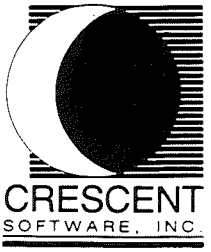

QUICKSCREEN
The Most Sophisticated Screen Management Software Ever Developed

Version 4.00

This manual was written and typeset by Jonathan Waldman with portions excerpted from Don Malin's specifications and from the QuickPak Professional documentation by Ethan Winer.

QuickBASIC is a trademark of Microsoft Corp.

CRESCENT SOFTWARE
32 SEVENTY ACRES
WEST REDDING, CT 06896
(203) 438-5300
Crescent's Support BBS: (203) 426-5958

LICENSE AGREEMENT

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

WARRANTY INFORMATION

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

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

TABLE OF CONTENTS

INTRODUCTION

INTRODUCTION

THANKS!

Thank you for purchasing QuickScreen from Crescent Software!

We have put every effort into making this the finest and most powerful screen building module available. We sincerely hope that you love it. If you have a comment, a complaint, or perhaps a suggestion for another product you would like to see, please let us know. We want to be your favorite software company.

REGISTRATION & UPGRADES

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 insuring that you are notified of possible upgrades and new products. Many upgrades are offered at little or no cost, but we cannot 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. Writing the serial number on the diskette will keep it handy.

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 number you are using. To determine the version number for any Crescent Software product simply display a directory of the original disk. The first thing that appears is similar to:

```
Volume in drive A is QScreen 4.0
```

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

QUICKSCREEN OVERVIEW

QuickScreen is both a text screen design tool and data entry form package. The screen design component allows you to create your own screens — called *display-only screens* — using a sophisticated editor. These screens can be displayed from BASIC programs at any time; they're ideal for on-line help or information screens; and they can be combined to create snazzy slide shows using the included DISPLAY utility.

With QuickScreen, you can also create screens to be used as data entry forms — often referred to as *data entry screens* or *forms*. This powerful ability allows you to quickly design screens which gather information on a prompt-by-prompt basis from a user. Of course, your own BASIC program can control the form and read the values it contains.

Forms are extremely flexible, and data from them can be saved to disk using methods as simple as random access files or as advanced as using data base management utilities available from other vendors of programmer's tools.

QuickScreen's editor can manage screens using any available text screen size, such as 25-, 43-, and 50-line mode, and its forms support a mouse automatically.

Text Mode Screen Designer

To make the task of designing screens as effortless as possible, QuickScreen's interactive editor is mouse-driven and has a variety of features:

- Block operations, such as Cut, Copy, Move, Paint, and Fill are fully-supported.
- An on-screen ruler line is available so that text can be easily measured and aligned.
- A full color chart and ASCII character chart are instantly accessible.
- Line and box drawing are fully-supported.
- Any block of text can be centered.
- Any area of the screen may be filled with any character.
- Any area of the screen may be "painted" to any color.

Data entry screens are created with the help of 19 pre-defined *field types*, such as a zip code or dollar value. Additionally, fields can be further customized:

- Fields can be protected from being changed; they can also be indexed and formatted in any way.
- Fields can support range checks and field calculations based on supplied formulas.
- Unique help messages can be associated with each field in a data entry screen.
- The QuickScreen editor allows you to test data entry at any time, and can readily generate *form definitions* in a variety of formats.

If you are creating such displays as title or help screens, QuickScreen reduces necessary typing by being able to import ASCII data from any DOS text file. On the other hand,

information you provide on a screen can be saved in an ASCII format, which makes it available another editor and to most word processing software.

BASIC Modules

QuickScreen's BASIC modules allow you to manage both display-only and data-entry screens.

DISPLAYING SCREENS

To display screens from BASIC you may use a variety of methods:

- Screens may be displayed directly from disk to video.
- Screens may be loaded from disk to RAM, then displayed to video from RAM.
- Screen library files can be used to combine screens into a single DOS file. Any screen in a library can be displayed when needed.
- Screens saved as object files may be linked when creating an .EXE file so that they are integrated into your application program.
- Screens can be displayed using impressive screen wipe effects.

DATA ENTRY

Managing data entry screens from BASIC is one of the more appealing features of QuickScreen. The supplied BASIC modules let you do the following:

- Control data entry screens automatically based on a *form definition*.
- Handle data entry and movement among fields automatically.
- Perform range checks and field calculations for applicable fields.
- Generate custom help messages for each prompt.
- Support multiple-page forms.
- Support polling so that programmers can take special actions based on the user's activity without having to modify QuickScreen's data entry routines.
- Enable programmers to preset and modify field values, as well as change the cursor position within a form, on-the-fly.
- Support a mouse without additional programming.

ADDITIONAL UTILITIES

QuickScreen is shipped with two useful utilities. The first is a TSR called SCRNCAP — a utility which is used to capture any text screen. These captured screens can be saved and later loaded by the QuickScreen editor or its BASIC modules.

The second is called DISPLAY — a BASIC module used for slide-show presentations. It uses a custom text-file script which you create, and is able to display screens using a variety of features under complete script control.

QuickScreen supports db/LIB®, a third-party add-on product from AJS Publishing. This library provides routines to read and write dBase®-compatible data files, and, when combined with QuickScreen's forms, you can create a powerful database system.

COMPATIBILITY

System

QuickScreen will run on IBM XT, AT, PS/1- and PS/2-class machines and compatibles. DOS 2.0 or above is needed.

Compiler Versions

QuickScreen is available for users of Microsoft BASIC only. This includes QuickBASIC version 4.x; BASCOM, version 6.x; and the BASIC Professional Development System, version 7.x. If you own an earlier version of BASIC we suggest that you contact Microsoft for an upgrade. We will be happy to assist you in making a decision to upgrade.

USING THIS MANUAL

Intended Audience

This manual is designed for users familiar with QuickBASIC and with the concepts of using libraries and compiling to create standalone programs. We have not attempted to unnecessarily duplicate information which is QuickBASIC-related and appears in the QuickBASIC documentation. But we have explained necessary steps for using this product effectively.

Notational Conventions

We have used some variations in type style mainly so that the manual is more clear and more enjoyable to read. The purpose for most type styles is clear (i.e., for topic headings, computer text, etc.), however there are a few uses which may require further explanation:

- Examples of computer program code are printed in a fixed-spaced font. For instance, consider the DO loop below:

```
'pause for a key press
DO
LOOP UNTIL LEN(INKEY$)
    .
    .
    .
```

  Notice also the use of vertical ellipses to convey that more program instructions may follow and the use of BASIC's single-quote REM symbol (') to present comments.

- Examples taken from screen displays are printed as graphic images. A sample screen appears on the next page.

- Pulldown commands are printed in boldface for clarity using this syntax:

  (Menu name) **pulldown menu command**

  For example, "(File) **New Screen...**" refers to the File menu and the **New Screen...** pulldown command. When a menu is discussed alone, the menu name, such as **File**, is in boldface.

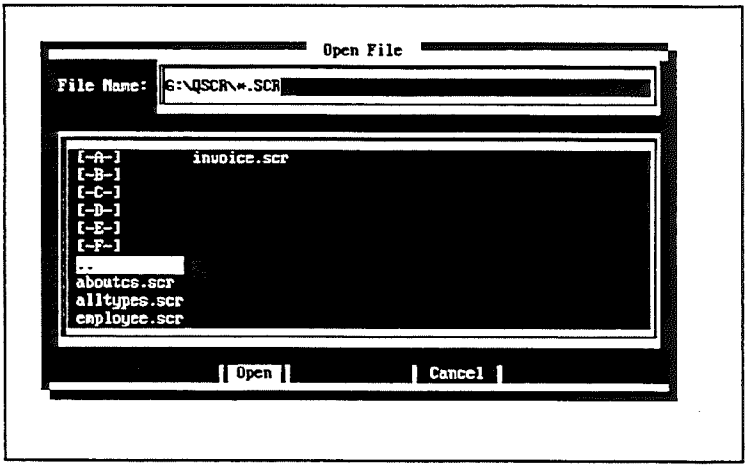

Figure 1: Sample Screen Shot

- DOS directories, file names, acronyms, and BASIC commands are printed in uppercase letters. For example:

  "The SCRNCAP.EXE program is a TSR."

- Instances when the computer input or output may vary (depending on your hardware, software, etc.) are shown in italics. For example, the QuickBASIC Quick Library support module will have a slightly different name depending on its version number. We therefore would refer to such a file as in the example below:

  LINK *PROGNAME*.OBJ,,BQLB*45* /Q

  Notice that not only is "45" italicized, but also the program name, which is specified by the user, is italicized. Italics in the main text usually represents terms which appear in the glossary.

- In some examples there may be optional features in a syntax. These features will be shown in square brackets. For example, the LET statement is optional in QuickBASIC:

```
[LET ]A = 10
```

- Keys on the keyboard are represented as the key name enclosed by angle brackets. Key names are taken from the standard IBM® extended keyboard. For example, <F1> is the F1 function key; <Enter> is the Enter or Return key. Certain keys are mentioned in terms of general function. For example, the direction keys typically include the up-, down-, left-, and right-arrow keys, and sometimes the <PgDn> and <PgUp> keys as well. When the need arises to refer to all of these keys as a group, we will refer to them as direction keys.

TECHNICAL SUPPORT

If you require technical support for QuickScreen, you will need your serial number before calling us at **(203) 438-5300**, between 9:00 a.m. and 5:00 p.m. EST, Monday through Friday. Please gather as much detail as possible about the problem before you call. Be prepared to provide the QuickScreen version number as well as the QuickBASIC version number. We can assist you best when you are able to describe the precise nature of any difficulties.

INSTALLATION

INSTALLATION INSTRUCTIONS

QuickScreen is now distributed using the popular and efficient .ZIP compression format. To help simplify the process of extracting specific files from the archive, we've created a front-end installation program named INSTALL. Upon starting, this program shows the number of bytes the extracted files will occupy, and even allows you to select those files you wish to extract.

To begin installation, place the QuickScreen distribution diskette in a disk drive. Then, log to that drive and type INSTALL (this example assumes the floppy is in A:):

```
C:\]A:
A:\]INSTALL
```

If you have a color graphics card with a monochrome monitor, you can use the /B command-line switch so that the installation program generates monochrome colors:

```
A:\]INSTALL /B
```

If you start INSTALL from a drive and/or directory different from the one containing the INSTALL.EXE program, the current drive and directory is used as the installation destination.

After the program starts, it displays its main screen (see Figure 2), and begins reading any .ZIP files in the same drive and directory as the INSTALL.EXE program. The second line of the screen displays the available function-key commands. Below this is a field where the installation destination drive and path may be specified. To the right of this is a display field where the amount of free disk space is displayed for the specified drive. The bottom-left portion of the screen contains a bar menu where the available .ZIP files are displayed along with the disk space required for installation. The bottom line of the menu displays the total disk space needed to

install all selected files in the menu. The bottom line of the screen displays comments about the currently-highlighted file.

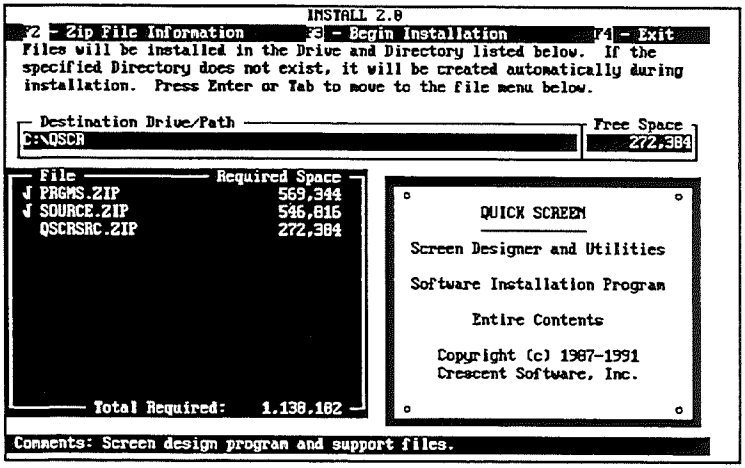

Figure 2: The Installation Start-Up Screen

Several function keys are operable from INSTALL. They are summarized in Table I.

The *Destination Drive/Path* field contains a default drive and path where all selected files will be installed. We suggest that you use \QSCR as the destination directory, if possible, however you can choose any valid DOS path name. If you specify a path which does not exist, it will be created during installation. If you change the drive letter, the amount of free space on that drive is displayed to the right after moving from this field.

KEY	FUNCTION	DESCRIPTION
<F2>	Info On Zip File	Displays the contents (i.e., file names and sizes) of the currently-highlighted .ZIP file. After viewing or selecting individual files, you can press <Esc> to go back to viewing the actual .ZIP files.
<F3>	Begins Install	Once you are satisfied with the selected files, you can start the installation process.
<F4>	Exit	To return to DOS you can exit the installation program.

Table I: Installation Function Keys

The <Tab> or <Shift> <Tab> keys move the cursor between the *Destination Drive/Path* field and the *File Display* box. Within the File display, the <Space Bar> or <Enter> key will toggle a check mark (✓) on or off. You can check entire .ZIP files or you can press <F2> to check individual files within ZIP files. If you are checking individual files, simply press <Esc> to go back to the .ZIP file list. When you are finished, all checked files are installed when <F3> is pressed.

The number of bytes displayed to the right of each file name is the space required on the destination drive to install that file. This number is the uncompressed size of the file — rounded up to the nearest cluster.

After the *Destination Drive/Path* has been specified and files have been selected, you can press <F3> to begin installation. If the total expanded size of all selected files exceeds the available disk

space, you will be asked whether to continue. You can answer "Y" for *Yes* if you are sure there is enough space on the target drive.

> ***If you are re-installing files then there may be more space available than displayed since existing files will simply be overwritten.***

If you are installing to an existing directory you will be asked if you wish to be prompted before existing files are overwritten. We suggest answering *Yes* if you are not sure about overwriting certain files.

After responding to the prompts mentioned above, the screen is cleared and installation continues. As files are installed, messages from the decompression utility, PKUNZIP.EXE, are displayed. If another diskette is required, you will be prompted to change the disk in the source drive. After doing this, you can select new files and proceed as before.

Once all selected files have been installed, the program displays a message indicating a successful installation.

Setting The DOS PATH

In order to make QuickBASIC and its support files available from any directory, the PATH environment variable is set, usually from the AUTOEXEC.BAT file read from your boot drive when your system starts. Setting the PATH merely requires that the directory names containing your compiler files are specified. If you are working on a system where several drives are available, you will want to specify the drive letter as well in your PATH statement.

If you install QuickScreen on C:\QSCR, but your QuickBASIC files are on D:\QB, the PATH variable should be set as follows:

```
PATH = D:\QB
```

If your system uses multiple drive letters for several disk partitions we suggest including the drive letter of each directory in your PATH statement. Doing this ensures that the directory can be properly located.

Some users will need to specify several paths so that QB and BC are properly found. In the Microsoft BASIC Professional Development System, the QBX executable and BC are in separate subdirectories by default: \BC7\BIN and \BC7\BINB, respectively. In this case, you would need to specify both paths:

```
PATH = D:\BC7\BIN;D:\BC7\BINB
```

Notice that multiple paths are separated by semicolons. This way, many drive/directory combinations can be searched:

```
PATH = D:\QB;C:\DOS;C:\WINDOWS;E:\GAMES
```

The sequence in your PATH statement is significant, for it indicates the order in which the paths are to be searched. And of course, the more entries you have, the longer it may take DOS to complete its search.

If you want to see what your PATH is currently set to, simply type "PATH" at the DOS prompt.

The README File

After installing this product, you may want to check for the presence of a README file. Helpful information, as well as additions or changes to this manual, appear in such a file.

After logging onto your QuickScreen directory, simply enter the following DOS command to peruse it (< Ctrl-S > pauses the output until a key is pressed):

```
TYPE README
```

MAJOR FILES OF QUICKSCREEN

The following files are on your distribution diskette; similar file types are grouped together for clarity:

File Name	Description
QSCR.EXE	QuickScreen executable.
SCRNCAP.EXE	TSR screen-capture program.
DBLIBMOD.BAS	db/LIB® support module.
DEMDBLIB.BAS	Demo of db/LIB® support routines.
DEMOANY.BAS	Demo which loads a screen and form.
DEMOCUST.BAS	Demo of random-access and form-editing techniques.
DEMOINV.BAS	Demo containing fields with multiple-choice array, calculated fields, and multi-line text.
DEMOOBJ.BAS	Demo which displays an object-file screen.
DEMOPAGE.BAS	Demo illustrating the use of a two-page form.
DISPLAY.BAS	Slide-show module.
EVALUATE.BAS	Double-precision equation handler.
FORMEDIT.BAS	Form data entry handler.
FORMFILE.BAS	Loads information from form (.FRM) files.
FORMLIB.BAS	Loads information for a particular form stored in a form library (.QFL) file.
SCRNFILE.BAS	Loads and displays a QuickScreen (.SCR) file.

File Name	Description (continued)
SCRNLIB.BAS	Loads and displays screens stored in a screen library (.QSL) file.
QSCALC.BAS	Used for calculated fields in a form.
SCRNDISP.BAS	QuickScreen screen display module.
NOCALC.BAS	Used to "exclude" support for calculated fields.
NOMULT.BAS	Used to "exclude" support for multiple-choice fields.
NONOTES.BAS	Used to "exclude" support for notes fields.
FORMS.LIB	QuickScreen assembler library file for QuickBASIC 4.x or BASIC 6.0.
FORMS7.LIB	QuickScreen assembler library file for BASIC PDS 7.x.
FORMS.QLB	QuickScreen quick library file for QuickBASIC 4.x or BASIC 6.0.
FORMS7.QLB	QuickScreen quick library file for BASIC PDS 7.x.
*.BI	BASIC Include files.
*.CMD	DISPLAY command script files.
*.FRM	QuickScreen Form files.
*.MAK	QuickBASIC Make files.
*.QSL	QuickScreen Library files.

Files above which start with "NO" are used to "exclude" support for certain features serve a similar function as Microsoft's stub files. *Stub files* allow the linking process to work properly when entire modules are referenced but when these modules are either not desired or unavailable. Using stub files has the practical advantage of allowing you to make your .EXE programs smaller.

COPYING AND BACKING UP

Before you start to use the program you should first make a copy of the original diskette and then work with the copy. We know we don't have to tell you this, but reminding you could prevent a very frustrating situation should your distribution diskette become damaged.

QUICK START

QUICK START

If you are familiar with QuickBASIC and add-on libraries, you can get started quickly by running the QuickScreen demonstration programs. Naturally, you'll benefit most if you also examine the commented BASIC source code.

RUNNING THE DEMOS

QuickScreen includes five demonstration programs. You are encouraged to both experiment with these programs and copy statements from the demos into your own programs.

Starting with the simplest among them, the programs are:

- DEMOANY.BAS

  This is a basic example of how to load a form definition file, display a screen, and allow the user to perform data entry. We've called it DEMOANY since the program can work with any standalone screen (.SCR) and form (.FRM) file.

- DEMOOBJ.BAS

  This program shows how to display a screen saved as an object file, and shows how to assign a *wipe type* for the screen image. Further, this program illustrates how DATA statements generated by QuickScreen can be used in lieu of a form definition (.FRM) file.

  This example is particularly useful because it shows a form which contains each QuickScreen field type.

- DEMOCUST.BAS

  This example provides a customer information form, and shows how to store and retrieve information using random file techniques. This program accesses screens and forms which are stored in screen library (.QSL) and field library (.QFL) files, respectively. This program also provides a technique for clearing all fields so that a fresh form can be presented to a user.

- DEMOINV.BAS

  This demonstration shows an invoice form and the special features which make QuickScreen so powerful, such as the use of multiple-choice, calculated, and notes fields. Examples of advanced polling are demonstrated which report the user's activity on the form and make certain fields change their characteristics based on user-defined options. Additionally, this demo shows how a to create a moving highlight bar, so that fields change color as you enter them.

- DEMOPAGE.BAS

  Demonstrates using EditForm to allow data entry on a multi-page form.

- DEMDBLIB.BAS

  This demonstration of an employee information form requires db/LIB®, a product by AJS Publishing, which allows BASIC programmers to read and write dBASE-compatible files.

To run a demonstration program, you can follow these steps:

1. Change to your QuickScreen directory:

```
CD \QSCR
```

2. Start QB or QBX, and make sure to specify an appropriate quick library, such as FORMS.QLB:

```
QB /L FORMS
```

For BASIC 7.x, use the following:

```
QBX /L FORMS7
```

Loading the proper Quick Library furnishes the QuickBASIC environment with required external routines, and allows the demos to run properly.

If you own Crescent Software's QuickPak Professional do not attempt to load PRO.QLB or PRO7.QLB as your Quick Library since these files are too large to be used in this manner. You can create your own, smaller, Quick Library using QuickPak Professional's MAKEQLB utility. Please see page 263 for more information.

3. Select the (File) **Open** menu command, then choose the demonstration you wish to run from the dialog box which appears.

4. Once the program has been loaded into the QuickBASIC environment, you can press <Shift><F5> to run it.

RUNNING THE SLIDE SHOW

A slide show has been created which serves as a product overview for QuickScreen. We encourage you to run this presentation by compiling the DISPLAY.BAS program using the following commands:

```
BC DISPLAY /O;
BC SCRNDISP /O;
LINK DISPLAY+SCRNDISP,,,FORMS /E;
```

BC and LINK are supplied with QuickBASIC. The syntax above assumes that either all files are on the current directory or that a "SET PATH=" command is in effect. If you encounter difficulty in using these commands it may be helpful to consult your QuickBASIC and DOS user's manuals.

Once successfully compiled and linked, you should have a new file called DISPLAY.EXE. At this point you may run the presentation by typing the following command at the DOS prompt:

```
DISPLAY CommandFile [/B]
```

DISPLAY is the name of the slide show program, while CommandFile is the full path and file name of the script (.CMD) file you wish to run. For the demo you may omit the CommandFile parameter since DISPLAY will read DISPLAY.CMD by default. Therefore, typing the following command will be sufficient for now:

```
DISPLAY [/B]
```

The optional /B command-line switch will adjust the demo for use on monochrome systems.

For further information on the DISPLAY utility please see page 259.

THE SCREEN DESIGNER

QuickScreen's editor, referred to as a *Screen Designer*, exists as a compiled executable program called QSCR.EXE. It is possible to start this program immediately from the DOS prompt simply by typing "QSCR".

Once the program begins you should be able to use QuickScreen's intuitive interface right away. If attached to the computer, a mouse may be used. Also, help is available by pressing <F1>.

QuickScreen's user-interface is based on a comprehensive pulldown menu system and dialog boxes. The menu system organizes the major command categories as menu titles and pulldown commands. Dialog boxes query the user for additional information before certain commands are performed.

THE PULLDOWN MENU SYSTEM

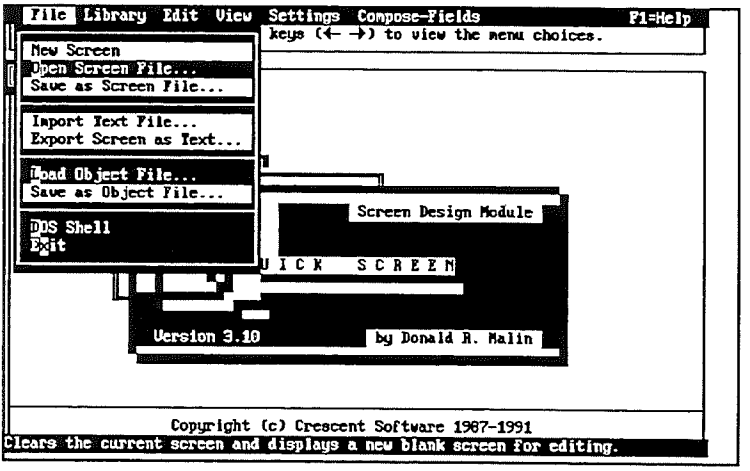

Figure 3: The Pulldown Menu System

Figure 3 depicts QuickScreen's introductory display with an active menu system. The menu system is said to be *active* whenever the menu bar is visible. On the first line of the screen appears the *menu bar*, and under each menu bar option there is a unique *pulldown menu*. Most major commands in QuickScreen are available by accessing this *menu system*, or by using *shortcut keys* which execute a command in one keystroke when the menu system is not active. These shortcut keys will be discussed, but for now it may be useful to know that they appear on Help screens and to the right of certain commands in a pulldown menu.

Menu System Contents

Table II summarizes the pulldown menu features. These features are fully-explored beginning on page 43.

Although it appears on the menu bar, Help has no corresponding pulldown menu. Instead, help may be obtained by selecting the Help menu and pressing <Enter> when the menu bar is active, or by pressing <F1> when the menu bar is inactive.

Using The Menu System

The menu system reflects a user-interface with which you are most likely already familiar. Very much like QuickBASIC's own menu system, QuickScreen's menus may be used with the keyboard or mouse. For this reason, and for the convenience of users not owning a mouse, these discussions are presented separately below.

Menu Name	Pulldown Menu Command Features
File	Specifies load and save options in a variety of formats, executes a DOS shell, and exits QuickScreen.
Library	Manages QuickScreen libraries by providing display, open, and save operations, as well as commands for adding screens to or replacing screens in a library.
Edit	Controls color, draws boxes and lines, fills regions of the screen, manipulates blocks of text, centers text, and repeats the last key pressed.
View	Accesses an ASCII character chart or ruler line, and allows screens to be viewed as they would appear on a monochrome monitor.
Settings	Sets the default paint color, toggles blinking, specifies line border characters, selects the number of lines on the display, and toggles the error beep.
Compose-Fields	Defines, rearranges, and prints data field definitions.

Table II: Summary of Menu Commands

■ Keyboard

The keyboard interface to the menu system is very extensive. In general, the direction keys are used to select a menu and a pulldown command. Once a command is chosen, <Enter> is pressed to execute it, or <Esc> is pressed to abandon the choice. Although it is common to rapidly develop a "feel" for the menu system, the summary in Table III may assist you in learning even faster. Please notice that some keys have more than one function.

When QuickScreen begins it presents the File pulldown menu depicted in Figure 3. At this point you may scan across the menu bar by using the <Left> and <Right> direction keys. Once the desired menu (such as *File*) is selected, you may press the <Up> and <Down> keys until the desired command (such as *New*) is highlighted. To execute the command simply press <Enter>, or press <Alt> plus a highlighted letter corresponding to the command of your choice.

Once the screen editor is in use, the menu system is deactivated and the menu bar will no longer be visible at the top of the screen. At this point you may activate it by pressing <Alt>, which will show the menu bar, or by pressing <Esc>, which will show the last pulldown menu used. Alternatively, you may access a particular pulldown menu by pressing <Alt> plus the first letter of the desired menu name. For example, to access the File pulldown menu, press <Alt-F>.

■ Mouse

When pressed, the right mouse button toggles the menu bar on and off the screen. Once the menu bar is displayed, you may move the mouse cursor over a menu item and press the left mouse button. To choose a pulldown menu command, simply move the mouse cursor over the desired option and click the left mouse button.

Some users may find that "dragging" the mouse produces better results: you may move the mouse cursor over the desired menu bar title and press and hold down the left mouse button. You may then select different pulldown menu commands simply by moving the mouse cursor along the pulldown menu. If the mouse button is released while the mouse cursor is over a command, then the command selected will be executed. You may also drag the mouse cursor along the menu bar to view other pulldown menus before making a selection.

Key(s)	Action on Menu System
<Esc>	Toggles the appearance of the menu bar; cancels a command which has been selected.
<Alt>	Toggles the appearance of the menu bar; highlights menu *hotkeys*.
<Alt-char>	Generates the pulldown menu of the menu bar option starting with the character pressed; executes a command having the hotkey of the character pressed.
char	When a pulldown menu is displayed, pressing a character accesses and executes the pulldown menu command having the same highlighted letter as the character pressed.
<Right/Left>	Selects menu options on the menu bar.
<Up/Down>	Selects commands in a pulldown menu.
<Enter>	Generates the pulldown menu of the menu bar option selected; executes the pulldown command selected.

Table III: Menu System Keyboard Interface Summary

If you do not wish to choose a command after you have activated the menu system simply move the mouse cursor away from the menu system and either click or release the left mouse button.

■ General Comments

Some pulldown menu commands are black while others are gray. The black commands are active; the gray commands are inactive and will produce no effect if selected. Once a screen is loaded into the QuickScreen editor many of the inactive choices will become active. For example, in the (File) **Save as Screen File** command will not be active until a screen has been created or loaded.

DIALOG BOXES

A pulldown menu choice followed by ellipses usually generates a dialog box. Dialog boxes provide an effective way to gather information from the user by making it easy to enter information or to select options.

Figure 4 depicts the *Open File* dialog box and highlights three major dialog box input elements: the *text box*, *list box*, and *command buttons*. The text box accepts a string of characters from the keyboard and allows the entry of a path and file name. The list box presents items in a columnar list and above it shows available files matching the wildcard specification shown in the text box. List boxes may hold many items, and their contents may be scrolled by using the direction keys. The command buttons carry out the designated command when chosen. Thus, pressing <Alt-C> using the example in Figure 4 cancels the dialog box.

What follows is a brief summary of the keyboard and mouse interface to QuickScreen's dialog box input elements.

■ Keyboard

When a dialog box is first presented the cursor will rest on a particular input element. This cursor, or *input focus*, may be moved to the next input element by pressing <Tab>, or to the previous input element by pressing <Shift><Tab>. The input focus may also be directed to a particular input element by pressing <Alt> plus a first letter of a dialog input element label.

Aside from these general directions, there are more specific ways of using each dialog box input element with the keyboard:

- Text box

  The text box accepts text which is typed by the user and it asks for specific information. When text is selected the entire string shown will be cleared when you begin typing. If you wish to edit the string without clearing it then use the direction keys before typing.

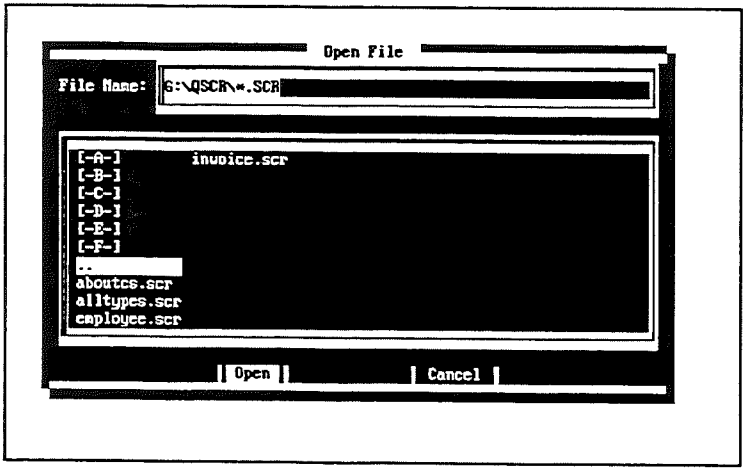

Figure 4: The Open File *Dialog Box*

- List box

  List box items are selected with the direction keys. When the desired item is selected you may press <Enter> to accept it.

- Check box

  The check box is toggled by pressing the space bar.

- Command button

  You may execute the highlighted command button at any time by pressing <Enter>. You may also <Tab> to a particular command button and press <Space> or <Enter>. Further, pressing <Alt> plus the first character of a command button will also execute it.

■ **Mouse**

If you have a mouse you may access dialog input elements by clicking on the desired element. More detailed instructions are summarized below.

- Text box

  The mouse is not useful for entering information into a text box. You may, however, direct the input focus to a text box by clicking on it.

- List box

  A list box item may be selected by double-clicking on it. "Double-clicking" refers to pressing the left mouse button twice in rapid succession. If a list box has more information, its contents may be scrolled by clicking the mouse on a border.

- Check box

  The check box is toggled by clicking it with the mouse.

- Command button

  You may execute a command button by clicking it with the mouse.

MENU ITEM INFORMATION

The purpose of this section is to present an overview of the menu system commands. Menu bar options are organized into subsections; corresponding pulldown commands are listed next to round bullets. The underlined letter in commands discussed below represents the hotkey for the command; these hotkeys appear on-screen as bright letters, and they allow immediate access to an item merely by pressing the highlighted key. Finally, pulldown menu choices followed by ellipses usually present a dialog box for further input.

This section attempts to be exhaustive. Further explanation, however, may be encountered when a particular command is discussed later in the manual.

File Menu

The **File** menu allows you to load and save screens in a variety of formats. These formats include a compressed QuickScreen file, and an object file which may be linked to a QuickBASIC program and combined in a final .EXE file. Screens may also be stored in a QuickScreen Library. The File menu also allows you to execute a DOS shell; and to exit QuickScreen.

- New Screen

  This option is used when you wish to design a new screen from scratch. If a screen is currently being edited and has been altered since last saved, then QuickScreen will prompt you to save before continuing. Once the screen is successfully saved a clear screen with a black background will appear.

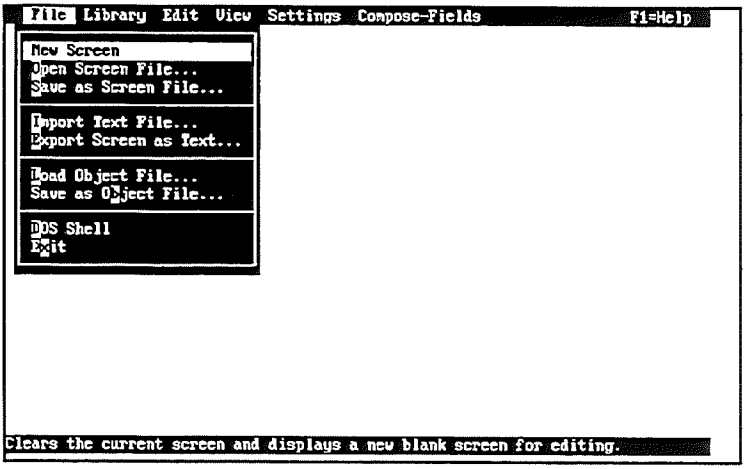

Figure 5: The File Menu

- Open Screen File...

  This selection is used to retrieve a screen which has already been designed and resides on disk. The default extension for QuickScreen screens in the file selection dialog box is .SCR, but this may be changed. Any screen which had been saved using the QuickScreen (File) **Save as Screen File...** command, BSAVE, or QBase, may be retrieved using this option. Further, any special wipes which were specified when the screen was saved will take effect when screen is displayed.

  Crescent Software's QBase screens and associated data fields, as well as BSAVEd screens, are automatically converted as they are retrieved into QuickScreen. When these screens are subsequently saved by QuickScreen, they will be in a new format and will not be readable without using QuickScreen-specific load processes.

- Save as Screen File...

  This option saves the screen currently being edited, and creates a QuickScreen file with the default extension .SCR. If you are saving a form, two check boxes appear in the Save dialog box. The first, *Save TYPE Structure*, creates a .BI file which stores a single BASIC TYPE declaration which encompasses all fields in the form. The second check box, *Save Fields To DATA*, creates a .DTA file which has BASIC DATA statements describing all attributes for each field.

  When saving, you will also be given the opportunity to specify how the screen is to be displayed using one of the wipe types discussed on page 93.

- Import Text File...

  This option is active only when a screen is being used. You may retrieve an ASCII text file from disk and incorporate it into the current screen at the position of the cursor. The maximum number of lines which may be imported in this manner is the same number corresponding to the lines available in the current screen mode. Thus, for a 43-line EGA display only the first 43 lines of the ASCII file will be retrieved. Therefore, this feature is useful for single-screen text files only. Once the text is retrieved it is treated like any other text in the editor.

- Load Object File...

  This selection allows QuickScreen object-saved screens to be retrieved for editing. Any special wipes which were specified when the screen was saved will take effect when screen is displayed.

- Save as Object File...

  This option saves a QuickScreen screen as a binary object file which may later be linked to a QuickBASIC program and combined in a final .EXE file. During the save process you will be asked to specify the wipe type which should be used when the screen is later displayed.

- DOS Shell

  This option executes a second copy of COMMAND.COM and thus allows you to perform DOS functions while QuickScreen remains fully in memory. In order for this feature to work properly the file COMMAND.COM should reside on the root directory of the C: drive, or on the A: drive for systems without a fixed disk. You may specify where the COMMAND.COM file resides by using the COMSPEC command in your AUTOEXEC.BAT file. For example, if COMMAND.COM is on C:\DOS you may place the following command in the AUTOEXEC.BAT file:

  ```
  COMSPEC=C:\DOS
  ```

  As always, if you change AUTOEXEC.BAT you must either run it or reboot your computer in order for the change to take effect.

- Exit

  Exit will end your session with QuickScreen. If the editor contains a screen which has been changed since the last save, then QuickScreen will prompt you to save the screen before exiting.

Library Menu

A library file is a single file which contains one or more screens. Compression techniques are used so that many screens may be stored using the least disk space. The **Library** menu allows a library file to be managed by providing open, save, and display options, as well as commands for adding screens to or replacing screens in the library currently loaded.

- Display Screen... or <F2>

  This option presents a dialog box of all screens in the currently-loaded library. The dialog box allows you to select a screen to be displayed or deleted from the library.

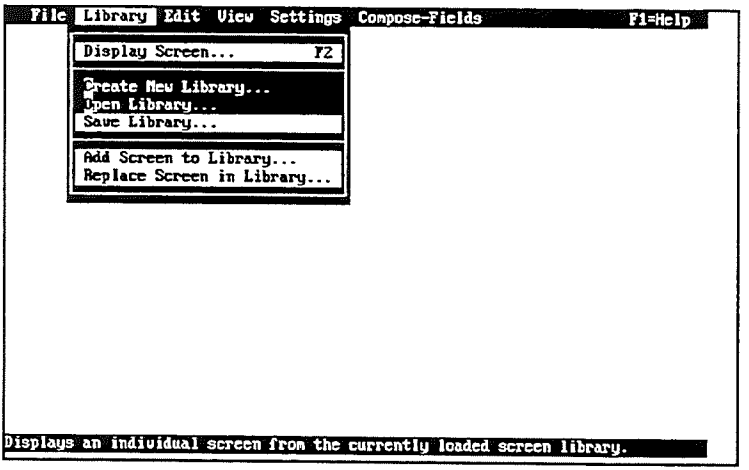

Figure 6: The Library Menu

- Create New Library...

  This option creates a new library as a filename with a .QSL extension. The file created will then be used as the active library for subsequent library commands until a new library is specified.

- Load Library...

  Loads a previously-stored library and makes available to QuickScreen all the screens it contains. If a library is currently active and has changed since last saved, then you will be asked to save it before continuing.

- Save Library...

  This option stores the library file to disk. It creates a .QSL screen library file, and, if forms are present, creates a .QFL form library file as well.

- Add Screen to Library...

  This command adds the screen currently being edited to the library. If the screen already exists in the library you must use the (Library) **Replace Screen in Library...** command, explained next. If the screen is also a form, you will encounter the *Save TYPE Structure* and *Save Fields To DATA* check box options (see the (File) **Save as Screen File...** command for a discussion).

- Replace Screen in Library...

  This option allows the contents of an existing library screen to be replaced with the screen currently being edited. The name of the screen remains the same. If the screen is also a form, you will encounter the *Save TYPE Structure* and *Save Fields To DATA* check box options (see the (File) **Save as Screen File...** command for further discussion).

Edit Menu

The **Edit** menu allows rectangular areas of text, also called *blocks*, to be colored, moved, cleared, and copied. Some of these operations allow text to be transferred to and from a reserved area of memory referred to as the *clipboard*. Further, options are included for replacing existing screen attributes, drawing boxes and lines, and for repeating the last key typed. Methods for setting the painting colors, box- and line-drawing line types are discussed under the Settings menu.

- Paint Block or <Ctrl-P>

  Changes the color of all characters in the currently-selected block of text to the current paint color (specified in the Settings menu). If no block is highlighted then the screen is painted as the cursor is moved.

- Replace Color...

  This option displays two color charts. The first requires you to select the color you wish to replace. The second requires that you select the color to be used for replacement. QuickScreen will then substitute the designated color with the new color on either the currently-selected block or on the entire screen if no block is defined.

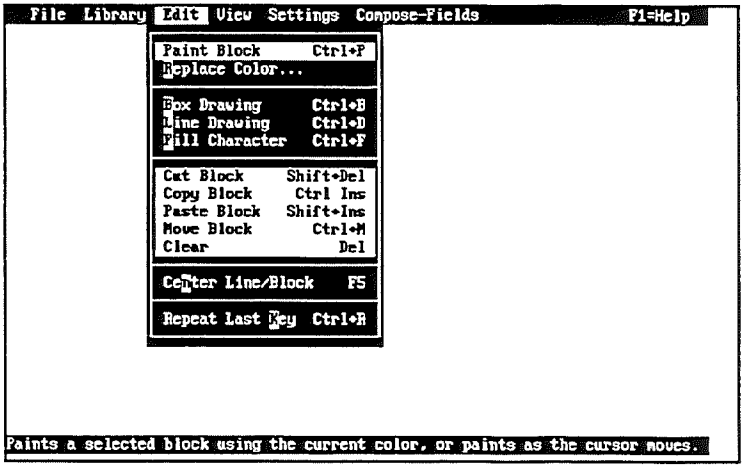

Figure 7: The Edit Menu

- Box Drawing or <Ctrl-B>

  This option toggles the operation of box drawing. When this command is chosen the current cursor position becomes the upper-left corner of a box. The arrow keys may then be used to adjust the opposing corner. If a block of text is currently selected, then choosing the **Box Drawing** command will draw a box the same size as the selected block.

- Line Drawing or <Ctrl-D>

  This command toggles the operation of line drawing. Lines are drawn by using the direction keys. Correct intersection characters will be added by QuickScreen whenever lines overlap one another.

- Fill Character or <Ctrl-F>

  A fill character is used to replace every character in a block whenever this command is chosen. If a block already is defined then characters will be filled within the borders of that block; otherwise text is filled as the cursor is moved and until <Enter> is pressed.

  This option displays an ASCII character chart which allows you to select a character to be used as the fill character.

- Cut Block or <Shift><Delete>

  This command "cuts" the currently-selected block of text by removing it from the screen and placing it into the clipboard. This operation will properly "cut" field information into the clipboard, which makes it possible to quickly erase fields from one location and transfer them to a new one using the (Edit) **Paste Block** operation.

- Copy Block or <Ctrl><Insert>

  This command copies the currently-selected block of text into the clipboard without disturbing the screen. The clipboard contents can then be "pasted" anywhere on the screen using the next command. Like the (Edit) **Cut Block** command, this operation will handle fields which are in a selected block of text.

- Paste Block or <Shift><Insert>

  This command "pastes" the information contained on the clipboard at the current cursor position and places the editor in block-move mode. Once the block has been properly positioned using the direction keys you may press <Enter> to actually make it a part of the screen. If <Esc> is pressed then the pasted block will be removed without disturbing the contents of

the screen. All paste operations transfer the clipboard contents (and any including field information) to the screen, and paste operations may be performed repeatedly without needing to refresh the clipboard contents.

- Move Block

  This option allows a selected block of text to be moved using the direction keys. The <Tab> and <Shift><Tab> keys are also active.

- Clear or <Delete>

  This command clears the currently-selected block of text. This option does not transfer any information to the clipboard and, therefore, should be used with care.

- Repeat Last Key or <Ctrl-R>

  This command repeats the last character entered. This is useful for repeatedly entering non-keyboard characters, such as the graphics characters available as part of the extended ASCII character set. This command may also minimize repeated access of the ASCII character chart (in the View menu).

View Menu

The View menu displays an ASCII character chart, ruler line, and allows color screens to be viewed as they would appear on a variety of monochrome scenarios. This lets you ensure that your chosen color combinations will be visible on monochrome systems.

- ASCII Chart or <Ctrl-A>

  This command generates an ASCII character chart. A character is selected using the direction keys, and then chosen by pressing <Enter>.

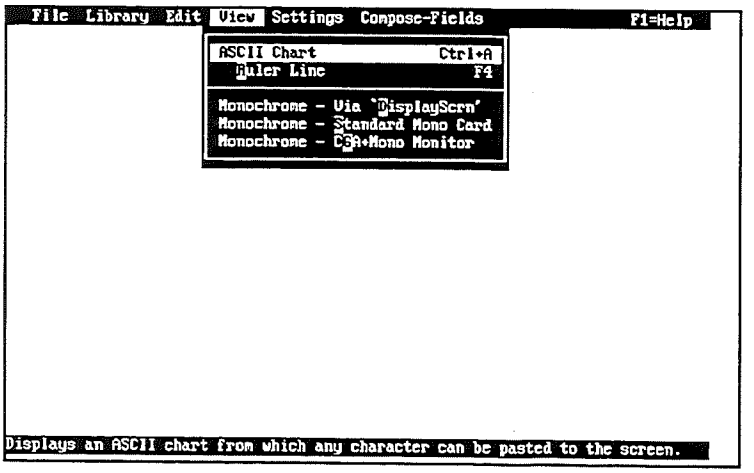

Figure 8: The View Menu

- Ruler Line or <F4>

  This option displays the current cursor position in terms of x- and y-coordinates, and reports the decimal values of both the color and character under the cursor.

- Monochrome - Via DisplayScrn

  This option displays the screen as it would appear on a monochrome screen using the DisplayScrn subprogram.

- Monochrome - Standard Mono Card

  This option displays the screen as it would appear on a display which uses a standard monochrome card.

- Monochrome - CGA+Mono Monitor

  This option displays the screen as it would appear on a monochrome screen attached to a CGA (Color Graphics Adapter).

Settings Menu

The Settings menu allows global settings to be changed. You may specify the default painting color, whether painted characters are to blink, drawing line types, the screen size, and whether the error beep is audible.

- Painting Color...

  This command displays a color chart which lets you select the color to be used for subsequent paint operations.

- Blinking Off

  This option is toggled whenever selected. When blinking is on, painted text will flash on and off; when off, painted text will not flash.

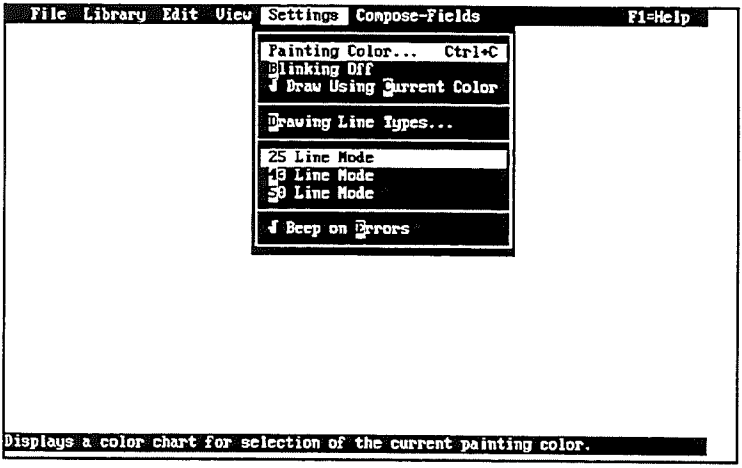

Figure 9: The Settings Menu

- Draw Using Current Color

  A check mark appears next to this option when it is active, where it causes all boxes, lines and text to be drawn using the current painting color. If this option is not active, then the colors already on the screen are used.

- Drawing Line Types...

  This option specifies border characters for boxes, and the character used for line draw. It is unique because it presents a new menu bar which contains options for the left, right, top, and bottom borders. You may then specify particular border characters to be used, or you can set all four border characters to the same character by using the *All* menu option.

- 25 Line Mode

  This command switches the current screen mode to display 25 horizontal lines.

- 43 Line Mode (displayed only for compatible hardware)

  Supported in EGA/VGA only, this option switches the current screen mode to display 43 horizontal lines.

- 50 Line Mode (displayed only for compatible hardware)

  Supported in VGA only, this option switches the current screen mode to display 50 horizontal lines.

- Beep on Errors

  This option toggles the operation of the error beep. When on, this option will cause QuickScreen error messages to be accompanied by an audible beep. A check mark will appear on the left of this option when the error bell is enabled.

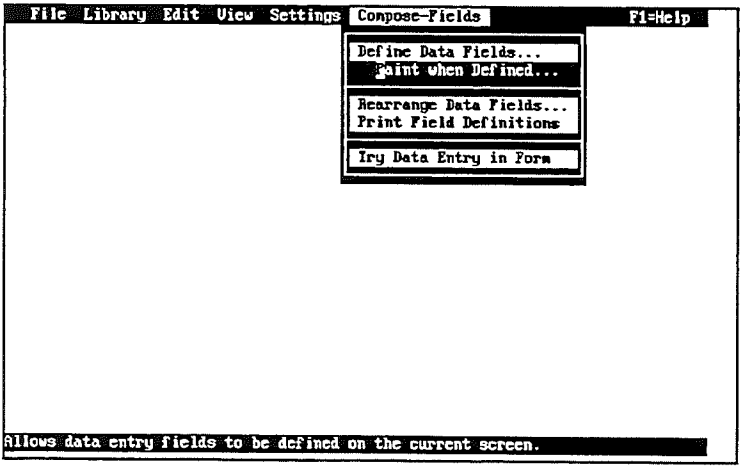

Figure 10: The Compose-Fields Menu

Compose-Fields Menu

QuickScreen has the ability to manage a large group of pre-defined fields. This feature allows data entry screens to be quickly and easily developed.

- Define Data Fields...

  This command defines fields to be added to the current screen thereby creating what ultimately is to be a form. Each field may be defined to accept very specific information. For example strings, numbers, or dates.

- Paint when Defined

  When checked, this option uses the current color to paint each entry fields as they are defined. If unchecked, fields become the color of the screen area over which they are placed.

- Rearrange Data Fields...

  This selection allows previously-defined data fields to be either moved from their current tab-order or deleted from the form.

- Print Field Definitions...

  This option creates a hardcopy of all field definitions for the current form and thus serves as a handy documentation utility. This printout information is sent to the device at LPT1:.

- Try Data Entry in Form

  The QuickScreen editor allows you to test how a form will operate when run from QuickBASIC. After choosing this command, the QuickScreen environment will allow you to enter data in any field of the form. When you have finished, simply press <Esc> in order to return to the editing mode.

FIELDS

QuickScreen has the powerful ability to create fields which gather user input when a screen is used from QuickBASIC. A screen with field definitions is called a form. When using forms from QuickBASIC, field information may be passed to and from a calling program.

In order to understand the use of fields in QuickScreen we first present to you the many field types available. Next, the **Field**

Settings dialog box is discussed, since each field may be customized to a certain extent. The next section discusses the power in using numeric formulas. And last, the entire process of defining fields is considered.

Field Types

The field type describes the data which is to be entered at a particular field. For example, there is a **Social Security Number** field type which accepts numerical information only in the form *###-##-####*. QuickScreen contains built-in logic for each field type, making additional formatting or syntax-checking by the calling program unnecessary.

Certain field types are fixed-length and generate mask characters. These are simply characters, such as the dashes in a social security number, that help to format a field on the screen. QuickScreen inserts mask characters for you and skips over them when the field is being used in a form.

What follows is a description for each field type available in QuickScreen.

- String

  Alphanumeric characters, both upper- and lower-case, are accepted. This field is useful for collecting any general single-line string information.

- Proper String

  Alphanumeric characters are accepted, and the first letter in each word is automatically capitalized during data entry. This field is useful for names and addresses.

- Upper Case String

  Alphanumeric characters are accepted, and all characters are automatically converted to upper-case during data entry. This field is useful for abbreviations and part numbers.

- Numeric

  Numeric characters are accepted, but are treated as a string. This field is useful for telephone numbers and zip codes.

- Multi Line Text

  Several lines of alphanumeric characters are accepted. This field is ideal for notes.

- Logical

  A pre-defined "true" or "false" character is accepted. This field is therefore useful for "yes/no" or "check/uncheck" fields.

- Integer

  An integer number in the default range -32768 to 32767 is accepted. This field is useful for entering an integer value such as the quantity of items in a sale.

- Long Integer

  A long-integer number in the default range -2,147,483,648 to 2,147,483,647 is accepted. This field is useful for entering a very large integer number.

- Single Precision

  A single-precision number in the default range ±3.402823 E+38 to ±1.401298 E-45 is accepted. This field is useful for general decimal numbers.

- Double Precision

  A double-precision number in the default range ±1.7976931 D+308 to ±4.940656 D-324 is accepted.

- Currency

  An amount of money in the double-precision range is accepted. The currency symbol for the field may be defined so that dollars, yen, or other currencies may be used.

- Date MM-DD-YYYY

  A date in the default range 01-01-1900 to 01-01-2065 and in the American format (MM-DD-YYYY) is accepted. The dash ("-") mask character is added automatically.

- Date DD-MM-YYYY

  A date in the default range 01-01-1900 to 01-01-2065 and in the European format (DD-MM-YYYY) is accepted. The dash ("-") mask character is added automatically.

- Phone Number

  A phone number in the form *(###) ###-####* is accepted. The parentheses and dash mask characters are added automatically.

- Zip Code

  A zip code in the form *#####-####* is accepted. The dash mask character is added automatically.

- Social Security Number

  A social security number in the form *###-##-####* is accepted. The dash mask characters are added automatically.

- Relational

  Allows you to specify what file and which field in that file is to be used for the current field. Although QuickScreen does not process these fields automatically, relational fields allow a calling program to access data in other form files.

- Multiple-Choice Array

  Presents a vertical menu of choices. This menu is defined in a string array by the calling program and is generated whenever this field is accessed.

- Command Button

  Command buttons return a single user-defined key code whenever they are selected in a form. Common example command buttons are "OK" and "Cancel", which typically return the key codes for <Enter> and <Esc>, respectively. Although the button fields may show any words and return any keycode you choose, they always return a single value.

  A command button is activated by clicking on it with a mouse, moving the text cursor to it and pressing <Enter>, or pressing the key which has been assigned to it.

Command buttons are the only fields in QuickScreen which do not have any data and thus do not occupy space in a data file.

Table IV presents a summary of QuickScreen's nineteen field types. The field type names appear in the **Field Types** dialog box, and are encountered when defining fields using the steps outlined on page 101.

Field Settings

Each field has a group of user-assigned attributes collectively called *Field Settings*. These attributes are set using dialog boxes, one of which is depicted below.

Figure 11, page 66, shows an example *Field Settings* dialog box which is generated for the String field type. It is important to realize that certain fields will generate dialog boxes containing slightly different options. For example, the Integer field type dialog box queries for a range of numbers which are to be accepted.

The following pages present an alphabetized list of input elements which are encountered for the *Field Settings* dialog box.

STRING	
String	Alphanumeric characters.
Proper String	Alphanumeric characters; the first letter in each word will be capitalized.
Upper Case String	Alphanumeric characters; each alphabetic character will be unconditionally capitalized.
Numeric	Numeric characters only.
Multi Line Text	Alphanumeric characters; several lines of text may be entered and edited.
NUMERIC	
Integer	-32768 to 32767
Long Integers	-2,147,483,648 to 2,147,483,647
Single Precision	±3.402823 E+38 to ±1.401298 E-45
Double Precision	±1.7976931 D+308 to ±4.940656 D-324
Currency	±1.7976931 D+308 to ±4.940656 D-324
GENERAL	
† Logic	A "true" or "false" character.
† Date (American)	MM-DD-YYYY (MM: month; DD: day; YYYY: year) 01-01-1900 to 01-01-2065
† Date (European)	DD-MM-YYYY 01-01-1900 to 01-01-2065
† Phone Number	(###) ###-####
† Zip Code	#####-####
† Social Security No.	###-##-####

Table IV: Available Field Types

Fields designated with a dagger "†" generate mask characters.

SPECIAL	
Relational	Relates current field to a field in another file.
Multiple Choice Array	Presents a vertical menu of choices.
Command Button	Returns a preassigned keycode in Frm.Keycode.

Table IV (continued): Available Field Types

- Currency Symbol

  The symbol to be used when displaying currency values, such as "$" for dollars or "¥" for Yen.

- Decimal Places

  Number of digits to be displayed after the decimal point. The value of the number presented will be rounded based on the internal decimal representation.

- "False" Character

  The character to be accepted as "false" in a logical field. The space bar will toggle "false" and "true" characters.

- Field Name

  A unique name, up to eight-characters in length, for the current field. This name may be used as a variable or a string, and can appear in field formula calculations.

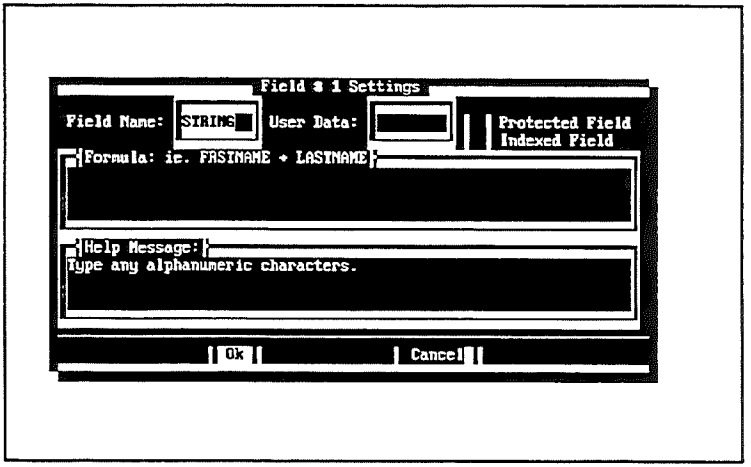

Figure 10: The Compose-Fields Menu

Realize that a constant, function, or operator name such as "ARCSIN" represents a reserved word and should not be used as a field name unless it will not be used in calculated fields.

- Formula

Formulas are considered to be string formulas if the example presented atop the Formula text box is:

```
FRSTNAME + " " + LASTNAME
```

Formulas are considered to be numeric formulas and define calculated fields, if the example presented atop the Formula text box is:

```
QUANTITY * UNITCOST + 1.3
```

While string formulas are limited to string concatenation, numeric formulas may use a variety of functions, constants, and operators. Please see the next section called Numeric Formulas for more detail.

- Help Message

  The text specified in the Help Message text box represents field-sensitive help which is presented as a window on a form when <F1> is pressed.

- Indexed Field

  This input element should be checked if the current field is to be indexed.

  Indexed fields are not processed by QuickScreen, however this information can be used by the calling program to determine which fields in a form are to be indexed.

- No Formatting

  This input element should be checked if a number is to be displayed as it was entered by the user. If this option remains unchecked, then numbers are right-justified.

- Protected Field

  This input element should be checked if the current field is to be protected from being modified by the user. In effect, the field will be a "display-only" field.

- Range

  Specifies the upper and lower limits for numeric input, or the date range for date input between which entered values must fall before being accepted in a form.

- Relational Field

  This option allows you to specify a file and field name for a relational field. A calling program may use this information to form a relational link to data in another file on disk.

  Relational fields are not processed by QuickScreen, however the related file name and the related field number are available to the calling program. See the discussion of the FieldInfo TYPE on page 118 for further information.

- "True" Character

  The character to be accepted as "true" in a logical field. The space bar will toggle between "true" and "false" characters during operation.

- User Data

  This field holds eight characters, and can be used for any purpose whatsoever. For example, you could store a flag here which would later be accessed by your own programs. Such a flag could tell whether the field has some type of security feature installed — such as requiring a password. The field could also hold the password itself! You could also use this field to hold a number which points to information contained in a secondary file. This would make comprehensive on-line help possible by calculating the number as an offset into a large help file.

Numeric Formulas

For a numeric formula you may use the same syntax you would normally use in BASIC. For example, you may calculate the total price for an item which costs PURCHASE dollars and is taxed at 6% as follows:

```
PURCHASE + (PURCHASE * .06)
```

Notice that parentheses may be used to group parts of a formula to develop a mathematical hierarchy. Although this is a very simple example, the formula can use a variety of constants, functions, and special operators. These are summarized below:

- Constants

  A few constants may be used in a formula expression. The field constants available are summarized in Table VII.
- Functions

  QuickScreen supports many functions available in BASIC as well as several that are not. The available functions are summarized below in Table VI.

- Math Operators

  Table VIII presents the QuickScreen math operators which may be used just like BASIC's.

- Relational Operators

  Relational operators, presented in Table IX, compare numerical values. When true, the formula expression evaluates to -1; when false the result is 0.

Name	Value
ARCSINH	Inverse Hyperbolic Sine
ASCCOSH	Inverse Hyperbolic Cosine
ARCTANH	Inverse Hyperbolic Tangent
ARCSECH	Inverse Hyperbolic Secant
ARCCSCH	Inverse Hyperbolic Cosecant
ARCCOTH	Inverse Hyperbolic Cotangent
ARCSIN	Inverse Sine
ARCCOS	Inverse Cosine
ARCSEC	Inverse Secant
ARCCSC	Inverse Cosecant
ARCCOT	Inverse Cotangent
SINH	Hyperbolic Sine
TANH	Hyperbolic Tangent
SECH	Hyperbolic Secant
CSCH	Hyperbolic Cosecant
COTH	Hyperbolic Cotangent
CSC	Cosecant
COT	Cotangent
SEC	Secant
SIN	Sine
COS	Cosine
TAN	Tangent
ATN	Inverse Tangent
LOG	Natural Log
EXP	Exponent
SQR	Square Root
CLG	Common Log
!	Factorial
ABS	Absolute Value

Table VI: QuickScreen Field Functions

Name	Value
PI	3.14159265358979323846
E	2.718281828459045

Table VII: QuickScreen Field Constants

- Boolean (Logical) Operators

You may use Boolean operators in a numerical formula to evaluate expressions to true, which is -1, or false, which is 0. Table X below summarizes QuickScreen's Boolean operators.

Operator	Purpose
^	Power
*	Multiplication
/	Division
\	Integer Division
MOD	Modulo
+	Addition
-	Subtraction

Table VIII: QuickScreen Field Math Operators

Operator	Purpose
=	Equal
>	Greater than
<	Less than

Table IX: QuickScreen Field Relational Operators

Boolean Operators
AND
NOT
OR

Table X: QuickScreen Field Boolean Operators

CREATING TITLE SCREENS

QuickScreen's editing features make it very easy to create screens and embellish them with character graphics and color. There are three main components to the QuickScreen editor: the environment settings, editing tools, and form data entry fields.

The following sections will refer to pulldown menus in terms of the commands available on them. It may be necessary to turn to page 43 if more detail is needed about a specific command.

THE ENVIRONMENT SETTINGS

The QuickScreen environment settings refer to global features which may be controlled. These options are:

- Line & box draw characters
- Paint color
- Blinking
- Screen size
- Error beep

Line & Box Draw Characters

Whenever you draw lines or boxes in the QuickScreen editor you will be accessing a defined set of border characters. These characters may be modified by using the (Edit) **Drawing Line Types...** command. When this command is executed you may select a specific character to be used for the each side of the box.

Line drawing accesses only the character defined as the left border character in the (Edit) Drawing Line Types command.

Paint Color

A default paint color is defined by using the (Settings) **Painting Color** command.

Blinking

Blinking causes foreground text to flash on and off. Blinking is toggled using the (Settings) **Blinking On/Off** command. When blinking is on, any subsequent paint operations will cause the selected foreground characters to blink. Similarly, when blinking is off, any painted foreground characters will cease to blink.

Screen Size

The number of lines on the display may be changed, depending on your video hardware, by accessing the (Settings) **Line Mode** command.

Error Beep

When QuickScreen generates an error message it can also generate an accompanying error beep. This error beep may be toggled on or off using the (Settings) **Beep on Errors** command. If on, a beep will be sounded with all error messages. If off, no beep will be generated by QuickScreen.

GENERAL EDITING

Once the environment settings are considered you may begin to design a screen. QuickScreen's editor provides many useful editing tools:

- Ruler Line
- Block Operations
- Box Drawing
- Line Drawing
- Painting
- Character Editing
- Line Editing

Ruler Line

If the (View) **Ruler Line** command is used, or <F4> is pressed, then a ruler line will be toggled to and from the screen. The ruler line is shown in Figure 12.

The ruler line displays the current cursor position in terms of x- and y-coordinates. The offset shows the position of the cursor relative to its location when the ruler was first displayed. Thus, the Row and Col values show the absolute cursor coordinate; the Offset shows the relative cursor coordinates.

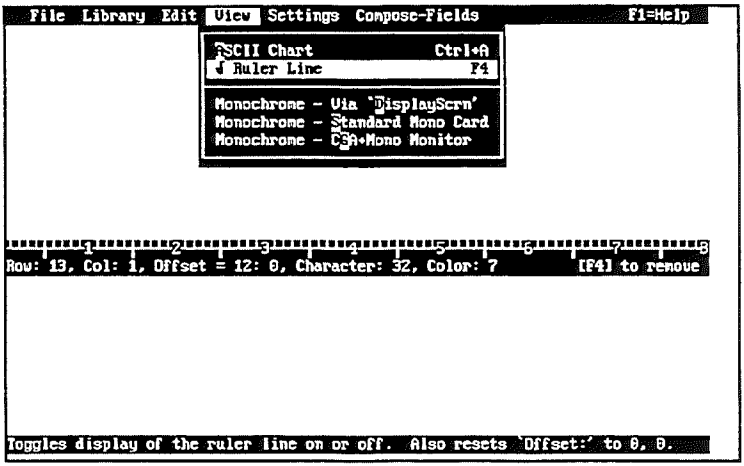

Figure 12: The Ruler Line

In addition, the ruler line gives the decimal ASCII and color values of the character above the cursor.

Block Operations

Function	Pulldown Command	Key Sequence
Marking	N/A	<Shift><Direction Key>
Moving	(Edit) Move Block	<Ctrl-M>
Centering	(Edit) Center	<F5>
Deleting	(Edit) Clear	<Delete>
Pasting	(Edit) Paste Block	<Shift><Insert>
Painting	(Edit) Paint Block	<Ctrl-P>

Table XI: Summary of Block Operations

Blocks are defined as rectangular areas, and are created when the <Shift> key is depressed while the direction keys are being used. In order to define a block the cursor is first positioned using the direction keys. This initial cursor location is the "starting point", and will be one of the corners of the final block, depending on how the block is created. For example, if <Shift> is pressed and the cursor is moved to the right and down from its starting point, then the starting point will be the upper-left corner. If, however, <Shift> is pressed and the cursor is moved to the left and up, the starting point will be the lower-right corner. Blocks may consist of a single character or the entire screen.

Once a block is defined, the text and fields it encompasses is said to be *selected* (selected text is always displayed in black and white). Understand that because block operations take into consideration the fields present in selected text, it's easy to manipulate fields on a form.

In order to move a block you may use the (Edit) **Move Block** command, or you may press <Ctrl-M>. Blocks may then be moved around the screen without disturbing unselected text by using the direction keys. The entire block is centered horizontally by pressing <F5>. When a final position is established, you may press <Enter>. If you wish to cancel the block move operation anytime before <Enter> is pressed, you may use <Esc>.

Blocks may be cut and assigned to the clipboard using <Shift><Delete>, or cleared without disturbing the clipboard using only <Delete>. The contents of the clipboard are transferred to the screen at the position of the cursor by pressing <Shift><Insert>, which immediately reveals the block and sets the block-move mode. The block may then be moved around the screen. When the block is positioned properly you may press <Enter>. If you wish to remove the block simply press <Esc>.

An entire block may be painted the default paint color by using the (Edit) **Paint Block** command or by pressing <Ctrl-P>.

The block operations discussed are summarized at the beginning of this section in Table XI.

Box Drawing

Boxes are drawn in one of two ways. The first method works just like defining a block. In other words, the cursor must be placed somewhere on the screen. This will be the box's starting point. Box drawing is then enabled using the (Edit) **Box Drawing** command or by pressing <Ctrl-B>. At this point the box may be drawn by moving the cursor around the screen with the direction keys. Press <Enter> to keep the box or <Esc> to remove it; the cursor will be returned to the starting point in either situation.

The second method for drawing a box requires that a block first be defined. If you wish to draw a box matching the size and position of a currently-defined block press <Ctrl-B>.

Recall that the border style of a box may be changed by accessing the (Settings) Drawing Line Types command.

Line Drawing

Lines are drawn using the direction keys after choosing the (Edit) **Line Drawing** command, or by pressing <Ctrl-D>. If a line overlaps an existing line, an appropriate cross-section character will be added by QuickScreen. In order to accept the lines drawn press <Enter>. Otherwise press <Esc> to cancel.

Recall that the line draw character may be changed by accessing the (Settings) Drawing Line Type command.

Painting

QuickScreen fully supports color video adapters and contains several features for manipulating color. The color of a character on the screen may be altered using a technique called *painting*. QuickScreen allows you to define a default paint color and to globally replace one color with another. Further, single characters and entire blocks may be painted using one keystroke.

- Replacing Color

  A particular color may be altered by using the (Edit) **Replace Color** command. You will first be asked for the color you wish to change; then you will be asked for the new color.

- Painting Characters

  The character above the cursor may be painted the default color by using the <Ctrl><Right> key combination. To unpaint text you may use the <Ctrl><Left> key combination.

- Painting Blocks

  A block of text may be painted by first selecting it, then by using the (Edit) **Paint Block** command. Alternatively, you can simply press <Ctrl-P>.

CHARACTER EDITING

The QuickScreen editor allows you to manipulate single characters using a variety of commands.

Deleting

The character above the cursor may be deleted by using <Delete>. The character to the left of the cursor may be deleted by using <Backspace>.

Inserting

The <Insert> key serves as an insert toggle. When QuickScreen's cursor is a large rectangle, the editor is in insert mode and characters entered will push existing characters to the right. If the cursor is small and flat then the editor is in overwrite mode, and any existing text under the cursor will be erased while typing.

ASCII Character Chart

A single character from the complete IBM® PC character set may be chosen when the (View) **ASCII Chart** command is used, or when <Ctrl-A> is pressed. Simply use the direction keys until the desired character is blinking. To accept the character press <Enter>, otherwise press <Esc>.

An extended ASCII character chart is provided for your convenience in Appendix D.

Repeating The Last Character

The last character typed may be repeated by using the (Edit) **Repeat Last Key** command, or by pressing <Ctrl-R>. This feature is useful when the character you are trying to type is not a keyboard character.

LINE EDITING

Just like characters may be managed, so can entire lines of text. This section provides detail on commands which operate on full lines rather than single characters.

Deleting

An entire line of text may be deleted from the screen and copied to the clipboard by pressing <Ctrl-Y>. When text is deleted in this manner, the remaining information on the screen is shifted up one line. If <Ctrl-Y> is inadvertently used you may "undo" its effect by "pasting" the contents of the clipboard in the usual manner: press <Shift><Insert>.

Inserting

A blank line may be inserted at the position of the cursor by pressing <Ctrl-N>. This will cause remaining information on the screen to be shifted down one line.

There is no way to recapture information that has scrolled beyond the last line of the screen.

Centering

The current line may be centered by pressing <F5>. You may center an entire selected block of text also by pressing <F5>.

VIEWING MONOCHROME SCREENS

When a screen is created you may want to see how it will appear on a monochrome display adapter. QuickScreen offers the unusual ability to design screens on a color monitor and view them on one of several monochrome modes. It is a fact that certain color combinations render characters invisible or unreadable on monochrome systems, and this feature may assist you in creating monochrome-compatible screens.

QuickScreen provides three different monochrome scenarios. These are available in the View menu and are addressed below.

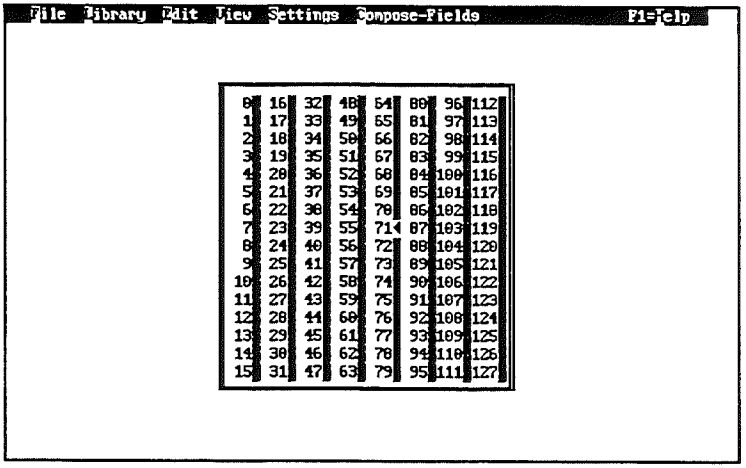

Figure 13: The Color Palette Chart

Figure 13 shows the color palette which is displayed when using the (Settings) **Painting Color...** menu command. It is shown for your reference only, and, unfortunately is only in black and white.

However, in the discussions which follow, we'll refer to numbers in this grid to tell how colors are remapped in various monochrome settings.

Monochrome - Via "DisplayScrn"

QuickScreen includes a subprogram called *DisplayScrn* which may be used to convert a full-color screen to color combinations which will appear readable when shown on a monochrome display. When this option is chosen while in the QuickScreen environment you will be able to see how the current screen would appear if it were displayed using the DisplayScrn routine. This routine maps colors as follows:

- A black foreground color on a non-black background displays as inverse video (black on black will be invisible).
- A normal foreground color displays a normal-intensity on a black background.
- A high-intensity foreground color displays a high-intensity on a black background.
- Blinking foreground colors will blink as expected.

When monochrome colors are shown using DisplayScrn, notice that black on black (color 0) is invisible and that other colors on the top row (values 16 through 112) appear as reverse-video attributes. Colors in the bottom-half of the chart (starting with the row containing values 8 through 120) appear as high-intensity (bright) attributes.

Monochrome - Standard Mono Card

This option shows how the current screen would appear if generated on a standard monochrome card, such as the original IBM® Monochrome Display Adapter (MDA), or some cards which use the Hercules® tradename and which attempt to emulate the MDA.

This routine maps colors as follows:

- A black foreground color on a non-white background is invisible.
- A black foreground color on a white background is in inverse video.
- A normal foreground color displays a normal-intensity on a black background.
- A high-intensity foreground color on a non-white background displays a high-intensity on a black background.
- A high-intensity foreground color on a white background displays as a high-intensity foreground in inverse video.
- Blinking foreground colors will blink as expected.

When monochrome colors are shown using a standard monochrome card, notice that colors from 0 to 96 on the color chart are displayed as black on black and are therefore invisible. Color 112 is displayed as black on white, and appears as inverse video. A large portion of the color chart, from the row containing values 1 through 113 and ending with the row containing values 8 through 120 is displayed as normal-intensity foreground on a black background. The bottom-left rectangle of the chart, starting with the row containing values 9 through 105, is displayed as a high-

intensity foreground on a black background. Finally, the rightmost column, containing values 121 through 127, is displayed as a high-intensity foreground and in inverse video.

Monochrome - CGA+Mono Monitor

Some users have a monochrome (green or amber, usually) screen attached to a Color Graphics Adapter (CGA). This option shows how the current screen will appear on such hardware.

This routine maps colors as follows:

- A black foreground color on a black background is invisible.
- A black foreground color on a non-black background is in inverse video.
- A foreground color on a black background color is normal-intensity on a black background.
- A foreground color on a color background is both invisible and in inverse video.
- A high-intensity foreground color on a black background is a high-intensity foreground on a black background.
- A high-intensity foreground color on a non-black background is a high-intensity foreground in inverse video.

When displaying screens using a CGA and a monochrome monitor, notice that color 0 is displayed as black on black, which renders it invisible, as expected. The row containing values 16 through 112 is displayed as black on white, and appears as inverse video. The column containing values 1 through 8 is displayed as normal-intensity on a black background. The rectagular area starting with the row containing values 17 through 113 and ending with the row

containing values 24 through 120 is completely invisible, and appears as inverse video only (characters are actually the same color as their background and therefore can't be seen). Finally, the bottom-right rectangular area starting with the row containing values 25 through 121 and ending with the row containing values 31 through 127 appears as high-intensity foreground characters and in inverse video.

SAVING SCREENS

QuickScreen can save screens in a variety of formats. These include the QuickScreen format, object-file format, and as images in a QuickScreen Library file. If a block is currently-highlighted when you save, then only the highlighted portion of the screen is saved. Otherwise, QuickScreen starts at the upper-left corner of the screen and uses the first character which is not a space and which has a color attribute other than 7 (white on black) to define a block to be saved. This process helps to compress images which do not occupy the entire display.

QuickScreen Files

The QuickScreen format uses a run-length encoding algorithm to compress the screen size on disk. The current screen is saved in this format using the (File) **Save as Screen File...** command. This generates the dialog box shown in Figure 14.

When screens are saved using the QuickScreen and Object file format, you may specify the way the screen is to appear when displayed. This is called the screen's *wipe type* ("wipe" is a term used by video engineers). The variety of wipe type is summarized in Table XII.

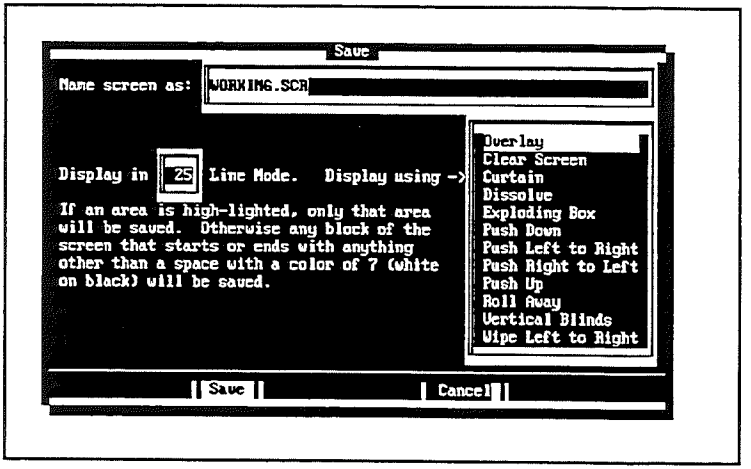

Figure 14: The Save *Dialog Box*

Object Files

Screens may be saved as object files using the (File) **Save as Object File...** command. These screen files are linked with a main program to create a standalone executable file. Unlike other screen formats, object files need not be present on the disk once they are linked to a program.

A wipe type may be selected when saving screens as object files.

Text Export

Screens may be saved as text using the (File) **Export Screen as Text...** command. Text files consist of ASCII characters only and therefore contain no color, special control codes, header

information, wipe type, or screen mode information. One advantage, however, is that ASCII screen can be read by nearly all editors or wordprocessors.

QuickScreen Library Files

Screens may be saved to a QuickScreen library. Please see the last section in this chapter entitled *Using QuickScreen Library Files* for more information.

RETRIEVING SCREENS

Screens created in the QuickScreen environment, which include QuickScreen screens, Object screens, and screens in Libraries, are easily retrieved into the environment. Any file which is ASCII may be imported to QuickScreen's environment. In addition, screens and forms designed using Crescent Software's QBase may be read and converted to a QuickScreen-compatible format. Once saved, however, they will no longer be compatible with QBase.

QuickScreen has built-in error checking to ensure that screens are indeed readable, and it will warn you when the current screen has not yet been saved before it attempts to load a new image.

#	Wipe Name	Wipe Description
0	Overlay (or Direct to Screen for .OBJ)	Places an image "on top of" the existing text.
1	Clear Screen	Restores a screen by first clearing the screen to black.
2	Opening Curtain	Restores a screen by pushing the existing screen outward from the center.
3	Dissolve	Restores a screen by "dissolving" the current screen.
4	Exploding Box	Restores a screen by "exploding" it onto the current screen.
5	Push Down	Restores a screen by pushing the existing screen down.
6	Push Left to Right	Restores a screen by pushing the existing screen from left to right.
7	Push Right to Left	Restores a screen by pushing the existing screen from the right to the left.
8	Push Up	Restores a screen by pushing the existing screen up.
9	Roll Away	Restores a screen by appearing to roll the existing screen away to the left.
10	Vertical Blinds	Restores a screen by appearing to open vertical blinds.
11	Wipe Left to Right	Restores a screen by sliding it from the left side of the display over the existing screen.

Table XII: QuickScreen Wipe Types

QuickScreen Files

QuickScreen screens, usually having the file extension .SCR, are retrieved using the (File) **Open Screen File...** menu command. The wipe type used when the screen was saved will take effect, and any field definitions will be available to the editor.

Object Files

Object screens, usually having the file extension .OBJ, are retrieved using the (File) **Load Object File...** menu command. Like QuickScreen screens, the wipe type used when the screen was saved will take effect, and any field definitions will be available to the editor.

Text Import

QuickScreen loads text files simply by reading the file contents one line at a time until the screen is full. Thus, if the screen is in 25-line mode then 25 lines will be read from the file; if the screen is in VGA 50-line mode then as many as 50 lines will be read. Text is retrieved at the row and column position of the cursor. The left margin of the imported text will be the column at which the cursor rests when text is retrieved.

QuickScreen Library Files

Screens may be loaded from a QuickScreen Library, which usually has the .QSL file extension. Library files usually represent a collection of screens and their use requires additional explanation, presented below.

USING QUICKSCREEN LIBRARY FILES

QuickScreen Library files are able to store many screens in a single DOS file. Library files are compressed and occupy the least amount of space on the disk.

Loading And Creating Library Files

A QuickScreen Library is opened using the (Library) **Open Library...** command. Alternatively, a Library may be created using the (Library) **Create New Library...** command.

When a library is opened you may add to it the current screen using the (Library) **Add Screen to Library...** command. This command also gives you the opportunity to specify a wipe type.

Displaying And Deleting Screens

Screens may be displayed from the Library by using the (Library) **Display Screen...** command or by pressing <F2>. Either method generates the dialog box shown in Figure 15.

The screens available in the library are shown in the list box. To display a screen simply highlight and use the *Display* command button. To delete the screen highlighted from the library, use the *Delete* command button. And to cancel the operation, use *Cancel*.

Saving A Library

You may choose the (Library) **Save Library...** command when you wish to update the Library to reflect the screens which have been added to or deleted from it.

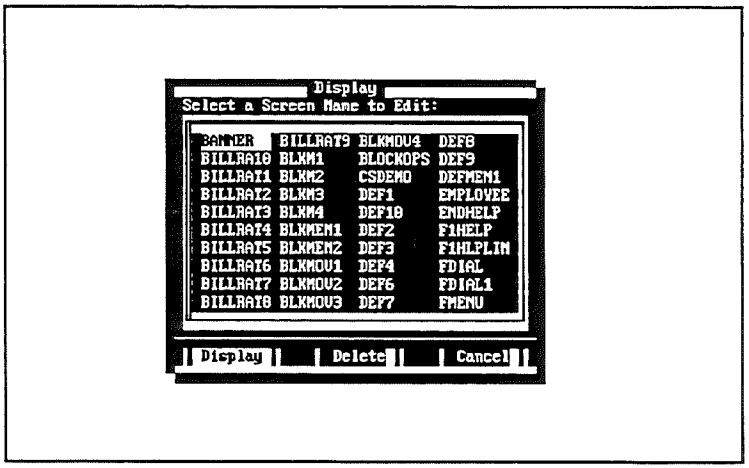

Figure 15: The Display Screen *Dialog Box*

Adding And Replacing Screens

Once a library is loaded, you can easily add the current screen to it by accessing the (Library) **Add Screen to Library** command. If the screen already exists in the library you cannot add it, but you can *replace* it using the (Library) **Replace Screen in Library** command. Naturally, you must save the library in order for these changes to be written to the library file.

CREATING DATA ENTRY FORMS

DEFINING FIELDS

1. Choose the **Define Data Fields** command
2. Position the cursor.
3. Choose a field type.
4. Adjust the field size.
5. Complete field settings.

The list above represents the sequence of steps needed to define any data field. Before you get started you should decide whether you want to paint the areas of the fields *before* defining them, or if you want QuickScreen to paint the fields *as* you define them. This latter option is chosen by ensuring that the (Compose-Fields) **Paint when Defined** option is checked. This way, you can choose a color which is different from the background so that fields can be easily seen immediately after they are created.

To create a field, first choose the (Compose-Fields) **Define Data Fields** menu option. Second, move the cursor to the starting position of the field you are defining, then press <Enter>. Third, choose a field type from those presented. Fourth, adjust the field size, keeping in mind that you must allow enough space to hold the field's data. You should consider that dashes, commas, and other "mask" characters occupy space in the field and should be considered when adjusting the size. If you are using a form and notice that a numeric field is filled with "%" symbols, then the data in that field is exceeding the field's size. This usually indicates that the field must be made larger.

Finally, the fifth step is to complete the field settings by specifying the field name, associated formula, help message, and other available options.

The field-definition procedure outlined becomes cyclic: step 5 is followed by step 2. This cycle persists until either <Esc> or <F10> is pressed after step 5. The final step 5 presents a dialog

box which is appropriate for the field being defined: this dialog box, therefore, is not always the same.

Try to define the fields in the same order you wish them to be used in the form. This reduces the liklihood that you will need to use the field Rearrange features, discussed later in this section.

Often, you'll notice ways that the form can be improved once you create it. Editing fields already on the screen is easy. Simply access the (Compose-Fields) **Define Data Fields** menu option. You can use the <+> and <-> keys to access the next and previous fields, respectively, in the form. Pressing <Del> deletes a field from the form, while pressing <Ins> inserts a field above the current field.

COPYING FIELDS

When designing a form it may be useful to copy fields already created. To do this, simply highlight the field by dragging the mouse or by using the <Shift><Direction Keys>, and press <Ctrl><Ins> to copy the field information to the clipboard. Once on the clipboard, you can position the cursor anywhere on the form and use <Shift><Ins> to copy information from the clipboard to the form. This method is suitable for copying single fields at a time, or several fields at once.

Whenever you "paste" field information from the clipboard, QuickScreen attempts to "intelligently" adjust the tab order for the entire form so that the copied fields are incorporated in a logical manner.

REARRANGING FIELDS

If you need to rearrange the order of fields on a form you may access the (Compose-Fields) **Rearrange Data Fields...** command.

This generates the dialog box similar to the one shown in Figure 16.

Fields may be reordered in the **Rearrange** dialog box by first selecting a field in the list box, and then executing the *Move* command button. You will then be able to position the selected field above or below another field in the list box. When <Enter> is pressed the field will be inserted at the indicated position.

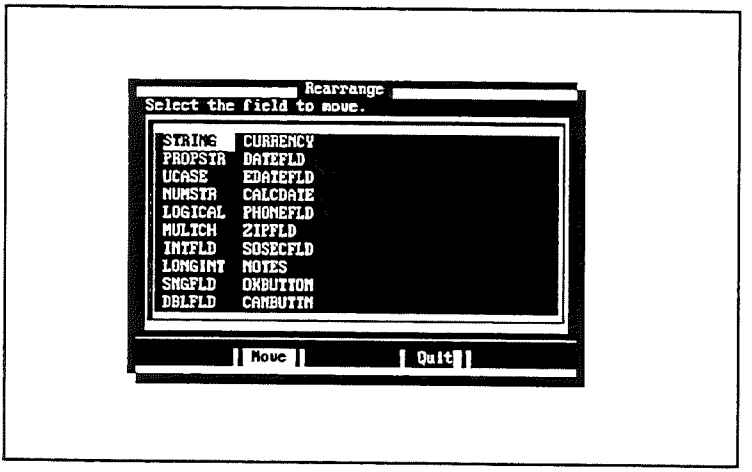

Figure 16: The Rearrange Fields *Dialog Box*

Alternatively, you can highlight the field on the screen and define it as a block. This way, the (Edit) **Move Block** command can be used to move the field anywhere on the screen. This method automatically updates the field's position and tab order. It is important to understand that you must highlight the leftmost character in the field before moving it this way.

PRINTING FIELD DEFINITIONS

The field definitions for the current screen can be printed to the device at LPT1: by using the (Compose-Fields) **Print Field Definitions** command. If the device at LPT1: is ready then the fields will be printed. The heading of the report will contain the field file name, the record length, time, and date. Page numbers will appear at the upper-right corner for your convenience.

The remainder of the report consists of a columnar table containing the headings summarized in Table XIII.

Heading Name	Meaning
Fld	The field number.
Offset	An integer pointer representing a byte offset in the entire field structure.
Name	The field name up to eight characters.
Type	The field type (see field type names beginning on page ?).
FldLen	The field length in bytes.
RecLen	The record length in bytes.
Located	The x- and y-coordinates of the field.
Related File	The file name for relation.
Choice Field #	The field number in the Related File for relation.
Index	Yes or No; tells whether the field is indexed.
Prot	Yes or No; tells whether the field is protected.
Range	The upper- and lower-range for allowable input.
Formula	The defined field formula.

Table XIII: Field Report Headings

SAVING A FORM

Creating a data entry screen usually takes both care and time. You will want to save often while designing or making changes to your form by using the (File) **Save** menu command.

When the Save dialog box appears, you can choose to generate an accompanying .BI or .DTA file. The .BI file contains the TYPE declaration for the fields in the form, and makes it possible to refer to fields by name rather than by number. The .DTA file contains ASCII program code containing BASIC DATA statements describing each field in the form. .DTA files can be used in lieu of loading a form definition file.

It is important to understand that QuickScreen provides a safeguard to protect you from destroying an existing form (.FRM) file inadvertently. If you load a form and make changes to it, QuickScreen first checks to see if a data file exists which has the same name as the form. For instance, if you are editing MYFORM.FRM and the file MYFORM.DAT or MYFORM.DBF exists, then QuickScreen will *not* overwrite the existing .FRM file (doing so could make the existing data file unreadable). Instead, a file with the extension of .NEW is created. In our example, therefore, MYFORM.NEW would be created instead of MYFORM.FRM.

QUICKSCREEN ROUTINES

PROCEDURE REFERENCE SECTION

QuickScreen allows the programmer to generate screens and process forms from BASIC using a variety of options. This flexibility necessarily brings some complexity. In order to make this section most useful, we first present some terms and concepts with which you should be familiar. Then, we introduce the important Include files which you should use in your calling programs. Next, we examine the variables which play a vital role in using QuickScreen from QuickBASIC and which appear in the demonstration programs. And last, we present documentation for the more useful QuickScreen BASIC and assembler routines.

Integers

Throughout the remainder of this manual we'll make reference to several important integer variables, such as WipeType, Action and ErrorCode. As you may know, such variables are represented in BASIC with a trailing "%". Thus, X% holds an integer variable. You will notice, however, that many examples and discussions which make use of integer variables omit the "%" symbol. The reason is that our sample programs and program fragments assume the presence of a "DEFINT A-Z" statement, which ensures that variables lacking a type identifier are integers by default.

We've retained the type identifier when discussing the syntax for the QuickScreen routines (beginning on page 130) to clearly show which parameters are integers.

Parameters And Arguments

PARAMETERS

A parameter is a variable which appears at the top of a subprogram or function heading. For example,

```
SUB QPrint0(T$, C%)
```

has T$ and C% as the parameter list. There must always be a one-two-one correspondence in both number and type among the arguments and the parameters used when implementing a routine. For instance, if there are five parameters then you must pass five arguments to the routine. And if the first parameter expects an integer, you will need to ensure that an integer is passed as the first argument.

ARGUMENTS

An argument is a variable or value used in a CALL to a subprogram or function. For example,

```
CALL QPrint0(Text$, Colr%)
```

has Text$ and Colr% as the argument list. The argument Text$ could be replaced by the literal "Hi there!" Similarly, the integer argument Colr% could be set to the integer 113. Arguments are passed to and used by the subroutine being called. When arguments are variable names (rather than literals) the subroutine being called may modify them and make their new values available to the calling program.

There are several arguments which are used in QuickScreen by a variety of routines. What follows is a discussion of each argument.

Action

Action is used by many of Crescent Software's pollable routines. The integer value contained in Action tells the called subprogram what it should do. QuickScreen uses a pollable routine called EditForm which has the responsibility of processing forms. When using EditForm, Action may be set to 1 or 3. Table XIV summarizes how these values affect EditForm.

Action Value	Meaning to EditForm
1	Initializes the current form for editing. Pads all Form$() elements to their proper lengths and formats. Displays the contents of all fields in the form. Resets Action to 3.
3	Keeps polling the current form while editing continues.

Table XIV: Action Values for EditForm

Attribute

Attribute may be set on entry to -1 to force a screen mode change when going from a large screen (i.e., greater than 25 lines) to a smaller one. If screen mode changes are not forced, then a smaller screen will occupy only a fraction of the video display. Attribute may also be set to -2 to suppress drop shadows from being displayed by the Lib2Scrn routine. You may both force a screen mode change and suppress drop shadows by using -3. Attribute entry and return values are summarized in Table XV and Table XVI, respectively.

Entry Value	Meaning
0	Screen mode changes will not occur; drop shadows will be displayed.
-1	Force screen mode change.
-2	Suppress drop shadows.
-3	Force screen mode change and suppress drop shadows.

Table XV: Attribute Entry Values

Formula	Value
Attribute MOD 256	Required screen lines for display
Attribute \ 256	Wipe type code

Table XVI: Attribute Return Values

ErrorCode

ErrorCode is used to report errors which may occur when trying to generate a screen. Its return values are summarized in Table XVII.

ErrorCode	Meaning
0	No error occurred.
1	Either a screen file was not found on disk or, if using a library, a screen was not found in the current library file.
2	A screen specified for display is too large for the current monitor.
3	The screen specified is not a QuickScreen-compatible screen.

Table XVII: ErrorCode Values

Form$() Array

The Form$() array is a conventional (not fixed-length) two-dimensional string array used to store information both about a form and about the fields it contains. The first subscript must be dimensioned to the total number of fields in the form; the second subscript is always dimensioned to 2.

A special area of the Form$() array is called the *form buffer*, and this occupies the array element of Form$(0, 0). The form buffer collects information from all fields and formats them into a fixed-length structure. This makes it possible to use random file I/O techniques to quickly load and save form information. Each field value is stored in the Form(0, 0) array element with one exception: data from notes fields are stored in a .NOT notes file. When a notes field is in the form buffer, 4 bytes are reserved and serve as an offset into the .NOT notes file. The position in the .NOT file pointed to contains a 2-byte (integer) value which gives the length of the note. In this way a linked list is created between the form buffer and the current notes file.

Form$(N, 0) contains the value of field N; Form$(N,1) contains the help message for field N; and Form$(N,2) contains the formula for field N. Note that some fields will not have a help message or formula, and, in such cases, Form$(N,1) and Form$(N,2) would be null.

The organization of the Form$() string array is summarized in Table XVIII.

Form$() Element	Description
Form$(0, 0)	Holds all data from fields as a contiguous string with numbers stored as IEEE formatted strings.
Form$(FieldNo, 0)	Holds data (numbers are stored as formatted strings).
Form$(FieldNo, 1)	Holds help message string.
Form$(FieldNo, 2)	Holds formula for calculated fields.

Table XVIII: Form() Layout

MonoCode

Monocode is a variable which controls whether a screen is displayed using color. If MonoCode is 0 then the screen will be displayed in color. If MonoCode is 3 then a screen will be converted to colors which are readable on monochrome systems. In most of the demonstration programs, MonoCode is set to 3 when the /B command line switch is used when starting the program from DOS.

MonoCode also accepts values 1 and 2 for generating different types of conversion to monochrome. We have found that a value of 3 produces the best results on most monochrome systems.

Screen Arrays

Many of the examples which follow will use integer screen array variables. It is always the case that these arrays must first be dimensioned, and that the number of elements in them must correspond to the size of the screen being used. For example, a full CGA text display must accommodate 80 columns multiplied by 25 lines, for a total of 2000 words, or 4000 bytes. An EGA 43-line mode display will require 80*43, for a total of 3440 words, or 6880 bytes. Similarly, a VGA 50-line mode display will require 80*50, for a total of 4000 words, or 8000 bytes.

TYPE Variables And Constants

Recall that "Include" files are ASCII text files containing BASIC source code. In general, they contain source code which is used by many programs. Placing such code in external files makes it easy to include them in a program without having to retype their contents each time. Also, if changes are made to an Include file, then programs which reference it will be updated the next time it is compiled. The $INCLUDE metacommand may be inserted anywhere in a program using this syntax:

```
'$INCLUDE: 'MyFile.BI'
```

When this line is encountered by the compiler, the contents of the path and file name enclosed in single quotes are read and compiled. If the file cannot be found on the directory specified, then the path stored in the INCLUDE environment variable is accessed. If the file still cannot be located, then the command is ignored and the compiler continues without generating an error.

There are four important QuickScreen-related Include files presented next. They are:

- DEFCNF.BI
- SETCNF.BI
- FIELDINF.BI
- FORMEDIT.BI

DEFCNF.BI

This file contains a TYPE which is also used by many other Crescent Software routines. The Config TYPE holds monitor and color information, and *must* precede the Include file SETCNF.BI, which DIMs the TYPE to a variable and sets values. The Config TYPE is presented next.

```
TYPE Config
   MonTyp AS INTEGER      'Monitor type
   PulBar AS INTEGER      'Horizontal bar color
   MenBox AS INTEGER      'Pulldown box color
   ActivCh AS INTEGER     'Active choice color (pulldown)
   InActCh AS INTEGER     'Inactive choice color (pulldown)
   HiLite AS INTEGER      'Active choice color (menu)
   InActHiLt AS INTEGER   'Inact. choice color (menu)
   NonMen AS INTEGER      'Normal screen color
   CurSize AS INTEGER     'Cursor scan lines
   Mouse AS INTEGER       'Mouse in-residence flag
END TYPE
```

MonTyp	Description
1	Monochrome adapter
2	Hercules card
3	CGA
4	EGA with monochrome monitor
5	EGA with color monitor
6	VGA with monochrome monitor
7	VGA with color monitor
8	MCGA with monochrome monitor
9	MCGA with color monitor
10	EGA with CGA monitor
11	IBM 8514A adapter

Table XIX: MonType Values for the Config User-Defined Type

All values for this type are set in SETCNF.BI.

SETCNF.BI

SETCNF.BI is the complementary Include file of DEFCNF.BI. It creates a new variable, Cnf, which is DIMed to the Config TYPE before its values are initialized. Mouse and MonTyp are set using special routines designed to detect and identify computer hardware. Mouse is set by calling InitMouse — an assembler function which returns -1 if a mouse is installed. MonTyp identifies the currently-active monitor type (it returns values which are summarized in Table XIX).

PulBar, MenBox, ActivCh, InActCh, HiLite, InActHiLt, and NonMen are all set to explicit integer color values, depending on whether the monitor detected is color (a "/B" command-line switch forces monochrome color assignments even if a color system

is detected). Color values are integers which appear in the color chart in Appendix C. CurSize is 0 to 7 for the number of scan lines in the cursor: 0 turns the cursor "off", and 7 creates a full block cursor.

The content of SETCNF.BI is shown below:

```
DIM Cnf AS Config
CALL InitMouse(Cnf.Mouse)
Cnf.MonTyp = Monitor
ClrDisp = Cnf.MonTyp = 3 OR (Cnf.MonTyp >= 5 _
   AND <= 9) OR Cnf.MonTyp = 11
IF INSTR(COMMAND$, "/B") THEN ClrDisp = 0
IF ClrDisp THEN
   Cnf.PulBar = 48
   Cnf.MenBox = 49
   Cnf.ActivCh = 48
   Cnf.InActCh = 52
   Cnf.HiLite = 31
   Cnf.InActHiLt = 64
   Cnf.NonMen = 30
ELSE
   Cnf.PulBar = 112
   Cnf.MenBox = 112
   Cnf.ActivCh = 112
   Cnf.InActCh = 112
   Cnf.HiLite = 15
   Cnf.InActHiLt = 80
   Cnf.NonMen = 7
END IF
IF Cnf.MonTyp >= 3 AND Cnf.MonTyp <= 5 THEN
   Cnf.CurSize = 7
ELSE
   Cnf.CurSize = 12
END IF
```

FIELDINF.BI

FIELDINF.BI includes the FieldInfo TYPE array and several constant assignments. This user-defined TYPE is required by a calling program to obtain information about a field in the currently-active form. The FieldInfo user-defined TYPE structure looks like this:

```
TYPE FieldInfo
   Fields            AS INTEGER
   Row               AS INTEGER
   LCol              AS INTEGER
   RCol              AS INTEGER
   StorLen           AS INTEGER
   FType             AS INTEGER
   RelFile           AS STRING * 8
   RelFld            AS INTEGER
   Indexed           AS INTEGER
   FldName           AS STRING * 8
   Decimals          AS INTEGER
   RelHandle         AS INTEGER
   Protected         AS INTEGER
   ScratchI          AS INTEGER
   LowRange          AS DOUBLE
   HiRange           AS DOUBLE
   ScratchS          AS STRING * 8
END TYPE
```

As in the demonstration programs, you must create and dimension the Fld() TYPE array as follows so that it is defined as the FieldInfo TYPE:

```
REDIM Fld(0) AS FieldInfo
```

DIMing any array to zero elements simply *defines* the array without committing a block of memory to it. Fld() will need to be redimensioned later to the number of fields in the current form using the NumberOfFields function. This way the array will be only as large as it needs to be.

When a form is loaded, the calling program may obtain specific information about each field using the Fld() TYPE array. For example, to find out whether the field number 3 is protected, the calling program would use a statement like:

```
IF Fld(3).Protected THEN ...
```

The program can also access a field's position on the screen and, for applicable fields, its low and high ranges for acceptable data entry.

In addition to examining the contents of the Fld() TYPE array, a calling program may change the values it creates. This means that

a field suddenly can be protected as the form is being processed. Or, based on values entered somewhere else on the form, low and high ranges for certain fields can be adjusted.

The Fld() TYPE array reserves element 0 for special use. For example, Fld(0).Row contains the record length, in bytes, of the entire current form. However, Fld(1).Row contains the row position for the field on the screen. Only Fld().Fields and Fld().Row make use of the zero element, however.

Table XX summarizes the FieldInfo TYPE elements. When "N" is mentioned, it applies to the subscript in the Frm() array: for Frm(0), N is equal to 0.

The constant assignments in the FIELDINF.BI file make it easy to use the FTYPE element of the FieldInfo TYPE. For example, if you need to know if the current field is a Proper String, you could use a statement similar to:

```
IF Fld(CurField).FType = PropStrFld THEN ...
```

FieldInfo's FType element constants are summarized in Table XXI. As above, the values designated "* Reserved *" should never be altered by your own programs.

FORMEDIT.BI

FORMEDIT.BI contains constant assignments and the user-defined TYPE called FormInfo which is constructed like this:

```
'$INCLUDE: "FORMEDIT.BI"
TYPE FormInfo
    StartEl AS INTEGER
    FldNo AS INTEGER
    PrevFld AS INTEGER
    FldEdited AS INTEGER
    KeyCode AS INTEGER
    TxtPos AS INTEGER
    InsStat AS INTEGER
    Presses AS INTEGER
    MRow AS INTEGER
    MCol AS INTEGER
    DoingMult AS INTEGER
    Edited AS INTEGER
END TYPE
```

The FormInfo TYPE elements are explained in Table XXII. It is suggested that you use the following statement to create the Frm TYPE variable:

```
DIM Frm as FormInfo
```

The Frm TYPE variable is used by a calling program to set the current field to be edited, to examine the last key pressed, to toggle the insert status of the forms editor, and to determine when data in a form has been altered. This last item is particularly useful if you are updating a file with information from a form. For example, each time you update a record in the file you should set Frm.Edited to 0. If any field values are changed then Frm.Edited will be set to -1, letting you know that it is necessary to write the form record to the file again.

You may read or set any of the Frm TYPE elements in your own program. However the elements Presses, MRow, and MCol should only be read: altering them will have no affect on the form.

Element	Description
Fields	When N=0, the number of fields in the form is returned. When N>0, the field's integer offset into the Form$(0, 0) array is returned.
Row	When N=0, the record length of the form is returned. When N>0, the screen row position for field N is returned.
LCol	The screen left column position for field N.
RCol	The screen right column position for field N.
StorLen	The number of bytes required to store the contents of field N.
FType	The field type number (see Table XVI).
RelFile	If a relational field, the base name of the file for relation is returned.
RelFld	If a relational field, the field number of the relational file is returned.
Indexed	0 when field is not indexed; -1 if field is indexed.
FldName	The name of the current field.
Decimals	The number of decimal places used for numeric fields. If -1 then numbers will not be formatted.
RelHandle	The file handle number for the related file. This number may be used for GET # and PRINT # BASIC statements.
Protected	0 if field is unprotected or -1 if protected.
ScratchI	* Reserved *
LowRange	Low range for numerical fields.
HiRange	High range for numerical fields.
ScratchS	Can be used to store any miscellaneous information, such as flags, etc.

Table XX: FieldInfo TYPE Elements

Constant Declaration	Field Type
CONST StrFld = 1	String
CONST UCaseStrFld = 2	Upper case string
CONST IntFld = 3	Integer
CONST SngFld = 4	Single precision
CONST DblFld = 5	Double precision
CONST DateFld = 6	Date (American)
CONST PhoneFld = 7	Phone number
CONST SoSecFld = 8	Social security number
CONST ZipFld = 9	Zip code
CONST MoneyFld = 10	Currency
CONST Relational = 11	Relational
CONST VirtualFld = 12	* Reserved *
CONST MultChFld = 13	* Reserved *
CONST LogicalFld = 14	Logical
CONST SequFld = 15	* Reserved *
CONST NumericStrFld = 16	Numeric string
CONST NotesFld = 17	Notes (multi-line text)
CONST LongIntFld = 18	Long integer
CONST EuroDateFld = 19	Date (European)
CONST RecNoFld = 20	* Reserved *
CONST TotRecsFld = 21	* Reserved *
CONST MultChAFld = 22	Multiple choice array
CONST PropStrFld = 23	Proper string
CONST Button = 24	Command button

Table XXI: FieldInfo FType Constants

FormInfo Element	Description
StartEl	Starting (base) element of the current form
FldNo	Current field number
PrevFld	Previous field number (different from FldNo only when first moving to a new field)
FldEdited	Returns -1 if a field has been changed
KeyCode	ASCII value of the last key pressed Extended keys return a negative value (i.e., <F1> = -59)
TxtPos	Cursor position relative to current field
InsStat	Current insertion mode status (-1 = insert on)
Presses	Number of mouse presses since last press
MRow	Mouse row number at last press
MCol	Mouse column number at last press
DoingMult	Returns -1 if handling a multiple-choice field
Edited	Returns -1 if anything on the form has changed

Table XXII: FormInfo TYPE Elements

To summarize, there are several important variables. The Cnf TYPE variable contains system environment information. The Fld() TYPE array provides information for each field in a form. The Frm TYPE variable gives information about the form itself, and about user-oriented events.

QuickScreen Routines

There are several BASIC modules which are supplied with QuickScreen in order to make its features accessible from within your own programs. The listing which follows summarizes this information (module names are uppercase; subprogram names are mixed-case):

EVALUATE.BAS: *Expression evaluation routine for calculated fields*

Evaluate	Returns the value of an expression.

FORMEDIT.BAS: *For handling data entry on a form*

EditForm	The main routine for data entry.
EndOfForms	Returns last field on a form/forms (For multi-page).
FixDate	Turns dates such as " 2- 3-1991" into "02-03-1990".
FldNum	Returns a field number given a field name.
Format	Places a formatted version of a number into the form.
Message	Used to display/clear a message box.
PrintArray	Displays the contents of the Form$() array on form.
SaveField	Formats data and saves it to the Form$(0, 0) buffer.
UnPackBuffer	Copies data from Form$(0, 0) into individual elements.
Value	Returns value of a numeric string like "$1,200.00".

FORMFILE.BAS: *For loading .FRM form definition files*

GetFldDef	Loads an .FRM file into the supplied arrays.
NumberOfFields	Determines the number of fields contained in a .FRM.

FORMLIB.BAS: *For loading form definitions from a .QFL forms library*

LibGetAddresses	Returns the starting and ending address of a form's data in the library.
LibGetFldDef	Loads field definitions into the supplied arrays.
LibNumberOfFields	Returns the number of fields contained on a form.

QEDITS.BAS: *Multi-line edit routine*

QEdit	Multi-line edit routine used for Notes fields.

QSCALC.BAS: *Support routines for doing field calculations*

CalcFields	Recalculates dependant fields based on a given field.
Tokenize	Resolves field name references in a formula to their field numbers.
WholeWordIn	Searches a string for a Whole Word version of a sub-string.

SCRNDISP.BAS: *Support routines for displaying screens*

ChangeClr	Search and replace routine for color.
DisplayScrn	Main routine for displaying screens with special effects.

SCRNFILE.BAS: *Support module to display QuickScreen files*

LoadScreen	Displays an .SCR screen file.

SCRNLIB.BAS: *Support routines to manage screen libraries*

Lib2Scrn	Displays a screen from a library array in memory.
LibFile2Scrn	Displays a screen from a library file (from disk).
LibNo	Returns a screen's number given its name.
LibScrName	Returns a screen's name given its number.
LoadScrnLib	Loads a screen library file into an array.
ScrnLibSize	Returns the memory size in words required to hold a screen library file.

VERTMENU.BAS: *Vertical menu routine*

VertMenu	Vertical menuing routine used for multiple choice fields.

Not all modules or subprograms and functions will be useful to you. However, a few, in particular, are required for certain tasks. Table XXIII presents a quick way to determine which modules and calls are required when working with certain types of QuickScreen files.

There are a number of BASIC and assembler subroutines that you may call from QuickBASIC. The following pages present an alphabetic summary of these routines. Each routine is discussed separately, and we have provided information about its program type, purpose, and calling syntax. Following the calling syntax is a brief explanation of the routine's arguments. Then, a detailed discussion of the routine and its arguments is presented. Finally, we have concluded each routine with either an example program segment or a reference to an example.

Some routines which are used by QuickScreen internally have been documented here so that they may be called directly from your own programs. These routines are not necessary in order to display or manage QuickScreen-screens, but they may be useful to you in some other capacity.

Many of the arguments used in this section have been discussed already. You may find it necessary to refer to earlier sections if more clarity is needed.

Method	Module/s	You call
Screen Files (.SCR)	SCRNFILE.BAS SCRNDISP.BAS	LoadScreen
Screen libraries (.QSL)	SCRNLIB.BAS SCRNDISP.BAS	ScrnLibSize LoadScrnLib Lib2Scrn or just LibFile2Scrn
Screen Object files (.OBJ)		
with wipe:	SCRNDISP.BAS	DisplayScrn
without wipe:	None	The screen name
Form files (.FRM)	FORMFILE.BAS	NumberOfFields GetFldDef
	FORMEDIT.BAS	EditForm
Form Library files (.QFL)		
	FORMLIB.BAS	LibNumberOfFields LibGetFldDef
	FORMEDIT.BAS	EditForm
Form DATA statements (.DTA)	FORMEDIT.BAS	EditForm

Table XXIII: Ways To Display Screens And Handle Forms

Calculated Fields

QSCALC.BAS (For support)
NOCALC.BAS (To remove support)

Multiple Choice fields

VERTMENU.BAS (For support)
NOMULT.BAS (To remove support)

Multi-line Notes fields

QEDITS.BAS (For support)
NONOTES.BAS (To remove support)

Table XXIV: QuickScreen's Optional Modules

BCopy

Assembler routine contained in FORMS.LIB

■ Purpose

Copies a block of memory (up to 64K in size) to a new location. It is used primarily to copy information from Form$(0, 0) to a TYPE structure.

■ Syntax

```
CALL BCopy(FromSeg%, FromAddr%, ToSeg%, ToAddr%, NumBytes%,_
   Direction%)
```

FromSeg%	- segment of the source location of the block
FromAddr%	- address of the source location of the block
ToSeg%	- segment of the destination
ToAddr%	- address of the destination
NumBytes%	- number of bytes to be copied
Direction%	- specifies direction of the copy (0 is forward; -1 is reverse)

■ Comments

BCopy is useful in a variety of situations, such as if you need to make a copy of an array or duplicate a range of elements. When using the routine with forms, you will most likely find it helpful when working with random file I/O. As you know, you can create a TYPE structure for your form when saving forms from the Screen Designer. For instance, in the supplied CUSTOMER.FRM file, the Customer TYPE is saved to CUSTOMER.BI, and looks like this:

```
TYPE CUSTOMER
   IDNO       AS INTEGER
   DATEIN     AS INTEGER
   NAME       AS STRING * 32
   COMPANY    AS STRING * 32
   ADDR1      AS STRING * 32
   ADDR2      AS STRING * 32
   CITY       AS STRING * 20
   STATE      AS STRING * 2
   ZIPCODE    AS STRING * 10
   WPHONE     AS STRING * 14
   HPHONE     AS STRING * 14
   NOTES      AS LONG
END TYPE
```

Using TYPE structures like the one above makes it easier to access a form's fields, and generally makes a program more readable. For instance, the GetRec routine retrieves a specific random-file record into the Form$(0, 0) form buffer. Once Form$(0, 0) is filled with data, you can use BCopy to duplicate the information it contains into a TYPE variable. This way, you can easily refer to field data using descriptive variable names. Thus, the value of the NAME field would be in CUSTOMER.NAME; the current ZIPCODE value would be in CUSTOMER.ZIPCODE, etc.

Be certain the length of Form$(0, 0) and the TYPE variable are the same — otherwise, a system crash may result.

The number of bytes may be up to 64K (i.e., 65535 bytes), though you will have to use a long integer (or negative number) to specify a value greater than 32767.

■ Example

The use of BCopy to copy information from Form$(0, 0) to a TYPE array is slightly different for BASIC 4.x and BASIC 7.x.

To copy information into the Customer TYPE variable from Form$(0, 0), use this statement in BASIC 4.x:

```
CALL BCopy(VARSEG(Form$(0, 0)), SADD(Form$(0, 0)),
   VARSEG(Customer), VARPTR(Customer), LEN(Customer), 0)
```

If you are using BASIC 7.x, you will need to use SSEG instead of VARSEG:

```
CALL BCopy(SSEG(Form$(0, 0)), SADD(Form$(0, 0)), _
   VARSEG(Customer), VARPTR(Customer), LEN(Customer), 0)
```

Box0

Assembler routine contained in FORMS.LIB

■ Purpose

Displays a box frame on text page zero.

■ Syntax

```
CALL Box0(ULRow%, ULCol%, LRRow%, LRCol%, Char%, Colr%)
```

ULRow% - upper-left row (y-coordinate)
ULCol% - upper-left column (x-coordinate)
LRRow% - lower-right row (y-coordinate)
LRCol% - lower-right column (x-coordinate)
Char% - box style (see discussion below)
Colr% - frame color (-1 uses the colors already on the screen)

■ Comments

The Box0 routine is a fast way to display a frame on text page zero. All you need to do is furnish the upper-left and lower-right coordinates, as well as the border style and color.

The Char value is set to an integer number from 1 to 4:

Char% value	Border style
1	single line all around
2	double line all around
3	double line horizontally, single line vertically
4	single line horizontally, double line vertically

The color value should be assigned from the color chart shown in Appendix C. If you want to draw a frame and use the colors already on the screen, you can set Colr to -1.

Example

This statement draws a border around the entire screen (in 25x80 text mode) using a double-line border and black-on-white as a color.

```
CALL Box0(1, 1, 25, 80, 2, 112)
```

ButtonPress

Assembler routine contained in FORMS.LIB

■ Purpose

ButtonPress reports how many times a specified mouse button was pressed since the last time it was called. It also returns the X/Y coordinate where the mouse cursor was located when that button was last pressed.

■ Syntax

```
CALL ButtonPress(Button%, Status%, Count%, X%, Y%)
```

Button% - button number (1, 2, or 3)Status% current button status (-1 if pressed; 0 if not pressed)

Count% - number of times a button has been pressed since ButtonPress was last called.

X% - x-coordinate of the mouse cursor when the button was pressed

Y% - y-coordinate of the mouse cursor when the button was pressed

■ Comments

ButtonPress is the only reasonable way to determine when the mouse buttons are active and need attention.

One important note is that ButtonPress resets the mouse-button counter (returned in Count) each time it is called.

■ Example

Please see VERTMENU.BAS for an example of implementing ButtonPress in a program.

CalcFields

BASIC routine contained in QSCALC.BAS

■ Purpose

To recalculate a field which is dependent on other input fields.

■ Syntax

```
CALL CalcFields(StartOfForm%, FldNo%, Form$(), _
   Fld() AS FieldInfo)
```

StartOfForm% - start of the form, equal to 0 for single-page forms; for multi-page forms, this number is equal to the offset in the Fld() TYPE array needed to point to first field of the desired form

FldNo% - number of the field you wish to recalculate

Form$() - form string array (see page 113)

Fld() - field information TYPE array (see page 118)

■ Comments

CalcFields should be used when information related to a calculated field is changed. CalcFields looks at the value of the specified field (contained in FldNo) and recalculates all other fields which depend on it.

This routine is useful only when you need to recalculate *specific* fields in a form.

Example

This example recalculates all fields that depend on the fifth field in the current form:

```
CALL CalcFields(0, 5, Form$(), Fld AS FieldInfo)
```

ChangeClr

BASIC routine contained in SCRNDISP.BAS

■ Purpose

To change any portion of the screen from one color to another.

■ Syntax

```
CALL ChangeClr(UlR%, UlC%, BrR%, BrC%, FromClr%, ToClr%)
```

ULRow% - upper-left row (y-coordinate)
ULCol% - upper-left column (x-coordinate)
BrR% - lower-right row (y-coordinate)
BrC% - lower-right column (x-coordinate)
FromClr% - original color to be changed
ToClr% - new color

■ Comments

This routine is useful if you want to create moving highlight bars or any other such effect which selectively colors a portion of the screen. The colors change immediately after a call to ChangeClr.

■ Example

To change the color of, say, field number three, you can access the Fld(3).Row, Fld(3).LCol and Fld(3).RCol array elements to determine the field's location on the screen. This example changes a black-on-white field to red-on-black:

```
CALL ChangeClr(Fld(3).Row, Fld(3).LCol, Fld(3).Row,
   Fld(3).RCol, 112%, 4%)
```

ClearScr0

Assembler routine contained in FORMS.LIB

■ Purpose

ClearScr0 clears all or a portion of the screen to a specified color.

■ Syntax

```
CALL ClearScr0(ULRow%, ULCol%, LRRow%, LRCol%, Colr%)
```

ULRow% - upper-left row (y-coordinate)
ULCol% - upper-left column (x-coordinate)
LRRow% - lower-right row (y-coordinate)
LRCol% - lower-right column (x-coordinate)
Colr% - color of the cleared area

■ Comments

ClearScr0 clears characters from an area of the screen you select. If Colr is set to -1, then text on the screen is cleared and colors remain as they are.

■ Example

This example clears a rectangle in the middle of a 25x80 text display page 0, and colors the area blue (color 7) before finishing:

```
CALL ClearScr0(7%, 20%, 17%, 60%, 7%)
```

Date2Num

Assembler function contained in FORMS.LIB

■ Purpose

Converts a date in string form to an equivalent integer variable.

■ Syntax

```
Days% = Date2Num%(D$)
```

Days% - the number of days before or after 12/31/79
D$ - a date in the form "MMDDYY" or "MM-DD-YY" or "MM/DD/YYYY", or any such combination

■ Comments and Example

Because Date2Num has been defined 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 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.

```
DEFINT A-Z
DECLARE FUNCTION Date2Num(X$)
DECLARE FUNCTION Num2Date(Dat)

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.

```
DEFINT A-Z
DECLARE FUNCTION Date2Num(X$)
DECLARE FUNCTION Num2Date(Dat)

INPUT "Enter a date in the form MM-DD-YYYY: "; 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 entered is valid.

Understand that while days before 12-31-1979 are returned by Date2Num as negative values, adding and subtracting will still be performed correctly.

Please see also the companion functions Num2Date and FixDate.

DisplayScrn

BASIC routine contained in SCRNDISP.LIB

■ Purpose

Displays a screen from an array using whatever special effect wipe is specified in the WipeType variable. This routine is called by Lib2Scrn and LoadScreen.

■ Syntax

```
CALL DisplayScrn(Scrn%(), Element%, MonoCode%, WipeType%)
```

Scrn%()	- integer screen array (see page 115)
Element%	- specifies the array element where the row and column information for the desired screen starts — usually the first element in the array
MonoCode%	- set to 0 for a color monitor, or to 3 for a monochrome system (see page 114)
WipeType%	- specifies which special effect wipe should be used when the screen is displayed (see Table XII)

■ Comments

This routine is called by your programs in order to restore a screen held in an integer array using a specific wipe. Scrn() must contain row and column information for the screen as well as the contents of the screen. It must be structured as follows:

```
'UlCol = upper-left column
'UlRow = upper-left row
'LrCol = lower-right column
'LrRow = lower-right row
Scrn%(Element%) = UlCol * 256 + UlRow
Scrn%(Element% + 1) = LrCol * 256 + LrRow
Scrn%(Element% + 2)... = contains the screen data
```

The Scrn() array may contain many screens as long as the Element offset is properly specified when the DisplayScrn routine is used. WipeType values are summarized on page 93.

■ Example

This example briefly shows how to store a screen in a screen array called Scrn(), and later display its contents using DisplayScrn, which can (if desired) wipe the image onto the display as well as convert it to monochrome colors.

```
DEFINT A-Z
    .
    .
    .
REDIM Scrn((LrRow - UlRow + 1) * (LrCol - UlCol + 1) + 2)
Scrn(0) = UlCol * 256 + UlRow
Scrn(1) = LrCol * 256 + LrRow
CALL MScrnSave(UlRow, UlCol, LrRow, LrCol, SEG Scrn(2))
    .
    .
    .
'display the old screen

Element = 0 : MonoCode = 0 : WipeType = 0
CALL DisplayScrn(Scrn(), Element, MonoCode, WipeType)
```

EditForm

BASIC routine contained in FORMEDIT.LIB

■ Purpose

To handle all user input, cursor, and mouse activity when processing a form from QuickBASIC. This routine is pollable so the calling routine can monitor user input as it occurs.

■ Syntax

```
CALL EditForm(Form$(), Fld(), Frm, Cnf, Action%)
```

Form$()	- form string array (see page 113)
Fld()	- field information TYPE array (see page 118)
Frm	- form information TYPE variable (see page 120)
Cnf	- system environment TYPE variable (see page 116)
Action%	- a flag used to control how the form behaves when called (see page 111)

■ Comments

The EditForm subprogram is a major routine in QuickScreen. It will allow calling programs to process forms, making additional programming virtually unnecessary.

When EditForm is called, it uses information in Form$(), Fld(), and Frm. Form$() is a conventional (not fixed-length) two-dimensional string array. The first subscript must be dimensioned to the total number of fields in the form. The second subscript must be dimensioned to 2. Fld() is a TYPE array which is DIMed to the FieldInfo user-defined TYPE (please see page 118). Both Form$() and Fld() may be

dimensioned using the NumberOfFields function as shown below.

```
Size% = NumberOfFields%(FormName$)
REDIM Form$(Size%, 2)
REDIM Fld(Size%) AS FieldInfo
```

Once form arrays are properly sized, they can be initialized and loaded using the GetFldDef routine (see pages 163 and 226 for further details):

```
CALL GetFldDef(FrmName$, StartEl%, Fld(), Form$())
```

To continue with the list, Frm is a TYPE variable which is DIMed to the FormInfo user-defined TYPE (please see page 120).

Cnf is a TYPE variable which is DIMed to the Config user-defined TYPE (please see page 116).

Action is either 1 or 3 was discussed earlier (please see page 111).

■ Example

The most effective way to poll EditForm is to wait for a particular keypress, such as <Esc>, to occur. When this happens, the form may be cleared from the screen and processing may continue.

In the example below we present a DO loop showing how to poll EditForm. The DO loop presented is terminated when <Esc> is pressed.

```
.
.
.
ACTION = 1
DO
   CALL EditForm(Form$(), Fld(), Frm, Cnf, Action)
LOOP UNTIL Frm.KeyCode = 27
           'Keep editing until user presses <Esc>
.
.
.
```

When the user finally presses <Esc>, the data entered into the form may be accessed by examining the contents of the Form$() string array.

EndOfForms

BASIC function contained in FORMEDIT.BAS

■ Purpose

Returns the number of the last field on any form.

■ Syntax

```
LastFld% = EndOfForms%(Fld())
```

LastFld% - the value of the last field on the form
Fld() - field information TYPE array (see page 118)

■ Comments

Because EndOfForms has been defined as a function, it must be declared before it may be used.

This function can be used to determine the last field number on a form, and is particularly useful for a multi-page forms.

Evaluate

BASIC routine contained in EVALUATE.BAS

■ Purpose

Returns the value of a mathematical expression. 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 lowercase "e" represents the constant, and an uppercase "E" is for the exponent.

■ Syntax

```
Answer# = Evaluate(Expression$)
```

Expression$ - string containing a mathematical expression, along with parentheses, operation keywords (such as ABS or SIN), and numbers; if the string expression is invalid, the string is returned in Expression$ with a leading percent sign (%) appended.

Answer# - receives the computed answer

■ Comments

Because Evaluate has been defined as a function, it must be declared before it may be used.

Scientific notation is supported using "E" (but not "e"). What follows is a list of operations supported by Evaluate:

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
ARCSIN	Arc Sine
ARCSINH	Arc Hyperbolic Sine
ATN	Arc Tangent
CLG	Common Log (base 10, what LOG really is)
COS	Cosine
COT	Cotangent
CSC	Cosecant
CSCH	Hyperbolic Cosecant
EXP	Exp
LOG	Natual 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 list shows math operators supported by Evaluate:

- ! Factorial
- ^ Exponentiation
- * Multiplication
- / Division
- \ Integer Division
- + Addition
- - Subraction (or unary minus, such as -15)
- < Less than
- = Equal to
- > Greater than

Example

```
X = EVALUATE("10 * (12^3+(4E-13))/LOG(8)")
```

Exist

Assembler function contained in FORMS.LIB

■ Purpose

Exist will quickly determine the presence of a file.

■ Syntax

```
There% = Exist%(FileName$)
```

FileName$ - file name or file specification
There% - assigned to -1 if FileName$ exists; 0 if FileName$ does not exist

■ 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 cause 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 could be created in the process. For this reason, we recommend the use of the Exist function.

It's important to know that the FileName$ may optionally contain a drive letter, a directory path, and either of the DOS wild card characters.

■ Example

This example returns -1 if there are .BAS files on the \STUFF directory of the B drive:

```
There% = Exist%("B:\STUFF\*.BA?")
```

FGet

Assembler routine contained in FORMS.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$)
```

Handle%	- handle assigned when the file was opened
Destination$	- 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 supplied FSeek routine.

The length of Destination$ is used to tell FGet how many bytes it is to read to ensure 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 first to 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.

■ Example

This example gets one byte of information from the current file, at the location specified by the DOS file pointer.

```
X$ = SPACE$(1)          'set one byte aside
                        'read the byte value from the file
CALL FGet(Handle%, X$)
```

FixDate

BASIC routine contained in FORMEDIT.BAS

■ Purpose

Changes the format of a date string.

■ Syntax

```
CALL FixDate(Dat$)
```

Dat$ - string containing the date in a variety of string formats

■ Comments

This subprogram is useful for ensuring the date format corresponding to MM-DD-YYYY are correctly-formatted. For instance, the routine ensures that all months and days have two numerical digits (single-digit months or days will have a leading zero). It also ensures that a century is appended to years entered as two digits. Thus, " 3- 4-91" will become "03-04-1991" after calling FixDate.

■ Example

```
NewDate$ = FixDate(Date$)
```

FldNum

BASIC function contained in FORMEDIT.BAS

■ Purpose

Returns the field number corresponding to a specified field name.

■ Syntax

```
FldNumber% = FldNum%(FldName$, Fld())
```

FldNumber% - number of the field named by FldName$
FldName$ - string containing the field name
Fld() - field information TYPE array (see page 118)

■ Comments

Because FldNum has been defined as a function, it must be declared before it may be used.

FldNum makes it easy to obtain the number of a field if all you have available is its name. The routine is useful for creating programs which do not have to be modified as your data entry form changes. It also makes source code more intelligible by allowing long variable names to refer to short field names.

■ Example

This example finds which field number holds a "discount rate". Then, the field number is used to access the form buffer so that the field's value is returned.

```
DiscountRateFld = FldNum("DISCRATE", Fld())
DiscountRate = VAL(Form$(DiscountRateFld, 0))
```

FOpen

Assembler routine contained in FORMS.LIB

■ Purpose

FOpen is used to open a disk file in preparation for reading or writing using the FSeek or FGet routines.

■ Syntax

```
CALL FOpen(FileName$, Handle%)
```

FileName$ - name of file to be opened
Handle% - handle assigned by DOS for all subsequent access; if errors occur when trying to open the file, Handle returns 0

■ Comments

FOpen will open any file, and will also accept an optional drive or directory as part of the file name. However, it will not create a file. If you are not sure whether a file exists you should first use the Exist function.

It is up to your program to store the handle number that DOS assigns, and use that handle whenever you access the file again.

■ Example

This example opens the file "MyForm.QSL" and assigns an integer file handle number to it.

```
CALL FOpen("MyForm.QSL", Handle%)
```

Format

BASIC routine contained in FORMEDIT.BAS

■ Purpose

Places a formatted version of a supplied number into a field of the current form. Often used in conjunction with the Value routine.

■ Syntax

```
CALL Format(Float#, Fld(FldNo), Form$(FldNo, 0))
```

Float#	- number you wish to be formatted
Fld(FldNo)	- field information TYPE array (see page 118) and the desired field number (FldNo) for which Float# is formatted
Form$(FldNo, 0)	- form string array (see page 113) and the desired field number

■ Comments

This routine makes it easy to place a number into a field which has been designated as a formatted field. For example, a currency field would format a number such as "5000" to "$5,000": the dollar-sign and comma are added after calling Format.

Although you can use this routine directly, we recommend using the SaveField routine (which itself calls Format) since it validates a field before updating the form buffer.

Example

This example converts "10000" to "$10,000" in field #5:

```
CALL Format(10000#, Fld(5), Form$(5, 0))
```

FSeek

Assembler routine contained in FORMS.LIB

Purpose

FSeek will position the DOS file pointer for a file that has been opened using the FOpen routine.

Syntax

```
CALL FSeek(Handle%, Location&)
```

Handle%	- handle assigned when the file was opened
Location&	- byte location in the file to seek

Comments

Unlike QuickBASIC's SEEK, FSeek considers the first byte in the 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.

One warning you should be aware of is seeking beyond the end of a file, which will cause its length to be extended. This is not a fault with FSeek, and in fact will happen with BASIC's SEEK command as well.

Example

This example seeks byte 8 in the header information to a QuickScreen library file. This byte tells how many screens are stored:

```
CALL FSeek("MyFile.QSL", 7&)
```

GetFldDef

BASIC routine contained in FORMFILE.BAS

■ Purpose

Retrieves information from a form file and places it in a structure for later reference by other routines. Also loads formulas and help messages into the Form$() data array.

■ Syntax

```
CALL GetFldDef(FrmName$, StartEl%, Fld(), Form$())
```

FrmName$	- name of the form (.FRM) definition file
StartEl%	- starting element in the Fld() array below which the form information is to be loaded
Fld()	- field information TYPE array (see page 118)
Form$()	- form string array (see page 113)

■ Comments

This routine allows a calling program to load a .FRM file so that it may be properly processed by EditForm. The NumberOfFields function (discussed on page 186) should be used before this routine in order to properly dimension the Fld() and Form$() arrays.

■ Example

An example of this routine is shown and discussed on page 226.

GetRec

BASIC routine contained in RANDOMIO.BAS

Purpose

To retrieve a specified record and any associated notes from a database.

Syntax

```
CALL GetRec(RecNo&, Form$(), Fld())
```

RecNo& - record number to retrieve
Form$() - form string array (see page 113)
Fld() - field information TYPE array (see page 118)

Comments

When called, this routine loads the specified record into the form buffer, Form$(0, 0). Once this is done, it is necessary to call the UnPackBuffer routine so that the remaining elements in the Form$() array are properly filled.

The data which is read by this routine is expected to be in a .DAT data file, while any associated notes fields are read from a .NOT notes file.

Usually, the routine OpenFiles is called before using either GetRec or SaveRec.

Example

Please see page 236 for an example.

LibFile2Scrn

BASIC routine contained in SCRNLIB.BAS

■ Purpose

Loads and displays an individual screen from a screen library file.

■ Syntax

```
CALL LibFile2Scrn(LibName$, ScrnName$, MonoCode%,_
   Attribute%, ErrorCode%)
```

LibName$	- the file name of the screen library (the ".QSL" extension is optional)
ScrnName$	- the name of the screen to display (8 characters maximum)
MonoCode%	- set to 0 for a color monitor, or to 3 for a monochrome system (see page 114)
Attribute%	- specifies drop shadow and screen mode change options (please see Table XV and Table XVI)
ErrorCode%	- returns a non-zero number if an error occurred (please see Table XVII)

■ Comments

This routine is ideal for implementing a help system in which help screens are only needed at certain times. In this system it is usually not practical to keep the entire screen library in memory.

Example

This routine requires no setup and can be used simply by specifying a screen library name and the name of the screen in the library which is to be displayed. MonoCode, Attribute, and ErrorCode have been discussed earlier.

The example below loads the "INTRO" screen of the "CUSTOMER.QSL"screen library file:

```
CALL LibFile2Scrn("CUSTOMER", "INTRO", 0, 0, ErrorCode)
```

If an error occurred, ErrorCode will be true, and will have a value of -1.

LibGetFldDef

BASIC routine contained in FORMLIB.BAS

■ Purpose

Fills the supplied arrays with data from a form stored in a form library file.

■ Syntax

```
CALL LibGetFldDef(LibName$, FrmName$, StartEl%, Fld(),_
   Form$(), ErrorCode%)
```

LibName$	- name of the form library file (the ".QFL" extension is optional)
FrmName$	- name of the form in the library
StartEl%	- specifies where in the Form$() and Fld() arrays information will loaded; StartEl% is the starting element
Fld()	- field information TYPE array (see page 118)
Form$()	- form string array (see page 113)
ErrorCode%	- returns a non-zero number if an error occurred (please see Table XVII)

■ Comments

This routine accesses form information for a specific screen in a form library and automatically parses information into the Fld() and Form$() arrays. The StartEl value makes it easy to preserve the lower portion of the supplied arrays and effectively allows several form files to be added to a single array.

Example

This example opens the BACKORDR.FRM form in the SALES.QFL form library file and fills the Fld() and Form$() arrays starting at the fifteenth element. Doing this preserves the form information already contained in the 0 through 14 elements of the Fld() and Form$() arrays:

```
CALL LibGetFldDef("SALES","BACKORDR", 15, Fld(), Form$(),_
   ErrorCode)
```

LibLoadDisplayForm

BASIC routine contained in DEMOCUST.BAS

■ Purpose

Simplifies loading and displaying data entry forms.

■ Syntax

```
CALL LibLoadDisplayForm(LibName$, FormName$, Form$(), _
   Fld() AS FieldInfo)
```

LibName$	- name of the screen/form (.QSL) library file
FormName$	- name of the form to load and display
Form$()	- form string array (see page 113)
Fld()	- field information TYPE array (see page 118)

■ Comments

This routine appears as a subprogram in DEMOCUST.BAS. In order to be used, it must be copied into your own program.

This routine automatically dimensions the Form$() and Fld() arrays by examining how many fields are in the form you have specified. These arrays are then filled with information from the form library (.QFL) file, and the screen is displayed from the screen library (.QSL) library file.

LibNo

BASIC function contained in SCRNLIB.BAS

■ Purpose

To return the number of a library screen given its name.

■ Syntax

```
ScrnNum% = LibNo%(NameInLib$, ScrnLib%())
```

ScrnNum%	- number of the screen in the library
NameInLib$	- screen name in library (must be 8 characters or less)
ScrnLib%()	- integer screen library array

■ Comments

Because LibNo has been defined as a function, it must be declared before it may be used.

This function is useful when you need to a screen in a screen library array by supplying its number rather than its name. LoadScrnLib should be used in preparation for this function.

■ Example

This example determines the number for the "INTRO" screen in the current screen library array:

```
ScrnNum% = LibNo%("INTRO", ScrnLib%())
```

LibNumberOfFields

BASIC function contained in FORMLIB.BAS

■ Purpose

Returns the number of fields for a form in a form library.

■ Syntax

```
TotalFields% = LibNumberOfFields%(LibName$, FormName$)
```

TotalFields% - total number of fields in the form specified
LibName$ - name of the form library file
FormName$ - name of the form

■ Comments

Because LibNumberOfFields has been defined as a function, it must be declared before it may be used.

This function is useful for determining how the Fld() and Form$() arrays should be dimensioned before being used. It is typically used before calling LibGetFldDef.

■ Example

This example returns the number of fields in the ACCOUNTS form of the BUDGET form library:

```
TotalFields% = LibNumberOfFields%("BUDGET", "ACCOUNTS")
```

LibScrName

BASIC function contained in SCRNLIB.BAS

■ Purpose

To return a screen name from a library given its number.

■ Syntax

```
ScrnName$ = LibScrName$(ScrnNo%, ScrnLib%())
```

ScrnName$	- return screen name
ScrnNo%	- number of the desired screen in the library
ScrnLib%()	- integer screen library array

■ Comments

Because LibScrName has been defined as a function, it must be declared before it may be used.

This routine makes it easy to obtain the name of a screen given its screen number. It is useful when controlling multi-page forms, since it allows you to increment or decrement a screen number counter in order to access particular screens in a library.

■ Example

This example shows how to fill an array with the screen names contained in a ScrnLib() integer screen library array:

```
REDIM Array$(ScrnLib(0))
        'ScrnLib(0) gives the last screen #
FOR N = 1 TO ScrnLib(0)
   Array$(N) = LibScrName$(N, ScrnLib())
NEXT N
```

Lib2Scrn

BASIC routine contained in SCRNLIB.BAS

■ Purpose

Displays a screen from a screen library array using whatever wipe type was specified when the screen was saved.

■ Syntax

```
CALL Lib2Scrn(NameInLib$, ScrnLib%(), MonoCode%, _
   Attribute%, ErrorCode%)
```

NameInLib$	- name of the screen in the current library
ScrnLib%()	- screen integer array used to hold the library image
MonoCode%	- set to 0 for a color monitor, or to 3 for a monochrome system (see page 114)
Attribute%	- specifies drop shadow and screen mode change options (please see Table XV and Table XVI)
ErrorCode%	- returns a non-zero number if an error occurred (please see Table XVII)

■ Comments

This routine may be called from your programs to display screens contained in QuickScreen Library files. The library must have been previously loaded using the steps outlined on page 212.

The NameInLib$ string variable is the name of the screen in the library you wish to display, and it must be 8 characters or less. ScrnLib() is the screen integer array.

Example

This example gets the size of the "CUSTOMER.QSL" screen library file using ScrnLibSize, before using LoadScrnLib to load the library into an integer screen array. Then, the screen called "INTRO" is generated from the screen array using the Lib2Scrn routine.

```
DimSize = ScrnLibSize("CUSTOMER")
REDIM ScrnArray(DimSize)
CALL Lib2Scrn("INTRO", ScrnArray(), MonoCode,  Attribute,_
   ErrorCode)
```

This routine is used by the DEMOCUST.BAS program.

LoadScreen

BASIC routine contained in SCRNFILE.BAS

■ Purpose

Loads and displays a QuickScreen screen file using whatever wipe type was specified when the file was saved.

■ Syntax

```
CALL LoadScreen(ScrName$, MonoCode%, Attribute%, ErrorCode%)
```

ScrName$ - full path and file name of the screen to be displayed

MonoCode% - set to 0 for a color monitor, or to 3 for a monochrome system (see page 114)

Attribute% - specifies drop shadow and screen mode change options (please see Table XV and Table XVI)

ErrorCode% - returns a non-zero number if an error occurred (please see Table XVII)

■ Comments

LoadScreen is able to load QuickScreen files from disk. If used, this subprogram requires that SCRNDISP.BAS be linked with your program.

■ Example

```
ScrName$ = "CSDEMO.SCR"
MonoCode = 0 : Attribute = 0
CALL LoadScreen(ScrName$, MonoCode, Attribute, ErrorCode)
```

LoadScrnLib

BASIC routine contained in SCRNLIB.BAS

■ Purpose

Loads a screen library into an integer array so that screens can be stored in and displayed from memory.

■ Syntax

```
CALL LoadScrnLib(LibName$, ScrnLib%(), ErrorCode%)
```

LibName$ - name of the screen library to load
ScrnLib%() - the array that will hold the library (this array must have been dimensioned to a size large enough to hold the library — please see the ScrnLibSize function to determine the correct size for this array)
ErrorCode% - returns a non-zero number if an error occurred (please see Table XVII)

■ Comments

This routine can be used in the beginning of a program to load an entire screen library into memory so that individual screens can be displayed without having to access a disk. Use the Lib2Scrn routine to display screens from memory.

■ Example

See the example for the Lib2Scrn routine.

MakeMono

Assembler routine contained in FORMS.LIB

■ Purpose

To convert the screen image contained in the integer array Scrn() to colors suitable for monochrome systems.

■ Syntax

```
CALL MakeMono(SEG Scrn%(0), ScreenSize%)
```

Scrn%(0) - screen integer array
ScreenSize% - size of the screen in words

■ Comments

The assembler MakeMono routine allows conversion of color screens for use on monochrome systems. The screen which is to be converted must already be placed in the Scrn() integer array.

ScreenSize specifies the size of the screen in words. Please see the discussion under Screen Arrays on page 115.

Message

BASIC routine contained in FORMEDIT.BAS

■ Purpose

To display text information to the user in a box.

■ Syntax

```
CALL Message(Msg$)
```

Msg$ - message string

■ Comments

This routine quickly displays any text you supply in a conventional (variable-length) string. When you call Message with a non-null string, the message is displayed in a box which is commonly known as a "window". If you call Message again with a null Msg$ (i.e., no text), then the window previously displayed with be removed, and the screen image the message covered will be properly restored.

Further, if your user starts his application with the "/B" command-line switch, then messages will be displayed in monochrome.

■ Example

```
CALL Message("This is a help message.")
```

Monitor

Assembler function contained in FORMS.LIB

■ Purpose

Monitor makes it easy to determine the type of display adapter currently active.

■ Syntax

```
MonType% = Monitor%
```

MonType% - integer value representing the detected monitor type currently in use

■ Comments

Because Monitor is designed 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. In the context of this QuickScreen product, which generates text-mode screens only, it's best to ensure that text mode is active before using Monitor.

The type of monitor detected will be returned as a number, discussed on page 117.

Monitor is useful 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 defaults.

Example

On an EGA system with a color monitor, Monitor returns the value 5 in MonType.

```
MonType% = Monitor%
```

MPaintBox

Assembler routine contained in FORMS.LIB

■ Purpose

MPaintBox changes the color of each character in a block of text. If a mouse is being used, this routine turns off the mouse cursor until all colors have been set.

■ Syntax

```
CALL MPaintBox(ULRow%, ULCol%, LRRow%, LRCol%, Colr%)
```

ULRow% - upper-left row (y-coordinate)
ULCol% - upper-left column (x-coordinate)
LRRow% - lower-right row (y-coordinate)
LRCol% - lower-right column (x-coordinate)
Colr% - color for the specified block

■ Comments

MPaintBox is useful for quickly "painting" an area of the currently-active text screen.

■ Example

This example paints the a 10x10 area at the upper-left portion of the display to blue on white.

```
CALL MPaintBox(1, 1, 10, 10, 113)
```

MQPrint

Assembler routine contained in FORMS.LIB

■ Purpose

To display a string very quickly at the current cursor location. The mouse cursor is turned off until printing is completed.

■ Syntax

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

X$ - string to be printed
Colr% - color to be used

■ Comments

MQPrint uses the currently-active text page for its operation. If the Colr value is -1, then the existing screen colors will be used.

MQPrint prints any character you send it. That is, unlike BASIC's PRINT statement, MQPrint does not interpret special "control" characters, such as CHR$(12).

■ Example

This example prints "Hello World!" at the current cursor location using the currently-displayed colors.

```
CALL MQPrint("Hello World!", -1)
```

MScrnSave and MScrnRest
Assembler routines contained in FORMS.LIB

■ Purpose

MScrnSave and MScrnRest save and restore portions of the screen. Each routine also turns off the mouse cursor while working.

■ Syntax

```
CALL MScrnSave(ULRow%, ULCol%, LRRow%, LRCol%, SEG A%(0))
```

or

```
CALL MScrnRest(ULRow%, ULCol%, LRRow%, LRCol%, SEG A%(0))
```

ULRow% - upper-left row (y-coordinate)
ULCol% - upper-left column (x-coordinate)
LRRow% - lower-right row (y-coordinate)
LRCol% - lower-right column (x-coordinate)
A%(0) - integer array which holds a portion of the screen

■ Comments

MScrnSave and MScrnRest work with the currently active page. MScrnSave saves an area of the screen to an integer array. The screen image can then be overwritten. Later, the orginal image stored in the integer array can be used when calling MScrnRest so that the original area of the screen is properly restored.

QuickScreen uses these routines extensively. For example, the screen image is first saved whenever a menu or message "pops up" over existing text. This way, the "underlying" screen

image can be easily restored when the pop up is no longer needed.

An integer screen array is used for several important reasons. First, it is organized in a manner similar to the screen, so that each character on the screen 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 — each screen occupies its own area in the array.

Example

The ScrnSave and ScrnRest routines make it possible to create text "windows" — areas of the screen which appear to pop up and pop down over existing text.

The trick in creating such an effect is to save a portion of the screen over which your window will appear. You can then write over the saved area of the screen since it can be easily restored with its prior contents.

It would be wasteful to save the *entire* screen each time you wished to create a window: doing this would be slow and would require large screen arrays to hold the screen image. Instead, we recommend calculating the precise area needed by the window. This way, arrays can be properly sized and the process will be as fast as possible.

To calculate the size of the screen array, you'll need to calculate the width (WDTH) and height (HGHT) of your window. Your window is specified in terms of its upper-left

and lower-right coordinates, so figuring out the width and height will require use of these coordinate values:

```
WDTH = LRCol - ULCol + 1
HGHT = LRRow - ULRow + 1
```

Where LRCol is the lower-right column; ULCol is the upper-left column; LRRow is the lower-right row; and ULRow is the upper-left row. The size of the array is then a simple calculation:

```
Size% = WDTH * HGHT
REDIM ScrnArray%(Size%)
```

The area of the screen is then saved like this:

```
CALL ScrnSave(ULRow, ULCol, LRRow, LRCol, ScrnArray%(0))
```

Now that the portion of the screen image you've specified is preserved in ScrnArray(), you can write to the screen anywhere in the specified rectangular area.

When it's time to restore the original screen image, you can call ScrnRest like this:

```
CALL ScrnRest(ULRow, ULCol, LRRow, LRCol, ScrnArray%(0))
```

It is important never to destroy the contents of ScrnArray() (if you intend to restore the screen image) since doing this can result in "garbage" on the screen.

To create several "layers" of windows you will need to create additional screen arrays and rectangular coordinates. This would allow you to overwrite and restore other areas of the screen as well, and can allow for very complex windowing schemes.

NumberOfFields

BASIC function contained in FORMFILE.BAS

Purpose

Returns the number of fields in a form.

Syntax

```
N% = NumberOfFields%(FormName$)
```

N%	- number of fields in FormName$
FormName$	- string containing the full path and file name of the form

Comments

Because NumberOfFields has been defined as a function, it must be declared before it may be used.

This function is used to dimension the Fld() TYPE array and Form$ data array to the proper number of elements before calling GetFldDef.

Example

This example returns the number of fields in the MyForm.FRM file:

```
NumFields% = NumberOfFields%("MyForm.FRM")
```

Num2Date

Assembler function contained in FORMS.LIB

Purpose

Num2Date converts a previously-encoded integer date to an equivalent date string.

Syntax

```
D$ = Num2Date$(Days%)
```

D$ - formatted date string
Days% - integer value from -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.

Please see the Date2Num discussion and example for more information.

OpenFiles

BASIC routine contained in RANDOMIO.BAS

■ Purpose

To open a random access database (.DAT) file, and to field-format it to the data buffer — Form$(0, 0). If there are multi-line notes fields contained in the form, a Notes database file (.NOT) is also opened.

■ Syntax

```
CALL OpenFiles(FormName$, Form$(), Fld() AS FieldInfo)
```

FormName$ - name of the database file to open (without the .DAT extension
Form$() - form string array (see page 113)
Fld() - field information TYPE array (see page 118)

■ Comments

OpenFiles looks in the current directory for the form name you provide. If you want to access files on a different drive/directory, then you must append the path in front of the FormName$.

If the form is found it and its associated notes file are opened; if the form file is not found then it is created.

Once the random file is open, Form$(0, 0) is fielded to it. Fld(0).RelHandle holds the handle for the .DAT file opened, and Fld(0).ScratchI hold the handle for any Notes file which was opened.

The routines GetRec and SaveRec can be used after OpenFiles has successfully worked.

PrintArray

BASIC routine contained in FORMEDIT.BAS

Purpose

To refresh the screen by redisplaying the contents of fields in the form.

Syntax

```
CALL PrintArray(FirstFld%, LastFld%, Form$(), Fld())
```

FirstFld% - starting field to be redisplayed
LastFld% - ending field to be redisplayed
Form$() - form string array (see page 113)
Fld() - field information TYPE array (see page 118)

Comments

Sometimes it is necessary to change a field from your own program. Refreshing the screen ensures that the user is aware of the exact content of each field on the form.

Example

This example refreshes only the third and fourth field of the currently-displayed form:

```
CALL PrintArray(3, 4, Form$(), Fld())
```

QEdit

BASIC routine contained in QEDITS.BAS

■ Purpose

QEdit is a text editor subprogram that may be called as a "pop-up" from within a BASIC program.

■ Syntax

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

Array$() - conventional (not fixed-length) string array that will hold the text being entered or edited (the size to which Array$() has been dimensioned determines the maximum number of lines that may be entered)
Ky$ - holds the last key pressed
Action% - indicates how QEdit is being invoked (see comments below)
Ed - TYPE variable that controls QEdit (see comments below)

■ Comments

The QEdit editing window may be positioned anywhere on the screen, and sized to nearly any number of rows and columns. QEdit automatically saves the underlying screen; it may be used in the 25-, 43-, or 50-line screen modes; and it supports word-wrap, a mouse, and horizontal/vertical scrolling.

All of the standard editing keys are supported. For example, <Home> and <End> move to the beginning and end of a line; the <PgUp> and <PgDn> 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.

The call for QEdit is fairly simple to set up. 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 wordprocessor 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 current line counter
WHILE NOT EOF(1)                 'read until the end
   LINE INPUT Array$(CurLine)    'get a line
   CurLine = CurLine + 2         'skip over next line
WEND
CLOSE #1                         'close the file
   .
   .
   .
```

Like VertMenu, the current cursor location indicates where to position the upper-left corner of the editing window. Arguments 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. Here's the QEdit calling syntax, once again:

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

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

Ky$ returns the key holding the last key pressed. For example, it will hold CHR$(27) if the user pressed <Esc> to exit QEdit.

The Action argument 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 <Esc> 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 multitasking 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 keypress. 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" TYPE 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. 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

This sets the number of rows to be displayed in the window. It can range up to 25 on a CGA. If an EGA or VGA is present, and WIDTH is used to set more screen lines before QEdit is called, then the window may occupy up to 43 (for EGA) or 50 (for VGA) lines.

Ed.Wide

This sets the number of columns (up to 80) to be displayed in the window.

Ed.Wrap

This 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

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

Ed.AColor

This sets the color of the edit window, according to values detailed in the color chart at the end of this manual.

Ed.Frame

Not supported.

The remainder of the parameters are intended to be read by your program, and do not have to be set before QEdit is called.

Ed.LSCol

This holds the current left screen column of the editable window.

Ed.LC

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

Ed.CurCol

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

Ed.TSRow

This holds the top screen row of the editable window.

Ed.TL

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

Ed.CurLine

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

Ed.UlCRow
Ed.UlCCol
Ed.BrCRow
Ed.BrCCol
Ed.CBlock

These are not supported.

Ed.Presses

This 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

This 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

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

Ed.Insert

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

Ed.Changed

This 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

This holds the number of active lines in the text string array.

Ed.MErr

This is an error flag 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.

SaveField

BASIC routine contained in FORMEDIT.BAS

■ Purpose

To validate and format a field before placing it in the Form$(0, 0) form buffer. This routine is often used before calling the PrintArray routine.

■ Syntax

```
CALL SaveField(FldNo%, Form$(), Fld(), BadFld%)
```

FldNo% - field to be examined
Form$() - form string array (see page 113)
Fld() - field information TYPE array (see page 118)
BadFld% - returns 0 when valid; -1 when invalid

■ Comments

This routine first validates the data in the field by checking high/low and acceptable ranges for the data. If the data in the field is not valid (i.e., it is out of the allowable range for the field, or it is an invalid date), then the BadFld flag will be set to -1. If BadFld is returned as 0, then the data is valid and SaveField updates the contents of the Form$(0, 0) form buffer with the current field's data.

■ Example

This example validates data in field three before updating the field buffer:

```
CALL SaveField(3, Form$(), Fld(), BadFld%)
```

SaveRec

BASIC routine contained in RANDOMIO.BAS

Purpose

To save information from a form to a specified record in a .DAT data file. Multi-line notes fields are written to a .NOT notes file.

Syntax

```
CALL SaveRec(RecNo&, Form$(), Fld())
```

RecNo& - record number to save
Form$() - form string array (see page 113)
Fld() - field information TYPE array (see page 118)

Comments

The data currently in the form buffer, Form$(0, 0), is saved to the random-access .DAT data file and any notes are saved to the .NOT notes file.

Usually, the routine OpenFiles is called before using either GetRec or SaveRec.

Example

Please see page 237 for an example.

ScrnLibSize

BASIC function contained in SCRNLIB.BAS

■ Purpose

Returns the size in words required to hold the screen library specified.

■ Syntax

```
DimSize% = ScrnLibSize%(LibName$)
```

LibName$ - name of the screen library on which to report
DimSize% - size to use in a subsequent DIM or REDIM statement

■ Comments

Because ScrnLibSize has been defined as a function, it must be declared before it may be used.

This function can be used to determine the size of an integer array required to hold a screen library file, and is usually used before the LoadScrnLib routine.

■ Example

See the example for the Lib2Scrn routine.

Tokenize

BASIC routine contained in QSCALC.BAS

■ Purpose

To replace field names with a padded fixed-length (23-character) field number.

■ Syntax

```
CALL Tokenize(Calc$, Fld())
```

Calc$ - formula string
Fld() - field information TYPE array (see page 118)

■ Comments

This routine is used internally so that field formula strings can be properly read. By replacing field name strings with field numbers, the routine can properly access data items in a form.

UnPackBuffer

BASIC routine contained in FORMEDIT.BAS

■ Purpose

Copies and formats information contained in the form array (Form$(0, 0)) and fills the Form$(FldNo,1) data array for each field.

■ Syntax

```
CALL UnPackBuffer(FirstFld%, LastFld%, Form$(), Fld())
```

FirstFld%	- starting field to be redisplayed
LastFld%	- ending field to be redisplayed
Form$()	- form string array (see page 113)
Fld()	- field information TYPE array (see page 118)

■ Comments

This routine is useful when using random access files to store and retrieve the contents of the Form Buffer, Form$(0, 0). If you are not using random access files then this routine is not needed.

One important note is that UnPackBuffer places information into the Form$() array only, and does nothing with the screen. To update the screen with the contents of each field, it is necessary to use the PrintArray routine.

Example

This example fills the Form$(FldNo, 1) array element for fields five through ten with information contained in the form buffer:

```
CALL UnPackBuffer(5, 10, Form$(), Fld())
```

Value

BASIC function contained in FORMEDIT.BAS

■ Purpose

Returns the value of a numeric string.

■ Syntax

```
StringValue# = Value#(NumString$, ErrorCode%)
```

NumString$ - numeric string
ErrorCode% - returns 0 if no overflow occurred; -1 if an overflow occurred

■ Comments

Because Value has been defined as a function, it must be declared before it may be used.

This function is used to convert a numeric string to a double-precision number. The numeric string can contain such characters as dollar signs, commas, exponent signs, etc. If an overflow occurred when the string was being converted to a number, then the ErrorCode value will be -1; otherwise it will be 0.

■ Example

This example converts $5,000 to 5000:

```
StringValue# = Value#("$5,000",ErrorCode%)
```

VertMenu

BASIC routine 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. It is used to support multiple-choice fields in a form.

■ Syntax

```
CALL VertMenu(Item$(), Choice%, MaxLen%, BoxBot%, Ky$, _
   Action%, Cnf)
```

Item$ - conventional (not fixed-length) string array containing the list of menu choices

Choice% - indicates which choice was selected, and may also be pre-loaded to force a given choice to be highlighted automatically

MaxLen% - maximum length of any menu choice, thus establishing the menu width (choices longer than MaxLen% will be displayed truncated)

BoxBot% - line on which the last line of the window is to rest

Ky$ - holds the last key that was pressed by the user

Action% - tells how VertMenu should be used (see comments below)

Cnf - system environment TYPE variable (see page 116)

Comments

Displaying a list of items in a bordered window is only one of the features of VertMenu. Its real power comes from the use of the Action variable, and the routine's ability to be polled.

The Action variable has six different possible settings, to tell VertMenu how it is to behave. Each of the possible Action values is described below. Mouse support is built in to the vertical menus, but you may easily remove that code if you don't need it. 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 may then be examined to see what selection the user chose.

When Action is set to 1, VertMenu simply saves the screen and displays the menu. Control is then returned to the calling program immediately, however Action is 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 redisplay 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 set to 5, and this simply tells VertMenu 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 using ScrnSave and ScrnRest.

If you are not using multiple-choice fields you should use the "blank" VertMenu subprogram which is part of NOMULT.BAS. This resolves references to VertMenu without needing to load the full VertMenu source code.

WholeWordIn

BASIC routine contained in QSCALC.BAS

■ Purpose

To locate a substring within a string, using math operators as delimiters.

■ Syntax

```
CALL WholeWordIn(Text$, Word$)
```

Text$ - string to be searched
Word$ - word to be search for in Text$

■ Comments

This routine is used internally so that field names and other "words" can be found in formula strings.

DEVELOPING IN THE QB/QBX ENVIRONMENT

Whether you are using QuickBASIC's QB.EXE or the BASIC Professional Development System's QBX.EXE, you'll need to make certain environment variables and library routines available to the compiler environment.

Steps needed to prepare the environment properly are summarized below.

1. Switch to the QuickScreen directory so that BASIC source (.BAS) and include (.BI) files are in the current directory (if include files are elsewhere, be sure to set the INCLUDE environment variable properly).

2. If you want to create .EXE files from within the environment, you should place all .LIB files in the same directory. Then, make sure that the LIB environment variable is set to the correct directory path. If the linker cannot locate needed .LIB files, it usually generates an "Unresolved external reference" error.

   Environment variables are usually set from a batch file or directly at the DOS prompt. The suggested method, however, is to add such commands to your system's AUTOEXEC.BAT file. This way, environment information will be established each time the computer is turn on.

   If you prefer, you can create a batch file in your QuickScreen directory which you can run before your QuickScreen sessions. An example SET command is:

   ```
   SET LIB = C:\QB\LIBS
   ```

3. In order to run QuickScreen programs in the environment it is necessary to load a Quick Library which, at the very least,

contains the assembly language routines on which QuickScreen depends.

Only one Quick Library can be used, and it must be loaded when starting QuickBASIC from DOS. Furthermore, loading Quick Libraries in either QB or QBX reduces the amount of conventional RAM available, so it is important to keep such libraries as small as possible. We suggest using Crescent Software's Quick Library Make Utility — included with this package (please refer page 263 for instructions on its use).

4. If you are using AJS Publishing's db/LIB® product, you can add it to a Quick Library by specifying its .LIB file when using MAKEQLB.

5. Upon starting QB or QBX, you can load a Quick Library by using the /L command line switch. The following example illustrate how a Quick Library called MYQLB is loaded for QuickBASIC or QuickBASIC Extended, respectfully:

```
QB /L MYQLB
```

or

```
QBX /L MYQLB
```

6. Once QuickBASIC has started you should inspect the currently-set path options. To do this, access the (Options) **Set Paths...** menu command (not all version of QuickBASIC support this feature). You should ensure that all shown paths are accurate.

7. If you are using QuickBASIC version 4.x, you can free-up memory in the environment by compiling modules such as FORMEDIT.BAS and SCRNDISP.BAS and adding them to a Quick Library. This way, the source code does not occupy precious "base" RAM.

Users of QBX should not add compiled BASIC modules to their Quick Libraries since Quick Libraries are stored in conventional memory, whereas BASIC source modules are loaded into high memory automatically.

To add to a Quick Library manually (i.e., without the aid of MAKEQLB.EXE), you will first need to compile source modules using the /O compiler switch. In general the following modules should each be compiled:

```
BC FORMEDIT /O;
BC SCRNDISP /O;
BC FORMFILE /O;
BC SCRNFILE /O;
BC QSCALC   /O;
BC EVALUATE /O;
BC QEDITS   /O;
BC VERTMENU /O;
```

After each compilation an .OBJ file will be created. You will never need to recompile any of these modules unless, of course, you change their source code.

To create a Quick Library you'll need to gather each .OBJ file and any required .LIB files and link them together. It is in this step that you can add any object-format screen files you want to display from your program. These commands create and append object files to the MYLIB.LIB file:

```
LIB MYLIB+FORMEDIT+SCRNDISP+FORMFILE+SCRNFILE;
LIB MYLIB+QSCALC+EVALUATE+QEDITS+VERTMENU;
```

Once the MYLIB.LIB file exists, you can combine it with the supplied FORMS.LIB (for BASIC version 7.x, use FORMS7.LIB) library with this command:

```
LIB MYLIB+FORMS[7].LIB;
```

A Quick Library is created using LINK and the /Q option. Since our example requires several segments of memory, we'll use the /SEG:512 LINK option:

```
LINK /Q/NOE/SEG:512 MYLIB.LIB,,NUL,BQLB45;
```

DISPLAYING SCREENS FROM YOUR PROGRAM

Once you have designed a screen you may display it from QuickBASIC using a variety of methods. The method which is applicable will be based on the format in which the screen had been saved. Since most screens will be saved using a preferred format, our discussion presents three main sections: Object, QuickScreen, and Library screens. Each section discusses how to manipulate screens when they contain fields and when they do not.

QuickScreen Screens

Displaying QuickScreen screens requires the LoadScreen sub-program which is in the BASIC module SCRNFILE.BAS. Any QuickScreen screen is easily displayed using a call like the following:

```
CALL LoadScreen(ScreenName$, MonoCode%, Attribute%, ErrorCode%)
```

ScreenName$ is the full path and file name of the screen you wish to display (if no file extension is specified then .SCR is used by default). MonoCode may be used to adjust the screen for use on monochrome systems, and is fully discussed on page 114. Attribute specifies drop shadow and screen mode change options, and is fully discussed on page 111. ErrorCode returns a non-zero number if an error occurred when in the LoadScrn routine, and is fully discussed on page 112.

Library File Screens

Screens can be conveniently stored in a single screen library file. Later, any screen in the library file can be retrieved using a variety of methods.

DISPLAYING SCREENS DIRECTLY FROM DISK

The LibFile2Scrn subprogram, stored in the BASIC module SCRNLIB.BAS, (see page 165) displays a specified screen directly from disk. While screens displayed in this manner require time needed for disk access, no screen arrays are needed — which spares memory.

LOADING A LIBRARY

Loading a library consists of three steps: using the ScrnLibSize function, dimensioning an integer array, and calling LoadScrnLib. When complete, the program segment will look like:

```
Size% = ScrnLibSize%(LibName$)
REDIM ScrnLib%(Size%)
CALL LoadScrnLib(LibName$, ScrnLib%(), ErrorCode%)
```

The file name specified in LibName$ should include the full path and filename of the needed library.

DISPLAYING SCREENS FROM MEMORY

Displaying screens stored in a QuickScreen library file requires the Lib2Scrn subprogram, present in the BASIC module SCRNLIB.BAS. Before library screens may be used, however, the library file must be loaded.

An example of displaying screens from library files is available in the program DEMOCUST.BAS — included on the distribution diskette.

```
CALL Lib2Scrn(ScreenName$, ScrnLib%(), MonoCode%, Attribute%,_
   ErrorCode%)
```

The argument ScreenName$ is the name of the screen in the library you wish to display, and it must be 8 characters or less. ScrnLib() is an integer array which holds the screen to be displayed, and it is dimensioned before calling the LoadScrnLib subroutine; MonoCode, Attribute and ErrorCode are discussed beginning on page 111.

Object Screens

Screens saved as object files are probably easiest to generate from your programs, particularly if they do not contain field definitions. Unlike the other screen formats, object screens are added to a program during the linking process and therefore will be entirely contained in the executable file. Although this makes it unnecessary to have special screen files on disk, you should be aware that if a large number of object screens are linked with your program, the final .EXE size will increase and may possibly require more memory to run.

If you are generating object screens which are also forms, you should make use of the .DTA files which can be generated when forms are saved. Doing so produces a true standalone .EXE file (i.e., no other support files would be required by the program). .DTA files generated by QuickScreen contain DATA statements which can be used by your program.

Using .DTA files is quite simple. First, you'll need to include the .DTA file at the top of your program, with a statement like this:

```
'$INCLUDE: 'formname.DTA'
```

This makes the DATA statements for the specified file available to your program. The next step is to read these data elements with BASIC's READ statement. The first value of data tells how many fields are in the form:

```
RESTORE formname.FieldInfo
READ NumFlds
```

Now that the number of fields is known, you can dimension the form arrays:

```
REDIM Fld(NumFlds) AS FieldInfo
REDIM Field$(NumFlds, 2)
```

Now you can fill the Fld() array with information. But before doing this, you'll need to reset the READ pointer to the beginning of the FieldInfo DATA statements using the RESTORE command:

```
RESTORE formname.FieldInfo
FOR N = 0 TO NumFlds
   READ Fld(N).Fields
   READ Fld(N).Row
   READ Fld(N).LCol
   READ Fld(N).RCol
   READ Fld(N).StorLen
   READ Fld(N).FType
   READ Fld(N).RelFile
   READ Fld(N).RelFld
   READ Fld(N).Indexed
   READ Fld(N).FldName
   READ Fld(N).Decimals
   READ Fld(N).RelHandle
   READ Fld(N).Protected
   READ Fld(N).ScratchI
   READ Fld(N).LowRange
   READ Fld(N).HiRange
   READ Fld(N).ScratchS
NEXT N
```

What follows in each .DTA file is the help messages associated with each field. As you know, this information is held in the Form$(N, 1) array elements:

```
FOR N = 0 TO NumFlds
   READ Form$(N, 1)
   CALL ReplaceChar(Form$(N, 1), "'", CHR$(34))
NEXT N
```

Although optional, it is useful to call ReplaceChar if you wish each single quote character to be replaced by a double quote. Be aware that possessives and contractions (such as *Sue's* or *Don't* in your help messages) will be altered by ReplaceChar.

Once the help strings are loaded into the Form$() array, the next step is to load the field formulas into the Form$(N, 2) array elements. This is done as follows:

```
FOR N = 0 TO NumFlds
   READ Form$(N, 2)
   IF LEN(Form$(N, 2)) THEN
      CALL ReplaceChar(Form$(N, 2), "'", CHR$(34))
      CALL Tokenize(Form$(N, 2), Fld())
   END IF
NEXT N
```

Whenever there's a formula present, all single quotes must be converted to double quotes using ReplaceChar, and then Tokenize is called to convert formula strings into a format readable by the QSCalc routine.

The steps discussed above are shown in the demonstration program DEMOOBJ.BAS, and you are encouraged to cut and paste code from this file to your own program.

Object-saved screens cannot be tested while in the QuickBASIC environment unless they are added to the Quick Library used when starting QuickBASIC. Object screens are added to Quick Libraries in the same manner as other .OBJ files, and you may want to consider adding screens to the FORMS.QLB or FORMS7.QLB Quick Library we have provided.

There are two ways to display object screens, and the appropriate method depends on the wipe type specified when the screen was saved. The Direct-to-Screen wipe requires fewer programming steps than other wipes.

DIRECT-TO-SCREEN WIPES

Screens saved with the Direct-to-Screen wipe (available when saving object screens only) may be generated using a single call. If the name of the screen is MyScreen, the following syntax is used:

```
CALL MyScreen(MonoCode%)
```

Notice that the name of the routine being called is the base name (the object file name less the ".OBJ" extension) of the screen. MonoCode, which is fully discussed on page 114, may be used to adjust the screen for use on monochrome systems.

OTHER WIPES

If a wipe type other than Direct-to-Screen had been specified when the object file was saved in the QuickScreen editor, then there are more steps needed to generate the screen. Three methods are discussed below.

Displaying Screens With Originally-Saved Wipe

You can display object screens using the wipe originally specified when the screen was saved in the QuickScreen editor. To do this, you will need to create an array which can hold the screen data and its 2-byte header. This array size will need to be adjusted for various text-mode resolutions, and the following formula can be applied:

```
ScreenBytes = (ScreenWidth * ScreenHeight) + 2
```

Thus, for a standard 25x80 display, the result is (25*80)+2, or 2002. This total bytes size is used when dimensioning the integer Scrn() array as such:

```
DIM Scrn(ScreenBytes)
```

Next, the MScrnSave routine is used to copy the contents of the current screen image into the Scrn() array, starting at byte 2 — that is, following the 2-byte header:

```
CALL MScrnSave(1, 1, 25, 80, SEG Scrn(2))
```

When the object screen file is called, the wipe type used when the object file was saved will be returned. In addition, the object screen is loaded into the Scrn() array as an overlay. Thus, if the object screen does not occupy the entire display, then the Scrn() array will contain portions of the old screen image as well as the new:

```
CALL MyScreen(WipeType, SEG Scrn(2))
```

Having the WipeType allows you to use DisplayScrn to wipe the image onto the display.

Next, the header bytes must be initialized. To calculate the value of Scrn(0) and Scrn(1) you may apply this general formula:

```
Scrn(0) = upper-left column * 256 + upper-left row
Scrn(1) = lower-right column * 256 + lower-right row
```

In our current 25x80-screen image example, the column numbers should be 1 through 80; the row number should be 1 through the total number of rows available in the current screen mode. The DisplayScrn routine is then called. For a full 25x80 screen, this calling program segment would appear like this:

```
DIM Scrn(2002)
CALL MScrnSave(1, 1, 25, 80, SEG Scrn(2))
CALL MyScreen(WipeType, SEG Scrn(2))
Scrn(0) = 1 * 256 + 1
Scrn(1) = 80 * 256 + 25
CALL DisplayScrn(Scrn(), Element, MonoCode, WipeType)
```

Displaying Screens With A New Wipe

Changing the wipe type to something other than the one specified when saving the object screen is a simple matter of furnishing a new WipeType value when calling the DisplayScrn routine. For instance, if you want to always use the "Opening Curtain" wipe type (number 2, in Table XII), you can hard-code this value in your program:

```
DIM Scrn(2002)
CALL MScrnSave(1, 1, 25, 80, SEG Scrn(2))
CALL MyScreen(WipeType, SEG Scrn(2))
Scrn(0) = 1 * 256 + 1
Scrn(1) = 80 * 256 + 25
CALL DisplayScrn(Scrn(), Element, MonoCode, 2)
```

Displaying Screens Without A Wipe

To display an object screen and ignore its wipe, you will use a technique whereby the screen image is first copied into the Scrn() integer array. You then may optionally convert the screen to colors suitable for a monochrome system using the MakeMono routine realizing that the size of the screen is specified in MakeMono's second argument. Finally, the screen is placed on the display from the integer array using MScrnRest.

The calling program segment discussed above is:

```
DIM Scrn(2000)
CALL MScrnSave(1, 1, 25, 80, SEG Scrn(0))
CALL MyScreen(WipeType, SEG Scrn(0))
IF MonoCode% THEN CALL MakeMono(SEG Scrn(0), 2000)
CALL MScrnRest(1, 1, 25, 80, SEG Scrn(0))
```

PERFORMING DATA ENTRY

Forms may be processed from QuickBASIC using various routines. When you wish to use forms you will need to know how to access and initialize field data, how to use multiple-choice fields, and how to poll the EditForm routine. We suggest studying and experimenting with the commented demonstration programs included on the distribution diskette.

GENERAL CONCEPTS

In QuickScreen it is important to understand that the screen image and a form with which it may be associated are separate files and are therefore handled independently. We've discussed ways to load and display screens, whether object screens, QuickScreen screens, or QuickScreen library screens. Once a screen is displayed, "forms" are able to automatically direct data entry activity.

Form data can be stored in a variety of ways. The first is a standalone (.FRM) file which has the same name as the screen with which it is associated. The second is a form library (.QFL) file, in which several forms are stored. Here, special routines are used to access form information so that it can be activated. And third, DATA statements (stored in a .DTA Include file) may be used in any program so that external routines and files are not required in order to begin form processing.

Data Entry

EditForm is a BASIC subprogram which handles all aspects of data entry in a form. It is used in a manner similar to INKEY$. Thus, EditForm continually polls for input while in a loop. While looping, a program can perform other operations before and after each call to EditForm. In this manner you can acheive multi-tasking behavior.

EditForm always furnishes current information to the calling program. Since you can read fields or even change them, you can tailor the operation of any field in a form.

GENERAL PROCEDURES

DemoAny.BAS

This program is a good starting point for understanding how QuickScreen forms are used from your own programs. Although the source code is commented, we think it will be helpful to provide more information here (some of the material is repeated in the following section entitled *Detailed Procedures*, which begins on page 227).

First, it is suggested that you set all numeric variables to integers by default, and that you declare the required BASIC and assembler routines — especially functions. These steps are accomplished as follows:

```
DEFINT A-Z

'-------- Declarations
DECLARE FUNCTION Monitor% ()
DECLARE FUNCTION NumberOfFields% (FormName$)
DECLARE SUB EditForm (Form$(), Fld() AS ANY, Frm AS ANY,_
   Cnf AS ANY, Action)
DECLARE SUB GetFldDef (FormName$, StartEl, Fld() AS ANY,_
   Form$())
DECLARE SUB LoadScreen (ScreenName$, MonoCode%, Attribute%,_
   ErrorCode%)
```

Next, we suggest that certain Include files are specified at the top of your programs. These files supply constant definitions and often-used TYPE variables to your programs (see the discussion which begins on page 115).

The Include files required for this example are FIELDINF.BI (contains the FieldInfo TYPE and associated constants),

FORMEDIT.BI (contains the FormInfo TYPE and associated constants), DEFCNF.BI (contains the Config TYPE) and SETCNF.BI (initializes the Config TYPE variables). BASIC source code which Includes these files looks like:

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

Since the DEMOANY.BAS module uses forms, you will need to dimension the Frm variable to the FormInfo TYPE. This makes available to your program and to the supporting QuickScreen routines information about the current form:

```
DIM Frm AS FormInfo 'FormInfo TYPE variable
```

In general you will want to design programs which run as easily on systems with a color display as on those with a monochrome display. For this reason, you should determine whether the monitor being used is color or black-and-white. To both determine the monitor type and make use of the returned information, we suggest using the value for the video adapter card, which is stored in low memory at location 463H (monochrome systems usually use location B4H). You can also make use of BASIC's COMMAND$ variable to check whether a "/B"switch was specified by the user when your program was started from DOS. If present, the "/B" option tells your program to force monochrome colors on all screens.

This logic is presented in BASIC below:

```
MonoCode% = ABS(PEEK(&H463) = &HB4 OR INSTR(COMMAND$, "/B"))
```

Before allocating memory to the arrays which are used to control the form, it is useful to determine the number of fields present. When using standalone .FRM files (as in this example) you will need to use the NumberOfFields function. If the form is in a library .QFL file, then you will need to use LibNumberOfFields.

In the next excerpt, we determine the number of fields in the form using the former method:

```
NumFlds% = NumberOfFields%(FormName$)
```

The integer value returned in NumFlds is used to dimension the Fld() and Form$() arrays.

Another array which can be dimensioned at this time is Choice$(). This array's first subscript provides the maximum number of choices you will need for any multiple-choice field, while the second subscript is one less than the total number of multiple-choice fields in the form. If your form has no multiple-choice fields, the Choice$() array must be dimensioned, and using zero as each subscript is a minimum requirement:

```
REDIM Fld(NumFlds) AS FieldInfo
REDIM Form$(NumFlds, 2)
REDIM Choice$(0, 0)
```

The next step is to load the form definition file. Once again the routine you'll need to use will depend on how the form was stored. For standalone (.FRM) files, you will need to use the GetFldDef routine; for form library (.QFL) files, use LibGetFldDef:

```
CALL GetFldDef(FormName$, Zero%, Fld(), Form$())
```

Next you can display the screen image with LoadScreen:

```
CALL LoadScreen(FormName$, MonoCode%, Attribute%, ErrorCode%)
```

At this point both the form and its image are on the screen. In order to allow input, the form has to be activated. This is done by calling EditForm with an Action of 1, which sets up internal pointers and displays initial field values from the Form$(N, 0) array elements.

After the first call to EditForm, Action will be automatically set to 3, so that polling can continue. It is up to you to examine

Frm.KeyCode so that certain keypresses can be recognized. In this example, both <Esc> (which has a keycode value of 27) and <F2> (which has a keycode of -60) are used to terminate the form:

```
Action = 1
DO
   CALL EditForm(Form$(), Fld(), Frm, Cnf, Action)
LOOP UNTIL Frm.KeyCode = 27 OR Frm.KeyCode = -60
END
```

DETAILED PROCEDURES

The prior section covered the fundamentals of the DemoAny program. The following sections provide slightly more detailed discussions for using forms.

Setting Up a Form

Before a form can be processed from QuickBASIC, there are some suggested as well as required steps which must be taken. The optional steps involve such things as clearing the screen before generating the form, initializing the mouse, printing explicit instructions to the user, and setting the insert status or other features in the Frm TYPE variable. It is beyond the scope of this user's guide to cover *all* such optional aspects of programming. Instead, we'll provide the basics below, and encourage you to experiment on your own, using portions of the included demonstration programs as building blocks for your own programs.

The best way to process forms in QuickBASIC is to follow the steps outlined below and discussed next.

- Specify Include files
- Dimension mandatory arrays
- Load and display the form
- Initialize field and form elements

SPECIFY INCLUDE FILES

As discussed earlier on page 224, you must make available certain TYPE declarations and constant assignments to the calling program. In addition, if you have generated .BI Include or .DTA data files for your forms, you may want to specify them at the top of your program.

The BASIC statements for required include files are summarized below:

```
'$INCLUDE: "DEFCNF.BI"
'$INCLUDE: "FIELDINF.BI"
'$INCLUDE: "FORMEDIT.BI"
'$INCLUDE: "SETCNF.BI"
'$INCLUDE: myform.BI      'this Includes the TYPE
                          '   structure for your form
'$INCLUDE: myform.DTA     'this Includes DATA
                          '   statements for your form
```

DIMENSION MANDATORY ARRAYS

The arrays which must be dimensioned are among those discussed in the section called ***Arguments*** (see page 110). Although these arrays are dimensioned to zero elements at the start, you should realize that all are REDIMed later when needed.

The required dimension statements are shown below, and they rely on the Include files mentioned in the previous section:

```
DIM Frm AS FormInfo
REDIM Fld(0) AS FieldInfo
REDIM Form$(0, 0)
REDIM Choice$(0, 0)
```

LOAD THE FORM

Form information is loaded into the form arrays using GetFldDef. This routine uses the Fld and Form$ array variables and requires that each be dimensioned before the call. These arrays are dimensioned using information from the NumberOfFields function.

Once the form is loaded, its screen image is displayed using any of the screen-display methods discussed earlier.

These program instructions are summarized below.

```
StartEl% = 0
Size% = NumberOfFields%(FormName$)
REDIM Fld(Size%) AS FieldInfo
REDIM Form$(Size%, 2)
CALL GetFldDef(FormName$, StartEl%, Fld(), Form$())
                          'now display the screen
```

INITIALIZE FIELD AND FORM ELEMENTS

Before and while a form is being used you may set certain field and form values. For example, you could use the Form$() array to assign default values to particular fields. Or you could use the Frm TYPE variable to set the form's insert status. These form variables were presented in the section entitled ***Arguments*** (please see page 110). We have chosen to illustrate here how to optionally set the insert status of the form, how to set a multiple-choice array, and how to create a few default field values.

Setting The Insert Status

If you wish to set the insert status, you will need to use the Frm TYPE variable. The following statement initially sets a form's insert status to "off":

```
Frm.Insert = 0   'set insert off
```

Setting Up Multiple-Choice Fields

If you are using multiple-choice fields in your form there are certain measures which you must take for them to work properly.

First, you will want to include the module VERTMENU.BAS instead of NOMULT.BAS. This way, the full VertMenu subprogram will be available to your program.

Multiple-choice fields require initializing the COMMON string array Choice$() (the COMMON attribute is assigned in the FORMEDIT.BI file). This is a two-dimension array which is dimensioned as follows:

```
DIM Choice$(MaxChoices%, NumChoiceFields%)
```

MaxChoices is the maximum number of choices which any vertical menu would need to hold. The zeroeth first subscript element (i.e., Choice$(0,N)) is always reserved for the field number(s) to which the multiple-choice array is to be linked.

The second subscript, NumChoiceFields, tells how many *unique* multiple-choice menus are needed, and it begins at 0 (some fields share the same multiple-choice information, so there could be more multiple-choice fields than this subscript indicates).

For example, suppose a form has only 2 multiple-choice fields and that the first, field number 2, is for soft drink selections while the

second, field number 6, is for T-Shirt sizes. Let's also suppose that there are 5 soft drinks and 3 T-shirt sizes available.

One way to redimension and initialize the Choice$() array for this example would be:

```
REDIM Choice$(0 to 5, 0 to 1)

Choice$(0, 0) = "2"       'field number 2
Choice$(1, 0) = "Pepsi"
Choice$(2, 0) = "Coke"
Choice$(3, 0) = "Dr. Pepper"
Choice$(4, 0) = "7-Up"
Choice$(5, 0) = "Canada Dry"

                          'field number 6, 15 and 16 use this
                          'multiple-choice array
Choice$(0, 1) = "6, 15, 16"
Choice$(1, 1) = "Small"
Choice$(2, 1) = "Medium"
Choice$(3, 1) = "Large"
```

Notice that the element corresponding to a 0 first-subscript is the field number of a multiple-choice field in the form, and that this field number need not be in any particular order. Furthermore, this zeroeth element can specify that several fields share a single multiple-choice list of items (these are specified in a comma-delimited list of field numbers). This allows the Choice$() array to be as small as possible.

Creating Default Field Values

The way to place default values into fields on a form is to assign elements in the Form$() array (see page 113 for more information). When using Form$() you will generally modify only the field's data, however you may want change the help text and formulas as well.

For instance, suppose that field 1 holds the current date and field 4 holds a tax rate. The calling program could initialize these fields like this:

```
Form$(1, 0) = DATE$
Form$(4, 0) = "7.5%"
```

One last comment regarding Form$() is that for notes (or multi-line text) fields it returns a single string. Blank lines in notes fields are returned as a CHR$(20).

Using EditForm

Once a form is properly initialized, the EditForm is called to allow data entry. EditForm works much like BASIC's INKEY$, and, as such, it is designed to be called repeatedly in a loop. When a routine is called in this way it is said to be *polled*. Such routines offer a tremendous level of flexibility and power since the calling program becomes an extension of the processing logic. For instance, polling offers the ability to modify a form *on-the-fly*, that is, *while* the form is being used.

EditForm processes each keystroke or mouse event immediately, and then returns to the calling program. Often, however, EditForm receives no input whatsoever, and simply returns to the calling program for its next iteration. What's important is that control is returned to the calling program *between keystrokes and mouse events*, which makes it possible to monitor input *as* it occurs.

Earlier, we discussed the form variables: the Form$() string array, the field information Fld() TYPE array, and the form information Frm TYPE variable. Each variable contributes or provides information about a form, and, together, these variables can be both accessed and changed while a form is being used.

These important form variables are reviewed below, in the context of how they can be used with EditForm.

FORM$()

In general, the Form$() string array (see page 113) is used to pre-fill a form or to examine data provided by the user. If needed, data in other fields can be changed based on information already entered. For example, if one field on your form is for Gender, and accepts "F" for Female and "M" for Male, then a subsequent field for a Salutation could be automatically pre-filled with "Mrs." or "Mr.", respectively. Doing this with EditForm is easy, as you'll see later.

FLD() TYPE ARRAY

The Fld() TYPE array (see page 118) determines field attributes. Changing information in this array enables you to protect fields or change valid data ranges for specific fields on-the-fly. For instance, an invoice form might ask for a saleperson number. *After* this field is complete, you could protect it so that once the order is taken, the saleperson's code cannot be changed.

FRM TYPE VARIABLE

The Frm TYPE (see page 120) variable provides more general information about a form, such as whether any information on it has changed; which field is currently-active; and which key was last pressed. Often it is desirable to change where the cursor is on a form, based on certain data. For example, if an order form allows purchases with a credit card, it usually also has fields for the credit card number and its expiration date. However, if the

sale is paid for by check, you may want to automatically skip over the credit card fields and jump to the Check Number field.

In the following program fragment, EditForm is initially called with an Action of 1, which fills and initializes the form. Then, after each polling cycle, the program checks to see if the user has left the current field. This test is accomplished by comparing the value of Frm.PrevFld (the previous field number) with Frm.FldNo (the current field number). This test is true for one polling cycle only — after this, Frm.PrevFld becomes Frm.FldNo. If the test is in fact true, the program starts a SELECT CASE statement which examines the value of Frm.PrevField to see which field the user just left.

In this example, the program checks whether "F" or "M" has been entered into the Gender field (field number 5). If the field is null, then no action is taken. If it contains an "F", then the Salutation field (field number 8) is set to "Mrs.". It is likewise set to "Mr." if the Gender field contains an "M":

```
Action% = 1
DO
   CALL EditForm(Form$(), Fld(), Frm, Cnf, Action%)
   IF Frm.PrevField <> Frm.FldNo THEN
      SELECT CASE Frm.PrevField
         CASE 5
                                'check whether field 5 has changed
            IF Frm.FldEdited
               IF Form$(5, 0) = "F" THEN
                  Form$(8, 0) = "Mrs."
               ELSE
                  Form$(8, 0) = "Mr."
               END IF
            END IF
            Action% = 1         'this forces the next call to
                                'EditForm to refresh the form
         CASE ELSE
            REM
      END SELECT
   END IF
LOOP Until Frm.KeyCode = 27
```

There are many tricks you can perform by manipulating the QuickScreen form variables which have been presented. One common example is building shortcut keys into a form so that a user can skip large portions over several fields at a time. For complex forms this is useful. To illustrate, you could decide that field 10 is to be associated with the <F2> key, while field 20 is to be associated with the <F3> key. This way, you could examine Frm.KeyCode each time EditForm is called. You could then test for <F2> or <F3> and set Frm.FldNo so that is points to a new field. This would allow a user to move instantly between fields 10 and 20 with a single keystroke!

RANDOM-ACCESS FILE I/O

One of the easiest ways to save and restore information from and to a form is to use random-file access techniques. This method relies on the fixed-length structure of the form buffer, Form$(0, 0). Using random-file I/O allows you to create a single database file and quickly access any record it contains.

QuickScreen provides a demonstration of the random-file access method in the source module DEMOCUST.BAS.

Setting Up

To prepare for using a random file, you will need to open the data (.DTA) file. This is achieved using the OpenFiles routine, which also opens associated notes files and ensures that the form buffer is properly sized to accommodate random-file data. This step is summarized as:

```
CALL OpenFiles(FormName$, Form$(), Fld())
```

Once the data files are open and the form arrays are initialized, you can use the GetRec and SaveRec routines to load and save records, respectively. Before retrieving records, you should know the upper limit — that is, the total number of records currently stored. This value is determined by dividing the length of the data file by the record length of the form being used:

```
LastRecord& = LOF(Fld(0).RelHandle) \ Fld(0).Row
```

Notice that the OpenFiles routine assigned to Fld(0).RelHandle the handle for the .DTA data file. Although you probably won't need to access it, OpenFiles also assigns to Fld(0).ScratchI the handle for the .NOT notes file.

Retrieving Records

To retrieve a record from the data file, simply call the GetRec routine and furnish a valid record number in RecNo:

```
CALL GetRec(RecNo&, Form$(), Fld())
```

This loads only the form buffer, Form$(0, 0), with information. You will need to use the UnPackBuffer routine to transfer form buffer contents to the individual form data elements:

```
CALL UnPackBuffer(FirstFld%, LastFld%, Form$(), Fld())
```

To fill the entire form with information from the form buffer, FirstFld should be 1, and LastFld should be assigned to Fld(0).Fields, which holds the number of fields in the form. If you wish only to fill a portion of the form with information from the form buffer, you can specify other values for FirstFld and LastFld, realizing also that several ranges of fields can be filled by calling UnPackBuffer several times with different values.

One entirely optional step, which serves only to simplify programming and increase readability of your source code, is to

copy information from the form buffer to a TYPE array corresponding to the current form. This allows you to refer to fields by the name in the TYPE structure, rather than by accessing the Form$() elements. Although simple, this process is a bit involved, and is fully-discussed in the example for the BCopy routine (see page 130 for more information).

The next step is to transfer information from the form data elements onto the screen. The easiest way to do this is to call EditForm with an Action of 1 (see page 232 for details). This ensures that all fields on the form are current. The alternative is to call the PrintArray routine, which provides more flexibility by allowing a specific range of fields to be updated.

Saving Records

Saving records is accomplished by calling the SaveRec routine. All you need to do is furnish the record number in RecNo:

```
CALL SaveRec(RecNo&, Form$(), Fld())
```

This statement writes information to the .DAT and .NOT data files opened earlier by the OpenFiles routine.

Clearing A Form

The best way to clear a form is to set each Form$(N, 0) element to null. You may then want to pre-set certain fields in the form before allowing a new record to be entered. The code fragment below clears the Form$() data elements, sets the date and time information, and sets Action to 1 before EditForm is called again.

```
        .
        .
        .
FOR N = 1 TO Fld(0).Fields              'Clear all fields
   Form$(N, 0) = ""
NEXT N

Form$(12, 0) = DATE$                    'Set date/time info.
Form$(15, 0) = TIME$
Action = 1                              'Prepare to refresh with
                                        '    EditForm
        .
        .
        .
```

NOTES FIELDS

Using Notes

If you plan to use multi-line edit notes in your form, you will need to replace the NONOTES.BAS stub-file module with QEDITS.BAS.

Information from notes fields are stored in a separate file with a .NOT extension. For each note field in a form, the Form$(0, 0) array element contains a long-integer pointer into the notes (.NOT) data file. This pointer accesses a two-byte integer in the notes file which gives the length of the string stored in the following bytes. The next note in the notes file immediately follows, and also begins with a two-byte integer giving the length of the note text, and so on. This arrangement makes a compact notes data file possible.

If you want to pre-fill a notes field with data, all you need to do is to fill the Form$() array with text. Recall that text is stored as a

single line of information, with CHR$(20) used whenever a blank line is needed. Thus, to pre-fill field 6 in your form, you would write to Form$(6,0):

```
Form$(6,0) = "This is line one of a note field and ":CHR$(20):_
   "this is line two."
```

Saving And Retrieving Notes Data

The best way to both save and recall information from notes fields is to use the SaveRec and GetRec routines in the RANDOMIO.BAS BASIC module. Please see pages 198 and 164 for more information.

RELATIONAL FIELDS

QuickScreen provides ways to store information which can be used to create *relational fields* — fields which two or more forms have in common. For instance, a customer number could be the record key to a CUSTOMER file, and the same customer number may also appear as a field in an INVOICE file. Related fields make it easy to access information stored in different data files which is linked together by a common field.

While QuickScreen does not provide support for related data files, it does offer a means by which your own programs can store and access information for related fields.

If you want to create a related field for field N in the current form, you would need to store information in the Fld(N).RelFile and Fld(N).RelFld field array elements. These two type elements describe the related file and field names, respectively.

If you want to access the related file by its handle rather than by its name, you can OPEN the file and store its handle in the Fld(N).RelHandle TYPE array element. Using the file handle provides fast access, but this technique naturally requires a file which is already open.

INDEXED FIELDS

Just like related fields, QuickScreen does not provide support for managing indexed fields, however it does provide a way to flag that certain fields are indexed.

For instance, if field N of the current form is indexed, you can store a flag in the Fld(N).Indexed field array element. Indexes are used to accelerate record searches and sorts, and most books on database design explain them in detail.

MULTI-PAGE FORMS

QuickScreen has the ability to manage multi-page forms. This feature exploits the ability of the Frm() and Form$() arrays to hold several forms at once. Each "page" of a multi-page form is a standalone screen — created separately in the Screen Designer. Screens are designed so that they appear to visually follow one another — either from top to bottom (for tall forms) or from left to right (for wide forms). Using various wipes, you can achieve the illusion of moving "up" and "down" through a long form. Thus, pressing <PgUp> could scroll a new page down from the top of the screen using the "Push Down" wipe, while <PgDn> could make use of the "Push Up" wipe. This *is* an effective illusion. Even though each screen is entirely separate, each accesses a unique range in the form arrays, making multi-page forms possible.

To illustrate the need for multi-page forms, suppose you want to allow 60 lines for item-entry on an invoice form. In 25-line mode,

you would need three different "pages". The first page would contain header information, such as customer information, and would also begin the columnar section which forms the line-item section of the invoice (this is where part numbers, descriptions, and price information is entered). The second page would probably consist entirely of the line-item section of the invoice. The last page would complete the line-item section, and also have "footer" information, such as price totals and special shipping instructions.

The DEMOPAGE.BAS demonstration program shows the basics of multi-page form processing and includes commented source code. It serves as an example, and, as written, is limited to a two-page form stored at the beginning of a form library file. The discussion which follows attempts to give a more generic approach to handling multi-page forms which may exceed two pages, and which may not be stored consecutively in the form and screen libraries.

Implementation

In terms of programming, the concept behind multi-page forms is simple. Every screen will be assigned a *number*. This way, your program can increment and decrement a screen counter to access the next or previous screen image, respectively. Typically, when a user presses <PgDn>, or moves beyond the last field on the current "page", you will increment the screen counter and display the next "page" of the form. Likewise, pressing <PgUp>, or moving beyond the first prompt, accesses the previous "page".

To illustrate this concept, recall the LibScrName function and Lib2Scrn routine. If a screen counter is stored in the variable ScrCounter, then consider the following code:

```
ScrName$ = LibScrName$(ScrCounter%, ScrnLib())
CALL Lib2Scrn(ScrName$, ScrnLib(), MonoCode%, Attribute%,_
   ErrorCode%)
```

This program fragment takes the number stored in ScrCounter to generate a screen name. The Lib2Scrn routine then uses the screen name to access the library and place the image on the display.

The technique above works only if the screens stored in the screen library are in the proper sequence. When you create a multi-page form, you should store the screens in the library file so the page one of a three-page form appears immediately before page two, and that page two appears immediately before page three. To change the order of screens in a screen library file, you can open the library in the Screen Designer and save each screen it contains to individual screen files. Then, you can clear the library and begin to build it from scratch — paying careful attention to the screen order.

Another way of handling multi-page forms is to create a numbered series for your screen names. For example, the pages of a five-page invoice form could be named INV1, INV2, INV3, INV4, and INV5. This way, your program can use the LibNo function to translate the screen name into a screen number. You'll still be able to use a screen counter variable since you can simply append its value to the base name for the screen. Consider the following:

```
ScrName$ = "INV"+LTRIM$(STR$(ScrCounter%))
CALL Lib2Scrn(ScrName$, ScrnLib(), MonoCode%, Attribute%,_
   ErrorCode%)
```

This technique is much like the one presented earlier, but it uses the LibNo function and relies on screens which have been numbered sequentially. The screen counter is then concatenated onto the base screen name to produce names such as INV1 or INV2.

To dimension form arrays for multi-page forms, use the LibSize function to calculate the size of a screen library array, and use the LibNumberOfFields function to determine the size of the form arrays. Since several screens are used, you will need to call

LibNumberOfFields for each screen in your multi-page form. This way, you can total the number of fields in each form.

To do this effectively, you should first determine the number of screens in your screen library, which is obtained by examining element zero of the screen library array (i.e., ScrnLib(0)):

```
            .
            .
            .
                          'Get required size for the array that
                          '   will hold the screen library
LibSize = ScrnLibSize%(LibName$)
                          'Create the array
REDIM ScrnLib(LibSize%)
                          'Load the screen library into it
LoadScrnLib LibName$, ScrnLib%(), ErrorCode%

NumFlds% = -1
NumFrms% = ScrnLib%(0)    'Get number of forms in library

FOR Scr% = 1 TO NumFrms%
                          'Get form name from screen number
   FormName$ = LibScrName$(Scr%, ScrnLib%())
                          'Add number of fields to total
   NumFlds% = NumFlds% + LibNumberOfFields%(LibName$,_
      FormName$) + 1
NEXT Scr%                 'DIM the field information array to
                          '  combined size of both forms
REDIM Fld(NumFlds%) AS FieldInfo
                          'DIM the form data array
REDIM Form$(NumFlds%, 2)
            .
            .
            .
```

The next step is to fill the newly-dimensioned Fld() TYPE array and the Form$() string array with the field data stored in the .QFL file. The best way is to use the LibGetFldDef routine, which is able to *add* form information, thereby building multi-page arrays. It is also best to use the Fld().Fields element of the Fld() array so that the total number of fields in a given form is known.

```
StartEl% = 0                    'Load first form to beginning of array
                                'Get form name from screen number
FOR Scr% = 1 TO NumFlds%
   FormName$ = LibScrName$(Scr%, ScrnLib())
                                'Load field information @ "StartEl"
   CALL LibGetFldDef(LibName$, FormName$, StartEl%, Fld(),_
      Form$(), ErrorCode%)
                                'Bump "StartEl" to next available
   StartEl% = StartEl% + Fld(StartEl%).Fields + 1
NEXT Scr%
```

Now that the form arrays are loaded and properly initialized, you can start to use the EditForm routine. While using EditForm, if <PgUp> is pressed then Frm.StartEl is assigned to 1 less than the starting field number for the current form. If <PgDn> is pressed then Frm.StartEl is assigned to 1 more than the highest field number on the current form. Thus, a calling program can check Frm.StartEl to determine whether either <PgUp> or <PgDn> has been struck.

> ***Because a multi-page form creates an extra element in the form arrays for each form "page", we suggest using the Form$(0, 0) form buffer (rather than the Form$(N, 0) data elements) when putting or getting records to and from a data file.***

Alternatively, you can also check Frm.Keypress for specific keystrokes (such as <PgUp>, <PgDn>, or function keys you wish to use) to determine whether the user is trying to access a prior or next "page" in the form.

This former technique (making use of Frm.StartEl) is demonstrated below:

```
      .
      .
      .
Action% = 1
DO                              'Poll the editing procedure
   CALL EditForm(Form$(), Fld(), Frm, Cnf, Action%)

   '----- If the user pressed PgUp or PgDn or moved off the top
   '----- or bottom of the form, "StartEl" will be updated by
   '----- "EditForm" so we need to check it.  The last value is
   '----- saved in "LastStartEl" for use as a comparison.
```

```
    IF Frm.StartEl <> LastStartEl% THEN
                            'Previous page?
       IF Frm.StartEl < LastStartEl% THEN
                            'Yes set previous page number
          Scr% = Scr% - 1
                            'Next page?
       ELSEIF Frm.StartEl > LastStartEl% THEN
                            'Yes set next page number
          Scr% = Scr% + 1
       END IF               'Display the screen
       CALL Lib2Scrn(LibScrName$(Scr%, ScrnLib()), ScrnLib(),_
          MonoCode%, -2, ErrorCode%)
                            'Save the new "StartEl"
       LastStartEl% = Frm.StartEl
    END IF                  'Keep editing until the user presses
                            '  the Escape key.
LOOP UNTIL Frm.KeyCode = 27
```

PROGRAMMING TIPS

The following sections provide additional programming tips which you may find both interesting and useful.

Manually Manipulating Data On-The-Fly

Since EditForm is polled, the calling program has the opportunity to carry out some task between polling iterations. The example show how to update the time on the screen each second. The time is printed on the first line near the right margin.

```
DO
   CALL EditForm(Form$(), Fld(), Frm, Cnf, Action)
   IF CLNG(TIMER) <> T& THEN
                            'Display the time
                            '   each second to
      T& = TIMER            '   show how things
      LOCATE 1, 70, 0       '   can be done while
      PRINT TIME$;          '   a form is being
   END IF                   '   edited.
LOOP UNTIL Frm.KeyCode = 27 'Keep editing until
                            '   user presses <Esc>.
```

When this routine executes, the time is updated while the form remains fully-capable of accepting user input. This example demonstrates how two activities (processing a form and updating the time) can seem to occur simultaneously.

Assigning Variables To Refer To Fields

QuickScreen has a function called FldNum (see page 157) which converts field names to numbers and makes it unnecessary to change program code when your form is modified.

For instance, if Form$(9, 0) currently points to a customer phone field, and you add two new fields to the form, the customer phone field could become Form$(12, 0). If Form$() array subscripts are used to access a field's data, it would be necessary to change array subscripts throughout your program. In the example provided, all Form$(9, 0) references would need to be changed to Form$(12, 0) so that they accurately point to the new location of the phone field.

Referring to fields using variable names is easy. Consider this program fragment:

```
DateFld% = FldNum%("INVDATE", FLD())
.
.
.
Form$(DateFld%, 0) = DATE$
```

This example shows how the variable DateFld is assigned to the field number corresponding to the field called "INVDATE". If the form changes and the field position of INVDATE is altered, this method ensures that the correct field receives the DATE$ information.

Updating Form Data Using SaveField

To update the Form$(0, 0) form buffer, the SaveField routine should be called. Recall that this routine (discussed on page 197) verifies data in the field specified before copying it to the form buffer.

```
CALL SaveField(DateFld%, Form$(), Fld(), BadFld%)
```

Recalculating Fields Using CalcField

If you change a field by changing a Form$(N, 0) element which affects a calculated value somewhere on the form, you will need to call CalcFields so that the field is properly recalculated. (See page 136 for more information.)

Converting Formatted Strings to Numbers

To quickly convert a formatted string to a double-precision number, you can use the Value function. If you need to extract a number from the IEEE string imbedded in the form buffer, you can use the appropriate conversion scheme from the following:

```
Num% = CVI(MID$(Form$(0, 0), Fld(FldNo).Fields, 2))
Num& = CVL(MID$(Form$(0, 0), Fld(FldNo).Fields, 4))
Num! = CVS(MID$(Form$(0, 0), Fld(FldNo).Fields, 4))
Num# = CVD(MID$(Form$(0, 0), Fld(FldNo).Fields, 8))
```

In each line above, MID$ is used to access a specific number of bytes from the form buffer. The starting character position, or offset, into Form$(0, 0) is supplied by Fld(FldNo).Fields. Then, the appropriate number of bytes are read, such as 2 for integer values. Once the string is properly read, the CVx operation converts the string to a number, and assigns the result to Num.

Redisplaying Form Data Using PrintArray

After you have made the desired changes to the form data, you can redisplay information in the form by calling the PrintArray routine (see page 189).

CREATING STANDALONE PROGRAMS

Standalone programs are generated by linking compiled object files using the LINK.EXE program supplied with your version of BASIC. Compiling programs for standalone use is a relatively easy task. However, you must first be aware of BASIC Make files before using the compiler and linker.

MAKE FILES

Make files are created by the QuickBASIC environment whenever a program requiring more than one module has been saved. They are ASCII files, with a .MAK file extension, and simply list the names of other modules which must be present in order for the main module to run. For example, the DISPLAY.BAS program also requires the SCRNDISP.BAS, SCRNFILE.BAS and SCRNLIB.BAS modules. All these modules must be compiled to object files and then linked together with the FORMS.LIB or PRO.LIB libraries (for BASIC version 7.1, use FORM7.LIB or PRO7.LIB).

COMPILING MODULES

BASIC source files are compiled using the BC.EXE command line compiler as such:

```
BC MyProg.BAS /O/S;
```

This will create an .OBJ file if the program compiled successfully. You will need to compile each BASIC module separately.

BASIC Program Name	Required BASIC Modules (.BAS files)
DISPLAY	DISPLAY, SCRNDISP, SCRNFILE, SCRNLIB
DEMDBLIB	DEMDBLIB, DBLIBMOD, FORMEDIT, NOCALC, QEDITS, SCRNDISP, VERTMENU, FORMFILE, SCRNFILE
DEMOCUST	DEMOCUST, FORMEDIT, NOCALC, NOMULT, SCRNDISP, QEDITS, FORMLIB, SCRNLIB, RANDOMIO
DEMOINV	DEMOINV, QSCALC, EVALUATE, FORMEDIT, QEDITS, SCRNDISP, VERTMENU, SCRNFILE, FORMFILE
DEMOPAGE	DEMOPAGE, QSCALC, EVALUATE, FORMEDIT, QEDITS, SCRNDISP, VERTMENU, FORMLIB, SCRNLIB

Table XXV: QuickScreen Programs And Required Modules

LINKING

Once you have compiled all your programs to .OBJ files, you will need to create a final standalone .EXE program. This is done by linking object files with the provided FORMS.LIB library (for BASIC version 7.x, use FORMS7.LIB).

If you are compiling and linking manually from DOS, then you would specify all your BASIC-compiled object modules, along with FORMS.LIB (or FORMS7.LIB), like this:

```
LINK PROG1.OBJ+PROG2.OBJ,,NUL,FORMS[7].LIB
```

If you prefer you can start LINK without any options, and wait for it to prompt you for the information it needs. To do this just type:

```
LINK
```

You may also specify more than one library when linking. For example, if you need assembler routines from both FORMS.LIB and a library file called MYSTUFF.LIB, you may tell LINK to use both of them:

```
LINK PROG1.OBJ+PROG2.OBJ,,NUL,FORMS[7] MYSTUFF
```

You may also add single object modules when linking, even if they are not present in a library at all:

```
LINK PROG1.OBJ+PROG2.OBJ+MYOBJECT.OBJ,,NUL, FORMS[7] _
   MYSTUFF
```

If you do wish to combine several libraries into a single .LIB 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 LIBRARY1.LIB+LIBRARY2.LIB+LIBRARY3.LIB
```

One useful link option you should be aware of is the /E command line switch. When LINK is invoked with /E, it creates an .EXE file in a special "packed" format. Not unlike the various archive programs, the 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. The /E switch is specified like this:

```
LINK PROG1.OBJ+PROG2.OBJ,,NUL,FORMS[7] MYSTUFF /E
```

A packed program will require less disk space, however it of course requires the same amount of memory when it is run.

QUICKSCREEN UTILITIES

SLIDE SHOW DISPLAY PROGRAM

The purpose for the DISPLAY program is to allow the creation of self-running slide show presentations. DISPLAY works by reading a special ASCII command file which you create, and then executing instructions that specify which images to display and how to display them. Although DISPLAY has been provided to you as a BASIC source module only, you may compile and link it so that it may be used as a command-line utility.

To do this, all modules associated with DISPLY.BAS must first be compiled to .OBJ files with BC. Inspecting the DISPLAY.MAK make file reveals four modules which are compiled like this:

```
BC DISPLAY /O;
BC SCRNDISP /O;
BC SCRNFILE /O;
BC SCRNLIB /O;
```

The next step is to link these modules together. Since DISPLAY contains assembler routines as well, it will be necessary to specify the FORMS.LIB library when linking (for BASIC version 7.x, use FORMS7.LIB):

```
LINK DISPLAY.OBJ+SCRNDISP.OBJ+SCRNFILE.OBJ+SCRNLIB.OBJ,,NUL,_
   FORMS[7].LIB
```

Linking as shown above creates DISPLAY.EXE, which you may run from DOS using the following syntax:

```
DISPLAY script.CMD [/B]
```

where script.CMD is the name of the script command file you wish for DISPLAY to use. If the "/B" command-line switch is used then DISPLAY will generate screens so that they are properly displayed on monochrome systems.

Command	Description
CLS	Clears the screen using the current background color
COLOR fore, back	Sets the foreground and background color for subsequent Message or CLS statements.
Goto labelname	Execution of the Display command file continues at the first command after the labelname specified.
KeyPress	Waits a for key press.
Labelname:	A label may be placed on its own line and must end with a colon. The label can be any name other than a Display command.
LOCATE y, x	Positions the cursor at row y and column x.
LoadLib filename.QSL	Loads the library specified in filename into memory.
Message "text"	Displays the literal string specified.
Pause n.n	pauses for the number of seconds specified. Notice that decimal values are allowed.
Display scrnname n	Displays the screen specified from the library which was loaded using the LoadLib command. The number n may be -1 to suppress mode changes, -2 to suppress drop shadows, or -3 to suppress both.
REM or '	Signifies a comment line the same way as in BASIC.

Table XXVI: DISPLAY Utility Commands

A DISPLAY command file is a text-only file, and may be created using your favorite text editor. The commands available are summarized in Table XXVI.

You should know that DISPLAY command names need not be uppercase when used in the script file. Also, there are several features which control DISPLAY while it is running. These controls are summarized in Table XXVII.

Key	Effect on Slide Show
+	Increases the speed of the slide show.
-	Decreases the speed of the slide show.
<Enter>	Sets speed to normal.
<Esc>	Ends the slide show.
<Space>	Freezes the slide show until a key is pressed.

Table XXVII: DISPLAY Runtime Keyboard Control

SCREEN CAPTURE PROGRAM

SCRNCAP.EXE is a "Terminate and Stay Resident" (TSR) utility which is provided for you on the QuickScreen distribution diskette. Written using Crescent Software's P.D.Q. alternate QuickBASIC library, this program occupies very little memory and allows you to capture text screens from within other application programs.

In order for SCRNCAP to run properly, you must start it from DOS. Before running SCRNCAP, you should consider the following points which apply to all TSR programs:

- It should not be installed while a program has "shelled" to DOS.
- The last TSR installed must be the first to be uninstalled.

Using SCRNCAP consists of a few simple steps which are summarized below.

- Run SCRNCAP from DOS.
- Start another program from which screens are to be captured.
- When the desired screen appears, press <Ctrl-S>.
- Specify a name for the screen, then press <Enter>.

If no file is specified when saving in the last step above, files are written to the current path. You may specify a different drive or directory before the file name if you wish.

All screens are saved in BASIC's BSAVE format. This makes them easy to load from the QuickScreen editor. Simply use the (File) **Open Screen File...** command and specify the screen you wish to edit.

One note of warning: SCRNCAP uses the screen name you specify and does *not* caution you if you will overwrite an existing screen with the same name. For this reason, be extremely careful when naming SCRNCAP screens to be saved.

When SCRNCAP no longer is required, you may remove it from memory by exiting any active applications to return to the DOS prompt. Then, run SCRNCAP again and use the "/U" command-line switch at DOS like this:

```
SCRNCAP /U
```

This completely uninstalls the program from memory.

QUICK LIBRARY MAKE UTILITY

To simplify the creation of custom Quick Libraries, we've included a utility called MAKEQLB.EXE. This utility examines a program and all its dependent modules, and creates 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, qlbname, listfile, lib1 lib2, bqlbname
```

You may also specify more than one file name to be examined, by separating each with a blank space:

```
MAKEQLB mainprog1 mainprog2, qlbname, listfile, lib1 lib2,_
   bqlbname
```

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 (if indeed you omit qlbname, be certain to retain the delimiting comma). If you specify "NUL" for the .QLB name, MakeQLB searches for unnecessary DECLARE statements and dead code, but will not create 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 generated .LST 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 .LIB library files 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. For other versions of BASIC, please see Table XXVII.

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.

BASIC Version	BQLB .LIB File Name
4.0	BQLB40.LIB
4.0b	BQLB41.LIB
4.5	BQLB45.LIB
6.0	*depends on QB version number*
7.x	QBXQLB.LIB

Table XXVII: BASIC BQLB .LIB File Names

APPENDICES

APPENDIX A:
MOUSE TIPS

MOUSE TIPS

If you have a mouse you will find QuickScreen's interface to be quite intuitive. The section entitled Menu System and Dialog Boxes clearly explains the mouse interface as it relates to the menu system.

In addition, you may use a mouse for other aspects of QuickScreen. Most mouse operations summarized below require you to hit <Enter> to "keep" whatever has been done with the mouse. If you wish to abandon the mouse function at any time, press <Esc>.

Blocks and Box Drawing

To define a block or to draw a box, place the mouse cursor at a starting point on the screen and press and hold down the left mouse button. Then, drag the mouse until the desired block or box has been drawn. When the left button is released, the block will remain selected or the box will remain drawn. If a block has been selected, you may unselect it by clicking the left mouse button or by defining a new block. If a box has been drawn, a new box may be drawn by repositioning the mouse cursor and by using the left mouse button again.

ASCII Chart & Color Chart

When either the ASCII Chart or Color Chart is presented, you may select a value from the chart by placing the mouse cursor over the desired choice and by pressing the left mouse button.

Drawing Lines

When in Line Draw mode, press and hold down the left mouse button to draw lines as the mouse cursor is moved. This allows free-hand drawing. The mouse cursor may be repositioned to a new screen location by releasing the left button.

APPENDIX B:
QUICKSCREEN EDITING KEYS

QuickScreen Editing Keys

Key Sequence	Result
<Ctrl-A>	ASCII chart
<Ctrl-B>	Box drawing
<Ctrl-D>	Line drawing
<Ctrl-F>	Fill character
<Ctrl-M>	Move marked block
<Ctrl-N>	Insert line above
<Ctrl-P>	Paint block
<Ctrl-R>	Repeat last character
<Ctrl-Y>	Delete line
<Ctrl><Right>	Paint character to right
<Ctrl><Left>	Unpaint character to left
<F1>	Help
<F2>	Display screen
<F4>	Ruler line
<F5>	Center line/block
<Delete>	Delete block
	Delete character under cursor
<Insert>	Insert mode toggle
<Backspace>	Delete character to left of cursor
<Shift><Delete>	Cut block
<Shift><Insert>	Paste block
<Shift><End>, <Delete>	Delete to end of line

APPENDIX C:
COLOR CHART

Color Chart

BACKGROUND

		Black	Blue	Green	Cyan	Red	Magenta	Brown	White
	Black	0	16	32	48	64	80	96	112
	Blue	1	17	33	49	65	81	97	113
	Green	2	18	34	50	66	82	98	114
F	**Cyan**	3	19	35	51	67	83	99	115
O	**Red**	4	20	36	52	68	84	100	116
R	**Magenta**	5	21	37	53	69	85	101	117
E	**Brown**	6	22	38	54	70	86	102	118
G	**White**	7	23	39	55	71	87	103	119
R	**Br Black**	8	24	40	56	72	88	104	120
O	**Br Blue**	9	25	41	57	73	89	105	121
U	**Br Green**	10	26	42	58	74	90	106	122
N	**Br Cyan**	11	27	43	59	75	91	107	123
D	**Br Red**	12	28	44	60	76	92	108	124
	Br Magenta	13	29	45	61	77	93	109	125
	Br Brown	14	30	46	62	78	94	110	126
	Br White	15	31	47	63	79	95	111	127

APPENDIX D:
ASCII CHARACTER CHARTS

Conventional ASCII Character Chart

000	(nul)	016	^P	(dle)	032	sp	048	0	064	@	080	P	096	`	112	p
001	^A	(soh)	017	^Q	(dc1)	033	!	049	1	065	A	081	Q	097	a	113	q
002	^B	(stx)	018	^R	(dc2)	034	"	050	2	066	B	082	R	098	b	114	r
003	^C	(etx)	019	^S	(dc3)	035	#	051	3	067	C	083	S	099	c	115	s
004	^D	(eot)	020	^T	(dc4)	036	$	052	4	068	D	084	T	100	d	116	t
005	^E	(enq)	021	^U	(nak)	037	%	053	5	069	E	085	U	101	e	117	u
006	^F	(ack)	022	^V	(syn)	038	&	054	6	070	F	086	V	102	f	118	v
007	^G	(bel)	023	^W	(etb)	039	'	055	7	071	G	087	W	103	g	119	w
008	^H	(bs)	024	^X	(can)	040	(	056	8	072	H	088	X	104	h	120	x
009	^I	(tab)	025	^Y	(em)	041	)	057	9	073	I	089	Y	105	i	121	y
010	^J	(lf)	026	^Z	(eof)	042	*	058	:	074	J	090	Z	106	j	122	z
011	^K	(vt)	027	^[	(esc)	043	+	059	;	075	K	091	[	107	k	123	{
012	^L	(np)	028	^\	(fs)	044	,	060	<	076	L	092	\	108	l	124	¦
013	^M	(cr)	029	^]	(gs)	045	-	061	=	077	M	093	]	109	m	125	}
014	^N	(so)	030	^^	(rs)	046	.	062	>	078	N	094	^	110	n	126	~
015	^O	(si)	031	^_	(us)	047	/	063	?	079	O	095	_	111	o	127	Δ

Extended ASCII Chart (character codes 128 - 255)

128	Ç	144	É	160	á	176	▒	192	└	208	╨	224	α	240	≡
129	ü	145	æ	161	í	177	▓	193	┴	209	╤	225	ß	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
131	â	147	ô	163	ú	179	│	195	├	211	╙	227	π	243	≤
132	ä	148	ö	164	ñ	180	┤	196	—	212	╘	228	Σ	244	⌠
133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	μ	246	÷
135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	·
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
139	ï	155	¢	171	½	187	╗	203	╦	219	▉	235	δ	251	√
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	ⁿ
141	ì	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	²
142	Ä	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
143	Å	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	

GLOSSARY

absolute cursor coordinate

The row and column positions of the cursor relative to the upper-left corner of the screen.

background color

The color of the background on which text is displayed on the screen.

blink

An attribute which makes a character flash on and off.

block

A rectangular area of text which has been selected using the <Shift> direction keys or the mouse. A block may consist of as little as a single character or as much as the entire display.

box draw characters

Any of the characters in the IBM® PC character set which are used to draw borders.

calculated field

Any field in a form for which a formula has been defined.

clipboard

An area of memory to which text can be temporarily placed and infinitely retrieved.

command line switch

Any parameter on the DOS command line which specifies an option for the program to be executed. QuickScreen, for example, uses the /B command line switch to adjust screens for a monochrome display.

dialog box

An input screen which collects information needed for carrying out a process. For example, QuickScreen's (File) **Open Screen File** pulldown command uses a dialog box.

direction keys

Keys which control the movement of the cursor. These keys typically include the <Up>, <Down>, <Left>, and <Right> keys, and sometimes include <PgUp> and <PgDn>.

fill character

The ASCII character that is to replace characters on the screen when the fill option is chosen.

foreground color

The color of characters on the screen.

form

A screen which has at least one field defined.

.FRM files

A file containing field definitions for a form.

hotkeys

Keys which directly access an item on a menu bar or pulldown menu. The characters corresponding to hotkeys are usually highlighted.

insert mode

The edit mode in which each character to the right of the cursor is moved to the right as new characters are entered.

menu bar

A component of the menu system which presents pulldown menu names on the top line of the screen.

menu bar option

One of the menu names on the menu bar. A menu bar option usually presents a pulldown menu.

overwrite mode

An edit mode in which new characters overwrite existing text on the screen.

pollable routine

A routine which may be repeatedly called in a loop. Pollable routines allow a calling program to carry out other tasks between each polling cycle, effectively simulating multi-tasking.

pulldown menu

A pop-up list of commands available for a given menu bar option.

relative cursor coordinates

The row and column position of the cursor relative to its position at the time the ruler line was displayed.

ruler line

A pop-up moveable window which presents information about the position of the cursor and the character on which the cursor rests.

selected text

Text which has been highlighted using the <Shift> direction keys or the mouse. A block is defined when text is selected.

shortcut keys

Single keys or key combinations which allow direct access to specific pulldown menu commands.

wipe or wipe type

A special effects technique of displaying screens. For example, one wipe makes a screen appear to "explode" into view.

INDEX

— D —