttons *must* start at Text$() element 1 and be consecutive. Note that the Command Button appearing in Text$ element 1 will always be the default. You may also have as many Command Buttons as will fit on the screen but there must be at least one.

Option Buttons are designated by empty parentheses and a descriptive string like this:

"( ) related text"

The "(" symbol defines this string as an Option Button. A CHR$(7) dot character must also be placed into one of the corresponding Response$ array elements. You may have as many Option Buttons as will fit on the screen, but they must be consecutive. Option Buttons may begin at any element in the Text$ array after element 1. If you prefer to use a character other than 7 for the Option Button indicator, search for and change the Dot$ variable in the DIALOG.BAS source code.

Check Boxes are designated by square brackets, and are also followed by a description as follows:

"[ ] related text"

The "[" symbol defines this string as a Check Box. A check mark ("X") may or may not be placed in one of the corresponding Response$ array elements. You may have as many Check Boxes as will fit on the screen, and they may be in any order. Check Boxes may begin at any element in the Text$ array higher than element 1.

Text entry fields are designated by curly braces as well as a string that describes its purpose:

"{23} related text"

The "{" symbol defines this string as a Text entry field. The number between the braces tells Dialog how long the text field is to be. To make this field accept numbers only, add 100 to the text length. The "related text" will be used as a heading for the field. You may have as many Text entry fields as will fit on the screen. Text Fields may begin at any element in the Text$ array after element 1.

Menu/List boxes are designated by vertical bars, and trailing descriptive text as shown here:

"|308| related text"

The "|" symbol defines this string as a Menu/List box. The number between the bars tells Dialog both the number of rows and the number of columns that are in the field. The right two digits indicate the number of rows. If more than one column is desired, add the number of columns * 100 to the number of rows. In this example, a menu 8 rows high by 3 columns wide is requested. Please note that the number of rows is a minimum number. The Menu/List field will expand to fill the right side of the dialog box no matter how many rows have been specified. You may have only one Menu/List field, and it may appear in any Text$ array element after element 1.

Plain Strings are designated by the absence of any delimiting characters:

"any other string"

Plain strings are automatically centered in the dialog box, although they may be offset left or right by padding them with spaces. Plain strings may be specified in any Text$ array element after element 1. If a title is desired for the dialog box, it must be assigned to element 0 in the Text$ array.

Please see the DEMODIAL.BAS, DEMODIAP.BAS and DEMODIA2.BAS demonstration programs for more information on using Dialog.

DirFile

assembler subroutine contained in PRO.LIB

Purpose:

DirFile provides an ideal menu for selecting a file name from a list of choices. The selection is made by using the arrow keys to move the cursor bar to the desired choice, and then by pressing Enter.

Syntax:

```
CALL DirFile(BYVAL VARPTR(Array$(1)), Count%, ScanCode%,_
     MsgColor%, FileColor%, HiLiteColor%, BoxColor%)
```

Where:

Array$(1) is the first element in the array to display, though any valid starting element may be specified. All of the elements are assumed to have a length of 12, and any additional characters will not be displayed.

Count% is the total number of items to display on entry, and it returns holding the array element number selected when DirFile is finished.

ScanCode% indicates which key the user pressed to exit the menu (see below).

MsgColor% is the color for the prompt and other messages that are displayed, FileColor% indicates what color to use for the file names, and HiLiteColor% is the color of the currently highlighted item.

BoxColor% tells DirFile what color to use when drawing the surrounding box.

Comments:

DirFile allows you to emulate the original Microsoft style of file directory selection in your own programs.ScanCode% is returned holding either the ASCII value of the terminating key, or a negative value to indicate an extended key. This allows you to trap any keys except the four arrow keys that navigate through the menu. For example, if ScanCode% is set to 13, the Enter key was used to exit DirFile. If ScanCode% is set to, say, -59, then the F1 key was pressed.

As with most of the QuickPak Professional routines that access video memory directly, both the foreground and background colors are contained in a single byte. The OneColor function should be used to calculate the value of the color byte.

DirFile always displays itself on the currently active video page.

DirFile is demonstrated in the program DEMOCM.BAS.

Editor

assembler subroutine contained in PRO.LIB

Purpose:

Editor is an assembly language text input routine that also allows editing an existing string.

Syntax:

```
CALL Editor(Ed$, ActiveLength%, ScanCode%, NumOnly%, _
     CapsOn%, NormalColor%, EditColor%, Row%, Column%)
```

Where:

Ed$ is the string being entered or edited, which must be pre-assigned to the correct maximum length (see below). ActiveLength% is then returned holding the actual length of the string after editing.

ScanCode% indicates how editing was terminated. You may also tell Editor to resume editing at any arbitrary cursor position (see below).

NumOnly% is 1 to accept numeric input only, or 0 to allow any characters. CapsOn% is a flag to tell Editor to automatically capitalize all incoming text—use 1 to capitalize or 0 not to.

NormalColor% is the color to restore the text to when editing has finished, and EditColor% is the color to use while the text is being entered.

Row% and Column% indicate where on the screen the beginning of the input field is placed.

Comments:

Editor is written in assembly language, and will thus occupy very little code space when compared to an equivalent input routine written in BASIC.

Because an assembler routine cannot create or assign a string, it is up to you to establish the string with the maximum length allowed. For example, if a new string is to be entered and it will be restricted to a length of fifteen characters, you would first assign Ed$ to a string of blanks:

```
Ed$ = SPACE$(15)
```

If Ed$ already contains information that is to be edited, you would instead pad it to a length of 15 with trailing blanks:

```
Ed$ = Ed$ + SPACE$(15 - LEN(Ed$))
```

ScanCode% is returned holding either the ASCII value of the terminating key, or a negative value to indicate an extended key. This allows you to trap any Alt or function keys. For example, if ScanCode% is set to 13, the Enter key was used to exit Editor. If ScanCode% is set to, say, -104, then Alt-F1 had been pressed.

ScanCode% may also be initialized before calling Editor to either resume editing at the cursor location last used, at any arbitrary cursor location, or at the beginning of the field. This lets you reenter Editor, for example after detecting F1 and displaying a help screen. Setting ScanCode% to 1 tells it to resume editing at the point where the cursor was located when Editor was last exited. If it is instead set to 2, editing will resume at the location specified by the Column% parameter. Editor considers any other value to mean that the string is being edited for the first time.

As with the other QuickPak Professional input and menu programs, Editor will display itself on whatever screen page is currently active.

Two additional keys that Editor recognizes are Alt-C, which clears the field, and Alt-R, which restores its original contents.

A complete demonstration of Editor is given in the DEMOCM.BAS example program.

Lts2Menu

BASIC subprogram contained in LTS2MENU.BAS

Purpose:

Lts2Menu is a Lotus 123 "look alike" menu, where a list of choices is displayed horizontally on a single line, along with a second line containing a prompt for the current item. A selection is made either by using the arrow keys to highlight a choice and pressing Enter, or by pressing a key that corresponds to the first letter of a choice.

Syntax:

```
CALL Lts2Menu(Item$(), Prompt$(), Choice%, Colr%)
```

Where:

Both Item$() and Prompt$() are conventional (not fixed-length) string arrays. Item$ contains a list of the menu items, and Prompt$ holds the text that describes each choice. The maximum length for any item or prompt is 78 characters.

Colr% is the color to use when displaying the choices, and Choice% returns holding the choice number that was selected.

Notice that the color that was specified is reversed to create a highlight for the current choice. The prompt color is also derived from the Colr% parameter, though this may be changed to suit. Comments in the source code show how to do this.

Comments:

As with many of the QuickPak Professional routines, both the foreground and background colors are combined into a single variable. The OneColor function may be used to calculate the correct value. You could also run the COLORS.BAS program to view all of the possible color combinations.

As shipped, Lts2Menu waits until the Enter key is pressed before returning to the calling program. However, comments in the source code show how to have it return as soon as the first letter of a choice has been pressed. If you make this modification, though, be aware that each choice must begin with a unique first letter.

If Escape is pressed, Lts2Menu will return showing a choice of zero. Both the Home and End keys are also recognized, which highlight the first and last choices respectively.

Notice that Lts2Menu always saves the underlying screen before it displays itself, and restores it again before returning to the calling program.

Lts2Menu and its companion LtsMenu are each shown in context in the DEMOLTS.BAS example program.

LtsMenu

BASIC subprogram contained in LTSMENU.BAS

Purpose:

LtsMenu is an alternate Lotus 123 "look alike" menu, where a list of choices is displayed horizontally on a single line, but without any prompt information. A selection is made either by using the arrow keys to highlight a choice and pressing Enter, or by pressing a key that corresponds to the first letter of a choice.

Syntax:

```
CALL LtsMenu(Item$(), Choice%, Colr%)
```

Where:

Item$() is a conventional (not fixed-length) string array containing a list of the choices, each of which may be up to 78 characters in length.

Colr% is the color to use when displaying the choices, and Choice% returns holding the choice number that was selected. Notice that the color that was specified is reversed to create a highlight for the current choice.

Comments:

As with many of the QuickPak Professional routines, both the foreground and background colors are combined into a single variable. The OneColor function may be used to calculate the correct value. You could also run the COLORS.BAS program to view all of the possible color combinations.

As shipped, LtsMenu waits until the Enter key is pressed before returning to the calling program. However, comments in the source code show how to have it return as soon as the first letter of a choice has been pressed. If you make this modification, though, be aware that each choice must begin with a unique first letter.

If Escape is pressed, LtsMenu will return showing a choice of zero. Both the Home and End keys are also recognized, which highlight the first and last choices respectively.

Notice that LtsMenu always saves the underlying screen before it displays itself, and restores it again before returning to the calling program.

LtsMenu and its companion Lts2Menu are each shown in context in the DEMOLTS.BAS example program.

MAMenu

assembler subroutine contained in PRO.LIB

Purpose:

MAMenu is a full-featured multi-column menu routine, which allows selecting items either with the keyboard or a mouse. This is a mouse-aware version of the original AMenu routine, which is described elsewhere.

Syntax:

```
CALL MAMenu(BYVAL VARPTR(Array$(1)), Selection%, Start%, Count%, _
     ScanCode%, NormalColor%, Hilight%, NumRows%, NumCols%, Gap%, _
     Row%, Column%)
```

Where:

Array$(1) is the first element in the string array, and Selection% is the element to be highlighted initially. On exit, Selection% holds the element that was selected.

Start% tells MAMenu which element to place in the upper left corner of the menu, and Count% holds the total number of elements to be displayed.

ScanCode% is returned holding the key that was used to exit MAMenu. If a normal key was pressed, ScanCode% will hold its ASCII value. For example, if Enter was pressed, then ScanCode% will be set to 13. If an extended key was used, then ScanCode% will hold a *negative* version of the key's extended code. That is, F1 would be returned as -59.

NormalColor% is the color to use for all of the selections except the one that is currently highlighted, and Hilight% is the color to use for the scroll bar.

NumRows% and NumCols% tell MAMenu how many rows and columns to display respectively, and Gap% is the number of columns between each cluster of items.

Row% and Column% indicate where to place the upper left corner of the menu.

Comments:

If the mouse is clicked while its cursor is outside of the active menu area, then MAMenu will return the following information:

ScanCode% = 1000	the left button was pressed
ScanCode% = 1001	the right button was pressed
ScanCode% = 1002	the middle button was pressed (on mice so equipped)
Row% and Column%	where the mouse cursor was located at the time the mouse button was pressed

If the mouse button is pressed while the mouse cursor is within the menu, then the mouse clicks will have the following effect:

Single click	the cursor bar will be relocated to the new position
Double click	a rapid double-click will exit MAMenu with ScanCode% set to 1003, and Selection% will hold the current selection

In order to use MAMenu within a BASIC program loop, it is assumed that the mouse button has been pressed once on entry. Therefore, an additional mouse button click within the time-out period will be treated as a double-click.

MAMenu and MAMenuT are demonstrated in the MOUSECM.BAS example program.

MAMenuT

assembler subroutine contained in PRO.LIB

Purpose:

MAMenuT is a full-featured multi-column menu routine, which allows selecting items either with the keyboard or a mouse. This is a "mouse-aware" version of the original AMenuT routine, which is described elsewhere.

Syntax:

```
CALL MAMenuT(BYVAL VARSEG(Array$(1)), BYVAL VARPTR(Array$(1)), _
     Selection%, Start%, Length%, Count%, ScanCode%, NormalColor%, _
     Hilight%, NumRows%, NumCols%, Gap%, Row%, Column%)
```

Where:

Array$(1) is the first element in the string array, and Selection% is the element to be highlighted initially. On exit, Selection% holds the element that was selected.

Start% tells MAMenuT which element to place in the upper left corner of the menu, Count% holds the total number of elements to be displayed, and Length% is the number of bytes that comprise each fixed-length element.

ScanCode% is returned holding the key that was used to exit MAMenuT. If a normal key was pressed, ScanCode% will hold its ASCII value. For example, if Enter was pressed, then ScanCode% will be set to 13. If an extended key was used, then ScanCode% will hold a negative version of the key's extended code. That is, F1 would be returned as -59.

NormalColor% is the color to use for all of the selections except the one that is currently highlighted, and Hilight% is the color to use for the scroll bar.

NumRows% and NumCols% tell MAMenuT how many rows and columns to display respectively, and Gap% is the number of columns between each cluster of items.

Row% and Column% indicate where to place the upper left corner of the menu.

Comments:

Please see the comments that accompany the MAMenu routine.

MaskIn

BASIC subroutine contained in MASKIN.BAS

Purpose:

MaskIn is a sophisticated "mask input" routine, which lets you specify the type of characters that are entered at each position in the field.

Syntax:

```
Mask$ = "Enter your name: " + STRING$(15, 1)
Text$ = SPACE$(15)
Call MaskIn(Mask$, Text$, Mski)
```

Where:

Mask$ contains the prompt text and bit-encoded mask characters, and Text$ returns holding the entered text. Mski is a TYPE variable that controls the field and prompt colors, the polling action, and indicates the last key that was pressed.

The mask characters are comprised of ASCII values 1 - 31 as follows:

	BIT
Accept letters, convert to upper case	1
Accept letters, convert to lower case	2
Accept digits	3
Accept math punctuation (-+,.)	4
Accept all punctuation	5

Note that combining bits 1 and 2 tells MaskIn to accept letters and not alter the capitalization.

Comments:

Like many of the other QuickPak Professional input routines, MaskIn can optionally be used in a polled mode. When the Action portion of the TYPE variable is set to 0, MaskIn works like a normal input routine. That is, it retains control until Enter, Tab, Shift-Tab, or Escape are pressed. The other supported action values are listed below, and they are compatible with the codes used by PullDown, VertMenu, QEdit, and the other pollable routines.

Notice that MaskIn does not support inserting or deleting characters, because that could let characters that are legal for one field position be moved into positions where they would be illegal. Therefore, the Delete key merely clears the current character. Also notice that MaskIn beeps if an illegal character is typed. If you prefer to have such keys be simply ignored, search for the BEEP statement and remove it.

Any combination of bits may be used as a field mask. For example, to accept only letters and numbers, and also force capitalization for the letters you would use CHR$(5) for that field position. A complete list of all the combinations is shown in the DEMOMASK.BAS demonstration program.

Text$ must be initialized to the appropriate length. If Text$ is shorter than the actual field size specified in Mask$, only as many characters as can will be returned.

Mski is a user-defined TYPE variable that controls the field colors and editing action, and also returns last key pressed. A TYPE variable is used to minimize the number of passed parameters, and thus reduces the size of your programs. Mski is defined as follows:

```
TYPE MaskParms
   Ky          AS INTEGER          'last key stroke entered
   Action      AS INTEGER          'action flag
   MColr       AS INTEGER          'message color
   FColr       AS INTEGER          'field color
END TYPE
Dim MSKI as MaskParms              'create the variable
```

When MaskIn returns, Ky will hold the ASCII value of the last key pressed. If an extended key was pressed, it will be returned as a negative number. For example, pressing Escape to exit will set MSKI.Ky to 27, and pressing F1 instead returns -59.

The Action flag controls how MaskIn reacts within the program as follows:

Action = 0:	Initializes (displays) the field and waits for keystrokes exiting when Enter, Escape, Tab, Shift-Tab, or the up or down arrow keys are pressed.
Action = 1:	Initializes the field, checks for keystrokes, and sets Action to 3 for a subsequent polled loop.
Action = 2:	Initializes only, does not check for keystrokes, and exits with an Action of 3.
Action = 3:	Does not initialize field, just checks for key activity.

MColr is the color to print the message or prompt portion of the mask string, using the QuickPak method of combined foreground/background colors.

FColr is the color to print the input field portion of the mask string.

See COLORS.BAS for a demonstration of the color combinations MaskIn expects.

MEditor

assembler subroutine contained in PRO.LIB

Purpose:

MEditor is a general purpose input routine that allows entering new text, as well as editing an existing string. This is a "mouse-aware" version of the original Editor subroutine which is described elsewhere in this manual.

Syntax:

```
CALL MEditor(Ed$, ActiveLength%, ScanCode%, NumOnly%, CapsOn%, _
     NormalColor%, EditColor%, Row%, Column%, CurrentColumn%)
```

Where:

Ed$ is the string to be entered or edited. This string must be initialized to the maximum acceptable length of the string being input.

ActiveLength% returns holding the active length of the string, not counting any trailing blanks.

ScanCode% is returned holding the key that was used to exit MEditor. If a normal key was pressed, ScanCode% will hold its ASCII value. For example, if Enter was pressed, then ScanCode% will be set to 13. If an extended key was used, then ScanCode% will hold a negative version of the key's extended code. That is, F1 would be returned as -59.

NumOnly% and CapsOn% may be optionally set to 1 to indicate that only numbers are acceptable, or that all letter will be forced to upper case. Otherwise, these should be zero.

EditColor% tells MEditor which color to use while editing, and NormalColor% is the color to restore the text to when editing is terminated.

Row% and Column% specify where to place the left edge of the field. On entry, CurCol% is the column at which to place the cursor, and on exit it holds the column where the cursor was last.

Comments:

If the mouse is clicked while its cursor is outside of the editing field, then MEditor will return the following information:

ScanCode% = 1000	the left button was pressed
ScanCode% = 1001	the right button was pressed
ScanCode% = 1002	the middle button was pressed (on mice so equipped)
Row% and Column%	where the mouse cursor was located at the time the mouse button was pressed

Please see the description for the Editor routine for a list of keystrokes and other features that are supported by MEditor.

MEditor is demonstrated in the MOUSECM.BAS example program.

MenuVert

assembler subroutine contained in PRO.LIB

Purpose:

MenuVert is a vertical menu program that accepts a list of choices in a conventional (not fixed-length) string array. The choices are displayed in a column, and a selection is made by using the up and down arrow keys to move a cursor bar to the desired choice. Pressing Enter then selects that choice. The PgUp, PgDn, Home and End keys are also supported.

Syntax:

```
CALL MenuVert(BYVAL VARPTR(Array$(1)), NumLines%, ScanCode%, _
     Choice%, NormalColor%, HiLiteColor%, Row%, Column%)
```

Where:

Array$(1) is the first element in the array to display, though any valid starting element may be specified.

NumLines% is the desired height of the menu, which must be no greater than the number of items in the array.

ScanCode% tells which key was used to make a selection (see below). On entry, ScanCode% *must* hold the total number of items in the array.

Choice% holds the element that was highlighted when MenuVert returns.

NormalColor% is the color to use for the choices, and HiLiteColor% is the color for the currently highlighted item. Both the foreground and background colors are contained in a single byte.

Row% and Column% determine the upper left corner of the menu on the screen.

Comments:

To minimize the number of passed parameters, ScanCode% is also used to tell MenuVert how large the array is. The length of the first array element is used for the menu width, and it is assumed that all items will be the same length.

ScanCode% is returned holding either the ASCII value of the terminating key, or a negative value to indicate an extended key. This allows you to trap any Alt or function keys. For example, if ScanCode% is set to 13, the Enter key was used to exit MenuVert. If ScanCode% is set to, say, - 84, then Shift-F1 had been pressed.

As with the other QuickPak Professional input and menu programs, MenuVert will display itself on whatever screen page is currently active.

A box is not drawn around the menu, though the Box routine would be ideal if you want that feature. MenuVert is shown in the DEMOCM.BAS example program.

MGetKey

assembler function contained in PRO.LIB

Purpose:

MGetKey first clears the keyboard buffer of any pending keys, and then waits until either a key or mouse button has been pressed. This is a "mouse-aware" version of the QuickPak Professional WaitKey routine.

Syntax:

```
ScanCode = MGetKey%(Row%, Column%)
```

Where:

ScanCode% is returned holding the key or mouse button that was pressed, and Row% and Column% indicate the location of the mouse cursor at the time the mouse button was pressed.

Comments:

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

ScanCode% is assigned a value that corresponds to the key that was pressed, or the mouse button if that was used. If a normal key was pressed, ScanCode% will hold its ASCII value. For example, if Enter was pressed, then ScanCode% will be set to 13. If an extended key was used, then ScanCode% will hold a *negative* version of the key's extended code. That is, F1 would be returned as -59.

If the mouse button was pressed, then MGetKey will return the following information:

ScanCode% = 1000	the left button was pressed
ScanCode% = 1001	the right button was pressed
ScanCode% = 1002	the middle button was pressed (on mice so equipped)

Row% and Column%	where the mouse cursor was located at the time the mouse button was pressed

MGetKey is demonstrated in the MOUSECM.BAS example program.

MMenuVert

assembler subroutine contained in PRO.LIB

Purpose:

MMenuVert is a complete vertical menu subroutine that accommodates selecting from a list of items. This is a "mouse-aware" version of the MenuVert routine described elsewhere.

Syntax:

```
CALL MMenuVert(BYVAL VARPTR(Array$(1)), Selection%, Start%, _
     ScanCode%, NormalColor%, Hilight%, NumRows%, Row%, Column%)
```

Where:

Array$(1) is the first element in the string array, and Selection% is the element to be highlighted initially. On exit, Selection% holds the element that was selected.

Start% tells MMenuVert which element to place at the top of the menu, and on entry, ScanCode% must be set to the total number of elements in the array.

On return, ScanCode% will hold the key that was used to exit MMenuVert. If a normal key was pressed, ScanCode% will contain its ASCII value. For example, if Enter was pressed, then ScanCode% will be set to 13. If an extended key was used, then ScanCode% will hold a *negative* version of the key's extended code. That is, F1 would be returned as -59.

NormalColor% is the color to use for all of the selections except the one that is currently highlighted, and Hilight% is the color to use for the scroll bar.

NumRows% tells MMenuVert how many rows to display on the screen, which should not be more than the number of elements specified in ScanCode%.

Row% and Column% indicate where to place the upper left corner of the menu.

Comments:

If the mouse is clicked while its cursor is outside of the active menu area, then MMenuVert will return the following information:

ScanCode% = 1000	the left button was pressed
ScanCode% = 1001	the right button was pressed
ScanCode% = 1002	the middle button was pressed (on mice so equipped)
Row% and Column%	where the mouse cursor was located at the time the mouse button was pressed

If the mouse button is pressed while the mouse cursor is within the menu, then the mouse clicks will have the following effect:

Single click	the cursor bar will be relocated to the new position.
Double click	a rapid double-click will exit MMenuVert with ScanCode% set to 1003, and Selection% will hold the current selection.

In order to use MMenuVert within a BASIC program loop, it is assumed that the mouse button has been pressed once on entry. Therefore, an additional mouse button click within the time-out period will be treated as a double-click.

MMenuVert is demonstrated in the MOUSECM.BAS example program.

NumIn

BASIC subprogram contained in NUMIN.BAS

Purpose:

NumIn provides the ability to enter or edit a numeric field in a BASIC program. The decimal point is skipped automatically, and Alt-C may be used to clear the field.

Syntax:

```
CALL NumIn(Number#, Max%, Places%, ExitCode%, Colr%)
```

Where:

Number# is a double precision value to be entered or edited.

Max% is the maximum number of digits to the left of the decimal point, and Places% is the number of digits to the right.

ExitCode% tells how editing was terminated, as shown below.

Colr% is the field color to be used, and it is packed into a single byte containing both the foreground and background colors.

Comments:

ExitCode% lets your program know whether the user pressed Enter to accept the field, the up arrow to go to the previous field, or Escape.

ExitCode% = 0	Enter, Tab, or the down arrow key was pressed, or the right arrow moved the cursor beyond the end of the field, or the field was filled.
ExitCode% = 1	Shift-Tab or the up arrow key was pressed.
ExitCode% = 2	Escape was pressed.

One important point to be aware of is that NumIn calls the CapNum subprogram to display the current setting of the Cap and NumLock keys. Therefore, any program that uses NumIn must also load or include CapNum.

Because NumIn is written in BASIC, it is simple to add additional exit codes or other features. You may also modify NumIn to accept a single precision variable if you'd like. Simply use Search and Replace to change every occurrence of "N#" to "N!" within the NumIn subprogram source code.

NumIn is shown in context in the DEMOIN.BAS example program.

PickList

BASIC subprogram contained in PICKLIST.BAS

Purpose:

PickList is a "front end" subprogram for VertMenu that allows selecting multiple items from a single menu.

Syntax:

```
CALL PickList (Items$(), Picked%(), NPicked%, Cnf)
```

Where:

Item$() is a conventional (not fixed-length) string array that contains the items to display, and elements in the Picked%() array indicate which ones were selected. NPicked% contains the total number of items that were chosen, and Cnf is a special TYPE variable that is used by VertMenu.

Comments:

PickList is useful in a variety of situations where multiple choices must be accommodated. For example, it could be used to select a group of files to be printed. Because PickList calls upon VertMenu to do most of the work, all of the features in VertMenu are available in PickList. The menu may be scrolled to display more items than will fit on the screen, a mouse is fully supported, and the menu colors are easily controlled via the Cnf TYPE variable.

While the menu is active, the user may press either Enter or the space bar to select an item. Their selection is indicated with a check mark that is placed to the right of the menu item. Pressing Enter or the space bar a second time removes the check mark. When selection is complete, pressing Escape returns control to the calling program.

Notice, the items may also be selected by double-clicking with a mouse

The Picked%() array indicates which items have been selected with a non-zero value, and also the sequence number for each selection. That is, if the item first selected in the list was, say, item 14, then Picked%(1) would hold the number 14.

The size to which Picked%() has been dimensioned controls the maximum number of items that may be selected. For example, if Picked%() is dimensioned to ten items, then only that many may be picked, regardless of how many items are actually in the Item$() array. Picked%() would usually be dimensioned to the same number of elements as the Item$() array.

You should pad each element in the Item$() array with three trailing spaces to make room for the check marks. This also tells PickList to add a separating divider between the items and the check marks. If an item does not contain at least three trailing spaces, PickList will not display the divider.

PickList is demonstrated in the DEMOPICK.BAS example program.

PullDown

BASIC subprogram contained in PULLDOWN.BAS

Purpose:

PullDown is a complete multiple-menu subprogram with many important capabilities including full support for a mouse. Besides being able to display more than one list of choices, it always saves the underlying screen, draws an attractive shadow automatically, and accommodates a separating divider between related groups of items.

Further, selected menu items may be allowed or disallowed at will. Finally, PullDown may be operated in a unique multi-tasking mode, whereby it is polled periodically to see if a choice has been selected.

Syntax:

```
CALL PullDown(Choice$(), Status%(), Menu%, Choice%, Ky$, _
     Action%, Cnf)
```

Where:

Choice$() is a two-dimensional array containing the list of choices for each menu. If any element is assigned to a hyphen only ("—"), it will be replaced by a separating line and not be selectable by the user.

Status%() is a parallel two-dimensional array that indicates which choices are active. Choices are deactivated by assigning any non-zero value to the elements that corresponds to a given item in the Choice$ array.

Menu% indicates which menu was active when a choice was selected, and may also be pre-loaded to force a given menu to be initially displayed. Choice% indicates which choice was selected, and may also be pre-loaded to force a given choice to be initially highlighted.

Ky$ holds the last key that was pressed by the user, and Action% tells PullDown how it is being used. Cnf is a special TYPE variable that contains information about the host PC.

Comments:

PullDown is explained in depth in the section entitled "Multi-Tasking Menus", and two complete demonstrations are also provided. DEMOPULL.BAS shows the minimum setup required for calling PullDown, and DEMOMENU.BAS illustrates some of its more advanced uses.

PullDnMS

BASIC subprogram contained in PULLDNMS.BAS

Purpose:

PullDnMS is a complete pull-down menu system similar to the regular PullDown subprogram described elsewhere. However, PullDnMS has been designed to behave exactly like the menus used in QuickBASIC 4.5 and later.

Syntax:

```
CALL PullDnMS (Choice$(), Stat%(), Menu%, Choice%, Ky$, Action%)
```

Where:

Choice$() is a 2-dimensional array that holds the menu choices.

Choice$(0, Menu) holds the menu titles, and Choice$(Choice, Menu) holds the selections for each menu. Using a CHR$(45) hyphen for a choice specifies a non-selectable dividing line across the menu.

Stat%() is a 2-dimensional array, which indicates whether an item is selectable, and also which letter in the string is to be highlighted for one-key selection.

Menu% is returned holding the menu that was active when the user made a selection. It may also be pre-loaded with a non-zero value to cause that menu to be displayed initially.

Choice% is returned holding the choice that was selected within a given menu.

Ky$ is returned holding the last key that was pressed by the user. For example, if Escape was pressed to exit the menu, Ky$ will hold CHR$(27).

The Action parameter specifies how the menu is to operate, and this is described in detail in the section entitled "Multi-Tasking Menus" elsewhere in this manual.

Comments:

One of the most important differences between PullDnMS and the original PullDown routine is in the operation of the Stat%() array. With PullDown, elements in the Stat% array merely control which menu choices may be selected by the user. Choices that are inactive are both dimmer in color, and ignored if they are pressed. However, PullDnMS expands the functionality of the Stat%() array by also allowing you to specify which character in the choice string is the "hot letter."

To save memory, the two bytes in each Stat%() element are used independently. If the low byte is non-zero, then that choice may not be selected by the user. The high byte then indicates which character is to be highlighted and made selectable with a single key. The example below shows how to define the choice "Save As", as used within the QuickBASIC 4.5 editor:

```
'              0123456789  <--- character position (zero-based)
Menu$(3, 0) = "Save As..."

Stat%(3, 0) = 5 * 256 + 1 '<--- highlight the 6th character, "A"
'                       ^ Optional to make the choice inactive.
```

The character position is intentionally zero-based rather than one-based, so the first letter will be the default if that element in Stat%() is left unassigned.

PullDnMS is demonstrated in the DEMOPLMS.BAS example program.

QEdit

BASIC subprogram contained in QEDIT.BAS

Purpose:

QEdit is a complete text editor subprogram that may be called as a "pop-up" from within a BASIC program. QEdit automatically saves the underlying screen and draws an attractive shadow, and it may be used in the 25, 43, or 50 line screen modes.

Syntax:

```
CALL QEdit(Array$(), Ky$, Action%, Ed)
```

Where:

Array$() is a conventional (not fixed-length) string array that will hold the text being entered or edited. The size to which Array$ has been dimensioned determines the maximum number of lines that may be entered.

Ky$ is returned holding the last key that was pressed. For example, Ky$ would be CHR$(27) if the user pressed Escape to exit QEdit.

Action% indicates how QEdit is being invoked. If Action% is set to 0 when QEdit is invoked, QEdit will save the underlying screen, and allow editing until the Escape key is pressed.

Ed is a special TYPE variable that controls a number of QEdit's parameters.

Comments:

The operation of QEdit is explained in depth elsewhere in this manual (see the Table of Contents), along with a detailed discussion of the Action% and Ed parameters. A complete demonstration is also provided in the DEMOEDIT.BAS example program.

There are actually three versions of QEdit; all have the same CALL name, but they are in separate files. QEDITS.BAS contains a smaller version that omits the block operations and thus adds less code to your programs. The QEdit contained in QEDIT7.BAS has the same features as QEdit, but it is optimized for use in the QBX editing environment. Since QBX will place small subprograms (less than 16K) into expanded memory, QEdit7 has been reorganized into several subprograms instead of one large one.

Please note that our QEdit subroutine in not the same as the QEdit text editor program published by SemWare.

ScrollIn

BASIC subprogram contained in SCROLLIN.BAS

Purpose:

ScrollIn is a virtual field input routine that allows editing text that is wider than the window showing on the screen.

Syntax:

```
CALL ScrollIn(Edit$, Start%, Wide%, MaxLen%, Filter%, Ky%, EdClr%, _
     NormalClr%)
```

Where:

Edit$ is the string to be edited, and may range from 0 to 32000 characters in length.

On entry, Start% specifies which character in Edit$ is to be placed at the left edge of the edit window. On exit, Start% holds the column where the cursor was when the field was exited.

Wide% is the width of the edit window.

MaxLen% is the maximum allowable length of the edited text. MaxLen% must be at least as great as Wide%. If MaxLen% = Wide% scrolling is disabled.

Filter% determines the type of text to be accepted by ScrollIn, and may be set to any of the following values:

0	All keys will be accepted
1	Integer characters
2	Integers, single/double precision characters only
3	User-defined
4	Converts all letters to upper case

Ky% returns the ASCII code for the key used to exit the routine. If an extended key was pressed, Ky% returns a negative value corresponding to the key's extended code. If Escape is pressed, ScrollIn restores Edit$ to its original contents. If the left mouse button is clicked outside of the edit window, ScrollIn responds as if Enter were pressed, but Ky% instead returns a value of 1000. The following values are returned when their corresponding exit keys are pressed.

Escape = 27
Enter = 13
Down Arrow = -80
Up Arrow = -72
PgUp = -73
PgDn = - 81

EdClr% is the color to use while editing, and is coded in the format used by the various QuickPak Professional video routines.

NormalClr% is the color to use when editing is complete.

Comments:

If the length of the text is greater than the size of the edit window, the text may be scrolled right or left using the standard cursor keys, or with a mouse by holding the left mouse button down on either the left- or right-most character in the edit window. All of the standard editing keys are supported; in addition, Alt-C clears the field, and Alt-R restores it to the original contents. Use BASIC's LOCATE statement to position the left edge of the field.

The Filter argument specifies which set of characters are accepted, based on three filter masks. The first two are defined within ScrollIn, using CONST strings named Filter1$ and Filter2$. You indicate which to use by setting Filter% to 1 or 2. If Filter% is assigned to 3, then ScrollIn uses a filter mask that you define. Simply define Filter3$ as shown at the start of the SCROLLIN.BAS source file.

If you do not require a mouse for your application, the block of mouse code in SCROLLIN.BAS can easily be removed. Simply search for "MMM" and do as the comments indicate.

ScrollIn is demonstrated in the DEMOSCRL.BAS example program.

See COLORS.BAS for a description of the combined color method.

Spread

BASIC subprogram contained in SPREAD.BAS

Purpose:

Spread is a complete spreadsheet subprogram that may be called as a "pop-up" from within a BASIC program. Spread automatically saves the underlying screen and draws an attractive shadow, and it may be used in the 25, 43, or 50 line screen modes.

Syntax:

```
CALL Spread(Wks$(), Format$(), ColumnWidths%(), Wide%, Rows%, Action%)
```

Where:

Wks$() is a normal (not fixed-length) two-dimensional string array that will hold the spreadsheet data being entered or edited. The size to which Wks$ has been dimensioned determines the maximum number of rows and columns that may be accessed.

Format$() is a two-dimensional table of formatting information for the numeric cell values.

ColumnWidths%() is a one-dimensional array containing a list of widths for each spreadsheet column.

Wide% is the total width of the spreadsheet window, and Rows% is the window height.

Action% indicates how Spread is being invoked.

Comments:

Spread is explained in depth elsewhere in this manual, and a complete demonstration is also provided in the DEMOSS.BAS example program.

TextIn

BASIC subprogram contained in TEXTIN.BAS

Purpose:

TextIn provides the ability to enter or edit a text field in a BASIC program. All of the standard editing keys are supported, plus Alt-C clears the field, and Alt-R will restore it to the original contents.

Syntax:

```
CALL TextIn(Text$, Max%, NumOnly%, CapsOn%, ExitCode%, Colr%)
```

Where:

Text$ is the text to be entered or edited, and Max% is the maximum allowable length of the field.

NumOnly% indicates whether numeric input only is to be accepted if non-zero. CapsOn% tells TextIn to automatically capitalize all incoming alphabetic characters if non-zero.

ExitCode% tells how editing was terminated, as shown below.

Colr% is the field color to be used, and it is packed into a single byte containing both the foreground and background colors.

Comments:

ExitCode% lets your program know whether the user pressed Enter to accept the field, the up arrow to go to the previous field, or Escape.

ExitCode% = 0	Enter, Tab, or the down arrow key was pressed, or the right arrow moved the cursor beyond the end of the field, or the field was filled.
ExitCode% = 1	Shift-Tab or the up arrow key was pressed, or the left arrow moved the cursor before the beginning of the field.
ExitCode% = 2	Escape was pressed.

One important point to be aware of is that TextIn calls the CapNum subprogram to display the current setting of the Cap and NumLock keys. Therefore, any program that uses TextIn must also load or include CapNum.

Because TextIn is written in BASIC, it is simple to add additional exit codes or other features.

TextIn is shown in context in the DEMOIN.BAS example program.

VertMenu

BASIC subprogram contained in VERTMENU.BAS

Purpose:

VertMenu is a comprehensive menu subprogram with many important capabilities including full support for a mouse. It always saves the underlying screen, and draws an attractive shadow automatically.

Syntax:

```
CALL VertMenu(Item$(), Choice%, MaxLen%,BoxBot%, Ky$, Action%, Cnf)
```

Where:

Item$() is a conventional (not fixed-length) string array containing the list of menu choices and Choice% indicates which choice was selected, and may also be pre-loaded to force a given choice to be highlighted initially. Using a string of CHR$(196) characters tells VertMenu to display a non-selectable separator for that entry.

MaxLen% is the maximum length of any menu choice, thus establishing the menu width. Choices that are longer than MaxLen% will be displayed truncated.

BoxBot% is the bottom screen line that the window is to extend to. That is, if BoxBot% is set to twenty, then the bottom border of the menu will be on line twenty. Notice that the upper left corner of the menu is established by the current cursor location.

Ky$ holds the last key that was pressed by the user, Action% tells VertMenu how it is being used, and Cnf is a special TYPE variable that contains information about the host PC.

Comments:

VertMenu is explained in depth in the section entitled "Multi-Tasking Menus", and a complete demonstration is provided in the DEMOVERT.BAS example program.

VertMenuT

BASIC subprogram contained in VERTMENT.BAS

Many people have asked us for a version of VertMenu that will work with fixed-length string arrays. VertMenuT will do this, but there is one important limitation inherent in the program — a fixed-length string array may not be passed to a BASIC subprogram. The only real solution is to create a TYPE array comprised solely of a fixed-length string component, and then pass the TYPE array to the subprogram.

While this does indeed work, it restricts the subprogram to accepting an array of only that length. If you modify VertMenuT to work with array elements of, say, 30 characters in length, you will not be able to use the same subprogram with strings of any other length. The example program DEMOVRTT.BAS uses a length of twelve, and presents a list of file names from the current directory for selection.

YesNo

assembler subroutine contained in PRO.LIB

Purpose:

YesNo provides a quick way to accept a Yes or No input.

Syntax:

```
CALL YesNo(YN$, Prompt$, ScanCode%, NormalColor%, EditColor%, Row%, _
     Column%)
```

Where:

YN$ is either a single blank space, or a default answer of "Y" or "N".

Prompt$ is a message to be displayed immediately to the left of the input box, much like BASIC's INPUT command. If Prompt$ is null, then no prompt message is displayed.

ScanCode% tells how YesNo was terminated (see below).

NormalColor% is the color to restore the field to when YesNo has finished, and EditColor% is the color to use while the field is being edited.

Row% and Column% indicate where on the screen the prompt message and editing are to occur.

Comments:

As with most of the QuickPak Professional input routines, both the foreground and background colors are combined into a single value. The OneColor function may be used to calculate the correct color.

ScanCode% is returned holding either the ASCII value of the terminating key, or a negative value to indicate an extended key. This allows you to trap any Alt or function keys. For example, if ScanCode% is set to 13 the Enter key was used to exit YesNo. If ScanCode% is set to, say, -80, then the down arrow key was pressed.

YesNo will operate on whatever screen page is currently active, and is demonstrated in the DEMOCM.BAS example program.

YesNoB

BASIC subprogram contained in YESNO.BAS

Purpose:

YesNoB provides a quick way to accept a Yes or No input.

Syntax:

```
CALL YesNoB(YN$, ExitCode%, Colr%)
```

Where:

YN$ is either null or a default "Y" or "N".

ExitCode% tells how editing was terminated, as shown below.

Colr% is the field color to be used, and it is packed into a single byte containing both the foreground and background colors.

Comments:

ExitCode% lets your program know whether the user pressed Enter to accept the field, the up arrow to go to the previous field, or Escape.

ExitCode% = 0	Enter, Tab, or the down arrow key was pressed, or the right arrow moved the cursor beyond the end of the field, or the field was filled.
ExitCode% = 1	Shift-Tab or the up arrow key was pressed, or the left arrow moved the cursor before the beginning of the field.
ExitCode% = 2	Escape was pressed.

One important point to be aware of is that YesNoB calls the CapNum subprogram to display the current setting of the Cap and NumLock keys. Therefore, any program that uses YesNoB must also load or include CapNum.

Because YesNoB is written in BASIC, it is simple to add additional exit codes or other features.

YesNoB is shown in context in the DEMOIN.BAS example program.

Chapter 6
Keyboard/Mouse Routines

AltKey

assembler function contained in PRO.LIB

Purpose:

AltKey reports if the Alt key is currently depressed.

Syntax:

```
Active = AltKey%
```

Where:

Active receives -1 if the Alt key is currently down, or 0 if it is not.

Comments:

Because AltKey has been implemented as a function, it must be declared before it may be used.

AltKey is designed to return -1 for a true value to also allow the use of the BASIC NOT operator:

```
IF AltKey% THEN . . .
```

or

```
IF NOT AltKey% THEN . . .
```

ButtonPress

assembler subroutine contained in PRO.LIB

Purpose:

ButtonPress will report how many times a specified mouse button was pressed since the last time it was called. It also returns the X/Y coordinates where the mouse cursor was located when that button was last pressed.

Syntax:

```
CALL ButtonPress(Button%, Status%, Count%, X%, Y%)
```

Where:

Button% is the button of interest, with a 1 indicating button 1, 2 meaning button 2, and 3 for button 3 (if the mouse has a third button).

Status% is the *current* button status, and has the same meaning as the information returned by the GetCursor mouse routine.

Count% tells how many times the button has been pressed since ButtonPress was last called. X% and Y% hold the mouse cursor position at the time the button was pressed. Use the GetCursor routine to determine the current mouse cursor location.

Comments:

ButtonPress is the only reasonable way to determine when the mouse buttons are active and need attention. Though the GetCursor routine will report the *current* button status, it would have to be polled repeatedly in a loop.

A good example of implementing ButtonPress can be found in the source code for both PullDown and VertMenu.

The comments that accompany the GetCursor routine provide an explanation of interpreting the X and Y values that are returned.

Notice that unlike GetCursor, ButtonPress resets the button-press counter to zero each time it is called.

CapsLock

assembler function contained in PRO.LIB

Purpose:

CapsLock reports if the Caps lock key is currently depressed.

Syntax:

```
Active = CapsLock%
```

Where:

Active receives -1 if the CapsLock key is currently down, or 0 if it is not.

Comments:

Because CapsLock has been implemented as a function, it must be declared before it may be used.

CapsLock is designed to return -1 for a true value to also allow the use of the BASIC NOT operator:

```
IF CapsLock% THEN . . .
```

or

```
IF NOT CapsLock% THEN . . .
```

CapsOff and CapsOn

assembler subroutines contained in PRO.LIB

Purpose:

CapsOff turns off the CapsLock key status, and CapsOn turns it on.

Syntax:

```
CALL CapsOff
```

or

```
CALL CapsOn
```

ClearBuf

assembler subroutine contained in PRO.LIB

Purpose:

ClearBuf will quickly clear the keyboard of any key strokes that are currently pending.

Syntax:

```
CALL ClearBuf
```

Comments:

The most common way to clear the keyboard buffer in a BASIC program is to use an INKEY$ loop like this:

```
WHILE INKEY$ <> "": WEND
```

ClearBuf provides a simple and concise way to accomplish the same action with the fewest code bytes.

Also see the related functions InStat and PeekBuf.

CtrlKey

assembler function contained in PRO.LIB

Purpose:

CtrlKey reports if the Ctrl key is currently depressed.

Syntax:

```
Active = CtrlKey%
```

Where:

Active receives -1 if the Ctrl key is currently down, or 0 if it is not.

Comments:

Because CtrlKey has been implemented as a function, it must be declared before it may be used.

CtrlKey is designed to return -1 for a true value to also allow the use of the BASIC NOT operator:

```
IF CtrlKey% THEN . . .
```

or

```
IF NOT CtrlKey% THEN . . .
```

GetCursor

assembler subroutine contained in PRO.LIB

Purpose:

GetCursor reports the current location of the mouse cursor, and which mouse buttons are currently depressed.

Syntax:

```
CALL GetCursor(X%, Y%, Button%)
```

Where:

X% and Y% return holding the current mouse cursor coordinates, and Button% is bit coded to indicate which buttons are currently down.

Comments:

The X% and Y% values returned depend in part on the number of pixels that are available on the screen. Unfortunately, this is true even in text mode.

As an example, if the mouse cursor is currently at the upper left corner of the screen, both the X and Y values will be returned as zero. However, if the cursor moves one character box to the right, the X value will immediately become eight.

The same thing happens when the cursor is moved downward, in which case the Y value suddenly jumps to eight. Thus, the resolution of a 25-line text screen is considered (to the mouse anyway) as being the same as that of a CGA display.

Running the short program below will illustrate this:

```
PRINT "Press a button to stop"
WHILE Button% = 0
   CALL GetCursor(X%, Y%, Button%)
   LOCATE 1, 1
   PRINT X; Y
WEND
```

The 43-line text mode available with an EGA or VGA adapter is handled similarly. But with the 50-line mode in the OS/2 DOS compatibility box, the Y value is instead incremented in steps of 7. In graphics mode the cursor position is reported as you would expect, depending on the available resolution.

The button information is represented by bits in the Button% variable, with bit 0 being on to indicate that button 1 is pressed, bit 1 for button 2, and so forth. The various bits may be easily isolated as shown below:

```
CALL GetCursor(X%, Y%, Button%)
IF Button% AND 1 THEN PRINT "Button 1 is pressed"
IF Button% AND 2 THEN PRINT "Button 2 is pressed"
IF Button% AND 4 THEN PRINT "Button 3 is pressed"
```

Unlike the ButtonPress routine, GetCursor does not reset the count of how many times the buttons have been pressed.

GetCursorT

assembler subroutine contained in PRO.LIB

Purpose:

GetCursorT reports the current location of the mouse cursor, and which mouse buttons are currently depressed.

Syntax:

```
CALL GetCursorT(Col%, Row%, Button%)
```

Where:

Col% and Row% return holding the current mouse coordinates, and Button% is bit-coded to indicate which buttons are currently being pressed.

Comments:

GetCursorT serves the same purpose as the original GetCursor routine, except it returns the mouse cursor coordinates in terms of rows and columns rather than pixels. The row and column information are based at one. That is, if the mouse cursor is in the upper left corner of the screen, GetCursorT will return Col% and Row% equal to one, not zero.

GrafCursor

assembler subroutine contained in PRO.LIB

Purpose:

GrafCursor greatly simplifies defining the shape of the mouse cursor for use in graphics mode.

Syntax:

```
CALL GrafCursor(X%, Y%, Cursor$)
```

Where:

X% and Y% define the cursor "hot spot", and Cursor$ is either a conventional or fixed-length string that contains the new cursor shape.

Comments:

As with many of the mouse services that Microsoft has designed, defining the shape of the mouse cursor is at best convoluted and confusing.

The example in the MOUSE.BAS demonstration program shows how GrafCursor can be used with a "pictorial" layout to quickly visualize how the cursor will appear.

The hot spot indicates which pixel the mouse cursor is considered to be on when in graphics mode. Even though the mouse cursor will span several pixels at one time, only one point can be considered to be the actual cursor location. Again, the example in MOUSE.BAS shows this in context.

Also see GRAFCURS.BAS which expands on MOUSE.BAS to show how custom graphics mouse cursor shapes can be defined for EGA and VGA displays.

HideCursor

assembler subroutine contained in PRO.LIB

Purpose:

HideCursor turns off the mouse cursor.

Syntax:

```
CALL HideCursor
```

Comments:

Any program that is to be "mouse aware" will need to turn on the mouse cursor before expecting a user to access the mouse. Likewise, it is only common courtesy to turn it off again before returning them to the DOS prompt. One very important point to be aware of regarding the HideCursor routine is how the current on and off status is maintained internally by the mouse driver.

Unlike the normal text cursor that is turned on or off with the BASIC LOCATE command, the mouse cursor keeps track of how many times it was turned on or off. Thus, if you call HideCursor twice in a row, you'll have to call ShowCursor twice before it will be visible again! However, having multiple calls to ShowCursor still only requires a single call to HideCursor to turn off the mouse cursor.

This behavior is one of the biggest problems programmers encounter when attempting to add mouse support to a program. Send your complaints directly to:

Bill Gates
Microsoft Corporation
16011 NE 36th Way
Box 97017
Redmond, WA 98073-9717

Fortunately, the InitMouse routine will reset everything back to normal, though at the expense of losing all of the current mouse settings.

InitMouse

assembler subroutine contained in PRO.LIB

Purpose:

InitMouse is used both to determine if a mouse is present in the host PC, and to reset the mouse driver software to its default values.

Syntax:

```
CALL InitMouse(There%)
```

Where:

There% receives -1 if a mouse is present, or 0 if no mouse is installed.

Comments:

Because InitMouse resets the mouse driver values (the mouse cursor color, its travel range and sensitivity, etc.), it would probably be called only once at the start of a program.

Understand that InitMouse doesn't actually detect the physical presence of the mouse hardware. Rather, the mouse driver software must also be installed before a mouse will be detected.

InStat

assembler function contained in PRO.LIB

Purpose:

InStat returns the number of characters that are currently pending in the keyboard buffer, but without removing them.

Syntax:

```
Count = InStat%
```

Where:

Count receives the number of characters that are currently pending.

Comments:

Because InStat has been implemented as a function, it must be declared before it may be used.

InStat is very valuable in situations where you need to see if a key is present, but do not want to remove it from the keyboard buffer. A good example would be when simulating multi-tasking in a BASIC program.

Most menu programs just sit there waiting for a keystroke from the user, which is fine as long as there's no other work to be done. But if you would like to call a subprogram that reads a file or sorts an array in the background, you could design the subprogram to periodically take a look at the keyboard. Then, if it sees that a key has been pressed, it would immediately return to the menu, and the key would still be waiting there in the buffer.

Also see the related functions PeekBuf and ClearBuf.

Keyboard

assembler subroutine contained in PRO.LIB

Purpose:

Keyboard provides a continual display of the current Cap and NumLock status, but without having to loop repeatedly to obtain the information.

Syntax:

```
CALL Keyboard(Row%, Column%, Color1%, Color2%, Mode%)
```

Where:

Row% and Column% indicate where on the screen the status is to be displayed.

Color1% and Color2% tell Keyboard what colors to use when removing and displaying the "CAP" and "NUM" messages respectively.

Mode% is either 1 to install the Keyboard routine, or 0 to remove it.

Comments:

The usual way to provide a continual display of the Cap and NumLock status is to constantly peek at low memory, often while the program is waiting for a key press. Keyboard instead intercepts the keyboard hardware interrupt, thereby freeing the BASIC program from doing any additional work.

While ON TIMER could also be used to periodically display the Cap and NumLock status, most programmers prefer to avoid event trapping at any cost. Event statements unfortunately cause a BASIC program to be both larger and slower.

Keyboard may be installed more than once, with subsequent installations simply to tell Keyboard to use a new location or set of colors.

Because interrupts are being redirected to the Keyboard subroutine, it is imperative that you install and remove it correctly. Equally important is ensuring that Keyboard is removed within any error handling routines that may be present. We recommended that you add Keyboard to a program *only* after it has been thoroughly tested and debugged. If a program ends prematurely and Keyboard has not been removed, a total crash of the system is guaranteed.

Neither Keyboard nor its companion program Clock will operate correctly in the QuickBASIC 4 environment. Further, if both Keyboard and Clock are being used in the same program, Keyboard must be installed first and removed last.

A full demonstration of Keyboard is given in the DEMORK.BAS example program.

KeyDown

assembler function contained in PRO.LIB

Purpose:

KeyDown reports if any keys are currently being pressed.

Syntax:

```
KeyIsDown = KeyDown%
```

Where:

KeyIsDown is set to -1 (True) if a key is currently being pressed, or 0 if no keys are pressed.

Comments:

Because KeyDown has been designed as a function, it must be declared before it may be used. KeyDown must also be installed before it will operate, and this is down by calling the InstallKeyDown routine.

KeyDown is useful in a variety of situations. We designed it to allow the Dialog routine to mimic the behavior of Microsoft's dialog boxes, which act on the key pressed only after it has been released. Like the WaitUp routine that waits in an endless loop until all mouse buttons have been released, you could write a similar BASIC subprogram for the keyboard using KeyDown:

```
SUB WaitKeyUp STATIC
   DO
   LOOP WHILE KeyDown%
END SUB
```

In order to detect when keys are pressed and released, KeyDown takes over the keyboard interrupt. This is why it must be installed. KeyDown automatically removes itself from the interrupt chain when your program ends.

However, a bug in QBX (the QB editor that comes with BASIC 7 PDS) prevents the automatic de-installation from working correctly. Therefore, you must call DeinstallKeyDown manually before ending your program if you are using QBX. De-installing is not necessary with QuickBASIC 4.0 or 4.5, nor with programs that are compiled to .EXE files.

Note that when multiple keys are pressed (such as Alt-F), Keydown returns -1 when Alt is first pressed. But as soon as either combination key is released KeyDown returns 0.

See the KEYDOWN.BAS demonstration program for an example of using Keydown.

Motion

assembler subroutine contained in PRO.LIB

Purpose:

Motion allows a program to establish the sensitivity of the mouse cursor motion.

Syntax:

```
CALL Motion(Value%)
```

Where:

Value% is the desired sensitivity ranging between 1 and 32767, with 1 being the most sensitive.

Comments:

Even though the mouse driver software allows setting the horizontal and vertical sensitivity separately, Motion uses the same value for both. This seems to be the most logical way to control a mouse, while eliminating yet another passed parameter. If you absolutely must be able to set these values independently, you should use the generic Mouse routine provided with QuickPak Professional like this:

```
CALL Mouse(15, 0, X%, Y%)
```

Where X% and Y% represent the sensitivity for the X and Y coordinates respectively.

The stated upper range for the motion sensitivity is 32767, however values beyond 100 or so are hopelessly insensitive.

You may be interested to know that Microsoft calls the unit of cursor distance for the mouse a "Mickey".

Mouse

assembler subroutine contained in PRO.LIB

Purpose:

Mouse provides access to all of the mouse services, and is the only way to use those that are not provided in a simplified form with QuickPak Professional.

Syntax:

```
CALL Mouse(AX%, BX%, CX%, DX%)
```

Where:

AX% is the number for the mouse service of interest, while BX%, CX%, and DX% assign and return the processor's registers.

Comments:

Mouse provides access to all of the mouse services. Most of the important ones are provided as a simplified call with QuickPak Professional. But there may be occasions when you need a mouse capability that we have not included.

Three important mouse services you may want to add to your programs are those that size, save, and restore the current mouse state. These are used extensively by the VertMenu subprogram to let it query the mouse buttons, but not destroy the information that is returned.

Similar to INKEY$, when ButtonPress calls the mouse driver software, the very act of requesting the number of button presses removes them from the internal "button buffer". If a button was pressed outside of the range that VertMenu is interested in, it first restores the mouse state, and then returns to the calling program. Of course, VertMenu saves the mouse state just prior to each call to ButtonPress.

Before the current mouse state can be saved, you must first ask the mouse driver how many bytes will be needed. The mouse state of course consists of more than just the button information. It also includes the cursor color, the cursor boundaries, and all of the other mouse parameters.

The number of bytes needed to save the current mouse state is obtained with Mouse service number 21. Next, a string must be assigned to a sufficient length to hold the information. Finally, Service 22 is used to copy the state to the string.

```
CALL Mouse(21, Bytes%, 0, 0)           'request the state size
Storage$ = SPACE$(Bytes%)              'make a string to hold it
CALL Mouse(22, 0, 0, SADD(Storage$)    'load the string
```

Restoring the mouse state again later is equally simple:

```
CALL Mouse(23, 0, 0, SADD(Storage$)    'restore the state
Storage$ = ""                          'free up the memory
```

MouseRange, 1, G, G1

assembler routine contained in PRO.LIB

Purpose:

MouseRange will return a range number that tells where the mouse cursor is located, based on an array of screen coordinates.

Syntax:

```
CALL MouseRange(SEG Array(Start), NumEls%, Row%, Col%, Button%, _
     RangeNum%)
```

Where:

Array(Start) is the first element in a special TYPE array, and NumEls% is the number of elements in that array. Row% and Col% return the current location of the mouse cursor, Button% indicates which mouse button is currently depressed, and RangeNum% holds the range within which the cursor is located.

Comments:

When writing a program that manages multiple windows at once, it can be very tedious to calculate in which window the mouse cursor is when a button has been pressed. MouseRange lets you create a table of screen coordinates that define the boundaries of each window, and it then does all of the searching to see within which window the mouse cursor is located. It can therefore be used to replace a call to GetCursor followed by many IF/ELSEIF or CASE statements.

Before MouseRange may be called you must first define a TYPE array, and dimension it to the number of windows that will be active at one time. This is shown below.

```
TYPE Area
     ULRow AS INTEGER
     ULCol AS INTEGER
     LRRow AS INTEGER
     LRCol AS INTEGER
     Alias AS INTEGER
END TYPE
DIM Array(NumWindows) AS Area
```

It is up to your program to fill in the corner parameters for each element in the array before calling MouseRange. The Alias portion of each TYPE element lets you assign window numbers that are not necessarily contiguous. For example, you could have five windows numbered 1, 3, 12, 22, and 7. These are the numbers that would be returned to you by MouseRange.

Setting an Alias parameter to -1 tells MouseRange to return the physical window number. For example, if the mouse cursor was in the range defined by Array(3), then RangeNum% would be set to 3.

Notice that ALIAS is a BASIC reserved word, and is used here for explanation purposes only.

The windows may of course overlap each other, and the highest window number that matches the mouse cursor location is the one that will be reported. This logic assumes that as each new window is opened it will be added to the next higher array element, and that you are interested in the most recently opened window in which the mouse cursor is located.

These are four versions of MouseRange supplied with QuickPak Professional, and all of them are set up and called the same way. MouseRange accepts ranges of screen coordinates in row and column units, and compares the current mouse position only when a button is pressed. If a button has not been pressed RangeNum is returned holding 0; however, Row and Column will hold the current mouse cursor location. When a mouse button has been pressed, MouseRange compares the current mouse location with the passed array of coordinates, and returns which range it falls in. MouseRangeG works in the same way, but all parameters are given and returned as pixels instead of rows and columns.

MouseRange1 and MouseRangeG1 are similar except they always return the appropriate range value, regardless of whether or not a mouse button is currently depressed. Note that these versions are slightly slower due to the constant comparisons required.

MouseRange is demonstrated in the MRANGE.BAS example program.

MouseState

assembler routine contained in PRO.LIB

All mouse drivers provide a way for a program to retrieve the current context of the mouse state. This includes the current mouse location, shape, and the number of times each button has been pressed. There may be situations where you want to save the current mouse state, and then restore it again later. For example, the QuickPak Professional PullDown menu program saves and restores the state to allow it to operate in a polled mode.

When a program calls ButtonPress to obtain the button press history for a given button, ButtonPress resets the button history counter. Therefore, PullDown first saves the mouse state, and then checks for button presses within its own active window. If a button was pressed outside of the current menu window, PullDown restores the mouse state and returns. This way another routine can detect and act on those button presses. Otherwise, PullDown acts on that button press.

Three routines are provided to save and restore the mouse state. MBuffSize is a function that must be declared, and it returns the size of the mouse state buffer. This information is needed to know how large a string must be in order to receive the information the mouse driver returns. MGetState and MSetState may then be used to obtain and restore the current state. A typical session would be as follows:

```
DECLARE FUNCTION MBuffSize%()
Buffer$ = SPACE$(MBuffSize%)       'make a string
CALL MGetState(Buffer$)            'load the current state
    ...                            'do whatever is needed
    ...
CALL MSetState(Buffer$)            'restore the state
```

MouseTrap

assembler subroutine contained in PRO.LIB

Purpose:

MouseTrap will establish the allowable range of movement for the mouse cursor.

Syntax:

```
CALL MouseTrap(ULRow%, ULCol%, LRRow%, LRCol%)
```

Where:

ULRow% and ULCol% specify the upper left corner of the range, and LRRow% and LRCol% indicate the bottom right boundary.

Comments:

MouseTrap is intended for use with the mouse text cursor, but in any screen mode. This is why the coordinates are given in rows and columns. The mouse services that trap the mouse cursor range (hence the name) normally expect you to supply the boundaries in pixels, even when the screen is in a text mode. (Ouch!)

MouseTrap accepts the row and column range you specify, and converts it to the correct pixel values before calling the mouse driver software. It will accommodate the 40 and 80 column modes automatically, as well as 25, 43 or 50 lines.

Notice that when an EGA (or VGA) is using a high resolution graphics screen *and* you have used WIDTH to set 43 or 50 text lines, you must specify the vertical bounds in pixels using the generic Mouse routine. You may prefer to use pixels in the graphics modes anyway, as shown below:

```
CALL Mouse(Service%, 0, Min%, Max%)
```

Service% is either 8 to establish the vertical range, or 7 to set the horizontal range, and Min% and Max% are the pixel numbers that define the limits. Notice that the lowest possible values are 0, not 1, and the highest value will depend on the current screen mode.

Also see the discussion of mouse coordinates that accompanies the GetCursor routine description.

NumLock

assembler function contained in PRO.LIB

Purpose:

NumLock will quickly tell if the NumLock key is currently depressed.

Syntax:

```
Active = NumLock%
```

Where:

Active receives -1 if the NumLock key is currently down, or 0 if it is not.

Comments:

Because NumLock has been implemented as a function, it must be declared before it may be used.

NumLock is designed to return -1 for a true value to allow the use of the BASIC NOT operator:

```
IF NumLock% THEN . . .
```

or

```
IF NOT NumLock% THEN . . .
```

NumOff and NumOn

assembler subroutines contained in PRO.LIB

Purpose:

NumOff turns off the NumLock key status, and NumOn turns it on.

Syntax:

```
CALL NumOff
```

or

```
CALL NumOn
```

PeekBuf

assembler function contained in PRO.LIB

Purpose:

PeekBuf provides a handy way to determine what key if any is the next one pending in the keyboard buffer, without actually removing it as INKEY$ does.

Syntax:

```
Char = PeekBuf%
```

Where:

Char receives the ASCII value of the pending key if it is a normal key, 0 if no keys are pending at all, or a negative value representing an extended key's scan code.

Comments:

Because PeekBuf has been implemented as a function, it must be declared before it may be used.

PeekBuf is very valuable in situations where you need to know if a key is present and also which one, but do not want to physically remove it from the keyboard buffer.

Also see the related functions InStat and ClearBuf.

RptKey

assembler subroutine contained in PRO.LIB

Purpose:

RptKey works much like BASIC's INKEY$, however it also returns the number of times an Alt, Ctrl, or Shifted key has been pressed.

Syntax:

```
CALL RptKey(Char%, Count%)
```

Where:

Char% is the ASCII value for a normal key, or the scan code for an extended key if Count% is not zero.

Count% is the number of times the key was pressed before the Alt, Ctrl, or Shift key was released.

Comments:

RptKey was inspired by the way that Borland's SideKick handles the PgUp and PgDn keys. Rather than attempt to repeatedly redraw the screen when these keys are held down, it instead waits until the key is finally released, and refreshes the screen only once.

Understand that RptKey works only with keys that are used in combination with Alt, Ctrl, or Shift. However, it is quite handy in those cases.

ScrlLock

assembler function contained in PRO.LIB

Purpose:

ScrlLock reports if the Scroll lock key is currently depressed.

Syntax:

```
Active = ScrlLock%
```

Where:

Active receives -1 if the Scroll lock key is currently active, or 0 if it is not.

Comments:

Because ScrlLock has been implemented as a function, it must be declared before it may be used.

ScrlLock is designed to return -1 for a true value to also allow the use of the BASIC NOT operator:

```
IF ScrlLock% THEN . . .
```

or

```
IF NOT ScrlLock% THEN . . .
```

SetCursor

assembler subroutine contained in PRO.LIB

Purpose:

SetCursor provides a simple way to set a new location for the mouse cursor.

Syntax:

```
CALL SetCursor(X%, Y%)
```

Where:

X% and Y% represent the new horizontal and vertical positions respectively.

Comments:

The valid X and Y coordinates you specify will depend on the current screen mode. For example, on a CGA graphics screen 1, the acceptable range would be between 0 and 319 for X%, and 0 to 199 for Y%.

Also see the comments and complaints about the mouse cursor coordinates in the sections that describe the GetCursor and HideCursor routines.

ShiftKey

assembler function contained in PRO.LIB

Purpose:

ShiftKey reports if either Shift key is currently depressed.

Syntax:

```
Active = ShiftKey%
```

Where:

Active receives -1 if either Shift key is currently down, or 0 if neither is.

Comments:

Because ShiftKey has been implemented as a function, it must be declared before it may be used.

ShiftKey is designed to return -1 for a true value to also allow the use of the BASIC NOT operator:

```
IF ShiftKey% THEN . . .
```

or

```
IF NOT ShiftKey% THEN . . .
```

ShowCursor

assembler subroutine contained in PRO.LIB

Purpose:

ShowCursor turns the mouse cursor on making it visible.

Syntax:

```
CALL ShowCursor
```

Comments:

For more information, see the comments and complaints that accompany the companion routine HideCursor.

StuffBuf

assembler subroutine contained in PRO.LIB

Purpose:

StuffBuf will insert a string into the keyboard buffer as if it had been entered at the keyboard manually.

Syntax:

```
CALL StuffBuf(X$)
```

Where:

X$ is a string containing up to fifteen keystrokes.

Comments:

StuffBuf is the key to many feats of seemingly magical programming. For example, it is the *only* way that a BASIC program can be coerced into running a .COM program or batch file. It could also be used just before ending a program to feed keystrokes in a subsequent application:

```
CALL StuffBuf("123" + CHR$(13) + CHR$(13) + "/FR")
END
```

When this program ends, it will start Lotus 123, bypass the opening title screen, and obtain a list of the available worksheets automatically.

The real beauty of StuffBuf, though, is being able to create and run a batch file. In fact, it is an important component in our QuickMENU program.

Of course, SHELL can be used to run a batch file temporarily, but that much less memory would be available to any programs the batch file runs. Further, some networks do not allow SHELL, and this is the only way it can be done.

StuffBuf is used to advantage in the QuickPak Professional spreadsheet program, where a key must be examined to see if the "LABEL" or "VALUE" indicator should be displayed. Once the key has been accepted, it is then placed back into the keyboard buffer where the Editor routine can use it too.

StuffBuf allows both normal and extended keystrokes to be placed into the keyboard buffer. The example below shows how to specify the characters "ABC" followed by the F1 function key:

```
CALL StuffBuf("ABC" + CHR$(0) + CHR$(59))
```

It is imperative that no more than fifteen keystrokes be specified when using StuffBuf. However, extended keys that are preceded with a CHR$(0) count as only one keystroke.

TextCursor

assembler subroutine contained in PRO.LIB

Purpose:

TextCursor provides an easy way to initialize the mouse cursor in text mode, and define its color.

Syntax:

```
CALL TextCursor(FG%, BG%)
```

Where:

FG% and BG% indicate the colors to be used.

Comments:

The Microsoft mouse driver software always displays the cursor when the ShowCursor routine is called and the screen is in text mode. However, some of the "clone" mouse drivers do not. To be sure that the mouse cursor will be displayed, call TextCursor once at the beginning of your program. This is not needed if the program is operating in a graphics mode.

Besides allowing you to initialize the mouse text cursor, the TextCursor routine has a few other interesting capabilities as well. Normally, defining the text cursor color is at best an exercise in frustration. For example, the Microsoft Mouse Programmer's Reference Guide devotes four pages to charts and examples, just to explain how the color system works. If you don't have a copy of this otherwise useful book, trust me, it is *very* confusing.

TextCursor greatly simplifies this, and it allows you to set new colors in a variety of ways. For example, if you use 0 for FG% and 4 for BG%, then when the mouse cursor passes over a character on the screen, the character will be displayed in black on a red background. This is the color scheme used by the QuickBASIC editor.

Any other color combination will be drawn as you specify, though understand that the mouse cursor color is really the background variable. The foreground value merely indicates what color the text is to become when the cursor is on it.

But TextCursor also takes one of two possible codes for either the foreground, the background, or both. If either color parameter is set to -1, then you are telling TextCursor to maintain that portion (foreground or background) of the current character color when the mouse cursor passes over it. Of course, if both FG% and BG% are set to -1, you'll never see the cursor.

When either parameter is assigned to -2, then the character's foreground or background color will become "inverted" as the mouse cursor passes over it. For example, White will become Black, Green turns to Magenta, and Blue becomes Brown.

WaitKey

assembler subroutine or function contained in PRO.LIB

Purpose:

WaitKey first clears the keyboard buffer of any keys that may be pending, and then waits until a key is pressed.

Syntax:

```
CALL WaitKey                      'if declared as a subprogram
```

or

```
X = WaitKey%                      'if declared as a function
```

Where:

If WaitKey has been declared as a function, X will receive either the ASCII value for the key that was pressed, or a negative number representing an extended key.

Comments:

The usual way to pause for a key press is with an empty INKEY$ loop, such as:

```
WHILE INKEY$ = "": WEND
```

However, when you are asking a user to acknowledge an error it is generally a good idea to ensure that a stray key stroke waiting in the keyboard buffer is not inadvertently accepted as a confirmation. This further complicates the necessary code:

```
WHILE INKEY$ <> "": WEND          'clear any pending key strokes
WHILE INKEY$ = "": WEND           'wait for a key
```

WaitKey simplifies this by performing both operations in a single routine. If WaitKey is declared as a subprogram, calling it will cause the PC to pause until a key is pressed. If it has been declared as a function, then the key that was pressed will also be available for examination by your program.

WaitScan

assembler function contained in PRO.LIB

Purpose:

WaitScan waits for any key to be pressed, and then returns the actual keyboard scan code for that key.

Syntax:

```
Keycode = WaitScan%
```

Where:

KeyCode receives the scan code for the key that was pressed, including otherwise illegal keys such as "5" on the number pad, or the Alt key by itself.

Comments:

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

We do not recommend using WaitScan for several reasons; however, a number of customers have asked for it. One problem with WaitScan is that different keyboards produce different scan codes for the same key. For example, the original IBM PC keyboard uses different codes than the enhanced (AT style) keyboard, which in turn is different from the keyboards provided with the Tandy 1000 series of computers.

Another problem is that WaitScan is not compatible with SideKick. Because SideKick "steals" the keyboard interrupt even after it has been loaded, there is simply no way to make WaitScan work correctly in that case.

WaitScan operates by first installing itself in the keyboard interrupt chain, and then waits for a key to be pressed. After remembering the scan code for the key, the original keyboard interrupt is reinstated.

WaitUp

assembler subroutine contained in PRO.LIB

Purpose:

WaitUp halts a program's execution until no mouse buttons are being depressed.

Syntax:

```
Call WaitUp
```

Comments:

WaitUp merely waits in an endless loop until no mouse buttons are being pressed. This is similar to waiting in an INKEY$ loop until there are no keys still pending in the keyboard buffer. Of course, WaitUp returns immediately if no mouse is installed.

WaitUp is useful in those situations where clicking on a choice invokes another routine that also polls for mouse button presses. To avoid having the subsequently called routine act on the earlier presses, it is necessary to pause until all buttons are released.

A call to WaitUp creates only 5 bytes of compiled code, which is much more efficient than the equivalent BASIC loop shown below:

```
DO
  Call GetCursor(X%, Y%, Button%)
LOOP WHILE Button%
```

Chapter 7
Miscellaneous Routines

AddUSI

assembler function contained in PRO.LIB

Purpose:

AddUSI adds two integers on an unsigned basis, without creating an overflow error if the total exceeds 32767.

Syntax:

```
Sum = AddUSI%(X%, Y%)
```

Where:

X% and Y% are the integers to be added, and Sum% receives the result.

Comments:

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

In Microsoft BASIC, integer numbers and variables may range from -32768 to 32767, which is a total of 65536 possible values. However, assembler routines can optionally consider this same range of values as spanning from 0 to 65535. Therefore, a variable's address that is greater than 32767 will be reported by BASIC's VARPTR as a negative value. This behavior can cause problems when addresses that may exceed 32767 are being manipulated. For example, if a TYPE variable begins at address 32765 and 10 is added to access a portion of the TYPE, QuickBASIC will create an Overflow error.

AddUSI simply adds the two values in assembly language, which of course does not generate errors. Even if the result exceed 65535, the adding will merely "wrap" around and pass zero. AddUSI is invaluable in those situations where BASIC might create an error even when the values are legitimate.

ASCIIChart

BASIC subprogram contained in ASCCHART.BAS

Purpose:

ASCIIChart is meant to be used as a pop-up utility in your programs. When called it saves the underlying screen, and accepts the up and down arrow keys, the PgUp and PgDn keys, or Escape to exit.

Syntax:

```
CALL ASCIIChart(ULRow%, ULCol%, Height%)
```

Where:

ULRow% and ULCol% tell where to locate the display, and Height% specifies how many rows it is to occupy.

Comments:

ASCIIChart adjusts the way PgUp and PgDn work to accommodate the current height automatically. As shipped, the box is drawn in yellow on red, and the characters are bright white on red. The colors are defined at the very beginning of the source code, and may be easily changed. Run the COLORS.BAS program to see a list of the possible color combinations.

ASCIIChart is shown in context in the DEMOPOP.BAS example program.

BCopy

assembler subroutine contained in PRO.LIB

Purpose:

BCopy will copy a block of memory (up to 64K in size) to a new location.

Syntax:

```
CALL BCopy(FromSeg%, FromAddr%, ToSeg%, ToAddr%, NumBytes%, _
     Direction%)
```

Where:

FromSeg% and FromAddr% indicate the source location of the block, and ToSeg% and ToAddr% tell where to copy it to. NumBytes% is, well, you can figure that one out. Direction% is either 0 to copy in the forward direction, or -1 to copy backwards.

Comments:

BCopy is useful in a variety of situations, for example if you need to make a copy of an array or duplicate a range of elements.

The number of bytes may be up to 65535, though you will have to use a long integer (or a negative number) to specify a value greater than 32767.

BCopy could also be used to swap strings in and out of a dynamic array in far memory to conserve string space, as shown below.

```
X$ = "This is a test"              'make a test string
L% = LEN(X$)                       'remember its length
DIM A%(L% / 2)                     'make an array to hold it
CALL BCopy(GetDS%, SADD(X$), VARSEG(A%(0)), VARPTR(A%(0)), L%, 0)

N$ = SPACE$(L%)                    'make a new string
CALL BCopy(VARSEG(A%(0)), VARPTR(A%(0)), GetDS%, SADD(N$), L%, 0)
PRINT N$                           'show that it worked
```

Here we're using the QuickPak Professional GetDS% function along with BASIC's SADD to locate the string in near memory, and VARSEG and VARPTR to find where the array is in far memory. BCopy may then be used both to copy the string to the array, as well as back again when the string is needed.

You could also use VARSEG instead of GetDS% to specify the segment for any type near data. However, if you are copying to or from a string using BASIC PDS far strings, you must instead use SSEG.

BCopyT

assembler subroutine contained in PRO.LIB

Purpose:

BCopyT will copy one or more elements in a TYPE array to another array, or to any location in memory. Alternately, BCopyT can be used to move any contiguous block of memory, even if the number of bytes exceeds 65536.

Syntax:

```
CALL BCopyT(SEG FromEl, SEG ToEl, ElSize%, NumEls%)
```

or

```
CALL BCopyT(BYVAL FromSeg%, BYVAL FromAdr%, BYVAL ToSeg%, _
    BYVAL ToAdr%, NumBytes%, 1)
```

Where:

FromEl is the starting element of the source array and ToEl is the starting element in the destination array. ElSize% is the size of each element in bytes, or a special code (see below). NumEls% is the number of elements to be copied.

With the alternate call form, FromSeg%, FromAdr%, ToSeg%, and ToAdr% define the source and destination segments and addresses. NumBytes% then tells how many bytes are to be copied.

Comments:

Where the original BCopy routine is intended for copying a block of memory up to 64K in length, BCopyT does not have that limitation. BCopyT is also designed to simplify the calling syntax when TYPE variables and array elements are being moved.

The first syntax shown above is for copying one or more elements in a numeric or TYPE array, while the second would be used for copying any contiguous block of memory. The actual number of bytes to copy is calculated within BCopyT by multiplying ElSize% times NumEls%. If you intend to copy 32K bytes or less (32768), simply set ElSize% to the number of bytes, and use 1 for NumEls%. To copy, say, 128K, you could set ElSize% to 32768 and use 4 for NumEls%. Any similar combination will also work.

BCopyT also recognizes the special codes used by the various TYPE sort routines for ElSize%, as shown in the table below.

-1	2-byte integer	
-2	4-byte long integer	}these two are the same,
-3	4-byte single precision	}and are interchangeable
-4	8-byte double precision	
+n	TYPE variable of length n	

Because this routine must be able to work with any type of variable, you should use the AS ANY option when declaring it:

```
DECLARE SUB BCopyT(SEG FromEl AS ANY, SEG ToEl AS ANY, _
    ElSize%, NumEls%)
```

If you intend to use both syntax forms in the same program, then you must not declare it at all.

BLPrint

assembler subroutine contained in PRO.LIB

Purpose:

BLPrint is an LPRINT substitute that eliminates the need for ON ERROR in case the printer is off line, or becomes unavailable during printing.

Syntax:

```
CALL BLPrint(LPTNumber%, X$, ErrCount%)
```

Where:

LPTNumber% is either 1, 2, or 3 to indicate which parallel printer to use, and X$ is the string to be printed. ErrCount% reports if the string was printed without error. If an error occurs, ErrCount% instead reports how many characters were successfully printed.

Comments:

Even though the PRNReady routine will report if a printer is ready at the time it is called, it does not prevent against an error caused by the printer running out of paper, or being placed off-line in the middle of a job.

Because BLPrint goes directly to the PC's BIOS, BASIC does not get a chance to add a carriage return or line feed to the string. You must do this in your BASIC program, as shown below. However, this also provides you with greater control over how the text is printed.

```
CALL BLPrint(1, X$ + CHR$(13) + CHR$(10), ErrCount%)
```

For example, if you omit the CHR$(10) line feed, the print head will be returned to the beginning of the same line, without advancing to the next one. This lets you overstrike characters or easily perform underlining and bold printing, without having to know those codes for a particular printer.

If the entire line was printed successfully, BLPrint returns -1 as a status code. Any other value indicates the number of characters that were actually printed, which allows you to resume printing at the correct place.

In the tests we performed, most printers accepted either the entire line or nothing at all. That is, deselecting the printer in the middle of a line was not detected until the line was finished. However, because some printers (most notably the Hewlett-Packard LaserJet) do not work this way, you should use RIGHT$ as shown in the BLPRINT.BAS example program in case only part of the line was printed.

Calc

BASIC subprogram contained in CALC.BAS

Purpose:

Calc provides a handy pop-up calculator that can be added to your BASIC programs.

Syntax:

```
CALL Calc(ULRow%, ULCol%, FG%, BG%)
```

Where:

ULRow% and ULCol% tell where the upper-left corner of the calculator is to be located, and FG% and BG% are the foreground and background colors to use.

Comments:

Besides its calculating abilities, Calc can also send its output to a printer. Instructions to begin printing are on the screen, so no additional training is required for those people who are using your program.

Eighteen lines are required on the screen, so be sure to take that into account when locating the calculator display. Of course, Calc works equally well in the 43 or 50 line screen modes available with an EGA and VGA display adapter.

You may also consider using the NumOn routine just prior to calling Calc, to allow the operator to use the keys on the number pad. Of course, you should also call NumOff when they are finished.

Calc is shown in context in the DEMOPOP.BAS example program.

Calendar

BASIC subprogram contained in CALENDAR.BAS

Purpose:

Calendar is a pop-up calendar that will display any month of any year.

Syntax:

```
CALL Calendar(Month%, Day%, Year%, ULRow%, ULCol%, Color1%, _
     Color2%, Action%)
```

Where:

Month%, Day%, and Year% tell Calendar what month to display and what day to highlight, and ULRow% and ULCol% specify the upper left corner of the window.

Color1% controls the border color, and Color2% is for the rest of the display.

Action% is either set to 1 to display the calendar, or 0 to remove it and restore the original underlying screen.

Comments:

Calendar highlights the specified day by reversing the foreground and background components of Color2%. If you do not want any day to be highlighted, set Day% to 0 before calling Calendar.

Calendar is shown in context in the DEMOPOP.BAS example program. Also, see the COLORS.BAS program description for a table of color combinations.

Chime

assembler subroutine contained in PRO.LIB

Purpose:

Chime provides five different types of beep tones, and five short attention-getting trill sounds.

Syntax:

```
CALL Chime(Number%)
```

Where:

Number% is between 1 and 10.

Comments:

When someone using your program presses the wrong key or is about to overwrite a file, you probably want to notify them with a sound as well as a warning message. Many programmers use the BEEP statement for this purpose. While BEEP is certainly adequate, it is admittedly a rather boring sound.

SOUND and PLAY allow nearly any combination of tones, however using either adds considerably to size of a program. In a test program compiled by QuickBASIC 4.5 using a single SOUND statement increased the code size by 11K, while PLAY added a whopping 14.5K. Chime offers a variety of short sounds, but adds less than two hundred bytes to your program.

The only real difference between the sounds created by Chime and those produced by PLAY or SOUND is that tones are played in the "foreground" only. Where BASIC's PLAY and SOUND return to your program immediately and continue playing the notes in the background, Chime waits until the tones have completed before it returns.

The tone sequences used by Chime are stored in a table which is kept in the code segment. Storing data in the code segment prevents it from stealing string space from your programs. The table is organized such that it is easy to modify or expand, as shown in the comments in the assembler source code.

Chime is demonstrated in the QPSOUND.BAS example program.

Clock and Clock24

assembler subroutines contained in PRO.LIB

Purpose:

Clock provides a continual display of the current time, but without having to loop repeatedly in the BASIC program. Clock24 is identical, except it displays the time using the 24-hour military format.

Syntax:

```
CALL Clock(Row%, Column%, Colr%, Mode%)
```

Where:

Row% and Column% indicate where on the screen the time is to be displayed, Colr% tells Clock what color to use, and Mode% is either 1 to install the routine, or 0 to remove it.

Comments:

The usual way to provide a continual display of the time is to constantly print it in a loop, often while the program is waiting for a key press. Clock instead intercepts the timer hardware interrupt, thereby freeing the BASIC program from doing any additional work.

While ON TIMER could also be used, most programmers prefer to avoid event trapping at any cost, because those statements cause a program to be both larger and slower.

Clock may be installed more than once, with subsequent installations simply to establish a new location or color.

Because interrupts are being redirected to the Clock routine, it is imperative that you install and remove it correctly. Equally important is ensuring that Clock is removed within any error handling routines that may be present.

We recommended that you add Clock to your program *only* after it has been thoroughly tested and debugged. If a program ends prematurely and Clock has not been removed, a total crash of the system is guaranteed.

If both Clock and Keyboard are being used in the same program, Clock must be installed last and removed first.

One important warning you should be aware of is when chaining from a program that uses Clock. It is essential to add a brief delay after uninstalling Clock, and before issuing the CHAIN command. Otherwise, the chained-to program may overwrite the clock routine before it has released the timer interrupt. The following code is recommended:

```
CALL Clock(Row%, Column%, Colr%, 0)
CALL Pause(1)
CHAIN "Program"
```

Also see the COLORS.BAS program description for a table of color combinations.

A complete demonstration of Clock is given in the DEMORK.BAS example program.

Compare

assembler function contained in PRO.LIB

Purpose:

Compare will compare any two blocks of memory, and report if they are the same.

Syntax:

```
Same = Compare%(Seg1%, Adr1%, Seg2%, Adr2%, NumBytes%)
```

Where:

Seg1% and Adr1% tell where the first block is located, and Seg2% and Adr2% point to the second. NumBytes% is either the number of bytes to be compared, or a special code indicating a type of variable (see below). Same then receives -1 if the two blocks are identical, or 0 if they are not.

Comments:

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

Not unlike AddUSI, Compare was designed to serve a very special, but important purpose. Because the KeySort subprogram must be able to work with any arrangement of a TYPE array, it is mandatory that the element comparisons be performed by examining memory. For example, we couldn't use something like:

```
IF Array(X).StringPart = Array(X + 1).StringPart
```

The whole point of a TYPE array is that the programmer can decide how it is to be organized. But BASIC doesn't provide a way to get at individual members of a TYPE variable or array element, other than by specifying its name. Thus, Compare provides an easy solution to the problem.

However, there are no doubt other, similar situations where Compare could be useful. For example, to see if a block of elements in one array is the same as those in a second one.

NumBytes% may also be coded using the same system as the QuickPak Professional sort routines:

-1	2-byte integer	
-2	4-byte long integer	} these are the same and
-3	4-byte single precision	} are interchangeable
-4	8-byte double precision	
+n	n-byte fixed-length string or any block of data	

CompareT

assembler function contained in PRO.LIB

Purpose:

CompareT will compare any two TYPE variables, and report if they are the same.

Syntax:

```
Same = CompareT%(SEG Type1, SEG Type2, NumBytes%)
```

Where:

Type1 and Type2 are two TYPE variables or elements in a TYPE array, and NumBytes% indicates their length in bytes. Same then receives either -1 if they are the same, or zero if they are not.

Comments:

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

Even though QuickBASIC will allow you to assign one TYPE variable to another or exchange them via the SWAP statement, it is not possible to simply compare them. For example, either of the statements below will result in an error message:

```
IF Type1 = Type2 THEN . . .
```

or

```
IF Type1 <> Type2 THEN . . .
```

Before CompareT, the only solution was to compare each component of the TYPE individually:

```
IF Typ1.A = Typ2.A AND Typ1.B = Typ2.B AND Typ1.C = Typ2.C AND...
```

This quickly becomes tedious when there are many components to each TYPE variable. Many individual comparisons also create a lot of wasted code that takes an undue amount of time to execute.

CompareT is modeled after the more generalized Compare routine, but it is designed to be simpler to use when comparing two TYPE variables.

Notice that when declaring CompareT you must use the "AS ANY" option so QuickBASIC will let you pass it any type of variable:

```
DECLARE FUNCTION CompareT%(SEG X AS ANY, SEG Y AS ANY, NumBytes%)
```

Also notice that once the function has been declared using SEG, it is not necessary to use SEG later each time CompareT is used.

Date2Day

BASIC function contained in DATE2DAY.BAS

Purpose:

Date2Day accepts an incoming date string, and returns the appropriate day of the week (1 - 7).

Syntax:

```
Day = Date2Day%(D$)
```

Where:

D$ is a string such as "MMDDYY", "MM-DD-YYYY", and so forth, and Day receives a value corresponding to the day of the week.

Comments:

Because Date2Day has been designed as a function, it must be declared before it may be used.

Also see the related BASIC function Num2Day, that instead accepts the incoming date as an integer.

Date2Num

assembler function contained in PRO.LIB

Purpose:

Date2Num converts a date in string form to an equivalent integer variable.

Syntax:

```
Days = Date2Num%(D$)
```

Where:

D$ is a date in the form of "MMDDYY" or "MM-DD-YY" or "MM/DD/YYYY", or any such combination, and Days receives the number of days before or after 12-31-1979.

Comments:

Because Date2Num has been designed as a function, it must be declared before it may be used.

Date2Num is a very powerful routine with two important uses. Besides allowing what would otherwise be an eight character string to be packed to only two bytes, it also provides an easy way to perform date arithmetic.

Date2Num will operate on any date that is within the range 01-01-1900 to 11-17-2065. Invalid dates and dates that fall outside of that range will return -32768 to indicate an error.

Once a date has been converted to the equivalent integer value, you may add or subtract a number of days, and then use the companion function Num2Date to convert the result. The example below shows this in context:

```
D$ = "09-17-88"
Start% = Date2Num%(D$)
Later% = Start% + 30
After30$ = Num2Date$(Later%)
PRINT "Thirty days after "; D$; " is "; After30$
```

Because Date2Num and Num2Date are set up as functions they may also be used within a print statement directly, along with optional calculations:

```
PRINT "30 days after"; D$; " is " Num2Date$(Start% + 30)
```

Date2Num and Num2Date are also useful for verifying if a given date is valid, which eliminates tedious calculations that you would have to perform to take possible leap years into consideration.

The only requirement for the date validation example below is that the original date must be in the form "MM-DD-YYYY", because this is the format returned by Num2Date.

```
INPUT "Enter a date in MM-DD-YYYY form ", D$
Dat% = Date2Num%(D$)
IF Num2Date$(Dat%) = D$ THEN
   PRINT D$ " is a good date!"
ELSE
   PRINT "Please try again."
END IF
```

What we are doing here is asking for an original date, and then converting it to an equivalent number. If after converting it back to a string again we have the same date that we started with, then the date was valid.

Understand that while days before 12-31-1979 are returned by Date2Num as negative values, adding and subtracting will still be performed correctly.

Date2Num is demonstrated in the DEMODATE.BAS example program.

Also see the companion function Num2Date.

DayName

assembler function contained in PRO.LIB

Purpose:

DayName accepts an integer value between 1 and 7, and returns an equivalent day name as a string in the form "Sun", "Mon", "Tue", and so forth.

Syntax:

```
D$ = DayName$(Day%)
```

Where:

Day% is an integer within the range of 1 and 7 inclusive, and D$ receives the equivalent day name in the form of a string.

Comments:

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

Because DayName is a function it may be used directly in a PRINT or assignment statement, or in combination with other BASIC or QuickPak Professional functions:

```
PRINT "Today is " DayName$(Day%)
```

or

```
Today$ = DayName$(WeekDay%(DATE$))
```

Also see the related function MonthName.

Demo123

Lotus 123 example program contained in DEMO123.BAS

Purpose:

Demo123 is an example program that shows how to read and write files that can be processed by Lotus 123. This program is derived from an article we wrote that appeared in the December 13, 1988 issue of PC Magazine. Along with the sample routines for accessing Lotus 123 files, a brief discussion of the Lotus file format is given in comments in the DEMO123.BAS program.

DirTree

BASIC subprogram and example contained in DIRTREE.BAS

Purpose:

DirTree reads a disk's entire directory structure, and returns it in two string arrays suitable for displaying.

Syntax:

```
CALL DirTree(Tree$(), FullPath$(), Levels%)
```

Where:

Tree$(1) is initially assigned to the root directory of a drive, and the rest of the array receives the formatted directory structure. FullPath$() is returned holding the actual directory names, and Levels indicates the total number of directory levels that were encountered.

Comments:

The DirTree subprogram is combined with a demonstration program in a single file, so you must copy it to your own program.

It is essential that you seed the first element in Tree$() with the root directory of a drive:

```
Tree$(1) = "C:\"
```

The proper setup and use of DirTree is shown in the DIRTREE.BAS demonstration program.

EDate2Num

assembler function contained in PRO.LIB

Purpose:

EDate2Num accepts a date in the European "DDMMYY" format, and returns a corresponding integer value.

Syntax:

```
Days = EDate2Num%(Dat$)
```

Where:

Dat$ is a date in the form of "DDMMYY" or "DD-MM-YYYY" or "DD.MM.YYYY" or any such combination, and Days receives the number of days before or after 31-Dec-1979 (12-31-1979).

Comments:

Because EDate2Num has been designed as a function, it must be declared before it may be used.

EDate2Num is nearly identical to the Date2Num routine meant for use with American dates, but instead accepts dates in the European format.

See the description for Date2Num for more information about storing dates as integers.

Also see the related routine ENum2Date.

EMS Manager

assembler subroutines contained in PRO.LIB

The QuickPak Professional EMS Memory Manager is a complete set of subroutines that allow you to store and retrieve any type of data using expanded memory. Because memory is allocated in 16K blocks, EMS is very useful for storing arrays, text and graphic screens, and very long strings. It is not appropriate for storing individual variables or short strings.

Each of the EMS routines is explained in detail on the pages that follow, along with a brief example showing the correct usage. All of these routines are written in assembly language, and are contained in PRO.LIB.

Before any EMS routines may be used in a program, the EmsLoaded function must be invoked to determine if EMS memory is installed, and if the appropriate driver software is loaded. This is not a requirement of the EMS hardware, however EmsLoaded performs some initialization that allows the other routines to operate more quickly.

To be compatible with as many PC systems as possible, none of these routines require version 4 or later of the EMS software. The EmsVersion function is provided solely in the interest of completeness. However, future versions of QuickPak Professional may include routines that take advantage of features available only in version 4.

For most applications you will use four of these routines as follows. The EmsLoaded function will first be used to see if EMS memory is installed and available. The Array2Ems and Ems2Array routines may then be used to copy data to and from EMS memory. EmsRelMem will be used to release the EMS memory when it is no longer needed. It is very important that you release the memory that was allocated before your program ends, so it may be used by other programs. Although the discussion that follows describes storing and retrieving arrays, Array2Ems and Ems2Array may in fact be used with *any* contiguous block of memory.

When Array2Ems is called, the correct amount of memory will be allocated for you automatically, based on the number of elements you are storing and the size of each element. Besides allocating memory, Array2Ems also returns a "handle" number that will be used to retrieve the array later. This handle remains active until the EmsRelMem routine is called to "close" the handle and release the memory.

Each time Array2Ems is called, a new handle is obtained. Thus, if you intend to save the same array more than once in EMS memory you should call EmsRelMem before each subsequent save. However, when Ems2Array is used to retrieve an array, the memory is *not* automatically released. Therefore, you may retrieve the array as many times as you'd like, and call EmsRelMem only once when the memory is no longer needed.

A list of the EMS error codes is shown in the table on the following page. Similar to the way errors are reported for the various QuickPak Professional DOS routines, EMS errors are detected by querying the EmsError function. This function returns the status of the most recent EMS service, and is either zero meaning no error occurred, or it contains an error code. The "official" EMS errors have values of 128 or higher, and we have added a few of our own starting at 1.

EMS ERROR CODES

Hex	Dec	Meaning
00H	0	No error
01H	1	EmsLoaded hasn't been used yet to initialize these routines
02H	2	The element length was given as zero
03H	3	The number of elements was given as zero
80H	128	Internal error in EMS device driver
81H	129	Hardware malfunction
83H	131	Invalid EMS handle
84H	132	Function requested is undefined
85H	133	No more handles are available
86H	134	Memory deallocation error
87H	135	More pages were requested than exist in the system
88H	136	More pages were requested than are currently available
89H	137	Zero logical pages requested
8AH	138	Logical page number requested is out-of-range for handle
8BH	139	Physical page number requested is out-of-range

Many of these routines have been designed as functions, and therefore must be declared before they may be used in a program. A complete demonstration is given in the DEMOEMS.BAS example program.

Also see DEMOEMS2.BAS which shows how to manipulate two-dimensional numeric arrays in EMS memory.

EMS FUNCTIONS

EmsError — function

Reports the status of the most recent EMS operation.

```
IF EmsError% THEN PRINT "Error number"; EmsError%; "occurred."
```

EmsGetPFSeg — function

Returns the physical page frame segment that EMS is using in DOS memory.

```
PRINT "The physical page frame segment is"; EmsGetPFSeg%
```

This function is needed only if you intend to access EMS memory pages directly. The EMS driver software makes a page of expanded memory available to an application by mapping it onto a segment in normal DOS memory. This function reports which segment that is.

EmsLoaded — function

Returns -1 if the EMS driver software is loaded, or 0 if it is not.

```
IF EmsLoaded% THEN
   PRINT "EMS memory is loaded on this PC."
ELSE
   PRINT "Sorry, this PC does not have EMS."
END IF
```

EmsNumPages — function

Returns the number of 16K pages currently assigned to the specified handle.

```
PRINT Handle%; "is currently using"; EmsNumPages%(Handle%); _
     "pages."
```

EmsPageCount — function

Returns the total number of 16K EMS memory pages present in a system.

```
PRINT "This PC has a total of"; EmsPageCount%; "16K pages."
```

EmsPagesFree — function

Returns the number of currently available 16K EMS memory pages.

```
PRINT "This PC has"; EmsPagesFree%; "16K pages available."
```

EmsVersion — function

Returns the version number for the EMS driver software *times 100*. For example, EMS driver version 3.40 will be reported as the value 340.

```
PRINT "This PC is using EMS version"; EmsVersion% / 100
```

The major and minor portions of the version may be easily isolated as shown below.

```
Major = EmsVersion% \ 100
Minor = EmsVersion% MOD 100
```

EMS SUBROUTINES

Array2Ems — subroutine

Copies all or part of an array or other block of memory into EMS memory.

```
CALL Array2Ems(SEG Array(Start), ElSize%, NumEls%, Handle%)
```

or

```
CALL Array2Ems(BYVAL Segment%, BYVAL Address%, NumBytes%, 1, _
    Handle%)
```

Where Array(Start) is any numeric or TYPE array, ElSize% is the size of each element in bytes, NumEls% is the total number of elements to copy into EMS memory, and Handle% is the handle returned by Array2Ems. The second example shows how to store any contiguous block of memory.

The ElSize% parameter would be 2 for an integer array, 4 for a long integer or single precision array, and 8 for a double precision array. Array2Ems also accepts the negative code values used by the QuickPak Professional TYPE sort routines. To store fixed-length string and TYPE arrays in expanded memory, ElSize% will be the length of each element. However, to store a fixed-length string array you must first define it as a TYPE. This is described in the section entitled "Calling with Segments".

To store a conventional (not fixed-length) string array in EMS memory you must first store it in an integer array using the QuickPak Professional StringSave routine. Then the integer array may be copied into EMS memory. To retrieve the string array you would use Ems2Array to copy it to an integer array, and then use StringRest to place it back into the string array. StringSave and StringRest are described in the section entitled "String Manager Routines".

Array2Ems may also be used to store a single item, or any contiguous block of memory by specifying the number of bytes in ElSize%, and using 1 for NumEls%. The actual number of bytes copied into expanded memory is calculated within Array2Ems by multiplying ElSize% times NumEls%. If the number of bytes is 16K (16384) bytes or less, simply set ElSize% to the number of bytes, and use 1 for NumEls%. To store, say, 64K you would specify ElSize% as 16384, and set NumEls% to 4. Any similar combination will also work. The example below shows how to save a single text screen from a color display.

```
CALL Array2Ems(BYVAL &HB800, BYVAL 0, 4000, 1, Handle)
```

Then to display the screen again later you would use:

```
CALL Ems2Array(BYVAL &HB800, BYVAL 0, 4000, 1, Handle)
```

If there is not enough EMS memory available when Array2Ems is called, the EmsError function will be set to either 135 or 136.

Ems2Array — subroutine

Retrieves an array or other block of memory from EMS memory.

```
CALL Ems2Array(SEG Array(Start), ElSize%, NumEls%, Handle%)
```

or

```
CALL Ems2Array(BYVAL Segment%, BYVAL Address%, NumBytes%, 1, _
    Handle%)
```

Where Array(Start) is any type of numeric or TYPE array, ElSize% is the size of each element in bytes, NumEls% is the total number of elements to copy from EMS memory, and Handle% is the handle that was assigned by Array2Ems when the array was stored. It is essential that the array being restored has been sufficiently dimensioned to hold the information being copied to it.

This is the exact opposite of Array2Ems, and the parameters have the same meaning as in that routine.

EmsAllocMem — subroutine

Allocates a specified number of 16K pages.

```
CALL EmsAllocMem(NumPages%, Handle%)
```

Where NumPages% is the number of 16K blocks of memory being requested, and Handle% is returned to identify the memory for later use. Because Array2Ems allocates the correct amount of memory automatically, this routine is needed only when using the EmsSetPage service to manipulate EMS memory manually.

EmsGet1El — subroutine

EmsGet1El allows retrieving a single element from expanded memory.

```
CALL EmsGet1El(SEG Value, ElSize%, ElNum%, Handle%)
```

Where Value is any variable, and ElSize% is either its length in bytes, or a special code that indicates the length (see below). ElNum% is the element number (based at one, not zero), and Handle% is the EMS handle that was assigned when the array was first saved.

EmsGet1El allows you to retrieve a single element from an array that has been saved in expanded memory, when you don't want to have to retrieve the entire array. Another important use would be to access a single screen from among several that are being stored in EMS memory. Because the same routine may be used in a program to process different types of variables, you should declare it using the "AS ANY" option:

```
DECLARE SUB EmsGet1El(SEG Value AS ANY, ElSize%, ElNum%, Handle%)
```

The ElSize% variable may optionally be the special size code that is used by the various TYPE array sorts.

Also see the companion routine EmsSet1El which allows you to assign a single element.

EmsRelMem — subroutine

Release all memory associated with a specified handle.

```
CALL EmsRelMem(Handle%)
```

EmsSetError — subroutine

Allows a BASIC program to set or clear the EmsError value.

```
CALL EmsSetError(Value%)
```

EmsSetPage — subroutine

Allows access to individual pages in EMS memory.

```
CALL EmsSetPage(Page%, Handle%)
```

Where Page% is the desired page number, and Handle% is the handle that was assigned when EmsAllocMem or Array2Ems was first called.

Although the EMS driver software uses zero to indicate the first page, this routine accepts page numbers starting at one. Therefore, if you EmsAllocMem was used to request, say, four pages, then pages one through four will be valid. You may also use EmsSetPage to get at memory that was allocated with Array2Ems, however it will be up to you to determine in which page a given array element is stored.

EmsSet1El — subroutine

EmsSet1El allows assigning a single element in an array that is stored in expanded memory.

```
CALL EmsSet1El(SEG Value, ElSize%, ElNum%, Handle%)
```

Where Value is any variable or constant, and ElSize% is either its length in bytes, or a special code that indicates the length (see below). ElNum% is the element number (based at one, not zero), and Handle% is the EMS handle that was assigned when the array was first saved.

EmsSet1El allows you to assign a single element into an array that has been saved in expanded memory, when you don't want to have to retrieve the entire array, make the assignment, and then save it back again. Another important use would be to store multiple screen images in EMS memory.

Because the same routine may be used in a program to process different types of variables, you should declare it using the "AS ANY" option:

```
DECLARE SUB EmsSet1El(SEG Value AS ANY, ElSize%, ElNum%, Handle%)
```

The ElSize% variable may optionally be the special size code that is used by the various TYPE array sorts.

Also see the companion routine EmsGet1El which allows you to retrieve a single element.

Empty

assembler subroutine contained in PRO.LIB

Purpose:

As its name implies, Empty is an empty procedure that does absolutely nothing. However, Empty is invaluable when timing BASIC functions, precisely because it takes such little time to execute.

Syntax:

```
CALL Empty(AnyVariable)
```

Where:

AnyVariable is any type of variable or BASIC function.

Comments:

We discovered the need for Empty while comparing BASIC functions, to see which was the fastest. One common method for benchmarking a block of code is to use BASIC's TIMER, and then executing the code in a FOR NEXT loop many times. Then to determine how long it took, simply divide the total time by the number of iterations. When testing a BASIC subprogram this is fairly easy to do, as shown below:

```
Start! = TIMER
FOR X = 1 TO 1000
   CALL SubProgram(...)
NEXT
Elapsed! = TIMER - Start!
PRINT USING "Each iteration took #.###### seconds."; Elapsed! / 1000
```

But when a function is being tested, you can't just use CALL to invoke it. Rather, you must assign its output which takes time (especially for string functions), or print it, which takes even more time. Empty gives you a way to do something with the output of a function, in as little time as possible:

```
CALL Empty(SomeFunction(..))
```

ENum2Date

assembler function contained in PRO.LIB

Purpose:

ENum2Date converts a previously encoded integer date to an equivalent string in the European format.

Syntax:

```
Dat$ = ENum2Date$(Days%)
```

Where:

Days% is an integer variable or value within the range -29219 to 31368, and Dat$ receives the date in the form of "DD-MM-YYYY".

Comments:

Because ENum2Date has been designed as a function, it must be declared before it may be used.

ENum2Date is nearly identical to the Num2Date routine meant for use with American dates, but instead returns dates in the European format. See the description for Date2Num for more information about storing dates as integers.

Also see the related routine EDate2Num.

Evaluate

BASIC function contained in EVALUATE.BAS

Purpose:

Evaluate is a full-featured expression evaluator. It accepts a formula in an incoming string, and returns a double precision result. Capitalization is ignored (in keywords such as LOG and SIN), *except* for the "E" used for scientific notation. To Evaluate, a lower case "e" represents the constant, and an upper case "E" is for the exponent.

Syntax:

```
Answer = Evaluate#(Expression$)
```

Where:

Expression$ is of the form

```
10 * (12 ^ 3 + (4E-13)) / LOG(8)
```

and Answer receives the computed answer. If the incoming string is invalid (for example a mismatched number of parentheses), the incoming string will be returned with a leading percent sign (%) appended to it.

Comments:

Evaluate is set up as a function, with a demonstration contained in the same file. The demo is set up to display a variety of sample formulas, from which you may select one to try out. As with the other QuickPak Professional BASIC functions, you must copy the function source code into your own programs.

Scientific notation is also supported using "E" (but not "D" or "e").

For example: 10E+3 or 5E-19

Evaluate recognizes two constants, "PI" and "e".

The table below illustrates the operations supported by Evaluate:

Table 7-1

ABS	Absolute Value
AND	Logical AND
ARCCOS	Arc Cosine
ARCCOSH	Arc Hyperbolic Cosine
ARCCOT	Arc Cotangent
ARCCOTH	Arc Hyperbolic Cotangent
ARCCSC	Arc Cosecant
ARCCSCH	Arc Hyperbolic Cosecant
ARCTANH	Arc Hyperbolic Tangent
ARCSEC	Arc Secant
ARCSECH	Arc Hyperbolic Secant
ARCSIN	Arc Sine
ARCSINH	Arc Hyperbolic Sine
ATN	Arc Tangent
CLG	Common Log (base 10, what LOG really is)
COS	Cosine
COT	Cotangent
COTH	Hyperbolic Cotangent
CSC	Cosecant
CSCH	Hyperbolic Cosecant
EXP	Exp
LOG	Natural Log (base e, what BASIC calls LOG)
NOT	Logical NOT
OR	Logical OR
SINH	Hyperbolic Sine
SECH	Hyperbolic Secant
SEC	Secant
SIN	Sine
SQR	Square Root
TAN	Tangent
TANH	Hyperbolic Tangent

The following table shows the math operators supported by Evaluate.

Table 7-2

!	Factorial
^	Exponentiation
*	Multiplication
/	Division
\	Integer Division
+	Addition
-	Subtraction (or unary minus such as -15)
<	Less than
=	Equal to
>	Greater than

Extended

assembler subroutine contained in PRO.LIB

Purpose:

Extended will download a replacement font file to an Epson printer, enabling it to print the entire IBM extended character set.

Syntax:

```
CALL Extended
```

Comments:

Extended always sends its output to the printer connected to LPT1: To send the codes to LPT2: you should run the PSwap program described elsewhere in this manual.

Factorial

assembler function contained in PRO.LIB

Purpose:

Factorial provides an extremely fast way to obtain a factorial value.

Syntax:

```
Answer = Factorial#(Number%)
```

Where:

Number% is a number between 0 and 170, and Answer receives its factorial. If number% is negative or greater than 170, Answer instead receives -1.

Comments:

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

Computing factorials takes an enormous amount of time because so many double precision numbers must be multiplied. The approach we have taken here is to instead calculate all of the answers ahead of time, and place them into a table. This way, Factorial can look up the correct answer very quickly.

This table is stored in the function's code segment, to avoid stealing string space from BASIC. You may edit the table to shorten it if you don't need the full range of factorials this program can accommodate. We recommend that you remark out the table entries rather than remove them permanently. If you do shorten the table, be sure to also change the test that compares the incoming value to 170. Of course, you will have to reassemble the FACT.ASM source file.

FileView

assembler subroutine contained in PRO.LIB

Purpose:

FileView is an assembly language version of the ViewFile file browsing subprogram. FileView supports pollable operation, and also lets you browse multiple files simultaneously.

Syntax:

```
REDIM Buffer%(1 TO 16384)
CALL FileView(FileName$, Ky%, Action%, FVInfo, SEG Buffer%(1))
```

Where:

FileName$ is the file to browse.

Ky% is the last key pressed or a negative version of the key's extended code.

Action% determines the behavior of the routine as follows:

0 = FileView assumes full control and returns on Escape
1 = Initializes the re-entrant mode, sets Action = 3
2 = Updates the screen, then sets Action to 3
3 = Re-enters and processes any keys pending
4 = Sets Action = 3 and exits
5 = Terminates and closes the file

FVInfo is the TYPE variable containing window information and the current operating parameters.

Buffer%() is 32K buffer that ViewFile needs to hold the file contents.

Comments:

Although the ViewFile subprogram has been a part of QuickPak Professional for a long time, it is written in BASIC and is thus much larger than this assembly language version. However, we will continue to include ViewFile because it displays scroll bars, and saves and restores the underlying screen which FileView doesn't. Also because ViewFile is written in BASIC, most programmers can see more clearly how such a routine is written.

FileView is designed to be reentrant, using the same action methods and parameters as PullDown and VertMenu. See the section entitled "Multitasking menus" elsewhere in this manual for a description of the Action parameter and polling techniques in general.

Before calling FileView the calling program must create an integer array of 32K (16384 elements) to serve as a file buffer, and pass that array to FileView.

Note that FileView does not furnish a menu bar. If one is desired, simply clear the screen before calling FileView, print the desired menu bar on line 24 or 25, set FVInfo.Rows to 23 or 24 (respectively), and then call FileView.

FileView recognizes the four cursor keys, PgUp, PgDn, Home, End, and Escape. Pressing Escape returns control to the calling program. Note that FileView does not save and restore the underlying screen that was displayed at the time of the call. If this is needed, it must be done from the calling program.

FileView also lets the calling program specify where in the file to begin displaying (but only when using the polled mode, and only after calling it once to start at the beginning of the file). See FILEVIEW.BAS for an example of implementing this feature.

A TYPE variable called FVInfo is used to pass a number of arguments at one time. FVInfo is constructed as follows:

```
TYPE FVInfo
  Colr        AS INTEGER          'text color, defaults to white on black
  ULRow       AS INTEGER          'these describe the window's corners
  ULCol       AS INTEGER          '(usually 1, 1, 25, 80)
  LRRow       AS INTEGER
  LRCol       AS INTEGER
  HorizOffset AS INTEGER          'see below
  LoPtr       AS INTEGER          'used internally, do not change!
  FileHandle  AS INTEGER          'the DOS file handle, do not change!
  EndOfFile   AS INTEGER          'used internally, do not change!
  LineNumber  AS LONG             'top line displayed, do not change!
  TabStop     AS INTEGER          'spaces per Tab stop
  FileSeek    AS LONG             'see below
  FileOffset  AS LONG             'used internally, do not change!
END TYPE
```

The Colr portion of the TYPE specifies the combined foreground/background color using the same coding method as the QuickPak Professional video routines. See the discussion that accompanies the COLORS.BAS program for more information.

The HorizOffset parameter specifies the left margin within the window (in columns). It is used internally by FileView, but you may also assign it to force the screen to be shifted right by specifying a positive number.

TabStop sets the width (number of columns) when expanding Tab characters. The normal value is 8, and you can also use a value of zero to disable Tab expansion and display the CHR$(8) symbol instead.

FileSeek may be used to begin displaying at any arbitrary location in the file. For example, if you were to add a search capability that examined the file for a string, you would need to tell FileView the offset in the file where that string was found. You may modify FileSeek only if FileView is operating in a polled mode, and only after it has been called once to start at the beginning of the file. Note that you do not have to provide the exact offset in the file where the line begins. If the offset is, say, fifteen characters into the line, FileView will still display the entire line.

FudgeFactor

assembler function contained in PRO.LIB

Purpose:

FudgeFactor returns a long integer value that directly corresponds to the processing speed of a PC.

Syntax:

```
Fudge = FudgeFactor&
```

Where:

Fudge receives a value that indicates the relative speed of a PC.

Comments:

Because FudgeFactor is designed as a function, it must be declared before it may be used.

FudgeFactor is intended to be used with the Pause3 routine. However, it does provide a direct way to estimate the overall speed of the host PC it is running on. The table below shows the results we obtained using FudgeFactor on a variety of popular personal computers.

Brand/Model	Speed	CPU	Result
Epson Equity	4.77 MHz.	8088	2003
Epson Equity	10 MHz.	8088	4228
NEC PowerMate	8 MHz.	80286	7346
Hyundai AT clone	10 MHz.	80286	9345
IBM Model 80 PS/2	16 MHz.	80386	19135
DELL System 310	20 MHz.	80386	33814

FudgeFactor is demonstrated in the PAUSE3.BAS example program.

GetCMOS

BASIC example program contained in GETCMOS.BAS

GETCMOS.BAS is a demonstration program that shows how to access the data in the CMOS RAM of an AT or compatible computer. Several useful pieces of information are in there, including the floppy drive types and total system memory (including extended).

GetCPU

assembler function contained in PRO.LIB

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, 386 or 486 to indicate the presence of an 8086/88 (or NEC V20/30), 80286, 80386, or 80486 CPU.

Comments:

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

GetCPU can be used to determine whether or not certain CPU specific instructions can be executed on the host system such as our XMS memory management routines.

See GETEQUIP.BAS for an example of using GetCPU.

GetDS

assembler function contained in PRO.LIB

Purpose:

GetDS returns BASIC's current internal data segment.

Syntax:

```
Segment = GetDS%
```

Where:

Segment receives the default value of the DS register in a BASIC program.

Comments:

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

Even though DEF SEG will establish BASIC's data segment as the current one for subsequent BSAVES, PEEKS, and so forth, there is no direct way to know what that segment is.

Under QuickBASIC 4 you could ask for the VARSEG of any normal (non-array) variable, and get the same result. But the other Microsoft compilers do not offer that feature.

GetDS is used in the DATA.BAS example program to load data that has been stored in the code segment into a string. It would also be useful with the QuickPak Professional QBLoad and QBSave routines when loading or saving data in BASIC's default segment.

GetEquip

assembler subroutine contained in PRO.LIB

Purpose:

GetEquip returns several items from the equipment list kept in the low-memory area of a PC.

Syntax:

```
CALL GetEquip(Floppies%, Parallels%, Serials%)
```

Where:

Floppies% returns holding the number of floppy disks, Parallels% holds the number of parallel printer adapters, and Serials% tells how many serial ports are installed.

Comments:

GetEquip is useful in a variety of situations. It is particularly handy when used in conjunction with an automatic installation program for applications you create. For example, knowing that a PC has only one floppy disk drive helps you avoid the "Insert disk for drive B" nuisance message that DOS displays. Likewise, if a given PC has only one parallel or serial port, then you could avoid a potential lockup caused by trying to access a device that isn't there.

LockUp

assembler subroutine contained in PRO.LIB

Purpose:

LockUp causes an immediate system freeze that can only be cleared by turning off the PC's power switch.

Syntax:

```
CALL LockUp:
```

Comments:

Whenever a DOS program is exited, any data that had been in the computer's memory is still present. Anyone familiar with DEBUG could then come along and easily browse through that data. Even using Ctrl-Alt-Del will not necessarily clear out the contents of memory.

Because LockUp requires the PC's power to be turned off before it can be used again, you are assured that any abandoned data will not be viewed.

Also see the related routine ReBoot.

MakeQLB

BASIC utility program contained in MAKEQLB.BAS

Unlike most of the QuickPak Professional routines that are added to your own programs, MakeQLB is intended to serve as a stand-alone utility.

MakeQLB will examine a program and all of its dependent modules, and create a new Quick Library containing only those routines that are necessary. This is important when the programs you develop are very large, because it eliminates the wasted memory taken by routines that are not used. MakeQLB also allows you to easily combine routines from multiple library files, without having to extract each individual object module.

MakeQLB knows which routines are to be included by examining your main program for CALL statements, and by searching for DECLARE statements when the CALL keyword is not used. MakeQLB also searches include files to any level and the .MAK file if one is present, to account for all of the modules in a complete program.

MakeQLB will automatically report any subprograms or functions that have been declared but are not being used. Of course, those routines will not be added to the resultant Quick Library. It will also report any subprograms and functions that are present but never called. As an option, you may specify a file that contains a list of all the routines that are to be included in the library, rather than having MakeQLB examine your source files.

MakeQLB uses an interface similar to the LINK and LIB programs, and you may either enter the parameters on a single line, or wait for MakeQLB to prompt you for them. The command line syntax is as follows:

```
MAKEQLB mainprog|routines.lst, qlbname, listfile, lib1 _
    lib2, bqlbname
```

You may also specify more than one file name to be examined, by separating each with a blank space.

We have enhanced MAKEQLB in this version of QuickPak Professional to allow the inclusion of explicit object files, as well as automatically adding those contained in one or more .LIB libraries.

If you are running MakeQLB and supplying all of the parameters on the command line, list the object file names intermixed with source file names separated by a space. If you are letting MakeQLB prompt you for the names, also enter one or more BASIC and object files in answer to the source file prompt. Note that you must include the .OBJ extension, so MakeQLB will know what you mean. The complete syntax is as follows:

```
MAKEQLB basicprogram [basicprogram2] [object.obj], quicklib[.qlb],_
     listfile[.lst], library[.lib] [library2[.lib], [bqlb##][;]
```

This next example tells MakeQLB to examine the BASIC program file MYPROG.BAS, and also explicitly include OBJECT1.OBJ and OBJECT2.OBJ in the Quick Library.

```
   makeqlb myprog object1.obj object2.obj, , , pro, bqlb45
```

If you are using BASIC 7.0 then the last parameter should be QBXQLB.LIB.

Mainprog is the main BASIC program to examine, with a .BAS extension assumed. If a file name with a .LST extension is given, MakeQLB will instead use the procedure names contained in that file when creating the Quick Library.

Qlbname is the name of the resultant Quick Library. If the name is omitted, a library will be created with the same name as the main program, but with a .QLB extension. However, you must add a delimiting comma if the qlbname parameter is not used. If you specify "NUL" for the .QLB name, MakeQLB will merely search for unnecessary DECLARE statements and dead code, but without creating a Quick Library.

The listfile that is created contains a list of all the routines that are being added to the Quick Library. This file defaults to a .LST extension, and is in the correct format that MakeQLB requires to create a library from a list of procedure names. This way, if you need to add a routine or two to the Quick Library later on, you can simply edit the list file. Creating a Quick Library from a list file is of course much faster than examining an entire BASIC program. If the listfile parameter is omitted, the same name as the main program will be used, but with a .LST extension. To tell MakeQLB not to create a list file, use the reserved name NUL for that parameter.

The lib1 and lib2 parameters are library files (.LIB extension) that contain the procedures being added to the Quick Library. One or more library names may be specified, with a blank space used to delimit each name. If no library name is given, the name PRO.LIB is assumed.

The last parameter tells MakeQLB which "bqlb" support library is to be specified when linking. The default name is BQLB45.LIB, which is the library that comes with QuickBASIC version 4.5.

MakeQLB works by creating an object file that contains the list of procedure names. By establishing these procedures as External, they will be included in the Quick Library automatically when MakeQLB invokes LINK. The dirty work of extracting each routine from the various .LIB files is thus handled entirely by LINK.

Marquee

BASIC subprogram contained in MARQUEE.BAS

Purpose:

Marquee provides a cute way to display a scrolling message like a movie marquee.

Syntax:

```
CALL Marquee(X$, Row%, Colr%)
```

Where:

X$ is the message to be displayed, Row% is the screen row to display on, and Colr% is the color to use.

Comments:

Marquee is intended more as an example than a complete subprogram. Therefore, if you want to add it to your own programs, you should load MARQUEE.BAS as a module and copy the subprogram. The technique for doing this is described in the section on the QuickPak Professional BASIC functions.

Marquee assumes an 80 column screen when it calculates where to center the message, though this is easy to modify if you are using 40 columns. Simply change the line

```
LOCATE 40 - L% \ 2
```

to

```
LOCATE 20 - L% \ 2
```

The color parameter must be coded in the format used by the various video routines. See the discussion that accompanies COLORS.BAS for information about combining foreground and background colors into a single byte.

MathChip

assembler function contained in PRO.LIB

Purpose:

MathChip will report if an 80x87 math coprocessor chip is installed in the host PC.

Syntax:

```
There = MathChip%
```

Where:

There receives either -1 if a coprocessor is installed, or 0 if one is not.

Comments:

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

MathChip is furnished as a companion to the GetCPU function, to provide information about the host computer.

MaxInt and MaxLong

assembler functions contained in PRO.LIB

Purpose:

MaxInt compares two integer variables, and returns the value of the higher one. MaxLong is similar, but is intended for use with long integers.

Syntax:

```
Higher = MaxInt%(Value1%, Value2%)
```

or

```
Higher = MaxLong&(Value1&, Value2&)
```

Where:

Value1 and Value2 are the two values being considered, and Higher receives the higher of the two values.

Comments:

Because MaxInt and MaxLong have been designed as functions, they must be declared before they may be used.

These routines provide a convenient way to avoid code such as:

```
IF X% > Y% THEN
   A% = X%
ELSE
   A% = Y%
END IF
```

A single assignment may be used instead:

```
A% = MaxInt%(X%, Y%)
```

MaxInt is used extensively in the QEdit text editor supplied with QuickPak Professional to quickly determine the various boundaries of borders and rows being displayed.

Also see the companion functions MinInt and MinLong.

MinInt and MinLong

assembler functions contained in PRO.LIB

Purpose:

MinInt compares two integer variables, and returns the value of the smaller one. MinLong is similar, but is intended for use with long integers.

Syntax:

```
Smaller = MinInt%(Value1%, Value2%)
```

or

```
Smaller = MinLong&(Value1&, Value2&)
```

Where:

Value1 and Value2 are the two values being considered, and Smaller receives the lower of the two values.

Comments:

Because MinInt and MinLong have been designed as functions, they must be declared before they may be used.

These functions provide a convenient way to avoid code such as:

```
IF X% < Y% THEN
   A% = X%
ELSE
   A% = Y%
END IF
```

A single assignment may be used instead:

```
A% = MinInt%(X%, Y%)
```

MinInt is used extensively in the QEdit text editor supplied with QuickPak Professional to quickly determine the various boundaries of borders and rows being displayed.

Also see the companion functions MaxInt and MaxLong.

MonthName

assembler function contained in PRO.LIB

Purpose:

MonthName accepts an integer value between 1 and 12, and returns an equivalent month name as a string in the form “Jan”, “Feb”, “Mar”, and so forth.

Syntax:

```
M$ = MonthName$(Month%)
```

Where:

Month% is an integer within the range of 1 and 12 inclusive, and M$ receives the equivalent month name in the form of a string.

Comments:

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

Because MonthName is a function it may be used directly in a PRINT or assignment statement, or in combination with other BASIC or QuickPak Professional functions:

```
PRINT "The current month is " MonthName$(Month%)
```

or

```
Month$ = MonthName$(VAL(LEFT$(DATE$, 2)))
```

Also see the related function DayName.

MsgBox

BASIC subroutine contained in MSGBOX.BAS

Purpose:

MsgBox provides a quick and attractive way to display a message with word wrap, automatically centered on the screen. The underlying screen is always saved, and it may be restored again later.

Syntax:

```
CALL MsgBox(Message$, Wide%, Cnf)
```

Where:

Message$ is a single continuous string to be displayed, Wide% is the desired width of the text (up to 74), and Cnf is a special TYPE variable that must be defined. If Message$ is null, the most recently displayed message is cleared, and the underlying screen restored.

Comments:

The top line of the MsgBox display is placed at the current cursor line, so you should use LOCATE to set that before you call MsgBox.

When MsgBox is called, the first thing it does is check the length of the message string. If it is not null, it first saves the underlying screen, and then displays the message. To clear the message and restore the original screen, simply call MsgBox again with a null string.

Be sure that you don't call MsgBox with a null string, unless it has already been called at least once before. Also be aware that the message should always be cleared before a new one is displayed. Otherwise, there will be no way to restore the original screen.

The width is limited to 74 because MsgBox draws a border around the text, and adds an extra blank space to make the text easier to read. Two additional columns are needed to accommodate the shadow.

The Cnf variable is described in the section entitled "DEFCNF and SETCNF", but briefly, it contains a table of information about the host PC. It is used by MsgBox to determine appropriate colors to use, based on the type of monitor that is present.

Cnf is defined in the DEFCNF.BI Include file. The benefit of isolating the color definitions to a single file is that you may customize them to your own preferences, and they will then be reflected in all of the programs that use DEFCNF.

MsgBox is shown in context in the DEMOPOP.BAS example program.

Num2Date

assembler function contained in PRO.LIB

Purpose:

Num2Date converts a previously encoded integer date to an equivalent date string.

Syntax:

```
D$ = Num2Date$(Days%)
```

Where:

Days% is an integer variable or value within the range -29219 to 31368, and D$ receives the date in the form "MM-DD-YYYY".

Comments:

Because Num2Date has been designed as a function, it must be declared before it may be used.

If Days% is out of range, D$ returns "%%%%%%%%%%".

A complete discussion of the QuickPak Professional date conversion method is given in the section that describes the Date2Num routine.

Num2Day

assembler function contained in PRO.LIB

Purpose:

Num2Day accepts an integer number that represents a date in the QuickPak Professional format, and returns the appropriate day of the week (1 - 7).

Syntax:

```
Day = Num2Day%(D%)
```

Where:

D% is a date that has already been converted to the equivalent integer format, and Day receives a value corresponding to the day of the week.

Comments:

Because Num2Day has been designed as a function, it must be declared before it may be used.

Num2Day is useful in conjunction with the DayName$ function, as shown in the DEMODATE.BAS example program.

Also see the related BASIC function Date2Day, that instead accepts the date as a string.

Num2Time

assembler function contained in PRO.LIB

Purpose:

Num2Time converts a long integer that represents the number of seconds past midnight to an equivalent time in string form.

Syntax:

```
T$ = Num2Time$(Time&)
```

Where:

Time& is a long integer variable or value within the range 0 to 86400, and T$ receives the time in the form "HH:MM:SS".

Comments:

Because Num2Time has been designed as a function, it must be declared before it may be used.

A complete discussion of the QuickPak Professional time conversion method is given in the section that describes the Time2Num routine.

Pause

assembler subroutine contained in PRO.LIB

Purpose:

Pause will pause a program's execution for a specified period of time to a resolution as small as 1/18th of a second.

Syntax:

```
CALL Pause(Ticks%)
```

Where:

Ticks% is the number of 1/18ths of a second to pause.

Comments:

Before Pause, the only reasonable way to add a short delay to a program was with BASIC's TIMER function:

```
X! = TIMER
WHILE X! + .1 > TIMER
WEND
```

Pause minimizes both the amount of code you have to write, as well as the amount that BASIC would generate to accomplish the same function.

Pause works by examining the system clock data area in low memory, and waits until the specified number of clock ticks have occurred.

Pause2

assembler subroutine contained in PRO.LIB

Purpose:

Pause2 will pause a program's execution for a specified number of microseconds.

Syntax:

```
CALL Pause2(Microseconds%)
```

Where:

Microseconds% is the number of microseconds to pause.

Comments:

Pause2 is provided as a complement to Pause, to allow even greater control over delay times. One important use of very small delays is when communicating with external hardware devices. For example, schemes that employ a hardware block to effect copy protection often require this resolution when the protection device is being polled.

Pause2 is limited to delays of 27500 microseconds or less (0.0275 seconds).

To implement very small delays, Pause2 must reprogram the PC's timer chip. This in itself is not difficult to do, however the system timer must also be adjusted.

Approximately 18 times per second, the system timer generates an interrupt that updates the PC's internal clock. However, if this interrupt were to come along just at the moment Pause2 began its timing loop, several milliseconds would be lost while the timer interrupt is being processed.

Pause2 takes this into account by synchronizing the system timer to begin a new period at exactly the same time.

Pause3

assembler subroutine contained in PRO.LIB

Purpose:

Pause3 provides a simple method for obtaining delays with a resolution of 1 millisecond, without having to reprogram the PC's timer chips.

Syntax:

```
CALL Pause3(MilliSeconds%, Fudge&)
```

Where:

MilliSeconds% is the desired number of milliseconds to delay, and Fudge& was previously obtained using the QuickPak Professional FudgeFactor function.

Comments:

Even though QuickPak Professional provides the Pause2 routine for creating microsecond delays, that routine must reprogram the PC's timer chips as part of its operation. Unfortunately, this may cause undesirable side effects if Pause2 is called many times in a program. When such fine resolution is not required, Pause3 is probably a better choice.

Pause3 expects two parameters — the number of milliseconds to delay for, and a "fudge factor" that was determined earlier by the QuickPak Professional FudgeFactor function. FudgeFactor reports the relative speed of the host PC, which is then used by Pause3 to know how many loop instructions to perform internally for a 1 millisecond delay.

Because FudgeFactor creates a delay of up to 1/10 second or so, you should use it once at the beginning of your program, prior to calling Pause3. This is shown in context in the PAUSE3.BAS example program.

PDQTimer

assembler function contained in PRO.LIB

Purpose:

PDQTimer is an integer-only TIMER replacement that avoids the inclusion of BASIC's floating point code in your programs.

Syntax:

```
ThisTime = PDQTimer&
```

Where:

ThisTime receives the current timer count stored by the BIOS in low memory.

Comments:

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

PDQTimer returns a long integer value that corresponds to the BIOS timer count stored in low memory. Because it peeks that information directly, it does not add the enormous amount of floating point code that QuickBASIC's TIMER does.

The declare syntax and a typical usage example is as follows:

```
DECLARE FUNCTION PDQTimer&()

Start& = PDQTimer&                'start timing
FOR X& = 1 TO 100000              'time this loop
Z& = Z& + 1
NEXT
Done& = PDQTimer&                 'done timing

PRINT Done& - Start&; "18ths of a second have elapsed."
```

Miscellaneous

Peek1

assembler function contained in PRO.LIB

Purpose:

Peek1 will read a byte at a specified segment and address, and return its value.

Syntax:

```
Byte = Peek1%(Segment%, Address%)
```

Where:

Segment% and Address% indicate where the byte to be read is located, and Byte receives its value.

Comments:

Because Peek1 has been designed as a function, it must be declared before it may be used.

One of the problems with the usual method of peeking memory is that the segment to peek must be set with a DEF SEG statement. While there's nothing inherently wrong with using DEF SEG, it destroys any previous settings.

Of course, in most cases it really doesn't matter. However, when you are creating reusable modules that will be called at different times by different programs, using Peek1 can avoid potential problems.

Peek2

assembler function contained in PRO.LIB

Purpose:

Peek2 will read a word (two bytes) at a specified segment and address, and return its value.

Syntax:

```
Word = Peek2%(Segment%, Address%)
```

Where:

Segment% and Address% indicate where the word to be read is located, and Word receives its value.

Comments:

Because Peek2 has been designed as a function, it must be declared before it may be used.

One of the problems with the usual method of peeking memory is that the segment to peek must be set with a DEF SEG statement. While there's nothing inherently wrong with using DEF SEG, it destroys any previous settings.

In most cases it really doesn't matter. But, if you are creating reusable modules that will be called at different times by different programs, using Peek2 can avoid potential problems.

Further, when two bytes must be obtained, Peek2 avoids the extra calculations that would otherwise be needed. The first example below uses the old method to look at the video buffer size in low memory, to see how many bytes the current screen occupies.

```
DEF SEG = 0
X = PEEK(&H44C) + 256 * PEEK(&H44D)
```

Compare that with the Peek2 approach:

```
X = Peek2%(0, &H44C)
```

Pokel

assembler subroutine contained in PRO.LIB

Purpose:

Pokel will write a new byte to a specified segment and address.

Syntax:

```
CALL Pokel(Segment%, Address%, Byte%)
```

Where:

Segment% and Address% indicate where the byte to be written is located, and the value held in Byte% is then stored in that location.

Comments:

One of the problems with the usual method of poking memory is that the segment to write to must be set with a DEF SEG statement. While there's nothing inherently wrong with using DEF SEG, it destroys any previous settings.

Of course, in most cases it really doesn't matter. However, when you are creating reusable modules that will be called at different times by different programs, using Pokel can avoid potential problems.

Poke2

assembler subroutine contained in PRO.LIB

Purpose:

Poke2 will write a new word (two bytes) to a specified segment and address.

Syntax:

```
CALL Poke2(Segment%, Address%, Word%)
```

Where:

Segment% and Address% indicate where the word to be written is located, and the value held in Word% is then stored in that location.

Comments:

One of the problems with the usual method of poking memory is that the segment to write to must be set with a DEF SEG statement. While there's nothing inherently wrong with using DEF SEG, it destroys any previous settings.

Of course, in most cases it really doesn't matter. However, when you are creating reusable modules that will be called at different times by different programs, using Poke2 can avoid potential problems.

Further, when two bytes must be written, Poke2 avoids the extra calculations that would otherwise be needed:

```
Value% = 12345
DEF SEG = Segment%
POKE Address%, Value% MOD 256
POKE Address% + 1, Value% \ 256
```

Poke2 replaces the above mess with:

```
CALL Poke2(Segment%, Address%, 12345)
```

Power and Power2

assembler functions contained in PRO.LIB

Purpose:

Power and Power2 will raise any number to a power, or 2 to a power respectively, simulating BASIC's x ^ n and 2 ^ n.

Syntax:

```
X = Power(Y, N)                    'returns X = Y ^ N
X = Power2(N)                      'returns X = 2 ^ N
```

Where:

X is assigned either 2 ^ N power, or Y ^ N power, as shown above.

Comments:

Because Power and Power2 have been designed as functions, they must be declared before they may be used.

When used with integer values these routines are much faster than using BASIC's exponentiation operator which requires floating point math.

These functions may be declared as either integer or long integer functions, depending on the expected range of return values.

PRNReady

assembler function contained in PRO.LIB

Purpose:

PRNReady will report whether a specified printer is available and on-line.

Syntax:

```
Ready = PRNReady%(LPTNumber%)
```

Where:

LPTNumber% is either 1, 2, or 3 to indicate the parallel printer to check, and Ready receives -1 if it is ready, or zero if it is not.

Comments:

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

PRNReady works by attempting to send two characters—a space and a backspace— and reports if the printing was successful. PRNReady begins by sending a CHR$(32) space to the specified printer through the BIOS. If the BIOS returns an error, PRNReady simply gives up and reports the error to the calling BASIC program. If it successful, a backspace is then printed to "undo" the forward space.

Comments in the assembler source code show how to modify PRNReady to attempt two trial printings rather than only one. You might want this because some printers are very slow when performing a form-feed, and might appear to be not ready even when they are. Most of the current PC's and PC clones use a BIOS that tries for a long enough time. However, some computers do not wait long enough for a slow printer to complete a form-feed before reporting a time-out.

BLPRINT.BAS shows PRNReady in use, as well as how to reduce the timeout delay when a printer is off-line.

PSwap

assembler subroutine contained in PRO.LIB

Purpose:

PSwap exchanges LPT1 and LPT2 each time it is called.

Syntax:

```
CALL PSwap
```

Comments:

PSwap lets you quickly exchange the port addresses for the first two parallel printers to allow LPrint statements to work with either one.

You should be aware that most software spoolers intercept the printer port addresses when they are installed, so PSwap will not work unless it is run *before* the spooler program is installed.

QPCli and QPSti

assembler subroutines contained in PRO.LIB

Purpose:

QPCli and QPSti are a pair of routines that disable and enable interrupts respectively.

Syntax:

```
CALL QPCli     'disable system interrupts
CALL QPSti     'reenable interrupts
```

Comments:

There is usually little need for routines like these to be called from BASIC. However, they were needed to properly access the CMOS RAM in the GETCMOS.BAS example program. Because it is possible for a system interrupt to come along and access the port at the same time as the BASIC program, some way is needed to prevent the conflict.

VERY IMPORTANT!

If you do not know how these routines are supposed to be used, please do not experiment with them!

QPPlay

assembler subroutine contained in PRO.LIB

Purpose:

QPPlay is a replacement for BASIC's PLAY statement that greatly reduces the amount of code needed to add music to your programs.

Syntax:

```
CALL QPPlay(Tune$)
```

Where:

Tune$ is a string that contains musical notes in the same format as required by BASIC's PLAY statement.

Comments:

Unlike BASIC's PLAY statement which can increase your program's size by as much as 14K, QPPlay adds only 1K or so.

The only limitations compared to the BASIC version of PLAY is that QPPlay will not operate in the background (MB and MF are ignored), and a dot to extend a note's duration is not supported. Since many programs do not need the ability to play music as a background task, this routine can afford a substantial savings in code size.

QPSolver

BASIC demonstration program contained in QPSOLVER.BAS

QPSolver is a complete environment for entering and editing variables and expressions to be evaluated using the QuickPak Professional Evaluate function.

QPSolver is similar to a BASIC interpreter, in that you can enter assignments as immediate commands, and then display the results using PRINT statements. Variables may be assigned from numbers, or from other variables. All of the functions that Evaluate supports may be used; for example, you may enter an expression such as X = ATN(Y).

Besides merely assigning and displaying variable values, you may also set up Watch expressions in much the same way that QuickBASIC allows. For example, the command *WATCH X * (Y ^ Z)* sets that as a Watch expression, and the display will be updated with each change to X, Y, or Z.

Finally, you can save and load sessions for later editing. The command *SAVE FILENAME* will create a file named FILENAME, and store the entire context of the current session. Likewise, entering *LOAD FILENAME* will load that file and restore the original variable values and Watch expressions.

Understand that QPSolver is not a true solver in the sense that MathCad and TKSolver are. However, this is the core program for a real solver we hope to eventually develop for inclusion in QuickPak Scientific.

QPSound

assembler subroutine contained in PRO.LIB

Purpose:

QPSound is nearly identical to BASIC's SOUND statement, but provides a considerable reduction in the amount of code added to a program.

Syntax:

```
CALL QPSound(Frequency%, Duration%)
```

Where:

Frequency% is the desired frequency in Hz. (Hertz, or cycles per second), and Duration% is the length of the sound in 1/18ths of a second.

The frequency must be within the range 37 to 32767, and the duration may range from 1 to 65535. We can't imagine why you'd want a duration longer than, say, a few seconds, but if you do, a long integer is needed to exceed 32767. Also, frequencies higher than 18000 Hz. or so will be inaudible to most people.

Comments:

The SOUND statement is certainly a useful BASIC feature, however using it adds considerably to size of a program. In a test program compiled by QuickBASIC 4.5 using a single SOUND statement increased the code size by 11K. QPSound is nearly identical to BASIC's version, but adds less than fifty bytes to your program.

The only real difference between QPSound and BASIC's SOUND is that the sound is played in the "foreground" only. Where BASIC's SOUND returns to your program immediately and continues to play the tone in the background, QPSound waits until it has completed before returning. QPSound is demonstrated in the QPSOUND.BAS example program.

QPSSeg and QPSegAdr

assembler functions contained in PRO.LIB

Purpose:

QPSSeg and QPSegAdr are replacements for the BASIC PDS SSEG and SSEGADD functions, but they also work with regular QuickBASIC.

Syntax:

```
LongInt& = QPSegAdr&(Any$)
Segment% = QPSSeg%(Any$)
```

Where:

LongInt& receives the segmented address of Any$, and Segment% receives the segment that hold the data in Any$.

Comments:

Because QPSSeg and QPSegAdr have been designed as a function, they must be declared before they may be used.

If you are like us and have to write code that will compile under both QuickBASIC and BASIC 7 PDS, you will appreciate QPSSeg and QPSegAdr. They return the same information as BASIC 7's SSEGADD and SSEG. However, the same routines are provided in both the QB and BC7 versions of the QuickPak Professional library.

QPUSI

assembler function contained in PRO.LIB

Purpose:

QPUSI (QuickPak Unsigned Integer) returns the low-word portion of a long integer.

Syntax:

```
LowWord = QPUSI%(LongInt&)
```

Where:

LowWord receives the two lowest bytes in LongInt&.

Comments:

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

QPUSI is useful for those situations where you are using a long integer to store integer information whose value may exceed 32767. If you then need to pass that as a regular integer by value using BYVAL, the only other way to obtain just the lower word is with PEEK and POKE. QPUSI solves this problem by directly returning just the low-word portion of the long integer.

ReBoot

assembler subroutine contained in PRO.LIB

Purpose:

ReBoot causes the host PC to perform a “warm” boot, as if the Ctrl-Alt-Del keys had been pressed.

Syntax:

```
CALL ReBoot
```

Comments:

For the most part, we prefer programs that are as friendly as possible, and ReBoot is decidedly unfriendly. However, if you are determined to lock out a user who, for example, doesn’t know the correct password, ReBoot will do it.

Also see the related routine LockUp.

ShiftIL and ShiftIR

assembler subroutines contained in PRO.LIB

Purpose:

ShiftIL shifts the bits in an integer variable a specified number of positions to the left, and ShiftIR shifts bits to the right.

Syntax:

```
CALL ShiftIL(IntVar%, NumBits%)
```

or

```
CALL ShiftIR(IntVar%, NumBits%)
```

Where:

IntVar% is the variable whose bits are to be shifted, and NumBits% tells how many positions to shift them.

Comments:

Shifting bits in a variable is not something that most programmers would need to do every day, but it does have a few uses. For example, like the AddUSI function, shifting bits to the left can be used to multiply a variable by 2, 4, 8, and so forth, without generating an overflow error that might otherwise occur. Each time the bits in an integer are shifted to the left one position, the variable is multiplied by two. Likewise, shifting right divides by 2 for each bit position shifted.

Another possible use is when creating a Line Style for use with the graphics LINE command. LINE accepts an optional style argument in an integer variable, to create a dotted line. The bits in the integer are used to indicate which pixels are to be on or off in the line. Thus, ShiftIL could be used to animate a line by redrawing it with a new pattern.

Also see the related routines ShiftLL, and ShiftLR.

ShiftLL and ShiftLR

assembler subroutines contained in PRO.LIB

Purpose:

ShiftLL shifts the bits in a long integer variable a specified number of positions to the left, and ShiftLR shifts bits to the right.

Syntax:

```
CALL ShiftLL(LongInt&, NumBits%)
```

or

```
CALL ShiftLR(LongInt&, NumBits%)
```

Where:

LongInt& is the variable whose bits are to be shifted, and NumBits% tells how many positions to shift them.

Comments:

Shifting bits in a variable is not something that most programmers would need to do every day, but it does have a few uses. See the discussion of the ShiftIL and ShiftIR routines for some possible applications.

Soundex

assembler function contained in PRO.LIB

Purpose:

Soundex returns a value that corresponds to the sound of a word.

Syntax:

```
Code$ = Soundex$(Word$)
```

Where:

Word$ is any single word or proper name, and Code$ receives a four-digit Soundex code such as "0122".

Comments:

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

This routine employs the standard Soundex algorithm to return a four-character string that loosely approximates how the string sounds.

Soundex is often used for searching a database of names, when the exact spelling is not known. To see if one string sounds like another, you would simply compare the results that Soundex returns:

```
IF Soundex$(First$) = Soundex$(Second$) THEN
   PRINT "They sound alike"
END IF
```

When a string is being converted to its equivalent Soundex code, vowels are removed, and consonants that are similar are given the same value. Note that when the same sound occurs twice or more in succession, only one of them is counted. Also note that with long strings the conversion ceases when all four digits are filled. Likewise, short strings are padded with trailing "0" characters.

The conversion codes are shown in the table below.

Letters	B	F	P	V	C	G	J	K	Q	S	X	Z	D	T	L	M	N	R
CodeNum	1	1	1	1	2	2	2	2	2	2	2	2	3	3	4	5	5	6

Thus, the string "QuickPak" would be coded as "2212" ("QCPK"). The first consonant, "Q", is translated to "2", and the vowels "ui" are ignored. The "c" is then converted to "2", but the following "k" is ignored because it has the same code as the "c". The "P" is translated to "1", the vowel "a" is ignored, and the final "K" is converted to "2".

It is important to understand that because of the many quirks in the English language, Soundex is not always reliable.

Soundex is demonstrated in the SOUNDEX.BAS file.

String Manager

The QuickPak Professional String Manager is a comprehensive set of routines that allow you to store entire string arrays in "far" memory, and retrieve them again later. These routines are intended to be used with conventional (not fixed-length) string arrays only, which are copied into dynamic integer arrays. Once this is done the string array may then be erased, or loaded with new information. The original array contents may be retrieved at any time.

Because the dynamic arrays used for storage do not impinge on BASIC's string space, you can maintain many separate string arrays in a single program, and each may be as large as BASIC permits. Another advantage to using dynamic arrays is that they can be dimensioned and erased as needed. They may also be saved to disk—or expanded memory if it is present—to accommodate even larger amounts of string data. For more information about using expanded memory, please see the description of the EMS Manager elsewhere in this manual.

Each of the string manager routines is described in detail on the pages that follow, along with a brief example showing the correct usage. All but two of these routines are written in assembly language. The assembler routines are contained in PRO.LIB, and the BASIC routines are in the files GET1STR.BAS and STRREST.BAS. These two routines must be written in BASIC because they assign strings, which cannot be done in assembly language. All of the string manager services are demonstrated in the DEMOSTR.BAS example program.

Unlike numeric, TYPE, and fixed-length arrays, string arrays are not kept in contiguous memory locations. Rather, a table of descriptors that hold the length and address of each string is contiguous, and the strings themselves are scattered around in "near" memory. Therefore, StringSave merely gathers up each string element in succession, and copies it to an integer array.

The strings are stored in the integer arrays exactly as they would be kept in a disk file. That is, a terminating carriage return and line feed is placed after the end of each string element. This lets you load and save an entire file very quickly by using the QuickPak Professional FGetA and FPutA routines.

FindLastSM — assembler function

FindLastSM (SM stands for String Manager) determines the actual number of bytes occupied by strings in an integer storage array.

```
'  first see how many bytes are in the entire array
NumBytes = (UBOUND(Array%) - LBOUND(Array%) + 1) * 2

'  now see how many bytes are actually being used
NumBytes = FindLastSM&(SEG Array%(1), NumBytes)
```

Where Array%(1) is the first element in the integer array, and NumBytes receives the number of active bytes in the array. NumBytes may be either an integer or long integer variable.

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

Not unlike the FindLast routine for string arrays, FindLastSM scans the stated array backwards, looking for the last non-blank element. FindLastSM is used internally by the Get1String and StringRestore BASIC functions, and it is unlikely that you would need to use it directly in your programs.

Get1Str — assembler subroutine

Copies a single string from an integer array. Get1Str is a low-level routine that is used by the BASIC Get1String function. It is very unlikely that you would need to call this routine directly, and it is documented here solely in the interest of being complete.

```
CALL Get1Str(Work$, SEG Array%(1), StringNumber%)
```

Where Work$ has already been filled with the correct number of spaces, Array%(1) is the first element in the storage array, and StringNumber% is the element number of the string to retrieve. If the array begins with element zero, you will of course use Array%(0) instead of Array%(1).

Get1String — BASIC function

Returns a single string from an integer array.

```
StringNumber% = 1
PRINT "String number one is: "; Get1String$(Array%(), _
    StringNumber%)
```

Where Array%() is the integer storage array, and StringNumber% is the string to be retrieved.

GetNext — assembler function

Tells StringRestore the length of the next string to retrieve. GetNext is a low-level routine that scans ahead in the integer storage array looking for the CHR$(13) that marks the end of the current string. Once it is found, the length of the string is calculated and returned to StringRestore. ThisAddress% is also modified to point to the next string element in storage for the next time it is called. It is extremely unlikely that you would need to call this routine directly.

```
Array$(X) = SPACE$(GetNext%(Segment%, ThisAddress%, _
     LastAddress%))
```

Where Segment% and ThisAddress% indicate where in the integer array GetNext is to begin scanning, and LastAddress% points to the last active byte in the integer array.

MidStrSave/MidStrRest — assembler subroutines

MidStrSave and MidStrRest are specialized versions of the string manager routines StringSave and StringRest, which are designed to save a MID$ portion of a string array.

To save:

```
DIM Storage%((NumEls% * NumChars%) \ 2)
CALL MidStrSave(BYVAL VARPTR(Array$(First)), NumEls%, _
    FirstChar%, NumChars%, SEG Storage%(0))
```

To restore:

```
Buffer$ = SPACE$(NumChars%)    'LEN(Buffer$) indicates length
CALL MidStrRest(Buffer$, StrNumber%, SEG Storage%(0))
ERASE Storage%                 'optionally free up the memory
```

Where Storage%() is an integer array used to store the portion of the string array, and Array$() is the string array from which the text is being captured.

NumEls% is the number of string elements to save, FirstChar% indicates with which character to begin saving each element, and NumChars% is the number of subsequent characters to save. If a string contains too few characters when being saved, it is padded with blank spaces in the storage array.

When restoring, strings are accessed one by one, and StrNumber% tells which one to retrieve.

These routines were designed specifically for use with the QEdit subprogram, to allow capturing and restoring text in columns. It is unlikely that you would need these in your own programs, and they are documented here solely for completeness.

Note that the string portions being saved and restored must be less than 256 characters in length.

NumStrings — assembler function

Returns the number of strings stored in an integer array.

```
DIM Array$(NumStrings%(SEG Array%(1), NumBytes))
```

Once an array has been loaded from disk or copied from expanded memory, you will need to know how many strings it contains. This is what NumStrings% is for. If the array is going to be sent to QEdit for editing, you will probably want to include additional elements when dimensioning the string array.

Of course, you would use Array%(0) instead of Array%(1) if that is the first element in the integer array. Also, the NumBytes parameter indicates the total size of the storage array, and may be either an integer or a long integer.

StringRest — assembler subroutine

Copies strings from an integer array into a string array. This is a low-level routine used by the BASIC StringRestore subprogram, and it is very unlikely that you would need to call it directly.

```
CALL StringRest(BYVAL VARPTR(Array$(1)), SEG Array%(1), _
    NumStrings%)
```

Where Array$(1) is the first element in the string array, Array%(1) is the first element in the integer array, and NumStrings% is the number of string elements being restored. Of course, you would specify element zero instead of element one for either array if that is appropriate. StringRest requires that the correct number of characters has already been set aside in each string element.

StringRestore — BASIC subprogram

Creates space in a string array and fills it with strings stored in an integer array.

```
CALL StringRestore(Array$(), Array%())
```

Before calling StringRestore you must first dimension Array$() to the correct number of elements. However, if the number of strings stored in the integer array is greater than Array$() has been dimensioned to, only UBOUND(Array$) elements will be restored. Likewise, if the number of elements in Array$() is greater than the number of strings stored in Array%(), only the number of strings actually present in Array%() will be copied.

StringSave — assembler subroutine

Copies all or part of a string array to an integer array.

```
CALL StringSave(BYVAL VARPTR(Array$(1)), SEG Array%(1), _
    NumStrings%)
```

Where Array$(1) is the first element in the string array, Array%(1) is the first element in the integer array, and NumStrings% is the number of string elements in Array$() to be saved. Of course, you would specify element zero instead of element one for either array if that is appropriate. It is essential that the integer array be sufficiently dimensioned to hold the string data. See the description for the StringSize function to see how to determine the number of elements needed.

StringSize — assembler function

Returns the number of bytes needed to store a string array in an integer array.

```
NumBytes = StringSize&(BYVAL VARPTR(Array$(Start)), _
    NumStringEls%)
```

Where Array$(Start) is the first element to save in the string array, and NumStringEls% is the number of elements in the string array to include.

This function returns the number of bytes needed rather than the number of elements, so you will know exactly how many bytes to specify when saving the integer array to disk. To determine how large to dimension the integer array to, simply divide the number of bytes by two. If you intend to dimension the integer array starting at element 1 (instead of 0), you should also add one extra element in case the number of bytes is an odd number.

```
DIM Array%(NumBytes \ 2)
```

or

```
DIM Array%(1 TO NumBytes \ 2 + 1)
```

StrLength — assembler function

Returns the length of a single string stored in an integer array. StrLength is a low-level routine used by the BASIC Get1String function. It is unlikely that you will need to access StrLength directly.

```
Length = StrLength%(SEG Array%(1), NumBytes, StringNumber%)
```

Where Array%(1) is the first element in the integer storage array, NumBytes is the total number of bytes in the array (as reported by StringSize), and StringNumber% tells which string's length to return. Notice that NumBytes may be either an integer or long integer. A long integer is required if the array exceeds 32767 bytes in size.

Sub1String — assembler subroutine

Sub1String will substitute a string contained in a "far" integer array with a new one of the same or different length.

```
CALL Sub1String(New$, SEG Array%(1), NumBytes%, StrNumber%)
```

Where New$ is the new string to assign into the array, Array%(1) is the first element in the integer storage array, NumBytes is the number of active bytes (see below), and StrNumber% indicates which string to replace. After Sub1String has performed the substitution, NumBytes is modified to reflect the new number of bytes taken. NumBytes may be either an integer or long integer.

Sub1String is meant to be used in conjunction with the string manager routines, and it is demonstrated in the DEMONSTR.BAS example program.

Because it is possible the new string may be longer than the original string, we recommend that you include enough extra elements when dimensioning the integer array. The demo program includes an additional 100 elements (200 bytes), but you may need even more, based on the number of substitutions you may make, and the added length of the new strings.

SysTime

assembler subroutine contained in PRO.LIB

Purpose:

SysTime obtains the current system time through DOS, and returns it in a string formatted to the hundredth of a second.

Syntax:

```
T$ = SPACE$(11)
CALL SysTime(T$)
```

Where:

T$ must first be assigned to a length of at least eleven characters, and SysTime then fills it with the system time in the form "HH:MM:SS:HH".

Comments:

Even though DOS (and SysTime) report the system time to a resolution of 1/100th second, it is really only accurate to about 1/18th second or so. The PC's hardware updates the current time only that often, so the last digit has little meaning.

SysTime could be used to advantage if you want to simulate the type of display used in televised sporting events, or to create extra tension in an action game. The program fragment below shows how to do this:

```
PRINT "Press a key to stop"
T$ = SPACE$(11)
LOCATE 1, 1
DO UNTIL LEN(INKEY$)
    CALL SysTime(T$)
    CALL QPrint(T$, -1, -1)
LOOP
```

Time2Num

assembler function contained in PRO.LIB

Purpose:

Time2Num converts a time in string form to an equivalent number of seconds after midnight.

Syntax:

```
Time = Time2Num&(T$)
```

Where:

T$ is a time in the form of "HH:MM:SS", and Time receives an equivalent long integer value representing the number of seconds.

Comments:

Because Time2Num has been designed as a function, it must be declared before it may be used.

Time2Num is a powerful routine with two important uses. Besides allowing what would otherwise be an eight character string to be packed to only four bytes, it also provides a simple and effective way to perform time arithmetic.

Once a time has been converted to its equivalent value, you may add or subtract any number of seconds, and then use the companion function Num2Time to obtain the result.

```
T$ = "14:03:22"
Start& = Time2Num&(T$)
Later& = Start& + 3600
NextHour$ = Num2Time$(Later&)
PRINT "One hour after "; T$; " is "; NextHour$
```

Because Time2Num and Num2Time are functions they may also be used within a print or assignment statement directly:

```
PRINT "One hour after"; T$; " is " Num2Time$(Later&)
```

Also see the companion function Num2Time.

Times2

assembler function contained in PRO.LIB

Purpose:

Times2 will multiply an integer variable times 2, without causing an overflow if the result exceeds 32,767.

Syntax:

```
Value = Times2%(Number%)
```

Where:

Value receives the value of Number% multiplied by 2.

Comments:

Because Times2 has been designed as a function, it must be declared before it may be used.

Times2 is a fairly specialized routine, but it is included because we have needed it several times in our own programming. For example, it can be used to determine the number of bytes in an integer array, given the number of elements. Notice that when the returned value exceeds 32,767, QuickBASIC will consider it to be a negative number.

TrapInt

assembler function contained in PRO.LIB

Purpose:

TrapInt will constrain an incoming value to within a specified upper and lower limit.

Syntax:

```
Limited = TrapInt%(Value%, LoLimit%, HiLimit%)
```

Where:

Value% is the value to be constrained, LoLimit% is the mimimum acceptable value, and HiLimit% is the maximum acceptable value.

Comments:

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

TrapInt is very useful for avoiding an "Illegal function call" error, for example when attempting to locate to an illegal row or column. The examples on the following page show how TrapInt might be used for this purpose, along with the BASIC code that would otherwise be needed.

```
INPUT "Enter a row and column ", Row, Column
LOCATE TrapInt%(Row, 1, 25), TrapInt%(Column, 1, 80)
```

vs.

```
IF Row < 1 THEN
   Row = 1
ELSEIF Row > 25 THEN
   Row = 25
END IF

IF Column < 1 THEN
   Column = 1
ELSEIF Column > 80 THEN
   Column = 80
END IF

LOCATE Row, Column
```

ViewFile

BASIC subprogram contained in VIEWFILE.BAS

Purpose:

ViewFile is a complete pop-up file browsing subprogram.

Syntax:

```
CALL ViewFile(FileName$, Wide%, High%, Colr%, HiBit%, Action%)
```

Where:

FileName$ is the name of an ASCII text file to view, Wide% is how wide the displayed window is to be, and High% is the window height. Colr% is the combined foreground and background color, HiBit% is 1 to clear the hi-bit of WordStar type files, and Action% tells ViewFile how it is to behave (see below). The upper left corner of the window is located at the current cursor position.

Comments:

Like most of the pop up utilities provided with QuickPak Professional, ViewFile accepts an action parameter to tell it to open or close its window. This also allows ViewFile to be called in such a way that control is returned to your program while the file is still open.

When ViewFile is called with Action% set to zero, it will behave in a conventional manner. That is, the underlying screen is saved, the user can view all they want, and when they press Escape the file is closed and control returns to the caller.

If Action is instead set to 1, pressing Escape from within ViewFile tells it to return, but with the file still open and the display intact. At that point, ViewFile will have already reset Action to 3, which is the correct value for the next time it is called. To close the file and restore the original screen, call ViewFile again with an Action of 5.

This is the same system used by the pulldown and vertical menu programs, as described in that part of this manual. Please understand that if you do not need or want this "simulated" multi-tasking capability, simply call ViewFile with an Action of 0.

The only errors that are likely to be reported by ViewFile would be caused by a file that isn't there, an invalid drive letter or path, or if the disk drive is not ready. If an error occurs, the QuickPak Professional DOSError% and WhichError% functions will be appropriately set.

ViewFile accommodates files with up to 16,384 lines, and it is amply demonstrated in the DEMOVIEW.BAS program. This is an arbitrary limit that may be expanded, however the /AH command line switch must then be used when starting QuickBASIC or the BC.EXE compiler. Also see the discussion that accompanies the COLORS.BAS routine for more information about combined colors.

All of the expected navigating keys are recognized by ViewFile, including the curson direction keys, PgUp, and PgDn. Also, the Ctrl-Left Arrow key will return the display to the leftmost column.

Also see the FileView assembly language version of this routine.

VLAdd

assembler subroutine contained in PRO.LIB

Purpose:

VLAdd will add two "very long" integers, and return the result in another one.

Syntax:

```
CALL VLAdd(Value1#, Value2#, Sum#, ErrFlag%)
```

Where:

Value1# and Value2# are very long integers in a double precision "alias", Sum# is a very long that receives the result, and ErrFlag% indicates if an overflow occurred.

Comments:

Very long integers are discussed in detail in the section of this manual entitled "Very Long Integers".

VLDiv

assembler subroutine contained in PRO.LIB

Purpose:

VLDiv will divide two "very long" integers, and return the result and remainder in two other ones.

Syntax:

```
CALL VLDiv(Dividend#, Divisor#, Quotient#, Remainder#, ErrFlag%)
```

Where:

Dividend# and Divisor# are very long integers in a double precision "alias", Quotient# and Remainder# are very longs that receive the result, and ErrFlag% indicates if a "divide by zero" was attempted.

Comments:

Very long integers are discussed in detail in the section of this manual entitled "Very Long Integers".

Dividing very long integers is extremely difficult, and while we were developing VLDiv we used CodeView to examine the code produced by QuickBASIC 4, hoping to discover some helpful tips or techniques.

You may be interested to know that QuickBASIC doesn't even attempt to divide *regular* long integers. It simply converts them to floating point values, and lets the floating point math library do the work!

VLMul

assembler subroutine contained in PRO.LIB

Purpose:

VLMul will multiply two "very long" integers, and return the result in another one.

Syntax:

```
CALL VLMul(Value1#, Value2#, Product#, ErrFlag%)
```

Where:

Value1# and Value2# are very long integers in a double precision "alias", Product# is a very long that receives the result, and ErrFlag% indicates if an overflow occurred.

Comments:

Very long integers are discussed in detail in the section of this manual entitled "Very Long Integers".

VLPack

assembler subroutine contained in PRO.LIB

Purpose:

VLPack accepts a very long integer value in the form of a string, and returns it packed to the correct format in a double precision "alias" variable.

Syntax:

```
CALL VLPack(Number$, Value#, ErrFlag%)
```

Where:

Number$ is a string up to nineteen digits long plus an optional minus sign, Value# is a double precision variable that receives the packed information, and ErrFlag% indicates whether the number was packed correctly.

Comments:

The only errors that are likely to occur when using VLPack is giving it a number that contains too many digits (or a null string), or including non-numeric characters.

Very long integers are discussed in detail in the section of this manual entitled "Very Long Integers".

VLSub

assembler subroutine contained in PRO.LIB

Purpose:

VLSub will subtract two "very long" integers, and return the result in another one.

Syntax:

```
CALL VLSub(Value1#, Value2#, Difference#, ErrFlag%)
```

Where:

Value1# and Value2# are very long integers in a double precision "alias", Difference# is a very long that receives the result, and ErrFlag% indicates if an underflow occurred.

Comments:

Very long integers are discussed in detail in the section of this manual entitled "Very Long Integers".

VLUnpack

assembler subroutine contained in PRO.LIB

Purpose:

VLUnpack accepts a very long integer value in the form of a double precision "alias", and returns it in string form suitable for being displayed or printed.

Syntax:

```
Number$ = SPACE$(20)
CALL VLUnpack(Alias#, Number$, ErrFlag%)
```

Where:

Alias# is a double precision variable that contains the very long value, Number$ is a string long enough to hold the returned information (up to twenty digits including a possible minus sign), and ErrFlag% indicates whether the number was successfully unpacked.

Comments:

The only error that is likely to occur when using VLUnpack is giving it a string that contains other than twenty digits to accommodate the returned result.

Very long integers are discussed in detail in the section of this manual entitled "Very Long Integers".

WeekDay

assembler function contained in PRO.LIB

Purpose:

WeekDay will return the day of the week (1 through 7) given a legal DOS date in a string form.

Syntax:

```
Day = WeekDay%(D$)
```

Where:

D$ is a string date in the form "MMDDYY", or "MM-DD-YY", or "MM/DD/YYYY", and so forth, and
Day receives the equivalent number.

Comments:

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

WeekDay calls on an existing DOS service to do the actual calculation, therefore, the legal range of dates is limited to that of DOS: 01-01-1980 through 12-31-2099.

Also see the related routines Num2Day and Date2Day.

Notice that WeekDay modifies the system date and time temporarily, and should *not* be called repeatedly inside a loop. Doing that will cause your system clock to lose time, or possibly even run backwards!

WordWrap

BASIC subprogram contained in WORDWRAP.BAS

Purpose:

WordWrap accepts text that is contained in a single long string, and prints it to the screen with word wrap.

Syntax:

```
CALL WordWrap(Message$, Wide%)
```

Where:

Message$ contains the text to be printed, and Wide% indicates the right margin where the text is to be wrapped.

Comments:

The QuickPak Professional MsgBox subprogram is far more sophisticated than this one, but WordWrap is provided for several reasons. One is that it is very easy to modify to use a printer instead of the screen. The other is that you may not need the extra capabilities (and code size) that MsgBox offers.

Printing always begins at column one, however comments in the WordWrap source code show how to modify it to start printing at any specified left margin.

XMS Manager

assembler subroutines and functions in PRO.LIB

The QuickPak Professional XMS Memory Manager is a complete set of subroutines that allow you to store and retrieve any type of data using extended memory. Extended memory is the memory starting above 1,024K on a 286 or better machine. These routines access this memory using Microsoft's *extended memory specification* version 2.0.

VERY IMPORTANT

These routines require an 80286 or later computer only. The GetCPU function will allow you to determine what processor is currently installed in the host PC.

These routines are designed to emulate the QuickPak Professional EMS routines. However, some routines are not directly applicable to XMS, and others have parameters passed by value instead of reference for increased speed and reduced code size. Therefore, it is important that you declare these routines before using them. Each XMS routine is explained in detail on the pages that follow, along with a brief example showing the correct usage. All of these routines are written in assembly language and are contained in PRO.LIB and PRO7.LIB.

ACCESSING XMS

To access XMS, you first need to load an XMS memory manager such as HIMEM.SYS which Microsoft provides with DOS 5.0 and Windows 3.0. Other memory managers such as QEMM386 and 386MAX also provide XMS memory.

The XMS specification provides three types of memory to your system. The first is regular extended memory; you allocate this memory in kilobytes and it can be used to store data only. The second type of XMS is the *high memory area* (HMA). This is the first 64K segment above the 1,024K boundary.

Because of a quirk in Intel processors, DOS programs can access this segment in real mode, and also execute code there. If you loaded DOS 5.0 high or are using QBX then you are taking advantage of the HMA. Only one program can control the HMA, and you cannot take advantage of it in your own BASIC programs.

The last type of memory is called *upper memory blocks* (UMB). This is the memory between 640K and 1024K that is available on 80386/486 and some 80286 machines. The advantage of using UMB memory over conventional XMS memory is that you can access it directly with the BCopy routine, instead of having to move it into lower memory first by calling an XMS routine.

USING THE XMS ROUTINES

Before any XMS routines may be used in a program, the XMSLoaded function must be invoked to determine if XMS memory is available. Unlike the EMS routines which are invoked internally using an interrupt call, the XMS routines are invoked by calling a specific address. This address is determined by XMSLoaded, which is why it must be called first.

For most applications you will use four of these routines as follows. The XMSLoaded function is first used to determine if XMS memory is installed and available. The Array2XMS and XMS2Array subroutines may then be used to copy data to and from XMS memory. Finally, XMSRelMem will be used to release memory when it is no longer needed.

One important difference between these XMS routines and the EMS routines is in the use of an internal BASIC routine called B_OnExit. B_OnExit is a hook into BASIC's runtime that lets you specify a routine that BASIC will call before your program terminates. What this means is that all XMS memory you allocate will be automatically released when your program terminates. This is very useful when working in the QB environment, and you are likely to stop and restart your program many times without calling the routines that free the XMS memory.

However, there may be times when you want some XMS memory to remain after your program has terminated. For instance, you can pass a large amount of data between executable programs by storing it in XMS, and then passing to the second program the handle of the XMS block. We have provided the routine KeepXMSHandle for just this purpose.

A quirk in the XMS specification requires that all memory moves be an even number of bytes. Although we allow you to use a data structure with an odd length in the XMSGet1El and XMSSet1El routines, we require that you have an even amount of bytes when using Array2XMS and XMS2Array. This is done to keep code size small and to increase the speed of the routines.

When using XMSGet1El and XMSSet1El, it is better if you work with a data structure with an even length. Although this is not strictly required, when these routines are used with an odd-length data structure they have to make two XMS memory accesses to retrieve the information. If you are using an odd-length data structure with the Array2XMS and XMS2Array routines, then you have to use an even number of elements. Although the discussion that follows describes storing and retrieving arrays, Array2XMS and XMS2Array may in fact be used with any contiguous block of memory.

When Array2XMS is called, the correct amount of memory will be allocated for you automatically, based on the number of elements you are storing and the size of each element. Besides allocating memory, Array2XMS also returns a "handle" number that will be used to retrieve the array later. This handle remains active until the XMSRelMem routine is called, or your program terminates and the memory is released back to the system.

Each time Array2XMS is called a new handle is obtained. Thus, if you intend to save data repeatedly to the same XMS memory you should call XMSRelMem before each subsequent save. However, when XMS2Array is used to retrieve an array the memory is not automatically released. Therefore, you may retrieve the array as many times as you'd like, and call XMSRelMem only once when the memory is no longer needed. This is the same method the EMS routines use, and we have found it to be the most flexible.

A list of the possible XMS error codes is shown in the table on the following page. Similar to the way errors are reported for the various QuickPak Professional DOS routines, XMS errors are detected by querying the XMSError function. This function returns the status of the most recent XMS service, and is either zero meaning no error occurred, or it contains an error code. The official XMS errors have values of 128 or higher, and we have added a few of our own starting at 1.

XMS ERROR CODES

Hex	Dec	Meaning
00H	0	No error
01H	1	XMSLoaded hasn't been used yet to initialize these routines
02H	2	The element length was given as zero
03H	3	The number of elements was given as zero
80H	128	Function not implemented
81H	129	VDISK device is detected
A0H	160	All available extended memory is allocated
A1H	161	All available extended memory handles are in use
A2H	162	Handle is invalid
A3H	163	Source handle is invalid
A4H	164	Source offset is invalid
A5h	165	Destination handle is invalid
A6H	166	Destination offset is invalid
A7H	167	Length is invalid
A8H	168	Move has invalid overlap
A9H	169	A parity error occurred
B0H	176	A smaller UMB is available
B1H	177	No UMBs are available
B2H	178	UMB segment number is invalid

Some of these routines have been designed as functions while others pass parameters by value, so it is important that they be declared before you use them. A complete demonstration including appropriate declarations is given in the DEMOXMS.BAS example program.

XMS FUNCTIONS

XMSError - Function

Reports the status of the most recent XMS operation.

```
DECLARE FUNCTION XMSError% ()
IF XMSError% THEN PRINT "Error number"; XMSError%; "occurred."
```

XMSLoaded - Function

Returns -1 if the XMS driver software is loaded, or 0 if it is not.

```
DECLARE FUNCTION XMSLoaded% ()
IF XMSLoaded% THEN
   PRINT "XMS memory is loaded on this PC."
ELSE
   PRINT "Sorry, this PC does not have XMS."
END IF
```

XMS SUBROUTINES

XMSAllocMem - Subroutine

Allocate a specific number of kilobytes of XMS memory.

```
DECLARE SUB XMSAllocMem (BYVAL NumK%, Handle%)
CALL XMSAllocMem (NumK%, Handle%)
```

Where NumK% is the number of kilobytes of XMS memory requested, and Handle% is returned to identify the memory for later use.

XMSRelMem - Subroutine

Releases all memory associated with a specified handle.

```
DECLARE SUB XMSRelMem (BYVAL Handle%)
CALL XMSRelMem (Handle%)
```

Array2XMS - Subroutine

Copies all or part of an array or other block of memory into XMS memory.

```
CALL Array2XMS (SEG Array(Start), ElSize%, NumEls%, Handle%)
```

or

```
CALL Array2XMS (BYVAL Segment%, BYVAL Address%, NumBytes%, _
    1, Handle%)
```

Where Array(Start) is any numeric or TYPE array, ElSize% is the size of each element in bytes, NumEls% is the total number of elements to copy into XMS memory, and Handle% is the handle returned by Array2XMS. The second example shows how to store any contiguous block of memory.

Because of a requirement in the XMS specification, the total number of bytes must be even. This limitation requires you to have an even length data size or an even number of array elements.

The ElSize% parameter would be 2 for an integer array, 4 for a long integer or single precision array, and 8 for a currency or double precision array. Array2XMS also accepts the negative code values used by QuickPak Professional TYPE sort routines. To store fixed-length string and TYPE arrays in extended memory, ElSize% will be the length of each element. However, to store a fixed-length string array you must first define it as a TYPE. This is described in the section "Calling with Segments".

To store a conventional (not fixed-length) string array in XMS memory you must first store it in an integer array using the QuickPak Professional StringSave routine. Then the integer array may be copied into XMS memory. To retrieve the string array you would use XMS2Array to copy it back to an integer array, and then use StringRest to place it back into the string array. StringSave and StringRest are described in the section entitled "String Manager Routines".

Array2XMS may be used to store a single item, or any contiguous block of memory by specifying the number of bytes in ElSize%, and using 1 for NumEls%. Be sure to make ElSize% an even number or else you will get an error 167 "Invalid Length". The actual number of bytes copied into extended memory is calculated within Array2XMS by multiplying ElSize% times NumEls%. If the number of bytes is 16K (16,384) bytes or less, simply set ElSize% to the number of bytes and use 1 for NumEls%. To store, say, 64K (65536 bytes) you would specify ElSize% as 16384 and set NumEls% to 4. Any similar combination will also work. The example below shows how to save a single text screen from a color display.

```
CALL Array2XMS(BYVAL &HB800, BYVAL 0, 4000, 1, Handle)
```

Then to display the screen later you would use:

```
CALL XMS2Array(BYVAL &HB800, BYVAL 0, 4000, 1, Handle)
```

If there is not enough XMS memory available XMSError will return 160. If there are no XMS handles available XMSError returns 161. If you specify an odd number of bytes to transfer, then XMSError will return error 167.

XMS2Array - Subroutine

Retrieves an array or other block of memory from XMS memory.

```
CALL XMS2Array (SEG Array(Start), ElSize%, NumEls%, Handle%)
```

or

```
CALL XMS2Array (BYVAL Segment%, BYVAL Address%, NumBytes%, _
    1, Handle%)
```

Where Array(Start) is any numeric or TYPE array, ElSize% is the size of each element in bytes, NumEls% is the total number of elements to copy from XMS memory, and Handle% is the handle that was assigned by Array2XMS when the array was stored. It is essential that the array being restored has been sufficiently dimensioned to hold the information being copied to it.

This routine is the exact opposite of Array2XMS, and the parameters have the same meaning as in that routine. XMS2Array does not release the XMS memory assigned to Handle%. If you want to release it you must call XMSRelMem.

XMSGet1El - Subroutine

XMSGet1El allows retrieving a single element from extended memory.

```
CALL XMSGet1El(SEG Value, ElSize%, ElNum%, Handle%)
```

Where Value is any variable, and ElSize% is either its length in bytes or a special code that indicates the length (see below). ElNum% is the element number (based at one, not zero), and Handle% is the XMS handle that was assigned when the array was first saved.

Although Array2XMS and XMS2Array routines require an even number of bytes, this isn't the case with XMSGet1El. However, it takes two XMS memory accesses to get an odd length element, and it takes three extended memory accesses to set an odd length element. Therefore, your program will run more quickly if you can keep your elements at an even length.

XMSGet1El lets you retrieve a single element from an array that has been saved in extended memory, when you don't want to have to retrieve the entire array. Another important use would be to access a single screen from among several that are being stored in XMS memory. Because the same routine may be used to process different types of variable, you should declare it using the AS ANY option:

```
DECLARE SUB XMSGet1El(SEG Value AS ANY, ElSize%, ElNum%, Handle%)
```

The ElSize% variable may optionally be the special size code that is used by the various QuickPak Professional TYPE array sorts.

Also see the companion routine XMSSet1El which assigns a single element to an array or block of data in XMS memory.

XMSSet1El - Subroutine

XMSSet1El allows assigning a single element in an array that is stored in extended memory.

```
CALL XMSSet1El(SEG Value, ElSize%, ElNum%, Handle%)
```

Where Value is any variable, and ElSize% is either its length in bytes, or a special code that indicates the length (see below). ElNum% is the element number (based at one, not zero), and Handle% is the XMS handle that was assigned when the array was first saved.

Although Array2XMS and XMS2Array routines require an even number of bytes, this isn't the case with XMSSet1El. However, it takes two XMS memory accesses to get an odd length element, and it takes three extended memory accesses to set an odd length element. Therefore, your program will run more quickly if you can keep your elements at an even length.

XMSSet1El lets you assign a single element from an array that has been saved in extended memory, when you don't want to have to retrieve the entire array, make the assignment, and then save it back again. Another important use is to store multiple screen images in XMS memory.

Because the same routine may be used to process different types of variable, you should declare it using the AS ANY option:

```
DECLARE SUB XMSSet1El(SEG Value AS ANY, ElSize%, ElNum%, Handle%)
```

The ElSize% variable may optionally be the special size code that is used by the various QuickPak Professional TYPE array sorts.

Also see the companion routine XMSGet1El which allows you to assign a single element.

XMSInfo - Subroutine

XMSInfo retrieves several useful items of information about the XMS memory in your system, and stores it in a TYPE variable. The TYPE structure used is as follows:

```
TYPE XMSInfoType
   XMSVersion    AS INTEGER
   DriverVersion AS INTEGER
   NumHandles    AS INTEGER
   FreeMem       AS INTEGER
   Largest       AS INTEGER
   HMAAvail      AS INTEGER
   LargestUMB    AS LONG
END TYPE
```

Here, XMSVersion is the version of the XMS specification your driver conforms to, and DriverVersion is the version of your particular driver. Both of these values are stored with the major version times 100. For example, version 6.25 will be reported as the value 625.

NumHandles is the number of free handles available. FreeMem is the total amount of XMS memory available in kilobytes, and Largest is the largest single block of XMS memory you can allocate at a single time in kilobytes.

HMAAvail is a boolean (true/false) value that tells you if the HMA (High Memory Area) is available. It will be -1 if it is free, or 0 if another program has control of it.

LargestUMB is the largest Upper Memory Block you can allocate, in bytes. This does not represent the total amount of free UMBs, however, because upper memory is usually fragmented.

DEMOXMS.BAS contains examples on calling this routine and displaying the information returned.

XMSSetError - Subroutine

Allows a BASIC program to set or clear the XMSError value.

```
DECLARE SUB XMSSetError (BYVAL Value%)
CALL XMSSetError (Value%)
```

KeepXMSHandle - Subroutine

Allows you to retain XMS memory after your program terminates.

```
DECLARE SUB KeepXMSHandle (BYVAL Handle%)
CALL KeepXMSHandle (Handle%)
```

One important difference between the XMS memory manager and the EMS memory manager is the use of the B_OnExit routine. B_OnExit is a hook into the QuickBASIC runtime that lets you tell BASIC to call a particular routine automatically when your program terminates. This allows all of your XMS memory to be released back to the system when your program terminates.

Using B_OnExit is very useful when working in the environment where you are apt to restart your program many times without first calling your normal termination routine. However, there are times when you want some XMS memory you have allocated to remain after your program has terminated. KeepXMSHandle lets you pass data between programs without having to store it on disk.

UMBAllocMem - Subroutine

Allocates a specific number of bytes of upper memory.

```
DECLARE SUB UMBAllocMem (BYVAL NumBytes%, Segment%)
CALL UMBAllocMem (NumBytes%, HandleSegment%)
```

Where NumBytes% is the number of bytes of upper memory requested, and Segment% is the segment of the block that was allocated. You would then use BCopy or BCopyT to move data in and out of this block. Upper memory is the memory located between 640K and 1024K on 80386/486 and some 80286 machines.

UMBRelMem - Subroutine

Releases all memory associated with a specified upper memory segment.

```
DECLARE SUB UMBRelMem (BYVAL Segment%)
CALL UMBRelMem (Segment%)
```

Chapter 8
String Manipulation Routines

ASCII

assembler function contained in PRO.LIB

Purpose:

ASCII obtains the ASCII value for a string exactly as BASIC's ASC function does, but it will not cause an "Illegal Function Call" error if the string is null.

Syntax:

```
A = ASCII%(Any$)
```

Where:

Any$ is any string, and A receives the ASCII value of its first character. If the string is null, ASCII instead returns -1.

Comments:

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

ASCII was created to avoid the extra steps needed to insure that a string isn't null before ASC can be used. Attempting to obtain the ASCII value of a null string using the BASIC ASC() function will cause an Illegal Function Call error. A quick look at QuickBASIC's ASC under CodeView also reveals that our ASCII is substantially faster.

Before ASCII, you would need to add additional code to your programs as shown below:

```
X$ = INKEY$
IF LEN(X$) THEN
   IF ASC(X$) => 65 THEN
     PRINT "You pressed a letter key"
   END IF
END IF
```

ASCII instead lets you use:

```
IF ASCII%(INKEY$) => 65 THEN
   PRINT "You pressed a letter key"
END IF
```

Blanks

assembler function contained in PRO.LIB

Purpose:

Blanks will report the number of leading blanks in the specified string. Both CHR$(32) and the CHR$(0) "null" character are recognized.

Syntax:

```
NumBlanks = Blanks%(Work$)
```

Where:

NumBlanks receives the number of leading blank and/or CHR$(0) bytes in Work$.

Comments:

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

Blanks is useful for finding where within a string the actual text begins. Other methods would require either stepping through the string a character at a time using MID$, or comparing the length of the string with the length of a copy that had been processed with LTRIM$.

Either of those methods would be extremely slow compared to using Blanks, and of course LTRIM$ does not consider CHR$(0) as a blank. Recognizing CHR$(0) is important with fixed-length strings, because QuickBASIC uses that to initialize each string.

Compact

assembler function contained in PRO.LIB

Purpose:

Compact compresses a string by removing all embedded blanks.

Syntax:

```
NewString$ = Compact$(Old$)
```

Where:

NewString$ is assigned from Old$, but with all CHR$(32) blanks removed.

Comments:

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

Unlike most functions, the original string is also compacted. In designing Compact we would have preferred to not tamper with the original string. However, that would require reserving sufficient memory to hold the longest possible string that Compact would act on.

If the original string must be preserved you should use parentheses to force BASIC to create a copy of it.

```
NewString$ = Compact$((Old$))
```

Encrypt and Encrypt2

assembler subroutines contained in PRO.LIB

Purpose:

Encrypt and Encrypt2 will take any conventional or fixed-length string, and encrypt it using a password you provide.

Syntax:

```
CALL Encrypt(X$, PassWord$)
```

or

```
CALL Encrypt2(X$, PassWord$)
```

Where:

X$ is the string to be encrypted, and PassWord$ is the password to be used. Notice that these routines are intended both to encrypt the string, and to decrypt it again later using the same password. Also notice that when using Encrypt2, the password must be less than 44 characters long.

Comments:

Encrypt2 is based on a fairly simple algorithm that is also reasonably secure. While the Secret Service or the FBI will probably find it inadequate, it would be pretty difficult for most folks to crack. Encrypt is somewhat less secure, and is provided solely for compatibility with earlier versions of QuickPak and QuickPak Professional. Also for compatibility, the FileCrypt BASIC subprogram still uses the original Encrypt routine. We recommend that you modify the BASIC source code to use Encrypt2, unless you already have files that were encrypted using Encrypt.

There are a few points to consider that will maximize the security provided by Encrypt.

1) Use a long password.
2) Don't select a password that could be easily guessed.
3) Double-encrypt the string.

Using a long password will minimize the chance of creating a recognizable repeating pattern.

If you use a password such as your last name or birthday, you will make it that much easier for someone to guess. Of course, don't go overboard with something like "J@GE#SVEY34KJFDBE!OSNBDN", or you'll never remember it yourself. MCI Mail uses a clever system of nonsense letters that are also easy to pronounce, such as "YAXALUPA" and "ZIGUMINI".

If the password is first encrypted with another one, the protection will be much more effective. The FileCrypt subprogram shows an effective use of this principle. Of course, you must then decrypt the string twice in the reverse order later.

Encrypt and Encrypt2 use the XOR command to apply the characters in the password against those in the string being encrypted. The equivalent BASIC algorithm for Encrypt is:

```
L = LEN(PassWord$)
FOR X = 1 TO LEN(X$)
    Pass = ASC(MID$(PassWord$, (X MOD L) - L * ((X MOD L) = 0), 1))
    MID$(X$, X, 1) = CHR$(ASC(MID$(X$, X, 1)) XOR Pass)
NEXT
```

Encrypt2 is nearly identical, except the characters in the password are altered as the encryption progresses.

Far2Str

assembler function contained in PRO.LIB

Purpose:

Far2Str will retrieve an ASCIIZ string from anywhere in memory, and return it as a conventional BASIC string.

Syntax:

```
Work$ = Far2Str$(BYVAL Segment%, BYVAL Address%)
```

Where:

Segment% and Address% specify where in memory the string is located.

Comments:

Because Far2Str has been designed as a function, it must be declared before it may be used.

An ASCIIZ string is a string whose end is marked with a CHR$(0) zero byte (hence the "Z"), such as the strings returned by DOS and the C language. We wrote this routine to simplify access to Jake Geller's Spell Engine program which (erroneously) assumes that all programmers use C.

If you are using an array to store far string data, then you can save a few bytes each time Far2Str is used by declaring and calling it like this:

```
DECLARE FUNCTION Far2Str$(SEG FarString AS ANY)
Work$ = Far2Str$(Array(0))
```

If you are accessing an ASCIIZ string in near memory, simply use VARSEG (AnyVar) as the segment parameter.

Note that Far2Str is limited to a maximum of 40 characters. This limit reflects the number of bytes reserved in near memory for the function output. You can of course change this by modifying the assembler source.

FUsing

assembler function contained in PRO.LIB

Purpose:

FUsing accepts an incoming number and image string, and returns it as a formatted string much like BASIC's PRINT USING would.

Syntax:

```
Formatted$ = FUsing$(STR$(Number), Image$)
```

Where:

Number is any number whether integer, single, or double precision, and Image$ indicates how the result is to be formatted. If the number will not fit within the allotted space, the first digit of the returned string will be 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, we are letting BASIC do most of the dirty work to interpret the floating point numbers. This also lets FUsing accept any type of numeric variable. Normally, assembler routines must be written to expect one type of variable only. If you do not use BASIC's STR$ function and directly pass a string to FUsing, be sure to add an extra space to the beginning of the number string that you are formatting, i.e., "1234" becomes " 1234". Most of the formatting codes that PRINT USING recognizes are supported, including commas, dollar signs, and leading asterisks.

It is very important that you not include any blank spaces within the image string. For example "*##.##*" is perfectly legal, while " *##.##*" will not work as expected.

The table below summarizes FUsing's capabilities:

#	represents each digit position
.	specifies a decimal point

Strings

+	causes the sign of the number (+ or -) to be added (the sign must be the first character in the field)
**	replaces leading spaces in the field with asterisks
$$	adds a dollar sign to the left of the number
**$	combines the effects of ** and $$
,	specifies that commas are to be added to the formatted string

FUsing is demonstrated in the DEMOCM.BAS example program.

Although BASIC accepts multiple commas in the image string, FUsing requires only one. If there is a decimal point within the string, the comma must be placed just before it. Otherwise, the comma must be the last character.

Also see the description for PUsing elsewhere in this manual.

InCount and InCount2

assembler functions contained in PRO.LIB

Purpose:

InCount will quickly report how many times one string occurs within another, and the search string may contain any number of "?" wild cards. InCount2 works the same way, but searching is case-insensitive.

Syntax:

```
Count = InCount%(Source$, Search$)
```

Where:

Source$ and Search$ are either conventional or fixed-length strings, and Count receives the number of times Search$ occurs in Source$.

Comments:

Because InCount and InCount2 have been designed as functions, they must be declared before being used.

InCount is used to advantage by the QuickPak Professional ExpandTab$ and Parse$ functions. Of course, it could have many uses in your own programs as well.

For example, if you accept command line switches such as "/D" or "/T", InCount would quickly tell how many have been entered by the user:

```
NumSwitches = InCount(COMMAND$, "/")
```

InCount2 performs the identical function, except that it ignores capitalization in both strings as it searches. Both routines accept any number of the "?" wild cards in the search specification.

InCountTbl

assembler function contained in PRO.LIB

Purpose:

InCountTbl returns the number of characters in a string that match any of the characters in a table.

Syntax:

```
Count = InCountTbl%(Source$, Table$)
```

Where:

Source$ is the string being examined, and Table$ holds a list of characters to search for.

Comments:

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

InCountTbl is useful in a variety of situations, for example when searching for one or more printer control codes that have been imbedded within a string. In the case of a print routine that must also deal with margins, it would be necessary to know the length of each string, but without regard to the printer codes. The following example shows this in context.

```
Work$ = CHR$(15) + "This is a test"     '15 is an Epson code
Table$ = CHR$(27) + CHR$(15) + CHR$(2)  'some control codes
ActualLen = LEN(Work$) - InCountTbl%(Work$, Table$)
'                                         the answer is 14
```

InstrTbl and InstrTbl2

assembler functions contained in PRO.LIB

Purpose:

InstrTbl and InstrTbl2 will quickly search a string for the first occurrence of any characters that are specified in a table string. InstrTbl honors capitalization, while InstrTbl2 does not.

Syntax:

```
Position = InstrTbl%(Start%, Source$, Table$)
```

Where:

Start% specifies where in the source string searching is to begin, Source$ is the string being examined, and Table$ contains one or more characters to search for. Position then receives the position where the first match was found. If there is no match, Position will receive zero.

Comments:

Because InstrTbl and InstrTbl2 have been designed as functions, they must be declared before they may be used.

Unlike BASIC's INSTR function, the Start% parameter must always be given. If you intend to search the entire source string, then Start% should be set to 1. A typical use of InstrTbl2 is given below:

```
Source$ = "FIND ME HIDING HERE"        'search this string
Table$ = "aeiou"                       'find the first vowel
Position = InstrTbl2%(1, Source$, Table$)
'                                       Position receives 2
```

Also see the related routines InstrTblB and InstrTblB2, which search backwards through a string.

InstrTblB and InstrTblB2

assembler functions contained in PRO.LIB

Purpose:

InstrTblB and InstrTblB2 will search a string backwards for the first occurrence of any characters that are specified in a table. InstrTblB honors capitalization, while InstrTblB2 does not.

Syntax:

```
Position = InstrTblB%(Start%, Source$, Table$)
```

Where:

Start% specifies where in the source string searching is to begin, Source$ is the string being examined, and Table$ contains one or more characters to search for. Position then receives the position where the first match closest to the end of the string was found.

Comments:

Because InstrTblB and InstrTblB2 have been designed as functions, they must be declared before they may be used.

Unlike BASIC's INSTR function, the Start% parameter must always be given. If you intend to search the entire source string, then Start% should be set to -1. A typical use of InstrTblB is given below:

```
Source$ = "FIND 987 HIDING"           'search this string
Table$ = "0123456789"                 'find the last digit
Position = InstrTblB%(-1, Source$, Table$)
                                       'Position receives 8
```

Also see the related routines InstrTbl and InstrTbl2, which search forward through the string.

Strings

LongestStr

assembler function contained in PRO.LIB

Purpose:

LongestStr returns the length of the longest string in an entire array.

Syntax:

```
MaxLength = LongestStr%(Array$())
```

Where:

MaxLength receives the length of the longest string contained in Array$().

Comments:

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

LongestStr is useful in a variety of situations, for example to determine how wide to make a menu that will display an entire array. Unlike most of the QuickPak Professional assembler functions that require passing an entire array by specifying its first element, LongestStr expects the array to be passed using empty parentheses. If you specify an array other than a conventional (not fixed-length) string, LongestStr will return a value of zero.

See the LONGSTR.BAS demonstration program for an example of how LongestStr is declared and used.

LowASCII

assembler subroutine contained in PRO.LIB

Purpose:

LowASCII will strip the "high bit" from all of the characters in a specified string.

Syntax:

```
CALL LowASCII(X$)
```

Where:

X$ is the string to be processed.

Comments:

LowASCII is primarily intended to fix data that is in the format used by WordStar and some other word processors. It is equally useful for printers that can't accept characters with an ASCII value greater than 127.

Also see the related routines Translate and RemCtrl.

Lower

assembler subroutine contained in PRO.LIB

Purpose:

Lower will quickly convert all of the alphabetic characters in a specified string to lower case.

Syntax:

```
CALL Lower(X$)
```

Where:

X$ is the string to be processed.

Comments:

Even though QuickBASIC 4 provides a built-in LCASE$() function, Lower is provided because it is considerably faster. One of the problems with LCASE$ (and UCASE$) is that they must be used as a function. That is, in order to convert a string it must be reassigned which takes time:

```
X$ = LCASE$(X$)
```

Lower instead scans through the named string very quickly, converting only the upper case alphabetic characters.

Also see the related routine Upper.

LowerTbl

assembler subroutine contained in PRO.LIB

Purpose:

LowerTbl will convert all of the characters in a string to lower case, and it also looks in a supplied table to determine how to handle foreign characters.

Syntax:

```
CALL LowerTbl(Work$, Table$)
```

Where:

Work$ is the string to be processed, and Table$ holds one or more pairs of lower and upper case extended characters.

Comments:

LowerTbl was written to accommodate our European friends, although it could also be used as a general purpose table-driven character substitution routine. The table string comprises pairs of characters, with the first in each pair being the lower case version, and the second its upper case counterpart. Though we could have hard-coded the conversions into the routine, it would need to be changed for different languages. For example, the character conversions that are used for German are not necessarily the same as those for other languages. A typical table would be assigned like this:

```
Table$ = "äÄöÖüÜñÑ"
```

Notice that normal characters are converted to lower case regardless of the table, and Table$ is needed solely to accommodate the additional characters.

Also see the related routine UpperTbl, which converts a string to upper case.

MidChar

assembler function contained in PRO.LIB

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 negative or past the end of Work$.

Comments:

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

MidChar (and its companion, MidCharS) lets you access characters in a string much faster than with BASIC's MID$. These were originally written for our P.D.Q. product, but they are so useful we have added them to QuickPak Professional as well.

The first example below shows how MidChar is used, and the second shows an equivalent BASIC statement.

```
Char = MidChar%(Work$, Position%)
Char = ASC(MID$(Work$, Position%, 1))
```

Every time you use the MID$ function a copy of the specified portion of the string is created, which takes time. Further, string operations and comparisons are always slower than integer operations. Therefore, MidChar is most useful when examining characters in the string, perhaps to parse a file name for the path or drive letter as shown below. MidChar is approximately five times faster than BASIC's MID$ when using QuickBASIC or near strings in BASIC PDS.

```
DEFINT A-Z
FileName$ = "C:\SUBDIR1\SUBDIR2\FILENAME.EXT"
FOR X = LEN(FileName$) TO 1 STEP -1      'walk backwards
  IF MidChar%(FileName$, X) = 92 THEN    '92 = ASC("\")
    Path$ = LEFT$(FileName$, X)          'isolate the path
    FileName$ = MID$(FileName$, X + 1)   'and then the name
    EXIT FOR                             'no need to continue
  END IF
NEXT
```

MidCharS

assembler subroutine contained in PRO.LIB

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, and it is much more efficient than using the statement form of BASIC's MID$ when inserting a single character.

Using MidCharS adds only 17 bytes of code to a program, compared to 25 for the following BASIC equivalent:

```
MID$(Work$, Position, 1) = Char$
```

Like MidChar, MidCharS is five times faster than BASIC with near strings.

NotInstr

assembler function contained in PRO.LIB

Purpose:

NotInstr reports the first location in one string that does *not* match any of the characters in another string.

Syntax:

```
Position% = NotInstr%(Start%, Searched$, Table$)
```

Where:

Position receives the offset of the first character in Searched$ that doesn't match any of the characters in Table$. If all characters match, Position will be zero.

Comments:

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

NotInstr is not strictly the inverse of BASIC's INSTR function. Where INSTR compares entire strings, NotInstr uses a table of single characters. The first position in the searched string that doesn't match any of them is then returned by the function. You can tell NotInstr to begin at any character in the searched string, or use a Start value of 1 to consider the entire string.

The following example displays 2, because the second character in Source$ does not match any of the characters in Table$.

```
Source$ = "QUICKPAK"
Table$ = "QKP"
PRINT NotInstr%(1, Source$, Table$) 'prints 2
```

You can see NotInstr in use by examining the DIRTREE.BAS program.

Also see the descriptions for the QInstr? and InstrTbl? series of INSTR variations.

Null

assembler function contained in PRO.LIB

Purpose:

Null will report if the specified string is either null, or is filled only with blank or CHR$(0) characters.

Syntax:

```
Empty = Null%(Work$)
```

Where:

Empty is assigned -1 (true) if the string is effectively null, or 0 (false) if it is not.

Comments:

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

Null was developed to overcome the clumsy code that is otherwise needed to determine if a string contains useful characters. The usual way to do this would be either:

```
IF Work$ = "" OR Work$ = SPACE$(LEN(Work$))
   THEN . . .                    'It's null
```

or

```
IF LTRIM$(RTRIM$(Work$)) = ""
   THEN . . .                    'Also null
```

Besides being considerably faster, Null considers a CHR$(0) byte the same as a CHR$(32) space. It is important to recognize CHR$(0) because that is what QuickBASIC uses to initialize fixed-length strings.

ParseString

assembler function contained in PRO.LIB

Purpose:

ParseString accepts a string containing delimited information, and returns portions of that string each time it is invoked.

Syntax:

```
Item$ = ParseString$(CurPos%, Work$, Delimit$)
```

Where:

CurPos% tells where in the string to begin processing, Work$ is the string being read, and Delimit$ is a table of one or more delimiting characters. Item$ then receives the next delimited component from Work$.

Comments:

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

This is an enhanced version of PDQParse that we include with our P.D.Q. library. Each time ParseString is invoked it returns the next portion of the string, much like READ returns the next available DATA item in a list. The example below shows how ParseString is set up and used.

```
CurPos% = 1                 'start reading at the beginning
Delimit$ = ";/-,"           'any of these characters will delimit
Work$ = ENVIRON$("PATH")    'for example: "C:\DOS;C:\UTIL;C:QB4"
DO                          'read and print each PATH item
   ThisItem$ = ParseString$(CurPos%, Work$, Delimit$)
   PRINT ThisItem$
LOOP WHILE LEN(ThisItem$)   'until there are no more
```

The CurPos% (current position) parameter lets you indicate where to begin reading, and would usually be set to 1 initially. If CurPos% is less than 1 (0 or negative) it is forced to 1. If CurPos% points past the end of the string, then ParseString will return a null (zero-length) string.

Each time ParseString is used, it advances CurPos% automatically to point just past the last delimiter. You can change CurPos% manually to move around in one string, and also use ParseString with more than one string at a time. To work with multiple strings you must maintain a separate version of the CurPos% variable for each string being read.

If any of the delimiters in Delimit$ are encountered, ParseString returns the portion of the string that precedes the delimiter. Note that only ten characters are allowed for Delimit$. If Delimit$ is longer than ten, only the first ten characters are used.

The PARSESTR.BAS demonstration program shows an example of using ParseString.

ProperName

assembler subroutine contained in PRO.LIB

Purpose:

ProperName converts the first letter of each word in a string to upper case.

Syntax:

```
CALL ProperName(Work$)
```

Where:

Work$ is typically a name such as "john doe, jr.", and ProperName converts it to "John Doe, Jr.".

Comments:

ProperName is useful for automatically enforcing the correct capitalization on name fields in a database program. We considered writing ProperName to also convert the remaining characters to lower case, but that could have introduced undesirable side effects. For example, if the incoming name was "MacDonald", the first "D" would no longer be correct. Even though the routine could be written to search for all occurrences of "Mc" and "Mac", it would still fail on names such as "DeVito" or "O'Malley".

QInstr and QInstr2

assembler functions contained in PRO.LIB

Purpose:

QInstr serves the same purpose as BASIC's INSTR function, except it accepts any number of "?" wild cards. QInstr2 is similar, but it ignores capitalization when examining the strings.

Syntax:

```
Position = QInstr%(StartPos%, Source$, Search$)
```

Where:

StartPos% tells QInstr where in the string to start looking, and Source$ is the string being examined. Search$ is the string whose position in Source$ is being sought, and Position% receives where in Source$ the first occurrence of Search$ is located.

Comments:

Because QInstr and QInstr2 have been designed as functions, they must be declared before they may be used.

Unlike BASIC's INSTR where the starting position is an optional parameter, it must be given when using QInstr and QInstr2. An assembler routine simply cannot be called with a different number of parameters than it expects.

To search a string starting at the beginning, set StartPos% to 1. Using 0 will always return 0 for Position, which is what BASIC's INSTR does.

In QuickPak Professional 1.XX versions QInstr2 was named QInstr, and there was no case insensitive version.

Also see the description for NotInstr and the InstrTbl? series of INSTR variations.

QInstrB and QInstrB2

assembler functions contained in PRO.LIB

Purpose:

QInstrB is similar to the QuickPak Professional QInstr function, except it scans the source string *backwards* looking for a specified substring. QInstrB2 is an alternate version that ignores capitalization in both strings. Like QInstr and QInstr2, these functions accept any number of "?" wild cards to match any character.

Syntax:

```
Position = QInstrB%(StartPos%, Source$, Search$)
```

Where:

StartPos% tells QInstrB where in the string to start looking, and Source$ is the string being examined. Search$ is the string whose position in Source$ is being sought, and Position% receives where in Source$ the last occurrence of Search$ is located. If no matches are found, Position will be zero. If StartPos% is set to -1, then the entire string is examined starting at the last character.

Comments:

Because QInstrB and QInstrB2 have been designed as functions, they must be declared before they may be used.

QInstrB and QInstrB2 are useful in a variety of situations. For example, to isolate the path portion of a complete file name you would need to know where in the string the last backslash (\) is located.

Unlike BASIC's INSTR where the starting position is an optional parameter, it must be given when using QInstrB and QInstrB2. An assembler routine cannot be called with a different number of parameters than it expects. Even though searching is performed backwards, the returned position indicates where in the source string the search string begins.

Also see the description for NotInstr and the InstrTbl? series of INSTR variations.

QInstrH

assembler subroutine contained in PRO.LIB

Purpose:

QInstrH is a "Huge" INSTR routine that will locate a string of text anywhere in the PC's normal 1MB address space.

Syntax:

```
CALL QInstrH(Segment%, Address%, Search$, NumBytes&)
```

Where:

Segment% and Address% identify where in memory you want to begin searching, Search$ is the string to locate, and NumBytes& specifies how many bytes are to be searched. If the string is found, Segment% and Address% show where it is in memory. If it is not found, then Segment% and Address% are both returned set to a value of zero.

Comments:

Please note that our method of returning zeros for the segment and address can fail if you are searching starting at the very bottom of memory and a match is found at the very first byte. However, it is very unlikely that you'll need to search the interrupt vector table using QInstrH.

QInstrH is demonstrated in QINSTRH.BAS.

QPLeft, QPMid, and QPRight

assembler functions contained in PRO.LIB

Purpose:

QPLeft serves the same purpose as BASIC's LEFT$() function, but it is faster in some situations. Similarly, QPMid and QPRight replace BASIC's MID$() and RIGHT$() functions respectively.

Syntax:

```
SubString$ = QPLeft$(Work$, NumChars%)
```

or

```
SubString$ = QPMid$(Work$, StartChar%, NumChars%)
```

or

```
SubString$ = QPRight$(Work$, NumChars%)
```

Where:

SubString$ receives NumChars% characters from Work$. StartChar% is relevant for QPMid$ only, and indicates how far into the string to begin extracting characters.

Unlike BASIC's MID$() function, the NumChars% argument is not optional with QPMid. However, QPMid recognizes a value of -1 for NumChars% to mean that all characters through the end of the string should be considered.

Comments:

Because these routines have been designed as functions, they must be declared before they may be used.

These functions are provided as fast replacements for the built-in equivalent functions in QuickBASIC. In many situations they will be faster than QuickBASIC, however they are always slower when used with BASIC PDS far strings due to the additional overhead required to access far data.

QPLen

assembler function contained in PRO.LIB

Purpose:

QPLen serves the same purpose as BASIC's LEN function, but it is considerably faster.

Syntax:

```
Length = QPLen%(Work$)
```

Where:

Length is assigned to the length of Work$.

Comments:

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

Because of the way BASIC's system of library calls is organized, even the simple act of obtaining the length of a string involves a lot of unnecessary code. The entire assembler source code for QPLen is shown below to illustrate how simple this function really is.

```
QPLen Proc Far
    Mov   SI,SP                  ;put the stack pointer into SI
    Mov   BX,[SI+04]             ;get the address for X$ descriptor
    Mov   AX,[BX]                ;put LEN(X$) into AX for the
                                 ;function output
    Ret   2                      ;return to BASIC
QPLen Endp
```

QPLen will always be faster than BASIC's LEN, however it is slower when using BASIC PDS far strings due to the additional overhead of accessing far data. Also, QPLen will be slower with either compiler when used with string expressions such as QPLen%(A$ + B$). QPLEN is not appropriate for determining the length of a TYPE variable.

QPSadd

assembler function contained in PRO.LIB

Purpose:

QPSadd serves the same purpose as BASIC's SADD function, but it is considerably faster and more compact.

Syntax:

```
Address = QPSadd%(Work$)
```

Where:

Work$ is any conventional (not fixed-length) string, and Address receives the starting address of its characters in near memory.

Comments:

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

Also see the related functions QPLen, ASCII, and GetDS.

QPSadd will always be faster than BASIC's SADD, however it is slower when using BASIC PDS far strings due to the additional overhead of accessing far data. Also, QPSadd will be slower with either compiler when used with string expressions such as QPSadd%(A$ + B$).

QPStrI and QPStrL

assembler functions contained in PRO.LIB

Purpose:

QPStrI and QPStrL serve the same purpose as BASIC's STR$() function, but they are considerably faster. QPStrI is intended for use with integer variables, and QPStrL is meant for long integers.

Syntax:

```
X$ = QPStrI$(IntVar%)
```

or

```
X$ = QPStrL$(LongInt&)
```

Where:

IntVar% and LongInt& are the numeric variables being converted to string form.

Comments:

Because QPStrI and QPStrL have been designed as functions, they must be declared before they may be used.

Either of these routines will be much faster than the equivalent QuickBASIC functions because each has been optimized to work with only one type of variable. QuickBASIC's STR$() function must be able to convert any numeric variable type, which of course adds to the complexity of that routine.

Unlike BASIC's STR$ function, QPStrI and QPStrL do *not* add the annoying leading blank on positive values.

Also see the related functions QPValI and QPValL which replace QuickBASIC's VAL() function for integers and long integers respectively.

QPTrim, QPLTrim, and QPRTrim

assembler functions contained in PRO.LIB

Purpose:

QPLTrim and QPRTrim serve the same purpose as BASIC's LTRIM$() and RTRIM$() functions, but they are faster in most situations. QPTrim trims both leading and trailing blanks in one operation. All three of these routines remove CHR$(0) characters as well as normal CHR$(32) blanks.

Syntax:

```
SubString$ = QPTrim$(Work$)
```

or

```
SubString$ = QPLTrim$(Work$)
```

or

```
SubString$ = QPRTrim$(Work$)
```

Where:

SubString$ receives the trimmed version of Work$.

Comments:

Because these routines have been designed as functions, they must be declared before they may be used.

These functions are provided as fast replacements for the built-in equivalent functions in QuickBASIC. In many situations they will be faster than QuickBASIC, however the exact difference depends on how they are being used. When both leading and trailing blanks must be removed, QPTrim will be much faster in all situations.

Recognizing CHR$(0) is particularly important in QuickBASIC 4 if fixed-length strings are being used. When QuickBASIC initializes a fixed-length string, it is filled with zero bytes rather than CHR$(32) blanks.

QPValI and QPValL

assembler functions contained in PRO.LIB

Purpose:

QPValI and QPValL serve the same purpose as BASIC's VAL() function, but they are considerably faster. QPValI is intended for use with strings whose values range from -32768 to 32767, and QPValL is meant for use with values that fall within the range accommodated by long integers.

Syntax:

```
X = QPValI%("12345")
```

or

```
X = QPValL&("123456789")
```

Where:

X receives the value of the specified strings. Of course, you would probably use a string variable rather than a quoted constant, and the examples are shown this way merely for clarity.

Comments:

Because QPValI and QPValL have been designed as functions, they must be declared before they may be used.

Either of these routines will be much faster than the equivalent QuickBASIC functions because each has been optimized to work with only one type of variable. QuickBASIC's VAL() function must be able to process a wide range of values, which of course adds to the complexity of that routine.

Also see the related functions QPStrI and QPStrL which replace QuickBASIC's STR$() function for integers and long integers respectively.

RemCtrl

assembler subroutine contained in PRO.LIB

Purpose:

RemCtrl quickly scans through a given string, and replaces any control characters with a specified new character.

Syntax:

```
CALL RemCtrl(X$, Replace$)
```

Where:

X$ is the string to be processed, and the first character in Replace$ is used as a replacement. The replacement may also be specified to be a blank space by using a null string for Replace$.

Comments:

Embedded control characters have long been a problem for programmers, because so many text files contain them, and so many printers can't handle them. A control character is any character that has an ASCII value less than 32.

Control characters are often used in communications programs to tell the modem at the other end when to start or stop transmission. Also, many word processors use various control characters to indicate margin settings and other formatting information.

RemCtrl provides a clean way to remove those characters from a string that will be printed, or displayed on the screen. Remember, BASIC also interprets certain control characters, for example, a CHR$(29) backs up the cursor one column, and a CHR$(12) clears the screen.

Even though RemCtrl merely substitutes a new character and doesn't truly remove the control characters, that would be easy to do as well. Simply specify an unlikely replacement such as CHR$(0) or CHR$(255), and then use the ReplaceString function to replace that with a null string.

Also see the related routines LowASCII and Translate.

ReplaceChar and ReplaceChar2

assembler subroutines in PRO.LIB

Purpose:

ReplaceChar replaces all occurrences of a specified character in a string with a different character. ReplaceChar2 is similar, but when searching it ignores capitalization.

Syntax:

```
CALL ReplaceChar(Source$, Old$, New$)
```

Where:

Source$ is the string in which the characters are being replaced, Old$ is the character to replace, and New$ is what it is to be replaced with.

Comments:

These routines are intended to replace single characters only. They will of course replace more than one occurrence of the character automatically, but only the first letter in Old$ and New$ is examined.

Also see the ReplaceString function contained in REPLACE.BAS which will replace all occurrences of a string of any size with a different string also of any size.

ReplaceCharT and ReplaceCharT2

assembler subroutines contained in PRO.LIB

Purpose:

ReplaceCharT and ReplaceCharT2 are alternate versions of ReplaceChar and ReplaceChar2 that replace all occurrences of one character in a string with another. Where ReplaceChar and ReplaceChar2 are meant for use with conventional strings, these new versions are meant for use with TYPE variables, fixed-length strings, or indeed, any arbitrary block of memory up to 64K (65,535 bytes) long.

Syntax:

```
CALL ReplaceCharT[2](SEG TypeVar AS ANY, BYVAL TypeLength%, _
  BYVAL OldChar%, BYVAL NewChar%)
```

Where:

TypeVar is either a single TYPE variable, fixed-length string, or an array element. TypeLength% is the number of bytes to process, and OldChar% and NewChar% are the ASCII values of the character to replace and the new replacement respectively. That is, to replace all CHR$(0) bytes with CHR$(32) spaces you would use 0 for OldChar% and 32 for NewChar%.

Comments:

We needed these routines for an in-house project that read someone else's database into a TYPE variable, and all of the trailing characters in each field were filled with CHR$(0) zero bytes instead of CHR$(32) spaces which we needed to remove with RTRIM$.

Note that TypeLength% can be specified as LEN(TypeVar), which eliminates having to change the length value if you later make changes to TypeVar.

ReplaceCharT searches for an exact match on CHR$(OldChar%), and replaces each occurrence with CHR$(NewChar%). ReplaceCharT2 ignores capitalization both when searching and replacing, using upper-case versions. If you will be replacing zero bytes or other control characters, using ReplaceCharT is slightly faster and smaller than ReplaceCharT2 because it avoids the extra testing for capitalization.

As with all of the other QuickPak Professional routines that accept a SEG argument, you can also replace the single SEG parameter with an arbitrary segment and address by passing both as integers using BYVAL:

```
DECLARE SUB ReplaceCharT(BYVAL Segment%, BYVAL Address%, _
  BYVAL NumBytes%, BYVAL Old%, BYVAL New%)
```

You can also process multiple elements in an array at once by faking the length parameter. For example, if you have a 100 element array and each element is, say, 200 bytes long, you can use 20000 for the length. If the length is larger than 32767 (but less than 65536) you must use an equivalent negative value or a long integer.

If you plan to use these routines with fixed-length strings, please see the section "Calling with segments" in the manual.

ReplaceString

BASIC subprogram contained in REPLACE.BAS

Purpose:

ReplaceString replaces all occurrences of a specified string with a different string. Both the string being searched for and its replacement may be any length.

Syntax:

```
CALL ReplaceString(Source$, Old$, New$)
```

Where:

Source$ is the string in which the substring is being replaced, Old$ is the string to be replaced, and New$ is what it is to be replaced with.

Comments:

This routine is written in BASIC because an assembler routine cannot change the length of a BASIC string. That would of course be necessary, unless both Old$ and New$ were the exact same length.

As with the QuickPak Professional functions, ReplaceString is intended to be added to your programs by copying it from the REPLACE.BAS file.

Also see the ReplaceChar and ReplaceChar2 routines which will replace single characters in a string at assembler speed.

ReplaceTbl

assembler subroutine contained in PRO.LIB

Purpose:

ReplaceTbl lets you easily replace all occurrences of one character with any other character using a lookup table.

Syntax:

```
CALL ReplaceTbl(Work$, Table$)
```

Where:

Work$ is the string to be examined, and Table$ is a table of character pairs (see below).

Comments:

ReplaceTbl is modeled after UpperTbl except it does not modify any characters that are not found in the table (UpperTbl also capitalized the string).

Table$ is set up using pairs of characters--the first character is the one to replace and the second is the replacement. The third is the next character to replace, and so forth. The following example replaces the digits "1" through "5" with the letters "A" through "E".

```
Work$ = "Testing 1,2,3,4,5"
CALL ReplaceTbl(Work$, "1A2B3C4D5E")
PRINT Work$                          'displays "Testing A,B,C,D,E"
```

Sequence

assembler subroutine contained in PRO.LIB

Purpose:

Sequence will increment the characters in a string.

Syntax:

```
CALL Sequence(Work$)
```

Where:

Work$ is the string to be sequenced.

Comments:

It is common in database applications to utilize sequenced fields, perhaps to assign account or part numbers automatically. For example, if the last customer account number on record is, say, “AAA-0134-52”, Sequence will increment it to be “AAA-0134-53”. Likewise, if the string were “009-AABD-99”, Sequence would change it to “009-AABE-00”.

Sequence will increment the characters in any of the following three categories — digits, upper case letters, and lower case letters. Imbedded delimiters are unaffected, and "wrapping" will extend across any blanks or non-numeric/ alphabetic characters.

Sequence requires that the string be initially “seeded” with a starting value such as “000000001”. When all available digit positions have been exhausted, Sequence will wrap around without indicating an overflow. That is, “999” will be sequenced to “000”. It is therefore up to you to ensure that a sufficient number of character positions be initially set aside.

Sequence is demonstrated in context in the SEQUENCE.BAS example program.

SpellNumber

BASIC function contained in SPELLNUM.BAS

Purpose:

SpellNumber accepts a number in the form of a string such as "12345", and returns a spelled-out English equivalent in the form of "Twelve Thousand Three Hundred Forty Five".

Syntax:

```
English$ = SpellNumber$(STR$(Number))
```

Where:

Number may be any type of variable or value (integer, single precision, and so forth), and English$ receives the spelled-out equivalent.

Comments:

SpellNumber processes the whole number portion of the value only. Because you may want to print or display the spelled-out answer as values or as dollar amounts, we have instead shown how to deal with the fractional portion in the demonstration included in SPELLNUM.BAS.

Comments in the demonstration show which lines to remove when loading SPELLNUM.BAS as a module in your own programs.

Translate

assembler subroutine contained in PRO.LIB

Purpose:

Translate will replace any occurrence of an extended "box drawing" character with an appropriate equivalent normal ASCII character.

Syntax:

```
CALL Translate(X$)
```

Where:

X$ is the string to be processed.

Comments:

The IBM standard extended character set is useful for drawing boxes and rule lines on the display screen; however, many printers simply cannot print them. Also, early CGA adapters don't contain the font definitions for characters beyond 127.

Translate quickly searches through a specified string, and replaces the characters commonly used for lines and corners with an appropriate equivalent. For example, the CHR$(196) horizontal line is replaced with a hyphen (—), and a corner is replaced with a plus sign (+).

Any character that couldn't possible be considered a "box" character is replaced with a plus sign.

Upper

assembler subroutine contained in PRO.LIB

Purpose:

Upper will quickly convert all of the alphabetic characters in a specified string to upper case.

Syntax:

```
CALL Upper(X$)
```

Where:

X$ is the string to be processed.

Comments:

Even though QuickBASIC 4 provides a built-in UCASE$() function, Upper is provided because it is considerably faster. One of the problems with UCASE$ (and LCASE$) is that they must be used as a function. That is, in order to convert a string it must be reassigned which takes time:

```
X$ = UCASE$(X$)
```

Upper instead scans through the named string very quickly, converting only the lower case alphabetic characters.

Also see the related routine Lower.

UpperTbl

assembler subroutine contained in PRO.LIB

Purpose:

UpperTbl will capitalize all of the characters in a string, and it also looks in a supplied table to determine how to handle foreign characters.

Syntax:

```
CALL UpperTbl(Work$, Table$)
```

Where:

Work$ is the string to be capitalized, and Table$ holds one or more pairs of lower and upper case extended characters.

Comments:

UpperTbl was written to accommodate our European friends, although it could also be used as a general purpose table-driven character substitution routine. The table string comprises pairs of characters, with the first in each pair being the lower case version, and the second its upper case counterpart. Even though we could have hard-coded the conversions into the routine, it would need to be changed for different languages. For example, the character conversions that are used for German are not necessarily the same as those for other languages. A typical table would be assigned like this:

```
Table$ = "äÄöÖüÜñÑ"
```

Notice that normal characters are converted to upper case regardless of the table, and Table$ is needed solely to accommodate the additional characters.

Also see the related routine LowerTbl, which converts a string to lower case.

Chapter 9
Video Routines

APrint

assembler subroutine contained in PRO.LIB

Purpose:

APrint will quickly print any portion of a conventional (not fixed-length) string array, and contain the display within a specified area of the screen.

Syntax:

```
CALL APrint(BYVAL VARPTR(Array$(First)), NumEls%, FirstChar%, _
     NumChars%, Colr%, Page%)
```

Where:

Array$(First) is the first element to be included, and NumEls% is the total number of elements to print.

FirstChar% is the first character in each string to be displayed, and NumChars% is the number of characters to print.

Colr% specifies the display color, coded in the format used by QuickPak Professional. If -1 is used for Colr%, the current screen colors will be maintained.

Page% indicates the video page to write to, which is relevant only when a color monitor is installed. If Page% is set to -1, APrint will use the current page.

Printing always begins at the current cursor location.

Comments:

Using APrint to display part of an array is similar to using MID$ to print part of a string. Besides specifying which character to begin printing with and how many, you also give APrint a starting element and number of elements.

This approach greatly simplifies construction of a "Browse" facility, as shown in the APRINT.BAS demonstration program. Before APrint, the only reasonable way to browse an entire array was to print the first screen, and then use a scroll routine to move it around. But the scrolling approach is extremely cumbersome at best.

If the operator presses the up arrow, you must first scroll the screen down a line, and then display the previous element at the top of the screen. And if the new top line is not as long as what had been there before it, any remnants from the earlier string would have to be cleared.

The problem gets even worse when the array must be moved left or right. Not only do you have to scroll the screen, but you must also write new characters down the left or right edge of the screen from each element.

APrint provides a far superior method, by letting you specify the element and character to be positioned in the upper left corner. Then to display another portion of the array, simply give APrint the new starting values and do it again.

Where other methods require you to think in terms of "virtual screens" or work with multiple arrays, APrint instead creates a window into the array, letting you easily view any portion.

Besides the ability to quickly display all or part of a string array, APrint also accommodates any text screen mode automatically. All of the internal calculations it performs to determine screen memory addresses are adjusted to work in either 40 or 80 columns, and 25, 43, or 50 lines.

The example below shows how simple it is to create a complete browse routine in QuickBASIC 4, not counting the additional code needed to open and read the file.

```
DEFINT A-Z
FirstEl = 1                             'start at first element
FirstChar = 1                           'and first character
DO
   LOCATE 1, 1
   CALL APrint(BYVAL VARPTR(Array$(FirstEl)), 25, _
        FirstChar, 80, 7, -1)

   DO
        X$ = INKEY$                     'get a key
        IF X$ = CHR$(27) THEN END       'end if Escape
   LOOP UNTIL LEN(X$) = 2               'wait for extended key

   SELECT CASE ASC(RIGHT$(X$, 1))
        CASE 80                         'down arrow
            FirstEl = FirstEl + 1

        CASE 72                         'up arrow
            IF FirstEl > 1 THEN FirstEl = FirstEl - 1

        CASE 75                         'left arrow
            IF FirstChar > 1 THEN FirstChar = FirstChar - 1

        CASE 77                          'right arrow
            FirstChar = FirstChar + 1

        CASE ELSE
   END SELECT
LOOP
```

See the COLORS.BAS description for more information about combining foreground and background colors to a single byte. Also see the related routines APrintØ, APrintT, and APrintTØ.

APrintØ

assembler subroutine contained in PRO.LIB

Purpose:

APrintØ will quickly print any portion of a conventional (not fixed-length) string array, and contain the display within a specified area of the screen.

Syntax:

```
CALL APrint0(BYVAL VARPTR(Array$(First)), NumEls%, FirstChar%, _
     NumChars%, Colr%)
```

Where:

Array$(First) is the first element to be included, and NumEls% is the total number of elements to print.

FirstChar% is the first character in each string to be displayed, and NumChars% is the number of characters to print.

Colr% specifies the display color, coded in the format used by QuickPak Professional. If -1 is used for Colr%, the current screen colors will be maintained.

Printing always begins at the current cursor location.

Comments:

APrintØ is nearly identical to APrint, except it prints on text page zero only. Because many programs do not need the ability to print on multiple pages, the additional code to accommodate that feature has been omitted. However, APrintØ recognizes all of the text modes automatically.

A description of how array printing is useful in a program is given in the discussion that accompanies the APrint routine.

Also see the related routines APrint, APrintT, APrintTØ and COLORS.BAS.

APrintT

assembler subroutine contained in PRO.LIB

Purpose:

APrintT will quickly print any portion of a fixed-length string array or the string component of a TYPE array, and contain the display within a specified area of the screen.

Syntax:

```
CALL APrintT(BYVAL VARSEG(Array$(First)), _
    BYVAL VARPTR(Array$(First)), ElSize%, NumEls%, _
    FirstChar%, NumChars%, Colr%, Page%)
```

Where:

Array$(First) is the first element to be included, and ElSize% is the total length in bytes of each element.

NumEls% is the total number of elements to print, FirstChar% is the first character in each string to be displayed, and NumChars% is the number of characters to print.

Colr% specifies the display color, coded in the format used by QuickPak Professional. If -1 is used for Colr%, the current screen colors will be maintained.

Page% indicates the video page to write to, which is relevant only when a color monitor is installed. If Page% is set to -1, APrintT will use the current page.

Printing always begins at the current cursor location.

Comments:

APrintT is nearly identical to APrint, except it is intended for use with fixed-length string arrays.

Comments in the APRINTT.BAS demonstration program show how to call APrintT without requiring all of the BYVAL/VARSEG/VARPTR nonsense. That topic is also discussed in depth in the appendix under the heading "Calling With Segments".

A description of how array printing is useful in a program is given in the discussion that accompanies the APrint routine.

See the COLORS.BAS description for more information about combining foreground and background colors to a single byte. Also see the related routines APrint, APrintØ, and APrintTØ.

APrintTØ

assembler subroutine contained in PRO.LIB

Purpose:

APrintTØ will quickly print any portion of a fixed-length string array or the string component of a TYPE array, and contain the display within a specified area of the screen.

Syntax:

```
CALL APrintT0(BYVAL VARSEG(Array$(First)), _
     BYVAL VARPTR(Array$(First)), ElSize%, NumEls%, _
     FirstChar%, NumChars%, Colr%)
```

Where:

Array$(First) is the first element to be included, and ElSize% is the total length in bytes of each element.

NumEls% is the total number of elements to print, FirstChar% is the first character in each string to be displayed, and NumChars% is the number of characters to print.

Colr% specifies the display color, coded in the format used by QuickPak Professional. If -1 is used for Colr%, the current screen colors will be maintained.

Printing always begins at the current cursor location.

Comments:

APrintTØ is nearly identical to APrintT, except it prints on text page zero only. Because many programs do not need the ability to print on multiple pages, the additional code to accommodate that feature has been omitted. However, APrintTØ recognizes all of the text modes automatically.

A description of how array printing is useful in a program is given in the discussion that accompanies the APrint routine.

AprintTØ may also be called without requiring BYVAL VARSEG and BYVAL VARPTR, as explained in the tutorial entitled "Calling With Segments".

See the COLORS.BAS description for more information about combining foreground and background colors to a single byte. Also see the related routines APrint, APrintØ, and APrintT.

ArraySize

assembler function contained in PRO.LIB

Purpose:

ArraySize will quickly calculate how many elements are needed in an integer array that is intended to hold a portion of the display screen.

Syntax:

```
Size = ArraySize%(ULRow%, ULCol%, LRRow%, LRCol%)
```

Where:

ULRow%, ULCol%, LRRow%, and LRCol% describe the area of the screen to be saved, and Size receives the number of elements that are needed.

Comments:

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

ArraySize is meant to be used in conjunction with ScrnSave and ScrnRest. Those routines have been designed to store a screen in an integer array, thus allowing you to save as many screens or parts of a screen as necessary. However, the size to which the arrays must be dimensioned depends on how large a portion of the screen is being saved.

ArraySize accepts the window boundaries, and then returns the number of elements needed to hold it. The calculations add one extra element to accommodate the use of OPTION BASE 1 in your programs.

The example below shows the minimum steps necessary to save the portion of the screen from 1,1 to 10,80.

```
DIM ScrnArray%(ArraySize%(1, 1, 10, 80))
CALL ScrnSave(1, 1, 10, 80, SEG ScrnArray%(0), -1)
```

Of course, you could also calculate the array size manually. This would make sense if the size of the screen being saved is always the same. For example, an entire screen (with 80 columns by 25 rows) requires 2000 elements.

The equivalent formula is:

```
ArraySize% = (LRRow - ULRow + 1) * (LRCol - ULCol + 1)
```

BlinkOff and BlinkOn

assembler subroutines contained in PRO.LIB

Purpose:

BlinkOff and BlinkOn let you exchange foreground color flashing for a high-intensity background color on EGA and VGA monitors.

Syntax:

```
CALL BlinkOff
CALL BlinkOn
```

Where:

Calling BlinkOff disables foreground color flashing, and calling BlinkOn re-enables flashing.

Comments:

Once BlinkOff has been called, specifying a foreground color greater than 16 will instead set the background color to high intensity. Calling BlinkOn resets the default flashing mode.

These routines are intended for use with EGA and VGA monitors only, and they work by setting the EGA/VGA palette register.

Normal monochrome and CGA monitors can also be set to exhange flashing for a high-intensity background by using OUT statements. To switch a monochrome display to use a high-intensity background use *OUT &H3B8, 9* and to restore flashing use *OUT &H3B8, 29.* To switch a CGA display to use a high-intensity background use *OUT &H3D8, 9* and to restore flashing use *OUT &H3D8, 29.*

Be warned that using these OUT statements with incorrect values has the potential to damage the monitor.

Box

assembler subroutine contained in PRO.LIB

Purpose:

Box will quickly draw a box frame on the screen.

Syntax:

```
CALL Box(ULRow%, ULCol%, LRRow%, LRCol%, Char%,_
    Colr%, Page%)
```

Where:

ULRow%, ULCol%, LRRow%, and LRCol% describe the box boundaries, and Char% indicates the box style (see the table below).

Colr% is the desired box color, coded in the format used by QuickPak Professional. If -1 is used for Colr%, the current screen colors will be maintained.

Page% tells Box which screen page to use, but that is relevant only with a color monitor. If Page% is -1 then the currently active page will be used.

Comments:

Box uses a simple code to indicate the style of box to be drawn. The table below shows what values to use for Char% to indicate the type of box.

1 = single line all around
2 = double line all around
3 = double line horizontally, single line vertically
4 = single line horizontally, double line vertically

If Char% is assigned to any other value, that ASCII character will be used for the entire box.

Also see the related routine BoxØ that writes to page zero only. Box is demonstrated in the DEMOCM.BAS example program.

BoxØ

assembler subroutine contained in PRO.LIB

Purpose:

BoxØ will quickly draw a box frame on the screen.

Syntax:

```
CALL Box0(ULRow%, ULCol%, LRRow%, LRCol%, Char%, Colr%)
```

Where:

ULRow%, ULCol%, LRRow%, and LRCol% describe the box boundaries, and Char% indicates the box style (see the description for the Box routine).

Colr% is the desired box color, coded in the format used by QuickPak Professional. If -1 is used for Colr%, the current screen colors will be maintained.

Comments:

BoxØ is nearly identical to Box, except it draws on text page zero only. Because many programs do not need the ability to use multiple screen pages, the additional code to accommodate that feature has been omitted.

See the description for the Box routine for details on specifying the box drawing characters.

BPrint

assembler subroutine contained in PRO.LIB

Purpose:

BPrint will print either a conventional or fixed-length string at the current cursor position through DOS.

Syntax:

```
CALL BPrint(X$)
```

Where:

X$ is the string to be printed.

Comments:

BPrint is provided with QuickPak Professional as a way to communicate with a PC's BIOS directly. Because QuickBASIC 4 writes directly to screen memory, it is not possible to do this in BASIC.

BPrint will be necessary only if you need to send a special code to the video BIOS, for example to activate a terminate and stay resident utility. With many screen builder programs, the only way you can tell them to display a screen is by sending a special command string through the PC's BIOS.

BPrint is also useful if you intend to send screen codes to ANSI.SYS from within a BASIC program.

Even though BPrint is intended to print only strings, you can easily print numbers using the BASIC STR$ function:

```
CALL BPrint(STR$(Number))
```

ClearEOL

assembler subroutine contained in PRO.LIB

Purpose:

ClearEOL (Clear to End of Line) erases the current screen line starting at the current cursor position.

Syntax:

```
CALL ClearEOL(Colr%)
```

Where:

Colr% specifies the display color, coded in the format used by QuickPak Professional. If -1 is used for Colr%, the current screen colors will be maintained.

Comments:

ClearEOL always operates on whatever screen page is currently active.

When using ClearEOL, understand that the line is cleared by printing a string of blank spaces. Therefore, to clear to a particular color, it is the *background* color that you will be specifying.

The color will be given as a single byte in the format used by QuickPak Professional. See the COLORS.BAS description for more information about combining foreground and background colors to a single byte.

ClearScr

assembler subroutine contained in PRO.LIB

Purpose:

ClearScr will clear all or a portion of the screen to a specified color.

Syntax:

```
CALL ClearScr(ULRow%, ULCol%, LRRow%, LRCol%, Colr%, Page%)
```

Where:

ULRow%, ULCol%, LRRow%, and LRCol% indicate which portion of the screen is to be cleared, and Colr% is the color to clear it to. If -1 is used for Colr%, the current screen colors will be maintained.

Page% indicates the video page to be cleared, which is relevant only when a color monitor is installed. If Page% is set to -1, ClearScr will use the current page.

Comments:

ClearScr is intended primarily for clearing only a portion of the screen, though it could of course be used for the entire screen. The best way to clear the entire screen is to use BASIC's CLS command. To have CLS clear to a different color, simply use the COLOR statement to set that color.

When using ClearScr, understand that the screen is cleared by printing blank spaces. Therefore, to clear to a particular color, it is the *background* color that you will specify.

The color will be given as a single byte in the format used by QuickPak Professional. See the COLORS.BAS description for more information about combining foreground and background colors to a single byte.

ClearScrØ

assembler subroutine contained in PRO.LIB

Purpose:

ClearScrØ will clear all or a portion of the screen to a specified color.

Syntax:

```
CALL ClearScr0(ULRow%, ULCol%, LRRow%, LRCol%, Colr%)
```

Where:

ULRow%, ULCol%, LRRow%, and LRCol% indicate which portion of the screen is to be cleared, and Colr% is the color to clear it to. If -1 is used for Colr%, the current screen colors will be maintained.

Comments:

ClearScrØ is nearly identical to ClearScr, except it clears text page zero only. Because many programs do not need the ability to work with multiple pages, the additional code to accommodate that feature is omitted. However, ClearScrØ recognizes all of the text modes automatically.

Colors

BASIC program contained in COLORS.BAS

Purpose:

Unlike most of the QuickPak Professional programs, Colors is not meant to be used as a callable module. Rather, it simply displays a chart showing all the possible color combinations.

You should run it whenever you need to select an appropriate color for a program, or to determine the color value that is needed for the various video routines. Because most of those routines require a combined foreground and background color variable, Colors is provided to show the correct value.

Once the program is displaying the color chart, you may press Shift-PrtSc to get a permanent copy for your wall.

For your information, the three formulas shown on the following page may be used to convert a foreground and background color into a single byte value. Three different ones are given because the ones that are simpler and have less capability also use less code.

The first formula uses a simplified method that does not take a flashing foreground or illegal background into account. If the foreground color is such that it should cause a flashing effect (greater than 15), the flashing will be ignored. And if the background color is illegal (greater than 7) it will actually *cause* the foreground to flash.

The second formula accommodates foreground flashing, but not an illegal background. The third is the most complicated, but it also prevent an illegal background color from causing unintentional flashing. This is the approach we used in the assembler OneColor function.

Doesn't accommodate flashing or trap an illegal background:

```
Colr = FG + 16 * BG
```

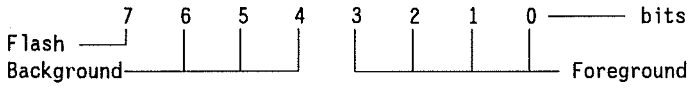
Allows flashing, but doesn't trap an illegal background:

```
Colr = (FG AND 16) * 8 + (BG * 16) + (FG AND 15)
```

Allows flashing and traps illegal background values:

```
Colr = (FG AND 16) * 8 + ((BG AND 7) * 16) + (FG AND 15)
```

The color for any given character on the display is contained in a single byte in screen memory, immediately following the corresponding character byte. It is organized as follows:



It is also fairly simple to extract the individual foreground and background components from a combined color byte:

```
FG = (Colr AND 128) \ 8 + (Colr AND 15)
BG = (Colr AND 112) \ 16
```

CsrSize

assembler subroutine contained in PRO.LIB

Purpose:

CsrSize will report the top and bottom scan lines that describe the current cursor size.

Syntax:

```
CALL CsrSize(Top%, Bottom%)
```

Where:

Top% and Bottom% are returned holding the top and bottom scan lines of the current text-mode cursor setting.

Comments:

In most programming situations your program will know the current cursor size because you set it yourself using the LOCATE command. The example below establishes a "normal" cursor size on a color monitor:

```
LOCATE , , , 6, 7
```

Thus, CsrSize would return 6 and 7 for Top% and Bottom%, respectively. But when you are creating re-usable modules that must restore the cursor to its original size when it is finished, CsrSize is the only way to know the correct values.

Notice that when the cursor is turned off (using LOCATE , , 0), the Top% parameter will be returned with a value greater than 30. This is because most programs hide the cursor by setting the top line to an otherwise illegal value. In our tests on EGA, VGA, CGA and Hercules display adapters, Top% was returned set to 39 when the cursor was off.

EGABLoad

BASIC subprogram contained in EGABSAVE.BAS

Purpose:

EGABLoad will load an EGA or VGA graphics screen from a specified disk file.

Syntax:

```
CALL EGABLoad(FileName$)
```

Where:

FileName$ is the name of a file that has previously been saved with the QuickPak Professional EGABSave routine.

Comments:

EGABLoad and its companion EGABSave are intended to be used with either an EGA or VGA display in SCREEN 9. Comments in the source code for these programs show how to accommodate the added resolution used by a VGA in SCREEN 12.

The file name you specify must not contain an extension, and if one is given it will be removed. The programs actually require four files to be saved, with one each for the Red, Green, Blue, and intensity (brightness) information.

A complete discussion of the concepts behind saving and loading graphics screen images is given in the Tutorial section of this manual.

An example of EGABLoad is given in the EGABSAVE.BAS demonstration program.

EGABSave

BASIC subprogram contained in EGABLOAD.BAS

Purpose:

EGABSave will save an EGA or VGA graphics screen to a specified disk file.

Syntax:

```
CALL EGABSave(FileName$)
```

Where:

FileName$ is the name of a file to use when saving the screen image to disk.

Comments:

EGABSave and its companion EGABLoad are intended to be used with either an EGA or VGA display in SCREEN 9. Comments in the source code for these programs show how to accommodate the added resolution used by a VGA in SCREEN 12.

The file name you specify must not contain an extension, and if one is given it will be removed. The programs actually require four files to be saved, with one each for the Red, Green, Blue, and intensity (brightness) information.

A complete discussion of the concepts behind saving and loading graphics screen images is given in the Tutorial section of this manual.

An example of EGABSave is given in the EGABSAVE.BAS demonstration program.

EGAMem

assembler function contained in PRO.LIB

Purpose:

EGAMem will report the amount of memory available on an EGA display adapter.

Syntax:

```
Memory = EGAMem%
```

Where:

Memory% receives the number of 64K banks installed on the display adapter. If an EGA is not installed, Memory instead receives 0.

Comments:

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

Most of the EGA adapters sold today contain the full 256K required to hold up to two screens with the full complement of colors. However, many PC's still have the original IBM brand of adapter with only 64K of memory.

The section in the QuickBASIC manual that describes the SCREEN command clearly shows which colors and screen pages may be used with an EGA, based on the amount of installed memory.

FillScrn

assembler subroutine contained in PRO.LIB

Purpose:

FillScrn will fill any rectangular portion of the display screen with the specified character.

Syntax:

```
CALL FillScrn(ULRow%, ULCol%, LRRow%, LRCol%, Colr%, Char%, Page%)
```

Where:

ULRow%, ULCol%, LRRow%, and LRCol% describe the area to be filled, and Colr% specifies which color to use. If Colr% is set to -1, the current screen colors will be honored.

Char% is the ASCII value of the character to print, and Page% indicates the video page to write to. If Page% is set to -1, then the currently active page will be used.

Comments:

FillScrn is similar to the QuickPak Professional ClearScr routine, except it lets you specify the character rather than using a blank space.

FillScrn can be used effectively to create interesting background patterns and textures in text mode using CHR$(176), CHR$(177), or CHR$(178). Of course, you
may use any other character as well.

FillScrnØ

assembler subroutine contained in PRO.LIB

Purpose:

FillScrnØ will fill any rectangular portion of the display screen with the specified character.

Syntax:

```
CALL FillScrn0(ULRow%, ULCol%, LRRow%, LRCol%, Colr%, Char%)
```

Where:

ULRow%, ULCol%, LRRow%, and LRCol% describe the area to be filled, Colr% specifies which color to use, and Char% is the ASCII value of the character to print. If Colr% is set to -1, the current screen colors will be honored.

Comments:

FillScrnØ is nearly identical to FillScrn, except it writes to text page zero only. Because many programs do not need the ability to access multiple video pages, the additional code to accommodate that feature has been omitted. However, FillScrnØ recognizes all of the text modes automatically.

GetColor

BASIC subprogram contained in GETCOLOR.BAS

Purpose:

GetColor will return BASIC's currently active foreground and background colors.

Syntax:

```
CALL GetColor(FG%, BG%)
```

Where:

FG% and BG% are the current foreground and background colors that BASIC is using for PRINT statements.

Comments:

When a subprogram needs to use the COLOR statement prior to printing, one serious drawback is that there is no way to know what those colors had originally been. When creating reusable modules that will be added to many programs, this prevents the subprogram from being able to "clean up" after itself completely.

Of course, this is one of the main reasons for using QPrint, APrint, and the other QuickPak Professional routines that accept a color argument. Even though BASIC provides a way to read the current cursor location with CSRLIN and POS(0), there is no such equivalent to obtain the colors.

The method used by GetColor is admittedly clunky, especially when you consider that the color information must be stored *somewhere* in memory.

GetColor obtains BASIC's current colors by first saving the character and color currently in the upper left corner of the screen. Next, a blank space is printed there, and then the SCREEN function is used to see what color was used. Finally, the original screen contents are restored.

GetVMode

assembler subroutine contained in PRO.LIB

Purpose:

GetVMode will report the current video mode, the currently active display page, the page size, and the number of rows and columns.

Syntax:

```
CALL GetVMode(Mode%, Page%, PageSize&, Rows%, Columns%)
```

Where:

Mode% is the equivalent BIOS screen mode, and Page% is the currently active page. The first page is Page 0, and not 1. PageSize& is the number of bytes of display memory being used to hold the current screen. Rows% and Columns% also indicate the size of the screen, but in terms of characters.

Comments:

GetVMode is useful when you are writing program modules that are needed by more than one program. In most cases, you will already know the current screen mode since your program would have previously set it. But if a subprogram may be called at various times by different programs, this information will be invaluable.

GetVMode is also used to advantage in the QuickDOS utility, to know if the PC is already in a 43 or 50 line mode.

The PageSize& variable is equally useful to determine the number of bytes that must be saved or loaded when storing screen images in a disk file. Saving and loading screen images is described in detail in the tutorial section of this manual.

HCopy

assembler subroutine contained in PRO.LIB

Purpose:

HCopy performs the same action as BASIC's PCOPY command, except it is designed to work with a Hercules or compatible display adapter in text mode.

Syntax:

```
CALL HCopy(FromPage%, ToPage%)
```

Where:

FromPage% is the source page to be copied, and ToPage% is the destination page to copy to.

Comments:

A Hercules display supports eight separate text pages, numbered from 0 to 7. However, QuickBASIC treats a Hercules display adapter operating in text mode as if it were an ordinary monochrome adapter. Therefore, attempting to use PCOPY will result in an "Illegal Function Call" error.

Even though QuickBASIC will display only text page zero, the adapter's memory may be used to store additional pages. For example, to display, say, page three, you could use HCopy to save page zero in an unused page, and then call it again to copy page three down to page zero.

HercThere

assembler function contained in PRO.LIB

Purpose:

HercThere will report if the QBHERC.COM or MSHERC.COM Hercules graphic support program has been loaded into memory.

Syntax:

```
Loaded = HercThere%
```

Where:

Loaded will be set to -1 if QBHERC.COM or MSHERC.COM program is resident, or 0 if it is not.

Comments:

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

Even though QuickBASIC supports graphics using a Hercules display adapter, a special TSR (terminate and stay resident) program must be run first. If this is not done, attempting to use the SCREEN 3 statement to enter graphics mode will cause an "Illegal Function Call" error.

The QuickPak Professional Monitor function will tell you if a Hercules display is installed in the host PC, but it does not detect if the necessary support program has been loaded. This is what HercThere is for.

QuickBASIC 4.0 comes with a program named QBHERC.COM that contains the routines necessary for Hercules graphics, but it was renamed to MSHERC.COM when QuickBASIC 4.5 was introduced. HercThere will detect if either QBHERC.COM or MSHERC.COM is loaded.

The following example shows one way you might use HercThere:

```
M = Monitor%
IF M = 2 THEN                       'Monitor% returns 2 for
                                    'a Hercules card
   IF NOT HercThere% THEN
        PRINT "Please run QBHERC.COM and start again."
   ELSE
        SCREEN 3
   END IF
END IF
```

You could also use the QuickPak Professional StuffBuf routine to create a batch file that runs QBHERC automatically, and then re-runs your program.

MakeMono

assembler subroutine contained in PRO.LIB

Purpose:

MakeMono will convert the colors on a text screen held in an integer array to those suitable for display on a monochrome monitor.

Syntax:

```
CALL MakeMono(SEG Array%(Element), Size%)
```

Where:

Array%(Element) is the location in the array where the screen begins, and Size% is the size of the screen in characters.

Comments:

One of the problems when designing screens to be displayed in a program is deciding what colors should be used. If you know the screen will be shown on a monochrome monitor, then you would avoid using a colored background unless the text foreground color is black (inverse colors). And if you are sure the screen will be displayed only on a color computer, then any color combination will be valid. However, the real problem is when the screen must be acceptable on either display.

MakeMono is intended to be used with screens that are kept in an array. See the discussion that accompanies the ScrnRest routine for an explanation of loading screens from disk into an array prior to displaying them.

MakeMono allows you to design a screen to be as attractive as possible in color, but be able to quickly modify it if your program detects a monochrome monitor. A simple algorithm is used by MakeMono when it convert the colors.

If the text foreground color is black (inverse), then the color is left unchanged. Otherwise the background is cleared to black, and the foreground is set to white. However, the intensity and flashing color attributes are left unchanged.

This provides readable text on *any* monochrome monitor, including the original Compaq portable and the AT&T 6300. Those displays are especially problematic because they look like a CGA to routines that are designed to detect the type of monitor.

MakeMon2

assembler subroutine contained in PRO.LIB

Purpose:

Like the original MakeMono routine, MakeMon2 will convert the colors on a screen held in an integer array to those suitable for display on a monochrome system. However, several additional options are available.

Syntax:

```
CALL MakeMon2(SEG Array%(Start), NumEls%, Code%)
```

Where:

Array%(Start) is the element in the array where the screen begins, NumEls% is the number of elements in use, and Code% is a conversion code that indicates how the colors are to be converted.

Comments:

MakeMon2 provides four different types of color conversion, as shown in the table below.

1	All colors are forced to white on black.
2	All colors are forced to black on white.
3	All colors are forced to white on black, unless the foreground is also black. In that case, the foreground is left unchanged and the background is forced to white, to preserve any inverse text. This is how the original MakeMono operates.
4	All colors are forced to black on white, unless the foreground is already white. In that case, the foreground is left unchanged and the background is forced to black. This is the opposite of the way MakeMono works.

See the description accompanying the MakeMono routine for more discussion of when and why this routine should be used.

Monitor

assembler function contained in PRO.LIB

Purpose:

Monitor reports the type of display adapter that is currently active.

Syntax:

```
Mon = Monitor%
```

Where:

Mon receives the current monitor type, coded as shown in the table below.

Comments:

Because Monitor is implemented as a function, it must be declared before it may be used.

Monitor recognizes all of the popular display adapter types, however it does not report which screen mode is currently active. That information is available by using the GetVMode function.

The type of monitor detected will be returned as follows:

1	Monochrome adapter
2	Hercules adapter
3	CGA adapter
4	EGA adapter with a monochrome monitor
5	EGA adapter with a color monitor
6	VGA adapter with a monochrome monitor
7	VGA adapter with a color monitor
8	MCGA adapter with a monochrome monitor
9	MCGA adapter with a color monitor
10	EGA adapter with a color CGA monitor
11	IBM 8514/A adapter

Knowing what type of monitor is installed in a system is very important if you intend to use graphics commands in your programs. For example, SCREEN 9 is legal only when an EGA or VGA display is present, and SCREEN 3 works only with Hercules monitors. Attempting to initiate an invalid screen mode will always result in an "Illegal Function Call" error from BASIC. With Monitor you can avoid that possibility by knowing which of the possible screen modes may be used.

Monitor is also helpful in determining appropriate colors for a program. For example, using colored backgrounds may result in unreadable text on a monochrome monitor. Be aware that some computers, such as the original Compaq portable and the AT&T 6300, have a CGA adapter connected to a monochrome monitor. In those cases Monitor will report a CGA. You might consider recognizing a command line switch such as /B to allow a user to override the program's assumptions.

MPaintBox

assembler subroutine contained in PRO.LIB

Purpose:

MPaintBox is similar to PaintBox, except it always turns off the mouse cursor before painting the screen. When it is finished the cursor is turned back on.

Syntax:

```
CALL MPaintBox(ULRow%, ULCol%, LRRow%, LRCol%, Colr%)
```

Where:

ULRow%, ULCol%, LRRow%, and LRCol% describe the area of the screen to be painted, and Colr% specifies the color to use.

Comments:

MPaintBox is used by the PullDown and VertMenu programs to allow them to cooperate with a mouse. It operates almost identically to the original PaintBox program, except the mouse cursor is turned off while the screen is being painted. If this is not done, the screen color at the mouse cursor location is destroyed when the mouse is subsequently moved.

MPaintBox always paints on the currently active screen.

MPRestore

assembler subroutine contained in PRO.LIB

Purpose:

MPRestore lets you redisplay any rectangular portion from a screen that has been saved to an array using ScrnSave or ScrnSaveØ.

Syntax:

```
CALL MPRestore(ULRow%, ULCol%, LRRow%, LRCol%, OriginalWidth%, _
     SEG Array%(Start))
```

Where:

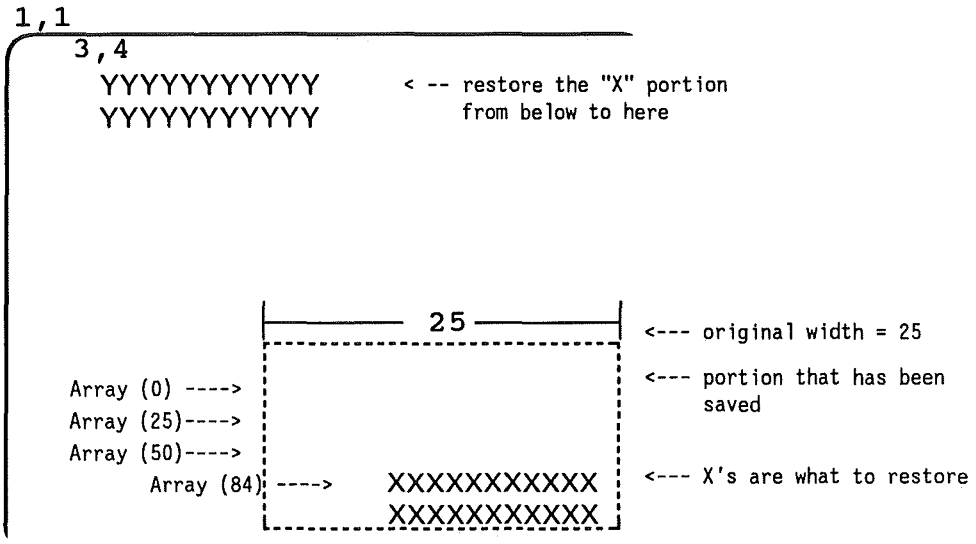
ULRow%, ULCol%, LRRow%, and LRCol% tell where the partial image is to be placed on the screen. OriginalWidth% is how wide (in characters) the original saved portion of the screen was, and Array%(Start) is where in the array the desired portion begins.

Comments:

MPRestore is a fairly specialized routine, but if you need it at all, you *really* need it. Where the normal ScrnRest routine can restore only the same size screen that had been saved earlier (though it may be restored anywhere), MPRestore lets you capture and redisplay just a rectangular portion from the storage array. We needed MPRestore to be able to re-size the editing screen in QEdit, though it undoubtedly has other uses as well.

We call it MPRestore because the Mouse cursor is disabled like MQPrint and the other "M" routines, and it lets you restore a Partial screen.

The figure below shows a representation of a 25 x 80 screen, a portion that had previously been saved with ScrnSave starting at Array%(0), and how to determine the correct starting element in Array%().



```
CALL MPRestore(3, 4, 4, 15, 25, SEG(Array%(84))
```

MQPrint

assembler subroutine contained in PRO.LIB

Purpose:

MQPrint is similar to QPrint, except it always turns off the mouse cursor before printing. When it is finished the cursor is turned back on.

Syntax:

```
CALL MQPrint(X$, Colr%)
```

Where:

X$ is the string to be printed, and Colr% specifies the color to use. If Colr% is set to -1, the current screen colors will be honored

Comments:

MQPrint is used by the PullDown and VertMenu programs to allow them to cooperate with a mouse. It operates almost identically to the original QPrint program, except the mouse cursor is turned off while the string is being displayed. If this is not done, the screen color at the mouse cursor location is destroyed when the mouse is subsequently moved.

MQPrint always prints on the currently active screen page.

MScrnSave and MScrnRest

assembler subroutines in PRO.LIB

Purpose:

MScrnSave and MScrnRest are similar to ScrnSave and ScrnRest, except they always turn off the mouse cursor while they are working.

Syntax:

```
CALL MScrnSave(ULRow%, ULCol%, LRRow%, LRCol%, SEG A%(0))
```

or

```
CALL MScrnRest(ULRow%, ULCol%, LRRow%, LRCol%, SEG A%(0))
```

Where:

ULRow%, ULCol%, LRRow%, and LRCol% describe the area of the screen to consider, and A%() is an integer array that is used to hold the portion of the screen.

Comments:

MScrnSave and MScrnRest are used by the PullDown and VertMenu programs to allow them to cooperate with a mouse. They are nearly identical to the original ScrnSave and ScrnRest routines, except they turn off the mouse cursor while the screen is being accessed. If this is not done, the screen color at the mouse cursor location is destroyed when the mouse is subsequently moved.

MScrnSave and MScrnRest always work with the currently active screen page.

OneColor

assembler function contained in PRO.LIB

Purpose:

OneColor accepts separate foreground and background color values, and returns them combined in a single byte for use with the QuickPak Professional video routines.

Syntax:

```
Colr = OneColor%(FG%, BG%)
```

Where:

FG% and BG% are the intended foreground and background colors, and Colr% receives the combined value.

Comments:

Because OneColor is implemented as a function, it must be declared before it may be used.

All of the QuickPak Professional video routines expect a single value to specify both the foreground and background colors. The colors are in fact stored this way by the PC's hardware, and providing them in this format allows the video routines to operate that much faster. Further, this saves variable memory in your programs by eliminating an extra parameter.

The PC's hardware uses a convoluted method to combine the foreground and background components of a color, and OneColor will save you that much additional code and effort.

The formula used by OneColor is:

```
Colr = (FG AND 16) * 8 + ((BG AND 7) * 16) + (FG AND 15)
```

A description of the color byte and how each bit affects the combined color is given in the discussion that accompanies the COLORS.BAS program.

PaintBox

assembler subroutine contained in PRO.LIB

Purpose:

PaintBox will paint any rectangular area of the screen, without disturbing the text that is already present.

Syntax:

```
CALL PaintBox(ULRow%, ULCol%, LRRow%, LRCol%, Colr%, Page%)
```

Where:

ULRow, ULCol%, LRRow%, and LRCol% describe the area of the screen to be painted, Colr% specifies the color to use, and Page% indicates which screen page is to be painted. If Page% is set to -1, then the currently active screen will be painted.

Comments:

PaintBox is useful in a variety of programming situations to highlight an area of a screen that already contains text. Using BASIC alone would require you to first set new colors with the COLOR command, and then print the text again.

We have used PaintBox in many of the BASIC programs that come with QuickPak Professional. For example, the Lotus menus use PaintBox to highlight and "un-highlight" the currently active choice. This saves time and reduces the size of the code by eliminating many COLOR, LOCATE, and PRINT statements.

PaintBox is also used to create the shadowed effect in the various BASIC pop-up utilities. In those cases PRINT statements couldn't possibly be used, because it is the underlying screen that is being shadowed. The subprograms have no way to know what text is already on the screen, thus it would be impossible to use COLOR and PRINT.

PaintBox is also used in QEdit to highlight the block of text as it is being marked and unmarked.

See the COLORS.BAS description for more information about combining foreground and background colors to a single byte. Also see the related routine PaintBoxØ.

PaintBoxØ

assembler subroutine contained in PRO.LIB

Purpose:

PaintBoxØ will paint any rectangular area of the screen, without disturbing the text that is already present.

Syntax:

```
CALL PaintBox0(ULRow%, ULCol%, LRRow%, LRCol%, Colr%)
```

Where:

ULRow, ULCol%, LRRow%, and LRCol% describe the area of the screen to be painted, and Colr% specifies the color to use.

Comments:

PaintBoxØ is nearly identical to PaintBox, except it paints on text page zero only. Because many programs do not need the ability to print on multiple pages, the additional code to accommodate that feature has been omitted. However, PaintBoxØ recognizes all of the text modes automatically.

See the COLORS.BAS description for more information about combining foreground and background colors in a single byte.

PrtSc

assembler subroutine contained in PRO.LIB

Purpose:

PrtSc sends a snapshot of the current text screen to a printer, as if the PrtSc key was pressed.

Syntax:

```
CALL PrtSc(LPTNumber%, Page%)
```

Where:

LPTNumber% is 1, 2, or 3 to indicate which parallel printer to send to, and Page% indicates which text screen page is to be printed.

If Page% is set to -1, then the current screen will be printed.

If a printer error occurs (invalid printer number, off-line, etc.), then LPTNumber% will be returned as -1.

Comments:

Unlike the screen print facility that is built into all PC's, PrtSc lets you send any text page to any parallel printer. Of course, the PrtSc subroutine is meant to be called from your programs so the operator doesn't have to press a key.

PrtSc accommodates all of the possible text screen modes including 40 and 80 columns, and 25, 43, and 50 lines, but it is *not* intended for use in any of the graphics modes. The QuickPak Professional ScrnDump routine is provided for printing graphics screens.

It is also possible to print just a portion of the screen. The example below has nothing to do with PrtSc, and it is provided simply to show how this could be done.

```
ULRow = 5: ULCol = 10             'portion of the screen to print
LRRow = 20: LRCol = 70

FOR X = ULRow TO LRRow
    LPRINT TAB(ULCol);            'optional left margin
    FOR Y = ULCol TO LRCol
          LPRINT CHR$(SCREEN(X, Y));
    NEXT
    LPRINT
NEXT
```

PrtScØ

assembler subroutine contained in PRO.LIB

Purpose:

PrtScØ sends a snapshot of the current text screen to a printer, as if the PrtSc key was pressed.

Syntax:

```
CALL PrtSc0(LPTNumber%)
```

Where:

LPTNumber% is 1, 2, or 3 to indicate which printer to send to.

Comments:

PrtScØ is nearly identical to PrtSc, except it prints from text page zero only. Because many programs do not need the ability to work with multiple pages, the additional code to accommodate that feature has been omitted. However, PrtScØ recognizes all of the text modes automatically.

PUsing

assembler subroutine contained in PRO.LIB

Purpose:

PUsing is provided as a replacement for the BASIC PRINT USING command, and it offers several enhancements.

Syntax:

```
CALL PUsing(STR$(Number), Image$, Color1%, Color2%, Page%, _
     Row%, Column%)
```

Where:

Number is any number whether integer, single, or double precision, and Image$ indicates how the result is to be formatted. Image$ may contain both leading and trailing text as shown below.

Color1% and Color2% indicate the colors for the text and numeric portions of Image$ respectively. Either color may be set to -1 to maintain the current screen colors.

Page% tells PUsing what screen page to print to, which may optionally be given as -1 to use the current page.

Row% and Column% specify where the printing is to be located on the screen.

Comments:

By using the STR$ function, we are letting BASIC do the dirty work of interpreting floating point numbers. This also lets PUsing accept any type of numeric variable. If you do not use BASIC's STR$ function and directly pass a string to PUsing, be sure to add an extra space to the beginning of the number string that you are formatting, i.e., "1234" becomes " 1234". Most of the formatting codes that PRINT USING recognizes are supported, including commas, dollar signs, and leading asterisks.

If the number will not fit within the allotted space, a percent sign is *not* used to indicate an overflow. Instead, any preceding text will be overwritten. The image string you give to PUsing will be in the following format:

```
   Image$ = "Some leading text ####,.## this is trailing"
```

By specifying the color, row and column, you are assured that the current cursor location and colors BASIC is using are not changed. This is especially important when creating reusable modules that will be called from different programs.

Unlike BASIC's PRINT USING, PUsing can accommodate any number of digits. Used in conjunction with exponential notation, PUsing can display extremely large numbers without creating an error.

The table below summarizes PUsing's numeric formatting capabilities:

#	represents each digit position
.	specifies a decimal point
+	causes the sign of the number (+ or –) to be added (the sign must be the first character in the field)
**	replaces leading spaces in the field with asterisks
$$	adds a dollar sign to the left of the number
**$	combines the effects of ** and $$
,	specifies that commas are to be added to the formatted string

Although BASIC accepts multiple commas in the image string, PUsing requires only one. If there is a decimal point within the string, the comma must be placed just before it. Otherwise, the comma must be the last character.

See the COLORS.BAS description for more information about combining foreground and background colors to a single byte.

PUsing is demonstrated in the DEMOCM.BAS example program.

Also see the description for FUsing elsewhere in this manual.

QPrint

assembler subroutine contained in PRO.LIB

Purpose:

QPrint will display a string very quickly at the current cursor location.

Syntax:

```
CALL QPrint(X$, Colr%, Page%)
```

Where:

X$ is the string to be printed, Colr% is the color to use, and Page% indicates which text page to print on.

If Colr% is -1, then the current display colors will be maintained. Likewise, if Page% is -1, the currently active text page is used.

QPrint always prints at the current cursor location.

Comments:

QPrint was originally developed for use with QuickBASIC 2 and 3, because those compilers send their display output through the PC's BIOS. Printing through the BIOS is notoriously slow, and QPrint provides an enormous boost in speed.

However, even with QuickBASIC 4, QPrint is noticeably faster. Further, QPrint accepts *any* characters. Even though the newer QuickBASIC 4 also writes directly to screen memory, it is slower than QPrint because it must examine every character to see if it is a special control code.

For example, a CHR$(9) means to advance the cursor to the next Tab position, CHR$(28) advances the cursor one column, CHR$(12) clears the screen, and so forth. QPrint simply prints the characters that you give it. Thus, any ASCII value may be displayed.

Even though QPrint can print only strings, the BASIC STR$ function may be used to convert a number into a string form:

```
CALL QPrint(STR$(Number), Colr%, Page%)
```

See the COLORS.BAS description for more information about combining foreground and background colors to a single byte.

QPrintØ

assembler subroutine contained in PRO.LIB

Purpose:

QPrintØ displays a string very quickly at the current cursor location.

Syntax:

```
CALL QPrint0(X$, Colr%)
```

Where:

X$ is the string to be printed, and Colr% is the color to use. If Colr% is -1, then the current display screen colors will be maintained.

QPrintØ always prints at the current cursor location.

Comments:

QPrintØ is nearly identical to QPrint, except it prints to text page zero only. Because many programs do not need the ability to print on multiple pages, the additional code to accommodate that feature has been omitted. However, QPrintØ recognizes all of the text modes automatically.

QPrintAny

assembler subroutine contained in PRO.LIB

Purpose:

QPrintAny provides a simple way for a BASIC program to utilize two monitors at the same time.

Syntax:

```
CALL QPrintAny(X$, Colr%, MonCode%, Row%, Column%)
```

Where:

X$ is the string to be printed, and Colr% is the color to use. If Colr% is set to -1, then the current screen colors will be maintained.

MonCode% tells QPrintAny what type of monitor it is to print on as shown in the table below, and Row% and Column% indicate where the text is to be printed.

Comments:

Even though many PC's have two monitors connected, only one of them may actually be active at a time. This makes it very difficult to write programs that can access both monitors at the same time.

QPrintAny provides a simple solution to this problem, by letting you specify the monitor that is *not* the currently active one. Even when BASIC or the BIOS consider a monitor to be inactive, it will still display whatever is in its display adapter's memory. Thus, when the color monitor is currently active, you would tell QPrintAny to print to the monochrome monitor.

The MonCode% variable accepts one of three possible values, to indicate both the correct video segment, and whether CGA "snow suppression" is required. The code is as follows:

1	Monochrome monitor at segment &HB000
2	CGA monitor at segment &HB800
3	EGA or VGA monitor at segment &HB800

QPrintAny will also accept a "negative" version of the code returned by the QuickPak Professional Monitor% function. An example of this is shown below.

```
Mon% = Monitor%
CALL QPrintAny(X$, Colr%, -Mon%, Row%, Column%)
```

The Row% and Column% variables assume the display is in an 80 column text mode. If QPrintAny is being used to print to a color monitor and you had previously initialized it to 40 columns, simply divide the Column% variable by 2.

Similarly, if you needed to clear the screen you could use:

```
CALL QPrintAny(SPACE$(2000), 7, MonCode%, 1, 1)
```

And to clear to the end of a given line use:

```
CALL QPrintAny(SPACE$(81 - StartColumn%), 7, MonCode%, _
    Row%, StartColumn%)
```

Because QPrintAny accepts the row and column rather than having to call the BIOS to obtain that information, it is considerably faster than any of the other QuickPak Professional quick print routines.

QPrintRC

assembler subroutine contained in PRO.LIB

Purpose:

QPrintRC will display a string very quickly at a specified row and column.

Syntax:

```
CALL QPrintRC(Work$, Row%, Column%, Colr%)
```

Where:

Work$ is the string to be printed, Row% and Column% tell where to place it on the display, and Colr% is the combined foreground and background color to use. If Colr% is set to -1, the current screen colors are used.

Comments:

QPrintRC is modeled after the QuickPak Professional QPrint0 routine, except it uses Row and Column arguments rather than relying on the current cursor position. It is thus much faster than QPrint0, because it doesn't have to call the BIOS to get those coordinates.

QPWindow

assembler subroutines contained in PRO.LIB

QPWindow is a collection of routines that provide a complete text windowing manager that lets you establish a windowed viewport on the screen, and print text in that window scrolling as needed automatically.

Four routines are provided to set the window boundaries, print text, clear the window, and locate the cursor within the window. The syntax for these routines is as follows:

```
CALL QPWindowInit(BYVAL ULRow%, BYVAL ULCol%, BYVAL LRRow%, _
    BYVAL LRCol%)

CALL QPWindowPrint(Text$, BYVAL Colr%)

CALL QPWindowCLS(BYVAL Colr%)

CALL QPWindowLocate(BYVAL Row%, BYVAL Column%)
```

QPWindowInit is called to set new window boundaries. It is not necessary to call QPWindowInit before using QPWindowPrint, though you usually would. The initial default values are 1, 1, 25, 80. QPWindowInit also checks the current cursor position, and moves it into the window if necessary.

QPWindowPrint prints the text specified containing it within the window and scrolling as necessary.

QPWindowCLS clears the window and places the cursor in the upper-left corner, and it uses the color you give for clearing.

QPWindowLocate sets the cursor position within the virtual window. Note that the QPWindowLocate values are virtual, and are relative to the window's upper left corner. Therefore, CALL QPWindowLocate(1, 1) places the physical cursor at the upper-left edge of the window, and not at the upper-left edge of the video screen.

You can freely mix calls to the window routines with regular PRINT and LOCATE statements. QPWindowPrint keeps track of where it last printed, and when the window needs to be scrolled. However, after calling QPWindowPrint the physical cursor is placed immediately after the text that was just printed. Likewise, QPWindowCLS positions the cursor at the upper-left corner of the window after it clears the windowed portion of the screen.

All of these routines work in screen modes other than 80 columns, and adjust their behavior automatically.

There is only minimal error trapping such as preventing against locating the cursor outside of the current window boundaries. But there's no added code to ensure, for example, that the upper-left window boundary is higher than and to the left of the lower-right boundary. QPWindowInit also makes no effort to ensure that you use legal values. This is done on purpose—if you have a display adapter that can show, say, 132 columns, then these routines will accommodate that. However, rows and columns are limited to a maximum of 255.

You can also use SetMonSeg to print to an integer array or other block of memory, to create virtual screens of any size that are built in the background and then copied to display memory using BCopy or ScrnRest or MPRestore.

These windowing routines are demonstrated in the WINDOW.BAS example program.

ReadScrn

assembler subroutine contained in PRO.LIB

Purpose:

ReadScrn will quickly read characters from the display screen, and store them in a specified string variable.

Syntax:

```
CALL ReadScrn(Row%, Column%, X$, Page%)
```

Where:

Row% and Column% tell ReadScrn where on the screen the characters are located, X$ receives the screen contents, and Page% indicates which video page to read. If Page% is set to -1, then ReadScrn will read from the currently active screen page.

The number of characters to be read is specified by the length of X$.

Comments:

ReadScrn is considerably faster than BASIC's SCREEN statement because it doesn't have to call the BIOS for each character. One of the things that makes the BIOS so painfully slow when accessing video memory is that it reads only one character at a time. Further, each time the BIOS reads a character, it must see what type of display is present, determine if it is in graphics mode, calculate the display memory address, and so forth.

Because ReadScrn does this only once ahead of time, it can read any number of characters about as quickly as the BIOS can read just one.

The length of X$ is used to tell ReadScrn how many characters to read to avoid the possibility of corrupting string memory. If ReadScrn had been designed to accept the number of characters in a separate variable and X$ wasn't long enough, a complete crash would be likely.

The example below shows how to read 25 characters from the screen at location 2, 10 from the currently active text page.

```
X$ = SPACE$(25)
CALL ReadScrn(2, 10, X$, -1)
```

ReadScrnØ

assembler subroutine contained in PRO.LIB

Purpose:

ReadScrnØ will quickly read characters from the display screen, and store them in a specified string variable.

Syntax:

```
CALL ReadScrn0(Row%, Column%, X$)
```

Where:

Row% and Column% tell ReadScrnØ where on the screen the characters are located, and X$ receives the screen contents.

The number of characters to read is specified by the length of X$.

Comments:

ReadScrnØ is nearly identical to ReadScrn, except it reads from text page zero only. Because many programs do not need the ability to read from multiple pages, the additional code to accommodate that feature has been omitted. However, ReadScrnØ recognizes all of the text modes automatically.

ScrnDump

assembler subroutine contained in PRO.LIB

Purpose:

ScrnDump will take a snapshot from a graphics screen, regardless of the mode*, and send it to nearly any type of graphics printer.

Syntax:

```
CALL ScrnDump(DPI$, LPTNumber%, Translate%)
```

Where:

DPI$ indicates the Dots Per Inch resolution when sending to a Hewlett-Packard LaserJet or compatible printer, or is a null string if printing on an Epson 9-Pin Dot Matrix or compatible printer.

LPTNumber% is either 1, 2, or 3, to tell ScrnDump which parallel printer port to use. If a printer error occurs, ScrnDump will return LPTNumber% set to -1.

Translate% is set to 1 if the screen colors are to be translated to equivalent tile patterns, or 0 to print all colors as solid black.

When printing on a laser printer ScrnDump positions the upper left corner of the image at the *printer's* current cursor position.

Comments:

ScrnDump will automatically recognize the current video mode, and determine the number of screen bytes being used and their organization. There are several different ways that screen memory may be organized, which makes designing a routine such as ScrnDump extremely difficult.

The CGA screen modes use a method called "interlacing," where consecutive screen memory addresses occupy alternating rows on the display. The EGA modes are even more complicated, because each color is contained in a separate bank of memory. A Hercules screen uses interlacing similar to the CGA, except that screen memory is organized into four groups instead of two. Further, the LaserJet and Epson printers expect their graphics data in two entirely different formats.

The table below shows the acceptable values for DPI$.

" 75"	lowest resolution, largest picture size
"100"	medium-large resolution, medium-large picture size
"150"	medium-small resolution, medium-small picture size
"300"	highest resolution, smallest picture size
""	send the output to an Epson or compatible

Notice that when sending to a LaserJet, DPI$ *must* contain exactly three characters. Thus, to print at 75 dots per inch you will need to add a leading blank space, as shown above.

The translate option is provided to distinguish the colors on the screen when they are printed in a single color on paper. ScrnDump is demonstrated in the SCRNDUMP.BAS example program.

*Does not support VGA screen 13.

ScrnRest

assembler subroutine contained in PRO.LIB

Purpose:

ScrnRest will restore a screen that had previously been saved with ScrnSave or ScrnSave∅.

Syntax:

```
CALL ScrnRest(ULRow%, ULCol%, LRRow%, LRCol%, SEG Array%(1), Page%)
```

Where:

ULRow%, ULCol%, LRRow%, and LRCol% describe the area of the screen, Array%() is an integer array used to hold the screen, and Page% indicates the page to restore to. If Page% is -1 the current screen is restored. Notice that a screen may be restored to any legal text page, regardless of which page it had been saved from.

Comments:

Besides its intended purpose for restoring a screen that has been saved with ScrnSave, ScrnRest can also display screens that have been loaded from a disk file. Saving and loading screens from disk are discussed separately in a tutorial, however those examples assume that BLOAD will be used to get the screen from disk onto the display.

Once a screen has been saved from display memory, it may be loaded into an array prior to being displayed. This method is particularly valuable when several screens are involved. That is, you would load different screens each into their own array, and then display them as they are needed.

The example below first loads a previously BSAVE'd text screen from a disk file into an integer array, and then calls upon ScrnRest to display it.

```
REDIM Array%(2000)                  'make room for the screen
DEF SEG = VARSEG (Array%(1))        'where to begin loading
BLOAD "screen.", VARPTR(Array%(1))              'load it
CALL ScrnRest(1, 1, 25, 80, SEG Array%(1), -1)  'show it
ERASE Array%                        'free up the memory
```

Other examples of saving and restoring a screen are provided with QuickPak Professional in the SCRNSR.BAS and DEMOMGR.BAS demonstration programs. Also see the related routines ScrnRestØ, ScrnSave, ArraySize, MakeMono, and WindowMgr.

ScrnRestØ

assembler subroutine contained in PRO.LIB

Purpose:

ScrnRestØ will restore a screen that had previously been saved with ScrnSave or ScrnSaveØ.

Syntax:

```
CALL ScrnRest0(ULRow%, ULCol%, LRRow%, LRCol%, SEG Array%(1))
```

Where:

ULRow%, ULCol%, LRRow%, and LRCol% describe the area of the screen being restored, and Array%() is an integer array used to hold the screen.

Comments:

ScrnRestØ is nearly identical to ScrnRest, except it restores to text page zero only. Because many programs do not need the ability to work with multiple pages, the additional code to accommodate that feature has been omitted. However, ScrnRestØ recognizes all of the text modes automatically.

ScrnSave

assembler subroutine contained in PRO.LIB

Purpose:

ScrnSave will save all or part of a text screen into an integer array, to allow restoring it again at a later time.

Syntax:

```
DIM Array%(1 TO 2000)
CALL ScrnSave(ULRow%, ULCol%, LRRow%, LRCol%, SEG Array%(1), Page%)
```

Where:

ULRow%, ULCol%, LRRow%, and LRCol% describe the area of the screen to be saved, Array%() is an integer array used to hold the screen, and Page% indicates which text page to save from. If Page% is -1, the current screen is saved.

Comments:

ScrnSave and its companion ScrnRest are used extensively by many of the QuickPak Professional BASIC subprograms. For example, all of the pop-up utilities save the screen before they do anything else, so the original underlying screen may be restored when they finish.

Screens are saved into integer arrays, so it is up to you to ensure that the arrays have been sufficiently dimensioned. Integer arrays are used for several reasons. The most important is that an integer array is organized in a manner similar to a display screen. That is, the number of screen characters saved exactly corresponds to the number of array elements that are needed. Further, by using an array you may reclaim memory when the saved screen is no longer needed—simply erase the array. Finally, arrays allow you to save as many screens as necessary, with each in its own array.

Examples of saving and restoring screens are given in SCRNSR.BAS and DEMOMGR.BAS. Also see ScrnRest, ScrnSave0, ArraySize, MakeMono, and WindowMgr.

ScrnSaveØ

assembler subroutine contained in PRO.LIB

Purpose:

ScrnSaveØ will save all or a portion of a text screen into an integer array, to allow restoring it again at a later time.

Syntax:

```
CALL ScrnSave0(ULRow%, ULCol%, LRRow%, LRCol%, SEG Array%(1))
```

Where:

ULRow%, ULCol%, LRRow%, and LRCol% describe the area of the screen to be saved, and Array%() is an integer array used to hold the screen.

Comments:

ScrnSaveØ is nearly identical to ScrnSave, except it saves from text page zero only. Because many programs do not need the ability to work with multiple pages, the additional code to accommodate that feature has been omitted. However, ScrnSaveØ recognizes all of the text modes automatically.

ScrollD, ScrollL, ScrollR, ScrollU

four assembler subroutines contained in PRO.LIB

Purpose:

ScrollD will scroll any portion of the display screen down a specified number of lines. ScrollL, ScrollR, and ScrollU scroll left, right, and up respectively.

Syntax:

```
CALL ScrollD(ULRow%, ULCol%, LRRow%, LRCol%, Lines%, Page%)
```

Where:

ULRow%, ULCol%, LRRow%, and LRCol% describe the area of the screen to be scrolled, Lines% is the number of lines to scroll, and Page% indicates which video page is to be scrolled. If Page% is set to -1, the current screen will be scrolled.

Comments:

For the most part, we recommend using one of the APrint routines to manipulate text that must be moved around. However, there are many situations where the screen must be able to be scrolled directly.

For example, the QuickPak Professional spreadsheet uses a two-dimensional array to hold the worksheet contents. When the screen must be shifted up or down a line, Spread uses ScrollU or ScrollD, and then updates only the bottom or top. The Calc subprogram also uses ScrollU to move the result from previous calculations within its window.

All four of these routines will operate correctly in any text mode, and on any legal screen page. After a line of text has been scrolled, its previous location is erased to the current screen color automatically.

A demonstration of all four scroll routines is given in the SCROLL.BAS example program.

SetMonSeg

assembler subroutine contained in PRO.LIB

Purpose:

SetMonSeg assigns a new video display segment that will be used by all of the QuickPak Professional video routines.

Syntax:

```
CALL SetMonSeg(NewSegment%)
```

Where:

NewSegment% is the new segment that will be used for subsequent video operations, or 0 to restore it to the appropriate value for the currently active monitor.

Comments:

There are two important reasons you would want to change the display segment used by the QuickPak Professional video routines. One is to cooperate with multi-tasking programs such as Windows and DesqView. These programs assign a new video segment for each application that is running, to prevent one program from overwriting another's output screen.

The second important reason is to support virtual screens. Virtual screens let you print in the background, without disturbing what is currently being displayed. This is typically done by printing to an array or other area of memory. Then, to display the screen later you would simply copy the array contents to display memory. With QuickPak Professional this is done using either the ScrnRest, ScrnRest0, or MScrnRest subroutines.

See VIRTUAL.BAS for a brief example that shows how to manipulate virtual screens using SetMonSeg, QPrintRC, and ScrnRest0.

SplitColor

assembler subroutine contained in PRO.LIB

Purpose:

SplitColor accepts a single byte that contains combined foreground and background colors, and returns the separate components.

Syntax:

```
CALL SplitColor(Colr%, FG%, BG%)
```

Where:

Colr% is the incoming combined color value, and FG% and BG% are the foreground and background colors.

Comments:

All of the QuickPak Professional video routines expect a single value to specify both the foreground and background colors. The colors are in fact stored this way by the PC's hardware, and providing them in this format allows the video routines to operate that much faster. Further, variable memory needed by a program is minimized by eliminating yet another extra parameter.

SplitColor lets you extract the foreground and background components from a combined color. The formulas it uses are:

```
FG = (Colr AND 128) \ 8 + (Colr AND 15)
BG = (Colr AND 112) \ 16
```

A description of the color byte and how each bit affects the combined color is given in the discussion that accompanies the COLORS.BAS program.

WindowMgr

BASIC subprogram contained in WINDOMGR.BAS

Purpose:

WindowMgr is a complete window manager that frees the programmer from having to dimension and erase arrays, and keep track of windows as they are opened and closed.

WindowMgr also clears the screen to a specified color, and optionally draws a box around the window.

Syntax:

```
CALL WindowMgr(WindowNumber%, Action%, ULRow%, ULCol%, LRRow%, _
     LRCol%, Colr%)
```

Where:

WindowNumber% is the window number to be opened or closed. If WindowNumber% is zero and a window is to be opened, then the next available window number is used automatically. If WindowNumber% is zero and the window is being closed, then the most recently opened window will be closed.

Action% tells WindowMgr whether to open or close a window. 1 means open, and 0 means close.

ULRow%, ULCol%, LRRow%, and LRCol% define the area of the screen for the current window. When a window is being closed, using a zero for any of the corner parameters tells WindowMgr to use the corners that were given when the window was originally opened.

Colr% indicates the color to use when clearing the screen and drawing the box. Colr% is a single value that represents the combined foreground and background colors. If Colr% is zero, then a box is not drawn.

Comments:

When more than one or two screens must be saved and restored in a program, WindowMgr will greatly reduce the amount of programming that is needed.

A description of using combined colors is given in the discussion that accompanies the COLORS.BAS program.

WindowMgr is demonstrated in context in the DEMOMGR.BAS example program.

Wipes

BASIC subprograms contained in WIPES.BAS

Purpose:

Wipes contains a variety of subprograms that perform interesting visual effects. Several subprograms are included along with a demo in the same file that illustrate scrolling the text screen in a number of imaginative ways. Try it, you'll like it.

Chapter 10
Reference

Routines Contained in QuickPak Professional:

ARRAY MANIPULATION

NAME	*PURPOSE*
AddInt	Adds a constant value to all of the elements in a specified portion of an integer array.
DeleteStr	Removes an element from a conventional (not fixed-length) string array.
DeleteT	Removes an element from a fixed-length string or TYPE array.
DimBits	Creates a BASIC string that will be used to hold a Bit array.
Fill2	Assigns all of the elements in a specified portion of an integer array to any value.
Fill4	Assigns all of the elements in a specified portion of a single precision array to any value.
Fill8	Assigns all of the elements in a specified portion of a double precision array to any value.
Find	Searches all or part of a conventional (not fixed-length) string array forward looking for the first occurrence of a string or sub-string.
Find2	Same as Find, except Find2 honors capitalization.
FindB	Same as Find, except FindB searches backwards.
FindB2	Same as Find, except FindB2 searches backwards and honors capitalization.

ARRAY MANUPULATION (Cont'd)

NAME	*PURPOSE*
FindExact	Searches an entire conventional (not fixed-length) string array for an exact match. Unlike Find, Find2, and FindB, and the other "Find" routines, this one requires an exact match.
FindT	Searches all or part of a fixed-length string array for any string or sub-string.
FindT2	Same as FindT, except FindT2 honors capitalization.
FindTB	Same as FindT, except FindTB searches backwards.
FindTB2	Same as FindT, except FindTB2 searches backwards and honors capitalization.
FindLast	Scans a conventional (not fixed-length) string array backwards for the last non-blank element.
GetBit	Returns the status (On or off) of an element in a QuickPak Professional Bit array.
IMaxC@	Searches an entire currency array and returns the element number of the largest value.
IMaxD#	Searches an entire double precision array and returns the element number of the largest value.
IMaxI%	Searches an entire integer array and returns the element number of the largest value.
IMaxL&	Searches an entire long integer array and returns the element number of the largest value.

ARRAY MANUPULATION (Cont'd)

NAME	*PURPOSE*
IMaxS!	Searches an entire single precision array and returns the element number of the largest value.
IMinC@	Searches an entire currency array and returns the element number of the smallest value.
IMinD#	Searches an entire double precision array and returns the element number of the smallest value.
IMinI%	Searches an entire integer array and returns the element number of the smallest value.
IMinL&	Searches an entire long integer array and returns the element number of the smallest value.
IMinS!	Searches an entire single precision array and returns the element number of the smallest value.
InitInt	Initializes all or a specified portion of an integer array with increasing values.
InsertStr	Inserts an element at any point in a conventional (not fixed-length) string array
InsertT	Inserts an element at any point in a fixed-length string, numeric, or TYPE array.
LongestStr	Returns the length of the longest element in an entire string array.
MaxC@	Returns the largest value in a specified portion of a currency array.
MaxD#	Returns the largest value in a specified portion of a double precision array.
MaxI%	Returns the largest value in a specified portion of an integer array.

ARRAY MANUPULATION (Cont'd)

NAME	*PURPOSE*
MaxL&	Returns the largest value in a specified portion of a long integer array.
MaxS!	Returns the largest value in a specified portion of a single precision array.
MinC@	Returns the smallest value in a specified portion of a currency array.
MinD#	Returns the smallest value in a specified portion of a double precision array.
MinI%	Returns the smallest value in a specified portion of an integer array.
MinL&	Returns the smallest value in a specified portion of a long integer array.
MinS!	Returns the smallest value in a specified portion of a single precision array.
Search	Scans all or a portion of a numeric array to find the first match of a given value. Searching may be performed either forward or backwards, and the match may be specified to be exact, less than or equal to, or greater than or equal to.
SearchT	Allows you to search for any type of data (numeric or string) in a TYPE array and is case sensitive.
SearchT2	Same as SearchT but is case-insensitive.
SetBit	Sets an element in a QuickPak Professional Bit array to either one or zero (true or false).

ARRAY SORTS

NAME	*PURPOSE*
ISortC	Places in order all or a specified portion of a currency array by sorting a parallel index array (ascending or descending order).
ISortD	Places in order all or a specified portion of a double precision array by sorting a parallel index array (ascending or descending order).
ISortI	Places in order all or a specified portion of an integer array by sorting a parallel index array (ascending or descending order).
ISortL	Places in order all or a specified portion of a long integer array by sorting a parallel index array (ascending or descending order).
ISortS	Places in order all or a specified portion of a single precision array by sorting a parallel index array (ascending or descending order).
ISortT	Places in order all or a specified portion of a fixed-length string or TYPE array by sorting a parallel index array (ascending or descending order).
ISortT2	Same as ISortT, except sorting is performed without regard to capitalization.
ISortStr	Places in order all or a specified portion of a conventional (not fixed-length) string array by sorting a parallel index array (ascending or descending order).
ISortStr2	Same as ISortStr, except sorting is performed without regard to capitalization.
KeySort	Sorts a user-defined TYPE array based on any number of keys (ascending or descending order).
SortD	Places in order all or a portion of a double precision array (ascending or descending order).

ARRAY SORTS (Cont'd)

NAME	*PURPOSE*
SortC	Places in order all or a portion of a currency array (ascending or descending order).
SortI	Places in order all or a portion of an integer array (ascending or descending order).
SortL	Places in order all or a portion of a long integer array (ascending or descending order).
SortS	Places in order all or a portion of a single precision array (ascending or descending order).
SortT	Places in order all or a portion of a fixed-length string or TYPE array (ascending or descending order).
SortT2	Same as SortT, except sorting is performed without regard to capitalization.
SortStr	Places in order all or a portion of a conventional string array (ascending or descending).
SortStr2	Same as SortStr, except sorting is performed without regard to capitalization.

DATE/TIME

NAME	*PURPOSE*
Clock	Provides a continual display of the current time, without having to loop repeatedly in a BASIC program.
Clock24	Alternate version of the Clock routine that displays the time in a 24-hour format, without the "am" or "pm" indicator.
Date2Day	Accepts an incoming date string, and returns the appropriate day of the week (1-7).
Date2Num	Converts a date in string form to an equivalent integer variable to allow date arithmetic.
DayName$	Accepts an integer value between 1 and 7, and returns an equivalent day name as a string in the form "Mon", "Tue", etc.
EDate2Num	Accepts a date in European "DDMMYY" format, and returns a corresponding integer value.
ENum2Date$	Converts a previously encoded integer date to an equivalent string in European format.
MonthName$	Accepts an integer value between 1 and 12, and returns an equivalent month name as a string in the form "Jan", "Feb", "Mar", etc.
Num2Date$	Converts a previously encoded integer date to an equivalent date string.
Num2Day	Accepts an integer number that represents a date in the QuickPak Professional format, and returns the appropriate day of the week (1-7).

DATE/TIME (Cont'd)

NAME	*PURPOSE*
Num2Time$	Converts a long integer that represents the number of seconds past midnight to an equivalent time in string form.
Pause	Pauses a program's execution for a specified period of time, to a resolution as small as 1/18th second.
Pause2	Pauses a programs execution for a specified number of microseconds.
Pause3	Pauses a programs execution for a specified period of time, to a resolution as small as 1 millisecond.
PDQTimer&	An integer-only TIMER replacement; does not require the use of the floating point math library.
SysTime	Obtains the current system time through DOS, and returns it in a string formatted to the hundredth of a second.
Time2Num&	Converts a time in string form to an equivalent number of seconds after midnight to allow time arithmetic.
WeekDay	Returns the day of the week (1-7) given a legal DOS date in a string form.

DIRECTORY

NAME	*PURPOSE*
CDir	Changes directories.
DCount	Reports the number of directories under the current directory that match a particular specification.
DirTree	Reads a disk's entire directory tree, and returns it in two string arrays suitable for display.
ExeName$	Returns the full name of the currently executing program, including the drive, path, and file name.
FullName	Accepts a file name, and returns it adding the complete path information.
GetDir$	Returns the current directory for either a specified drive or the default drive.
KillDir	Removes a specified directory.
MakeDir	Creates a directory.
NameDir	Renames a directory
ReadDir	Obtains a list of directory names from disk and loads them into a conventional (not fixed-length) string array.
ReadDirT	Obtains a list of directory names from disk and loads them into a fixed-length string array.

DISK AND DISK DRIVE

NAME	*PURPOSE*
DiskInfo	Reports a disk's sector and cluster makeup.
DiskRoom&	Returns the number of bytes available on the specified drive.
DiskSize&	Returns the total number in bytes of a specified disk drive.
FormatDiskette	Adds disk formatting capabilities to your programs.
GetDisketteType	Returns the type of floppy disk drive that is installed.
GetDrive	Returns the current default disk drive.
GetVol$	Returns the disk volume label for either the default or a specified drive.
GoodDrive	Determines whether a specified drive letter is valid.
LastDrive	Returns the last consecutively available drive in a PC.
NetDrive	Reports if a given drive is remote on a network.
PutVol	Creates or renames a disk volume label.
ReadTest	Reports whether a specified disk drive is ready for reading.
Removable	Reports if a given drive's media is removable (a floppy drive).
SetDrive	Allows changing the current default drive.
WriteSect	Writes new data to any disk sector from either a conventional or fixed-length string.

DISK AND DISK DRIVE (Cont'd)

NAME	PURPOSE
WriteSect2	Writes new data to a group of disk sectors from either a conventional or fixed-length string.
WriteTest	Reports if a specified disk drive is ready for writing.

ERROR HANDLING

NAME	*PURPOSE*
DOSError	Reports if an error occurred during the last call to a QuickPak Professional DOS routine.
ErrorMsg$	Returns an appropriate message given any of the BASIC error numbers for a DOS service.
SetError	Allows a BASIC program to set or clear the DOSError and WhichError functions.
SetLevel	Allows a BASIC program to set the DOS error level.
WhichError	Reports which error if any occurred during the last call to a QuickPak Professional DOS routine.

FILE MANAGEMENT

NAME	*PURPOSE*
ClipFile	Establishes a new length for a file either longer or shorter.
Exist	Determines the existence of a file.
FastLoad	Loads an entire text file into an array.
FastSave	Saves an entire text file to disk.
FClose	Closes a file opened with the QuickPak Professional FOpen Statement.
FCopy	Copies a file from within BASIC without requiring SHELL.
FCount	Reports the number of files that match a particular specification.
FCreate	Creates a file in preparation for writing to it with QuickPak Professional file handling routines.
FEof	Reports if the current DOS Seek location is at the end of a specified file.
FFlush	Flushes a file's buffers to disk without requiring the file to be closed.
FGet	Reads data from a disk file into a string.
FGetA	Reads data from a disk file to an array.
FGetAH	Retrieves an entire huge array of any size from disk in a single operation.
FGetR	Reads data from a random disk file.
FGetRT	Reads data from disk file into a TYPE variable.

FILE MANAGEMENT (Cont'd)

NAME	*PURPOSE*
FGetRTA	Reads data from a random disk file into a TYPE array.
FGetT	Reads binary data from a disk file into a TYPE variable.
FileComp	Reports if any two files are the same.
FileCopy	Serves as a "front end" to the FCopy routine, and allows the use of wild cards.
FileCrypt	Encrypts a file using a password provided by the calling program.
FileInfo	Returns all of the characteristics of a file.
FileSize&	Returns the length of a named file.
FileSort	Sorts a random access disk file on any number of keys.
FLInput$	Reads a line of data from a file opened with the QuickPak Professional FOpen routine.
FLoc&	Reports the current DOS file pointer position for files opened with the QuickPak Professional FOpen routine.
FLof&	Returns the length of a file opened with the QuickPak Professional FOpen routine.
FOpen	Opens a disk file in preparation for reading or writing data.
FOpenAll	Opens a file for any access mode including all of the variations required for a network.
FOpenS	Opens a file for read/write access on a network while allowing others read access only.

FILE MANAGEMENT (Cont'd)

NAME	*PURPOSE*
FPut	Writes data to a disk file from a string.
FPutA	Writes data to a disk file from an array.
FPutAH	Writes an entire huge array to disk.
FPutR	Writes data to a random disk file.
FPutRT	Writes data to a random disk file where the source data is either a fixed-length string or a TYPE variable.
FPutRTA	Same as FPutRT, except FPutRTA accepts a segmented for saving array data.
FPutT	Writes data to a disk file from either a fixed-length string or TYPE variable.
FSeek	Positions the DOS file pointer for a file opened with FOpen.
FStamp	Creates a new date and time for a specified file.
GetAttr	Returns the setting of a file's attribute byte.
Handle2Name	Returns the name of an open file, given the DOS handle.
KillFile	Deletes a specified file.
LineCount	Returns the number of lines of text in a specified file.
LockFile	Locks all or a portion of a network file.
NameFile	Renames a file.
QBLoad	Loads a BSaved or QBSaved file into memory much like QuickBASIC's BLoad.
QBSave	Saves an array of data to disk.

FILE MANAGEMENT (Cont'd)

NAME	*PURPOSE*
ReadFile	Obtains a list of file names from disk and loads them into a conventional (not fixed-length) string array.
ReadFileI	Obtains a list of file names, sizes, dates and times from disk, formats them and loads them into a conventional string array.
ReadFileT	Obtains a list of file names from disk and loads them into a fixed-length string array.
ReadFileX	Obtains a list of file names, sizes, dates and times from disk, formats them and loads them into separate components of a TYPE array.
ReadSect	Reads the contents of any disk sector into a string.
ScanFile&	Scans a file for a particular string.
SearchPath	Accepts the name of any executable file and returns its fully qualified name by searching the DOS path.
SetAttr	Sets the attribute byte for a specified file.
ShareThere	Reports if SHARE is installed in the host PC.
SplitName	Parses out the components in a file name and returns the drive letter, path name, file name and extension as separate items.
Unique$	Returns a file name that does not already exist on the default drive in a specified directory.
UnLockFile	Unlocks all or a portion of a network file.
Valid	Examines a string to see if it could be a valid DOS file name.

FINANCIAL FUNCTIONS

1. Sinking Fund Annutites:

NAME	*PURPOSE*
QPFV#	Calculates the future value of an annuity (sinking fund).
QPFVN#	Calculates the term (number of payments) of a sinking fund.
QPFVP#	Calculates the payment amount of a sinking fund.

2. Annuity Due:

NAME	*PURPOSE*
QPFVD#	Calculates the future value of an annuity due.
QPFVND#	Calculates the term (number of payments) of an annuity due/FV.
QPFVPD#	Calculates the payment amount of an annuity due/FV.

3. Ordinary Annuity:

NAME	*PURPOSE*
QPPMT#	Calculates loan payment (ordinary annuity).
QPPV#	Calculates present value of an ordinary annuity.
QPPVN#	Calculates the term (number of payments) of an ordinary annuity.

FINANCIAL FUNCTIONS (Cont'd)

4. Annuity due relationships:

NAME	*PURPOSE*
QPPMTD#	Calculates lease payments (annuity due).
QPPVD#	Calculates the present value of an annuity due.
QPPVND#	Calculates the term (number of payments) of an annuity due.

5. Other compound interest relationships:

NAME	*PURPOSE*
QPCINT#	Finds the future value of a savings account drawing compound interest.
QPCTERM#	Determines the number of compounding periods it will take an investment to grow to a pre-determined value.
QPIRR#	Calculates the internal rate of return.
QPNPV#	Calculates the net present value of future cash flows.
QPRATE#	Obtains the periodic interest rate required for an investment to grow to a pre-determined value in a specified time.

6. Depreciation:

NAME	*PURPOSE*
QPDDB#	Calculates double declining balance depreciation.
QPSLN#	Calculates straight line depreciation.
QPSYD#	Calculates sum-of-years-digits depreciation.

KEYBOARD

NAME	*PURPOSE*
AltKey	Reports if the Alt key is currently depressed.
CapsLock	Reports if the Caps Lock key is currently depressed.
CapsOff	Turns off the Caps Lock key status.
CapsOn	Turns on the Caps Lock key status.
ClearBuf	Clears the keyboard buffer of any pending keystrokes.
CtrlKey	Reports if the Ctrl key is currently depressed.
CapNum	Displays the current setting of the Caps Lock and NumLock keys.
InStat	Reports the number of characters that are currently pending in the keyboard buffer without removing them.
Keyboard	Provides a continuous display of the current Caps Lock and NumLock status, without having to loop repeatedly to obtain the information.
KeyDown	Reports if any keys are currently depressed.
MGetKey	Clears the keyboard buffer of any pending keys, and then waits until either a key or mouse button is pressed.
NumLock	Reports if the NumLock key is currently depressed.
NumOff	Turns off the NumLock key status.
NumOn	Turns on the NumLock key status.

KEYBOARD (Cont'd)

NAME	*PURPOSE*
PeekBuf	Returns what key if any is currently pending in the keyboard buffer without actually removing it.
RptKey	Works much like BASIC'S INKEY$; however, it returns the number of times an Alt, Ctrl, or shifted key has been pressed.
ScrlLock	Reports if the Scroll lock key is currently depressed.
ShiftKey	Reports if the Shift key is currently depressed.
StuffBuf	Inserts a string into the keyboard buffer as if it had been entered at the keyboard.
WaitKey	Clears the keyboard buffer of any pending keystrokes, and then waits until a key is pressed.
WaitScan	Waits for any key to be pressed, and then returns the scan code for that key.

MENUS

NAME	PURPOSE
AMenu	A multi-column menu routine that accepts a list of choices from a conventional (not fixed-length) string array. Choices are made by moving the cursor bar to the desired choice and pressing Enter.
AmenuT	Same as AMenu, except it is intended for use with fixed-length strings.
ASCIIChart	Displays a "scrollable" chart of ASCII characters and their corresponding decimal and hexadecimal values.
ASCIIPick	Presents a table of ASCII characters and waits until one is selected.
MASCIIPick	Same as ASCIIPick, except it supports the mouse for selection.
ColorPick	Presents a table of colors and their corresponding values, and waits until one is selected.
MColorPick	Same as ColorPick, except it supports the mouse for selection.
DirFile	Provides a menu for selecting a file name from a list of choices Choices are made by moving the cursor bar to the desired choice and pressing Enter.
Lts2Menu	A Lotus 123 "look alike" menu where a list of choices is displayed horizontally on a single line, along with a prompt for the current item. Choices are made by either pressing the arrow keys to highlight a choice, or by pressing a key that corresponds to the first letter of the choice.

MENUS (Cont'd)

NAME	*PURPOSE*
LtsMenu	Same as Lts2Menu, except it does not display a corresponding prompt.
MAMenu	Same as AMenu, except a mouse is supported for menu selection.
MAMenuT	Same as AMenuT, except a mouse is supported for menu selection.
MenuVert	A vertical menu program that accepts a list of choices from a conventional (not fixed-length) string array. Choices are made by moving the cursor bar to the desired choice and pressing Enter.
MMenuV	Same as MenuVert, except a mouse is supported for menu selection.
PickList	A "front end" subprogram for VertMenu that allows selecting multiple items from a single menu.
PullDown	A pull-down menu routine very similar to the QuickBASIC 4.0 environment's pull-down menu with full mouse support.
PullDnMS	A pull-down menu routine very similar to the QuickBASIC 4.5 environment's pull-down menu with full mouse and Hotkey support.
VertMenu	A vertical menu program that accepts a list of choices from a conventional (not fixed- length) string array. Choices are made by moving the cursor bar to the desired choice and pressing Enter. VertMenu always saves the underlining screen, and draws a box and an attractive shadow around the menu.
VertMenuT	Same as VertMenu, except for use with fixed-length string arrays.

MISCELLANEOUS

NAME	*PURPOSE*
BCopy	Copies a block of memory (up to 64K in size) to a new location.
BCopyT	Copies one or more elements in a TYPE array to another array, or to any location in memory. BCopyT can be used to move any contiguous block of memory, even if the number of bytes exceeds 65536.
Calc	Provides a handy pop-up calculator that you can add to your BASIC programs.
Calendar	Provides a pop-up calendar that will display any month of any year.
Compare	Compares any two blocks of memory, and reports if they are the same.
CompareT	Compares any two TYPE variables, and reports if they are the same.
DOSVer	Returns the version of DOS that is presently running on the host PC.
Empty	An empty procedure that does absolutely nothing, for use when timing BASIC functions.
EMS Manager	A collection of routines that allow you to store and retrieve any type of data using expanded memory.
FileView	A complete file browsing subroutine (assembler version of ViewFile).
FudgeFactor&	Returns a long integer value that roughly corresponds to the processing speed of a PC.
GetCMOS	Shows how to acces the data in the CMOS.RAM of an AT or compatible computer.
GetCPU	Returns an integer value that indicates the type of CPU installed on the host PC.

MISCELLANEOUS (Cont'd)

NAME	*PURPOSE*
GetDS	Returns BASIC's current internal data segment.
GetEquip	Returns several items from the equipment list kept in the low-memory area of a PC.
LoadExec	Executes another program and retrieves its exit code (the DOS error level).
LockUp	Causes an immediate system freeze that can only be cleared by turning off the PC's power switch.
MathChip	Reports if an 80x87 math co-processor chip is installed in the host PC.
Peek1	Reads one byte from a specified segment and address and returns its value.
Peek2	Reads a word (two bytes) at a specified segment and address, and returns its value.
Poke1	Writes a new byte to a specified segment and address.
Poke2	Writes a new word (two bytes) to a specified segment and address.
QPCli &QPSti	Disables and enables interrupts respectively.
ReBoot	Causes the host PC to perform a "warm" boot, as if the Ctrl-Alt-Del keys had been pressed.
SetCmd	Allows you to set the COMMAND$ that will be read by a program that is subsequently RUN or CHAINed to.
Soundex$	Returns a "sounds like" code that can be used to compare if two strings sound alike.
SpellNumber$	Accepts a number in the form of a string such as "12345", and returns a spelled-out English equivalent in the form of "Twelve Thousand Three Hundred Forty Five".

MISCELLANEOUS (Cont'd)

NAME	*PURPOSE*
Spread	A complete spreadsheet subprogram that may be called as a "pop-up" from within a BASIC program.
ViewFile	A complete pop-up file browsing subprogram in BASIC.
XMS Manager	A collection of routines that allow you to store and retrieve any type of data using extended memory.

MOUSE

NAME	*PURPOSE*
ButtonPress	Returns the number of times a specified mouse button has been pressed since the last time it was called. It also returns the X/Y coordinates where the mouse cursor was located when that button was last pressed.
GetCursor	Reports the current pixel X/Y location of the mouse cursor and which mouse buttons are currently depressed.
GetCursorT	Reports the current row/column X/Y location of the mouse cursor and which mouse buttons are currently depressed.
GrafCursor	Allows defining the shape of the mouse cursor.
HideCursor	Turns the mouse cursor off.
InitMouse	Used both to determine if a mouse is present in the host PC, and to reset the mouse driver software to its default values.
MBufSize	Returns the length of the buffer needed to save the current mouse state.
MGetState	Saves the current mouse state into a string.
Motion	Allows a program to establish the sensitivity of the mouse cursor motion.
Mouse	Provides access to all of the mouse services.
MouseRange	Returns a range number that tells where the mouse cursor is located, based on an array of screen coordinates in text mode.
MouseRangeG	Same as MouseRange but for graphics modes.

MOUSE (Cont'd)

NAME	*PURPOSE*
MouseTrap	Establishes the allowable range of movement for the mouse cursor.
MSetState	Restores a saved mouse state to the mouse driver.
SetCursor	Establishes a new location for the mouse cursor.
ShowCursor	Turns the mouse cursor on.
TextCursor	Initializes the mouse cursor in text mode and defines its color.

NUMERIC FUNCTIONS AND SUBS

NAME	*PURPOSE*
AddUSI	Adds two integers on an unsigned basis, without creating an overflow error if the total exceeds 32767.
Bin2Num	Accepts a binary number in the form of a string, and returns an equivalent value.
C2F!	Converts a celsius temperature to its Fahrenheit equivalent.
Eval#	Returns the value of a string similar to BASIC's VAL function, but without regard to dollar signs, commas, or any other punctuation.
Evaluate#	A full-featured expression evaluator, it accepts a formula in an incoming string, and returns a double precision result.
Factorial#	Provides an extremely fast way to obtain a factorial value.
F2C!	Converts a Fahrenheit temperature to its celcius equivalent.
MaxInt%	Compares two integer variables, and returns the value of the higher one.
MaxLong&	Compares two long integer variables, and returns the value of the higher one.
MinInt%	Compares two integer variables, and returns the value of the lower one.
MinLong&	Compares two long integer variables, and returns the value of the lower one.
Num2Bin$	Converts a number into an equivalent binary string with a fixed length of 16 digits.

NUMERIC FUNCTIONS AND SUBS (Cont'd)

NAME	*PURPOSE*
Num2Bin2$	Same as Num2Bin$, except Num2Bin2$ returns only as many digits as required to represent the number.
Pad$	Adds leading zeros to a number, padding it to a specified number of digits.
Power&	Raises any number to a power specified without using floating point math.
Power2&	Raises 2 to a power specified without using floating point math.
QPACOS#	Returns the Arc cosine of X.
QPASIN#	Returns the Arc sine of X.
QPATAN2#	Returns the 4-quadrant arc tangent of Y/X.
QPLOG10#	Returns log of X base 10.
QPSolver	A complete environment for entering and editing variables and expressions that are evaluated using the QuickPak Professional Evaluate function.
QPRound$	Rounds a number to a specified number of decimal places.
QPUSI	(QuickPak Unsigned Integer) Returns the low-word portion of a long integer, and it is useful for those situations where you are using a long integer to store integer information whose value may exceed 32767.
Rand!	Returns a random number between the specified upper and lower bounds.
ShiftIL	Shifts the bits in an integer variable a specified number of positions to the left.

NUMERIC FUNCTIONS AND SUBS (Cont'd)

NAME	*PURPOSE*
ShiftIR	Shifts the bits in an integer variable a specified number of positions to the right.
ShiftLL	Shifts the bits in a long integer variable a specified number of positions to the left.
ShiftLR	Shifts the bits in a long integer variable a specified number of positions to the right.
Signed	Takes an incoming unsigned integer value, and returns it in a signed form.
Times2	Multiplies an integer variable times 2, without causing an overflow if the value exceeds 32,767.
TrapInt	Constrains an incoming value to within a specified upper and lower limit.
UnSigned&	Takes an incoming signed integer value, and returns it in unsigned form.
VLAdd	Adds two "very long" integers and returns the result in another one.
VLDiv	Divides two "very long" integers, and returns the result and remainder in two other ones.
VLMul	Multiplies two "very long" integers, and returns the result in another one.
VLPack	Accepts a "very long" integer value in the form of a string, and returns it packed to the correct format in a double precision "alias" variable.
VLSub	Subtracts two "very long" integers, and returns the result in another one.

NUMERIC FUNCTIONS AND SUBS (Cont'd)

NAME	*PURPOSE*
VLUnpack	Accepts a very long integer value in the form of a double precision "alias" variable, and returns it in string form suitable for being displayed or printed.

PRINTER

NAME	*PURPOSE*
BLPrint	Similar to BASIC's LPRINT, except BLPrint returns an error code should the printer be off line, or becomes unavailable during printing.
Extended	Downloads a replacement font file to an Epson printer, enabling it to print the entire IBM extended character set.
PrinterReady	Similar to the PRNReady function, it avoids the excessive delays that can occur when a printer is turned on, but is off-line.
PRNReady	Reports whether a specified printer is available and on-line.
PrtSc	Sends a snapshot of the screen to a printer, as if the PrtSc key was pressed.
PrtSc0	Same as PrtSc, except PrtSc0 prints from text screen zero only.
PSwap	Exchanges LPT1 and LPT2 each time it is called.
ScrnDump	Takes a snapshot from a graphics screen regardless of mode, and sends it to an HP LaserJet (or compatible) or 9-pin Epson dot matrix (or compatible) graphics printer (Does not print SCREEN 13).

SAVE/LOAD/DISPLAY SCREENS

NAME	*PURPOSE*
ArraySize	Returns the number of elements in an integer array required to hold a portion of the display screen.
EGABLoad	Loads an EGA or VGA graphics image from a specified disk file.
EGABSave	Saves an EGA or VGA graphics screen to a specified disk file.
EGAMem	Reports the amount of memory available on an EGA display adapter.
MPRestore	Lets you display any rectangular portion from a screen that has been saved to an array using ScrnSave or ScrnSave0.
MScrnRest	Same as ScrnRest, except MScrnRest turns off the mouse cursor while it is working.
MScrnSave	Same as ScrnSave, except MScrnSave turns off the mouse cursor while it is working.
ScrnRest	Restores a screen previously saved with ScrnSave or ScrnSave0.
ScrnRest0	Same as ScrnRest, except ScrnSave0 restores to text page zero only.
ScrnSave	Saves all or part of a text screen into an integer array.
ScrnSave0	Same as ScrnSave, except ScrnSave0 saves from text page zero only.
Wipes	Allows you to display screens in a variety of interesting ways.

SOUND

NAME	*PURPOSE*
Chime	Provides five different types of beep tones, and five attention-getting trill sounds.
QPPlay	Replaces BASIC's PLAY statement, while greatly reducing the amount of code that is added to your programs (about 700 bytes compared to 14.5K!).
QPSound	Similar to BASIC's SOUND statement, but with a substantial reduction in code size.

STATISTICAL FUNCTIONS

NAME	*PURPOSE*
QPAVG#	Returns the average of the values in an array.
QPCOUNT	Returns the number of entries in an array.
QPMAX#	Returns the highest value in a list.
QPMIN#	Returns the lowest value in an array.
QPSTD#	Returns the population deviation of items in a list.
QPSUM#	Returns the sum of all values in an array.
QPVAR#	Returns the population variance of values in a list.

STRING MANIPULATION

NAME	PURPOSE
ASCII	Returns the ASCII value for the first character in a string, but it will not cause an "Illegal Function Call" error if the string is null.
Blanks	Reports the number of leading blanks in a specified string. Both CHR$(32) and CHR$(0) null characters are recognized.
Compact$	Compresses a string by removing all embedded blanks.
Delimit	Counts the number of delimiters in a string, by matching against a second string that contains a table of valid delimiters.
Encrypt	Encrypts a specified string using a password that you provide.
Encrypt2	Encrypts a specified string using a password that you provide. Somewhat more secure than the original Encrypt.
ExpandTab$	Accepts an incoming text string that contains embedded CHR$(9) Tab characters, and replaces them with the appropriate number of CHR$(32) spaces.
Far2Str$	Retrieves an ASCIIZ string from anywhere in memory, and returns it as a conventional BASIC string.
FUsing$	Accepts an incoming number and image string, and returns it formatted, much like BASIC's PRINT USING.
InCount	Reports how many times one string occurs within another, and the search string may contain any number of wild cards.

STRING MANIPULATION (Cont'd)

NAME	*PURPOSE*
InCount2	Same as InCount, but searching is case-insensitive.
InCountTbl	Returns the number of characters in a string that match any of the characters in a table.
InstrTbl	Searches a string for the first occurrence of any characters that are specified in a table string.
InstrTblB	Same as InstrTbl, except searching is performed backwards from the end of the string.
InstrTbl2	Same as InstrTbl, except InstrTbl2 is not case-sensitive.
InstrTblB2	Same as InstrTbl2, except InstrTbl2 is not case-sensitive and searching is performed backwards from the end of the string.
LastFirst$	Reverses the position of a first and last name in a string such that the last name comes before the first.
LastLast$	Reverses the position of a first and last name in a string such that the last name comes after the first.
LowASCII	Strips the "high bit" from all of the characters in a specified string.
Lower	Converts all characters in a specified string to lower case very quickly.
LowerTbl	Converts all characters in a specified string to lower case, and it also looks in a supplied table to determine how to handle foreign characters.

STRING MANIPULATION (Cont'd)

NAME	*PURPOSE*
MidChar	Returns the ASCII value for a single character within a string.
MidCharS	Inserts a single character into a string much faster then using the MID$ statement.
NotInstr	Returns the offset of the first character in a string that does not match any of the characters in another.
Null	Reports if a specified string is either null, or is filled with blank or CHR$(0) characters.
Parse	Extracts individual components from a string, and places each into a separate elements of a string array.
ParseStr	Accepts an incoming string that contains numbers separated by commas, and returns a new string consisting of equivalent ASCII characters.
ParseString	Accepts a string containing delimited information and returns portions of the string each time it is invoked.
ProperName	Converts the first letter of each word in a string to upper case.
QInstr	Serves the same purpose as BASIC's INSTR function, except it accepts any number of wild cards.
QInstr2	Same as QInstr, except QInstr2 is case-insensitive.
QInstrB	Same as QInstr, except it searches the source string backwards.
QInstrB2	Same as QInstr, except it searches the source string backwards, and is case-insensitive.

STRING MANIPULATION (Cont'd)

NAME	*PURPOSE*
QInstrH	Locates a string of text anywhere in the PC's normal 1 MB of address space.
QPHex$	Faster version of BASIC's HEX$ function that also returns a string padded to a specified number of digits.
QPLeft$	Smaller code version of BASIC's LEFT$ function.
QPMid$	Smaller code version of Basic's MID$ function.
QPRight$	Smaller code version of BASIC's RIGHT$ function.
QPLen	Smaller code version of BASIC's LEN function.
QPSadd	Smaller code version of BASIC's SADD function.
QPSegAdr&	Returns the segmented address of a conventional (not fixed-length) string as a long integer exactly like BASIC 7's SSEGADD However, the same routine is provided in both the QB and BC7 versions of the QuickPak Pro library.
QPSSeg	Returns the segment address of a Far String (BC7 only).
QPStrI$	Smaller code version of BASIC's STR$() function (Use with integers).
QPStrL$	Smaller code version of BASIC's STR$() function (Use with long integers).
QPTrim$	Removes both trailing and leading spaces and CHR$(0) characters from a string.
QPLTrim$	Smaller code version of BASIC's LTRIM$ function, also trims CHR$(0) characters.
QPRTrim$	Smaller code version of BASIC's RTRIM$ function, also trims CHR$(0) characters.

STRING MANIPULATION (Cont'd)

NAME	*PURPOSE*
QPValI%	Smaller code version of BASIC's VAL function that returns an integer result.
QPValL&	Smaller code version of BASIC's VAL function that returns a long integer result.
RemCtrl	Scans through a given string and replaces any control characters with a specified new character.
ReplaceChar	Replaces all occurrences of a specified character with a different character within a variable length string.
ReplaceChar2	Same as ReplaceChar, except it is case-insensitive.
ReplaceCharT	Same as ReplaceChar, except it can be used with Type variables, fixed length strings, or any block of memory up to 64K long.
ReplaceCharT2	Same as ReplaceCharT except it ignores capitalization when searching and it replaces characters using upper case versions.
ReplaceString	Replaces all occurrences of a specified string with a different string.
Sequence	Increments the characters in a string.
ShrinkTab$	Reduces the length of a string by replacing groups of blank spaces with CHR$(9) tab characters.
ReplaceTbl	Replaces all occurrences of one character with any other character using a lookup table.
StringMgr	A collection of routines that allows you to store entire string arrays in far memory.
Translate	Replaces an occurrence of extended "box drawing" characters with an appropriate equivalent ASCII character.

Reference

STRING MANIPULATION (Cont'd)

NAME	PURPOSE
UnParseStr$	Accepts an incoming string that contains ASCII characters, and returns the equivalent numeric values separated by commas.
Upper	Converts all alphabetic characters in a string to upper case very quickly.
UpperTbl	Capitalizes all of the characters in a string, and it also looks in a supplied table to determine how to handle foreign characters.
WordWrap	Accepts a single long string and prints it on the screen with word wrap.

TEXT/DATA ENTRY

NAME	*PURPOSE*
DateIn	Provides the ability to enter or edit date fields. The cursor automatically skips over the separating slashes, and Alt-C will clear the field.
Dialog	Generates Dialog boxes similar to QuickBASIC's based on the contents of a string array passed to it.
Editor	Text input routine that allows editing an existing string. Input may be limited to numbers or caps only, and both the normal and edit colors are specified.
MaskIn	A sophisticated "Mask Input" routine which allows you to specify the type of characters to be entered.
MEditor	Same as Editor, except it includes full mouse support.
NumIn	Provides the ability to enter or edit a numeric field. The cursor automatically skips over the decimal point, and Alt-C will clear the field.
QEdit	A complete text editor subprogram that may be called as a "pop-up" from within a BASIC program.
QEdit7	Same as QEdit, but optimized for PDS 7, works with QuickBASIC as well.
QEditS	A stripped-down version of QEdit without block operations or a window "frame".
ScrollIn	Single-line text-input routine that allows editing or entering a string of any length.
TextIn	A BASIC text input routine similar to Editor.

TEXT/DATA ENTRY (Cont'd)

NAME	*PURPOSE*
YesNo	Provides a quick way to accept a Yes or No input.
YesNoB	BASIC version of YesNo.

UTILITY PROGRAMS

NAME	*PURPOSE*
Demo123	An example program that shows how to read and write Lotus 123 files.
MakeQLB	Makes a Quick Library subset from a library or libraries.
ReadDirs	Demonstration utility that searches through all levels of sub-directories for matching files.
QuickDOS	Menu-driven DOS utility — copies, deletes, moves, sorts files.

VIDEO *(Text Mode Only Unless Otherwise Specified)*

NAME	*PURPOSE*
APrint	Prints any portion of a conventional (not fixed-length) string array, and contains the display within a specified portion of the screen.
APrintØ	Same as APrint, except APrintØ displays on text page zero only for less code.
APrintT	Prints any portion of a fixed-length string array, and contains the display within a specified portion of the screen.
APrintTØ	Same as APrintT, except APrintTØ displays on text page zero only.
Box	Draws a box frame on the screen.
BoxØ	Same as Box, except BoxØ displays on text page zero only.
BlinkOff	Turns the blink attribute off to allow high intensity background colors on EGA and VGA only.
BlinkOn	Turns the blink attribute on to allow flashing text on EGA and VGA only.
BPrint	Prints either a conventional or fixed-length string at the current cursor position through DOS.
ClearEOL	Erases the current screen line starting at the current cursor position.

VIDEO (Cont'd)

NAME	*PURPOSE*
ClearScr	Clears all or a portion of the screen to a specified color.
ClearScrØ	Same as ClearScr, except ClearScrØ displays on text page zero only.
Colors	Displays a chart showing all possible color combinations.
CsrSize	Reports the top and bottom scan lines that describe the current cursor size. Also determines if the cursor is on or off.
FillScrn	Fills any rectangular portion of the screen with a specified character.
FillScrnØ	Same as FillScrn, except FillScrnØ displays on text page zero only.
GetColor	Returns BASIC's currently active foreground and background colors.
GetVMode	Reports the current video mode, the currently active display page, the page size, and the number of rows and columns.
HCopy	Similar to BASIC's PCOPY command, except it is designed to work with a Hercules or compatible display adapter in text mode.
HercThere	Reports if QBHERC.COM or MSHERC.COM Hercules graphic support has been loaded into memory.
MakeMono	Converts the colors on a text screen held in an integer array to those suitable for display on a monochrome monitor.

VIDEO (Cont'd)

NAME	*PURPOSE*
MakeMon2	Similar to MakeMono, except MakeMon2 allows you to specify one of four different color conversions.
Marquee	Provides a cute way to display a scrolling message like a movie marquee.
Monitor	Reports the type of monitor display adapter currently active.
MPaintBox	Same as PaintBox, except MPaintBox always turns off the mouse cursor.
MQPrint	Same as QPrint, except MQPrint always turns off the mouse cursor before printing.
MsgBox	Provides a quick and attractive way to display a message with word wrap automatically centered on the screen.
OneColor	Accepts foreground and background color values, and returns them combined in a single byte for use with QuickPak Professional video routines.
PaintBox	Paints any rectangular area of the screen without disturbing the text that is already present.
PaintBoxØ	Same as PaintBox, except PaintBoxØ displays on text page zero only.
PUsing	Smaller code version of BASIC's PRINT USING command.
QPrint	Displays a string very quickly at the current cursor location.
QPrint0	Same as QPrint, except Qprint0 displays on text page zero only.

VIDEO (Cont'd)

NAME	*PURPOSE*
QPrintAny	Provides a simple way for a BASIC program to utilize two monitors at the same time.
QPrintRC	Displays a string very quickly at a specified row and column.
QPWindow	Provides a complete text windowing manager that lets you establish a windowed viewport on the screen and print text in that window, scrolling as necessary automatically.
ReadScrn	Reads characters from the display screen, and stores them in a specified string variable.
ReadScrn0	Same as ReadScrn, except ReadScrn0 reads from text page zero only.
ScrollD	Scrolls any portion of the display screen down a specified number of lines.
ScrollL	Scrolls any portion of the display screen left a specified number of columns.
ScrollR	Scrolls any portion of the display screen right a specified number of columns.
ScrollU	Scrolls any portion of the display screen up a specified number of lines.
SetMonSeg	Allows QuickPak Professional video routines to write to any arbitrary segment.
SplitColor	Accepts a single byte that contains combined foreground and background colors, and returns the separate components.
WindowMgr	A complete window manager that frees the programmer from having to dimension and erase arrays, and keep track of windows as they are opened and closed.

Index

Index